

WebSphere Development Studio Client Advanced Edition
for iSeries



EGL Reference Guide for iSeries

Version 5 Release 1

WebSphere Development Studio Client Advanced Edition
for iSeries



EGL Reference Guide for iSeries

Version 5 Release 1

Note

Before using this information and the product it supports, read the information in "Notices," on page 711.

Second Edition (May 2004)

This edition applies to version 5, release 1, modification 2 of WebSphere Development Studio Client Advanced Edition for iSeries (product number 5724-D46) and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright International Business Machines Corporation 1996, 2004. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Overview 1

Introduction to EGL	1
Development process	1
Run-time configurations	2
Use of a Java wrapper	3
Valid calls	3
Valid transfers	5
Source of additional information on EGL	5

EGL language overview 7

EGL projects, packages, and files	7
EGL project	7
Package	8
EGL files	8
Recommendations	9
Parts	11
Record types and properties	13
References to parts	16
Structure	19
Typedef.	20
Import	26
Background	26
Format of the import statement	26
Primitive types	27
Primitive types at declaration time	28
Relative efficiency of different numeric types	29
Character types	29
Numeric types	32
References to variables and constants	34
Program variables and constants	35
Function variables	35
Arrays and items	36
Rules for referencing variables	36
This	38
Operators and precedence	39
Overview of EGL properties and overrides	40
Item properties	40
Declaration properties	41
Syntax for specifying properties and their overrides	41
Syntax for overriding item properties in a record	42
Items	43
Arrays	64
Arrays of structure items	65
Static arrays of records and data items	67
Dynamic arrays of records and data items	68
EGL statements	70
Keywords	72
Transfer of control across programs	74
Exception handling	75
try blocks	76
Error-related system variables	77
I/O statements	78
Error identification	78
System words	79

Getting started 81

Setting EGL preferences	81
Switching to the EGL or EGL Web perspective	81
Creating a project to work with EGL	82
Creating an EGL Web project	82
Creating an EGL project	82
Specifying database options at project creation	83
Creating an EGL source folder	84
Creating an EGL package	84
Creating an EGL source file	85
Creating an EGL program part	85
Creating an EGL dataTable part	86
Creating an EGL library part	87
Opening a part	87
Locating an EGL source file in the Project Navigator	88
Creating a build file	88
Adding an import statement to a build file	89
Creating a build file with a build descriptor part	89
Creating a build file with a linkage options part	90
Creating a build file with a resource associations part	90
Editing an EGL build path	91
Working with build descriptor parts	92
Adding a build descriptor part	92
Editing general build descriptor options	93
Editing Java run-time properties in a build descriptor	93
Working with linkage options parts	95
Adding a linkage options part	95
Editing the callLink element of a linkage options part	96
Editing the asynchLink element of a linkage options part	97
Editing the transfer-related elements of a linkage options part	98
Working with resource association parts	99
Adding a resource associations part	99
Editing a resource associations part	100

EGL editor 103

Content assist	103
Using the EGL templates with content assist	103
Setting preferences for the EGL editor	104
Setting preferences for source styles	105
Setting preferences for templates	105

EGL debugger 107

Debugger mode	107
Debugger commands	107
Use of build descriptors	110
SQL-database access	111
call statement	111
System type used at debug time	112
EGL debugger port	112
Invoking the EGL debugger from generated code	112

Recommendations	113
Setting preferences for the EGL debugger	114
Starting a non-J2EE program in the EGL debugger	115
Creating a launch configuration in the EGL debugger	116
Preparing a server for EGL Web debugging	116
Starting a server for EGL Web debugging	117
Starting an EGL Web debugging session	117
Using breakpoints in the EGL debugger	118
Stepping through a program in the EGL debugger	118
Viewing variables in the EGL debugger	119

Build 121

Building EGL output	122
Invoking a build plan after generation	123
Generation	123
Generation in the workbench	124
Generation from the workbench batch interface	126
Generation from the EGL SDK	127
Generating for Java	129
Generating for COBOL	132

Web applications 135

Web support	135
Page Designer support for EGL	135
Binding controls to data areas in the page handler	135
Binding controls to functions	136
Using the Quick Edit view	137
Associating an EGL primitive type with a JSP	138
Associating an EGL data item with a JSP	138
Associating an EGL record with a JSP	139
EGL Web service	139
Run-time events	139
A deployment consideration	140
EGL Web Service Wizard	141
Creating an EGL Web service	143
Creating an EGL Web service definition file	147
Editing an EGL Web service definition file	148

Text applications 151

Segmentation in text applications	151
Modified data tag and modified property	152
Interacting with the user	152
Setting the modified property	152
Testing whether the form is modified	153
Examples	153
Creating an EGL formGroup part	154

Files and databases 157

Resource associations and file types	157
Resource associations part	157
File types	158
Record types and VSAM	158
For further details	159
Record and file type cross-reference	159
Logical unit of work	159
Unit of work for Java	160
MQSeries support	161
Connections	161

Include message in transaction	162
Customization	162
MQSeries-related EGL keywords	163
Direct MQSeries calls	165
MQ record properties	169
Options records for MQ records	169
SQL support	171
EGL statements and SQL	172
Result-set processing	174
SQL records and their uses	176
Database access at declaration time	179
Dynamic SQL	180
resultSetID	181
SQL examples	181
SQL-specific tasks	188
Database authorization and table names	197
Default database	198
Informix and EGL	199
SQL record internals	199
SQL data codes and EGL host variables	200
VSAM support	202
Access prerequisites	202
System name	203

Deploying EGL-generated Java output 205

Installing the EGL run-time code for Java	205
Setting up the J2EE run-time environment for EGL-generated code	206
Setting deployment-descriptor values	207
Understanding how a standard JDBC connection is made	208
Setting up a J2EE JDBC connection	209
Setting up the TCP/IP listener	210
Setting up the J2EE server for CICSJ2C calls	213
Generating into a project	214
Updating the deployment descriptor manually	216
Providing access to non-EGL jar files	217
Eliminating duplicate jar files	219
Setting the variable EGL_GENERATORS_PLUGINDIR	219
Generating into a directory	220
Deploying a linkage properties file	223
Updating the J2EE environment file	224
Updating the deployment descriptor manually	224
Providing access to non-EGL jar files	225
Eliminating duplicate jar files	227
Setting up EJB projects (additional tasks)	227
Generating deployment code for EJB projects	227
Setting the JNDI name for EJB projects	228
Deploying Java applications outside of J2EE	228
Setting up the UNIX curses library for EGL run time	228

Reference pages 231

Association elements	231
commit	231
conversionTable	231
duplicates	231
fileType	232
fileName	232

formFeedOnClose	232	sqlValidationConnectionURL	265
replace	232	sysCodes	266
system	233	system	266
systemName	233	targetNLS	267
text.	233	templateDir	267
Build descriptor part	234	VAGCompatibility.	268
Master build descriptors.	234	validateMixedItems	268
Precedence of options	235	validateOnlyIfModified	269
Example	235	validateSQLStatements	269
Setting the default build descriptors.	237	EGL build-file format.	269
Build descriptor options.	237	Build script reference.	271
bidiConversionTable	243	Build script	271
buildPlan.	243	Build scripts delivered with EGL	271
checkNumericOverflow	243	Options required in EGL build scripts	271
checkType	244	Symbolic parameters	272
cicsj2cTimeout	244	Predefined symbolic parameters for EGL	
clientCodeSet	245	generation	273
commentLevel	245	Build server	275
currencySymbol	245	COBOL reserved-word file	275
dbms	246	Format of COBOL reserved-word file	276
debugTrace	246	Compatibility with VisualAge Generator	276
decimalSymbol.	246	Data initialization	277
destDirectory	246	Data conversion	279
destHost	247	Data conversion when you generate a COBOL	
destLibrary	248	program	279
destPassword	248	Data conversion when the invoker is Java code	280
destPort	248	Conversion algorithm	281
destUserID	249	Bidirectional language text	282
eliminateSystemDependentCode	249	Dataltem part	284
enableJavaWrapperGen	250	Dataltem part in EGL source format.	284
fillWithNulls	250	DataTable part	285
genDataTables	250	Types of data tables	286
genDDSFile	251	Data-table generation.	286
genDirectory	251	Properties of the data table.	286
genFormGroup.	252	DataTable part in EGL source format	287
genHelpFormGroup	252	EGL reserved words	290
genProject	252	Words that are reserved outside of an SQL	
genProperties	254	statement.	291
initIORecords	255	EGL source format	292
initNonIOData	255	EGL statements.	293
J2EE	255	add.	293
leftAlign	256	Assignments.	296
linkage	256	call	299
math	256	case	302
nextBuildDescriptor	257	close	303
oneFormItemCopybook	257	Comments	305
positiveSignIndicator	257	converse	306
prep	258	delete	307
reservedWord	259	display	309
resourceAssociations	259	execute	309
serverCodeSet	259	exit.	313
sessionBeanID	259	forward	315
setFormItemFull	261	Function invocations	316
spacesZero	261	get	318
sqlDB	262	get next	324
sqlErrorTrace	262	get previous.	328
sqlID	263	goTo	332
sqlIOTrace	263	if, else.	332
sqlJDBCDataSource	263	move	333
sqlJNDIName	264	open	335
sqlPassword.	264	prepare	339

print	341	Identifying the programs or records to which an element refers	440
replace	341	asynchLink element	441
return	344	callLink element	443
set	345	transferToProgram element	459
show	353	transferToTransaction element	462
transfer	354	Name aliasing	463
try	355	How names are aliased	464
while	356	How COBOL names are aliased	464
EGL system limits	356	How Java names are aliased	465
Expressions	357	How Java wrapper names are aliased	466
Logical expressions	358	Naming conventions	468
Numeric expressions	364	PageHandler part	469
String expressions	365	Output associated with a page handler	469
Form part	366	Validation	470
Text forms	367	Run-time scenario	470
Print forms	368	PageHandler part in EGL source format	472
Form part in EGL source format	370	Program part	477
FormGroup part	376	Program part in EGL source format	478
FormGroup part in EGL source format	377	Program parameters	483
pfKeyEquate	380	Program part properties	485
Function part	380	Program data other than parameters	487
Function part in EGL source format	381	Record parts	490
Generated output	386	Basic record part in EGL source format	491
Generated output (reference)	387	Indexed record part in EGL source format	492
Build plan	392	MQ record part in EGL source format	493
COBOL program	393	Relative record part in EGL source format	495
Enterprise JavaBean (EJB) session bean	393	Serial record part in EGL source format	497
Java program, page handler, and library	394	SQL record part in EGL source format	498
J2EE environment file	394	Properties that support variable-length records	502
Java wrapper	395	Run unit	504
Library (generated output)	403	Structure item in EGL source format	505
Linkage properties file	404	Syntax diagram	506
Program properties file	404	System words in alphabetical order	508
Results file	405	Data conversion (system words)	518
Generation reference	405	Date and time (system words)	521
EGL command file	405	Exception handling and status (system words)	526
EGLCMD	407	File and database (system words)	537
EGL build path and eglpath	409	Java access functions	556
EGLSDK	410	Mathematical (system words)	580
Master build descriptor	413	String handling (system words)	597
in	416	Web-application system words	607
Examples with a one-dimensional array	417	Miscellaneous system words	610
Examples with a multidimension array	417	Use declaration	631
I/O error values	418	Background	631
duplicate	419	In a program or library part	632
endOfFile	419	In a FormGroup part	633
format	420	In a pageHandler part	634
noRecordFound	420	EGL Java run-time error codes	635
unique	421	EGL Java run-time error code CSO7000E	636
Java run-time properties	421	EGL Java run-time error code CSO7015E	637
In a J2EE environment	421	EGL Java run-time error code CSO7016E	637
In a non-J2EE Java environment	422	EGL Java run-time error code CSO7020E	637
Build descriptors and program properties	422	EGL Java run-time error code CSO7021E	637
For additional information	422	EGL Java run-time error code CSO7022E	638
Java run-time properties (details)	423	EGL Java run-time error code CSO7023E	638
Linkage properties file (details)	431	EGL Java run-time error code CSO7024E	638
Library part	433	EGL Java run-time error code CSO7026E	638
Library part in EGL source format	434	EGL Java run-time error code CSO7045E	639
Linkage options part	439	EGL Java run-time error code CSO7050E	639
Specifying when linkage options are final	439		
Elements of a linkage options part	439		

Contents vii

EGL Java run-time error code VGJ0168E	677	EGL Java run-time error code VGJ0609I	696
EGL Java run-time error code VGJ0200E	677	EGL Java run-time error code VGJ0610I	696
EGL Java run-time error code VGJ0201E	678	EGL Java run-time error code VGJ0611E	696
EGL Java run-time error code VGJ0202E	678	EGL Java run-time error code VGJ0612I	697
EGL Java run-time error code VGJ0203E	678	EGL Java run-time error code VGJ0614E	697
EGL Java run-time error code VGJ0204E	679	EGL Java run-time error code VGJ0615E	697
EGL Java run-time error code VGJ0215E	679	EGL Java run-time error code VGJ0616E	697
EGL Java run-time error code VGJ0216E	679	EGL Java run-time error code VGJ0617E	698
EGL Java run-time error code VGJ0217E	680	EGL Java run-time error code VGJ0700E	698
EGL Java run-time error code VGJ0250E	680	EGL Java run-time error code VGJ0701E	698
EGL Java run-time error code VGJ0300E	681	EGL Java run-time error code VGJ0702E	699
EGL Java run-time error code VGJ0301E	682	EGL Java run-time error code VGJ0703E	699
EGL Java run-time error code VGJ0302E	682	EGL Java run-time error code VGJ0705E	699
EGL Java run-time error code VGJ0303E	683	EGL Java run-time error code VGJ0706E	699
EGL Java run-time error code VGJ0304E	683	EGL Java run-time error code VGJ0707E	700
EGL Java run-time error code VGJ0305E	684	EGL Java run-time error code VGJ0708E	700
EGL Java run-time error code VGJ0306E	685	EGL Java run-time error code VGJ0709E	700
EGL Java run-time error code VGJ0307E	685	EGL Java run-time error code VGJ0710E	701
EGL Java run-time error code VGJ0308E	686	EGL Java run-time error code VGJ0711E	701
EGL Java run-time error code VGJ0315E	686	EGL Java run-time error code VGJ0712E	701
EGL Java run-time error code VGJ0320E	687	EGL Java run-time error code VGJ0750E	702
EGL Java run-time error code VGJ0330E	687	EGL Java run-time error code VGJ0751E	702
EGL Java run-time error code VGJ0331E	688	EGL Java run-time error code VGJ0752E	703
EGL Java run-time error code VGJ0350E	688	EGL Java run-time error code VGJ0754E	703
EGL Java run-time error code VGJ0351E	688	EGL Java run-time error code VGJ0755E	703
EGL Java run-time error code VGJ0352E	688	EGL Java run-time error code VGJ0770E	704
EGL Java run-time error code VGJ0400E	689	EGL Java run-time error code VGJ0800E	704
EGL Java run-time error code VGJ0401E	689	EGL Java run-time error code VGJ0801E	704
EGL Java run-time error code VGJ0402E	689	EGL Java run-time error code VGJ0802E	704
EGL Java run-time error code VGJ0403E	689	EGL Java run-time error code VGJ1000E	705
EGL Java run-time error code VGJ0416E	690	EGL Java run-time error code VGJ1001E	705
EGL Java run-time error code VGJ0450E	690	EGL Java run-time error code VGJ1002E	705
EGL Java run-time error code VGJ0500E	691	EGL Java run-time error code VGJ1003E	706
EGL Java run-time error code VGJ0502E	691	EGL Java run-time error code VGJ1004E	706
EGL Java run-time error code VGJ0503E	691	EGL Java run-time error code VGJ1005E	706
EGL Java run-time error code VGJ0504E	691	EGL Java run-time error code VGJ1006E	707
EGL Java run-time error code VGJ0505E	692	EGL Java run-time error code VGJ1007E	707
EGL Java run-time error code VGJ0506E	692	EGL Java run-time error code VGJ1008E	707
EGL Java run-time error code VGJ0507E	692	EGL Java run-time error code VGJ1009E	708
EGL Java run-time error code VGJ0508E	692	EGL Java run-time error code VGJ9900E	708
EGL Java run-time error code VGJ0510E	693	EGL Java run-time error code VGJ9901E	709
EGL Java run-time error code VGJ0511E	693		
EGL Java run-time error code VGJ0512E	693		
EGL Java run-time error code VGJ0600E	694		
EGL Java run-time error code VGJ0601E	694		
EGL Java run-time error code VGJ0603E	694		
EGL Java run-time error code VGJ0604E	695		
EGL Java run-time error code VGJ0607E	695		
EGL Java run-time error code VGJ0608E	695		
		Appendix. Notices	711
		Programming interface information	713
		Trademarks and service marks	713
		Index	715

Overview

Introduction to EGL

Enterprise Generation Language (EGL) is a development environment and programming language that lets you write full-function applications quickly and that frees you to focus on the business problem your code is addressing rather than on software technologies. You can use similar I/O statements to access different types of external data stores, for example, whether those data stores are files, relational databases, or message queues. The details of Java™ and J2EE are hidden from you, too, so you can deliver enterprise data to browsers even if you have minimal experience with Web technologies.

After you code an EGL program, you generate it to create Java or COBOL source; then EGL prepares the output to produce executable objects. EGL also can provide these services:

- Places the source on a deployment platform outside of the development platform
- Prepares the source on the deployment platform
- Sends status information from the deployment platform to the development platform, so you can check the results

EGL even produces output that facilitates the final deployment of the executable objects.

An EGL program written for one target platform can be converted easily for use on another. The benefit is that you can code in response to current platform requirements, and many details of any future migration are handled for you. EGL also can produce multiple parts of an application system from the same source.

Related concepts

“Development process”

“EGL projects, packages, and files” on page 7

“Generated output” on page 386

“Parts” on page 11

“Run-time configurations” on page 2

Related tasks

“Creating a project to work with EGL” on page 82

Related reference

“EGL editor” on page 103

“EGL source format” on page 292

Development process

Your work with EGL includes the following steps:

Setup

You set up a work environment; for example, you set preferences and create projects.

Create and open EGL files

You begin to create the source code.

Declaration

You create and specify the details of your code.

Validation

At various times (such as when you save a file), EGL reviews your declarations and indicates whether they are syntactically correct and (to some extent) whether they are internally consistent.

Debugging

You can interact with a built-in debugger to ensure that your code fulfills your requirements.

Generation

EGL validates your declarations and creates output, including source code.

Preparation

EGL prepares the source code to produce executable objects. In some cases this step places the source code on a deployment platform outside of the development platform, prepares the source code on the deployment platform, and sends a results file from the deployment platform to the development platform.

Deployment

EGL produces output that makes deployment of the executable objects easier.

Related concepts

"EGL debugger" on page 107

"Introduction to EGL" on page 1

Related tasks

"Generating into a directory" on page 220

"Generating into a project" on page 214

"Setting up the J2EE run-time environment for EGL-generated code" on page 206

Related reference

"EGL editor" on page 103

"EGL source format" on page 292

Run-time configurations

EGL provides the following kinds of generated code, among others:

- A Java program can be generated for any of several supported platforms. You can deploy the program outside of J2EE or in the context of any of the following J2EE containers--
 - J2EE application client
 - J2EE Web application
 - EJB container; in this case, you also generate an EJB session beanYou can use an EGL-generated Java program in an EGL Web service, but only if the program is non-interactive and receives control by a call. The program in this case may be deployed inside or outside of J2EE.
- A non-interactive COBOL program also can be generated to run on iSeries[™].

In addition, EGL provides a way to define a Web application that has the following characteristics:

- Delivers graphical pages to Web browsers
- Is able to store and retrieve data for a potentially large number of users
- Is embedded in a Java-based framework called JavaServer Faces

For details on this specialized support for Web applications, see *PageHandler part*.

Finally, you can use EGL to generate a Java wrapper, as described in the next section.

Use of a Java wrapper

The EGL-generated Java wrapper is a set of classes that let you invoke an EGL-generated program from non-EGL-generated Java code; for example, from an action class in a Struts- or JSF-based J2EE web application or from a non-J2EE Java program. The Java-to-EGL integration task is as follows:

1. Generate Java wrapper classes, which are specific to a generated program
2. Incorporate those wrapper classes into the non-generated Java code
3. From the non-generated Java code, invoke the wrapper-class methods to make the actual call and to convert data between these two formats:
 - The data-type formats used by Java
 - The primitive-type formats required when passing data to and from the EGL-generated program

Valid calls

The next table shows the valid calls to or from the EGL-generated code.

Calling object	Called object
An EGL-generated Java wrapper in a Java class that is outside of J2EE	An EGL-generated Java program (non-J2EE)
	An EGL-generated Java program in a J2EE application client
	An EGL-generated EJB session bean
	A CICS® COBOL program that was generated by VisualAge® Generator
An EGL-generated Java wrapper in a J2EE application client	An EGL-generated Java program (non-J2EE)
	An EGL-generated Java program in a J2EE application client
	An EGL-generated EJB session bean
	A CICS COBOL program that was generated by VisualAge Generator
An EGL-generated Java wrapper in a J2EE Web application	An EGL-generated Java program (non-J2EE)
	An EGL-generated Java program in a J2EE application client
	An EGL-generated Java program in the same J2EE Web application
	An EGL-generated EJB session bean
	A CICS COBOL program that was generated by VisualAge Generator

Calling object	Called object
An EGL-generated Java program that is outside of J2EE	An EGL-generated Java program (non-J2EE)
	An EGL-generated Java program in a J2EE application client
	An EGL-generated EJB session bean
	A CICS COBOL program that was generated by VisualAge Generator
	A non-EGL-generated program that was written in C or C++
	A non-generated program that was written in any language and runs under CICS
An EGL-generated Java program that is in a J2EE application client	An EGL-generated Java program (non-J2EE)
	An EGL-generated Java program in a J2EE application client
	An EGL-generated EJB session bean
	An EGL-generated CICS COBOL program
	A non-generated program that was written in any language and runs under CICS
	A non-generated program that was written in C or C++
An EGL-generated Java program in a J2EE Web application	An EGL-generated Java program (non-J2EE)
	An EGL-generated Java program in a J2EE application client
	An EGL-generated Java program in the same J2EE Web application
	An EGL-generated EJB session bean
	A CICS COBOL program that was generated in VisualAge Generator
	A non-generated program written in C or C++
An EGL-generated EJB session bean	An EGL-generated Java program (non-J2EE)
	An EGL-generated Java program in a J2EE application client
	An EGL-generated EJB session bean
	A CICS COBOL program that was generated by VisualAge Generator
	A non-generated program written in C or C++
An EGL-generated COBOL program on iSeries	An EGL-generated COBOL program on iSeries
	A non-EGL-generated program written in any language and running on iSeries
A non-EGL-generated program written in any language and running on iSeries	An EGL-generated COBOL program on iSeries
	A non-EGL-generated program written in any language and running on iSeries

Valid transfers

The next table shows the valid transfers to or from EGL-generated code.

Transferring object	Receiving object
An EGL-generated Java program that is outside of J2EE	An EGL-generated Java program (non-J2EE)
An EGL-generated Java program that is in a J2EE application client	An EGL-generated Java program in the same J2EE application client
An EGL-generated Java program in a J2EE Web application	An EGL-generated Java program in the same J2EE Web application
An EGL-generated program on iSeries	An EGL-generated COBOL program on iSeries
	A non-EGL-generated program (written in any language and running on iSeries)
An non-EGL-generated program written in any language and running on iSeries	An EGL-generated COBOL program on iSeries
	A non-EGL-generated program written in any language and running on iSeries

Related concepts

“COBOL program” on page 393

“Generated output” on page 386

“Introduction to EGL” on page 1

“Java program, page handler, and library” on page 394

“Java wrapper” on page 395

“PageHandler part” on page 469

Related tasks

“Setting up the J2EE run-time environment for EGL-generated code” on page 206

Source of additional information on EGL

The most recent copy of this document is at the following Web site:

<http://www.ibm.com/developerworks/websphere/studio/egldocs.html>

For details on migrating source code written in VisualAge Generator, see the *VisualAge Generator to Enterprise Generation Language Migration Guide* (file `vagenmig.pdf`), which is on the Web site mentioned earlier, in the root of the WebSphere® Studio Application Developer installation directory, and in the help system.

For details on run time issues when you generate COBOL output, see the EGL Server Guide for iSeries, which is on the Web site mentioned earlier and in the help system.

Related concepts

“Introduction to EGL” on page 1

EGL language overview

EGL projects, packages, and files

An EGL project includes zero to many source folders, each of which includes zero to many packages, each of which includes zero to many files. Each file contains zero to many parts.

EGL project

An EGL project is characterized by a set of properties, which are described later. In the context of an EGL project, EGL automatically performs validation and resolves part references when you perform certain tasks; for example, when you save an EGL file or build file. In addition, if you are working with page handler parts (the output of which is used to debug Web applications in the Websphere test environment), EGL automatically generates output, but only in this case:

- You have set the automatic build process after selecting these options: **Window > Preferences > Workbench > Perform build automatically on resource modification**
- You have established a default build descriptor as a preference or property

An EGL project is formed by selecting **EGL** or **EGL Web** as the project type when you create a new project. You assign properties while working through the steps of project creation. To begin modifying your choices after you have completed those steps, right-click the project name and when a context menu is displayed, click **Properties**.

The EGL properties are as follows:

EGL source folder

One or more project folders that are the roots for the project's packages, each of which is a set of subdirectories. A source folder is useful for keeping EGL source separate from Java files and for keeping EGL source files out of the Web deployment directories. It is recommended that you specify EGL source folders in all cases; but if a source folder is not specified, the only source folder is the project directory.

The value of this property is stored in a file named `.eglp` in the project directory and is saved in the repository (if any) that you use to store EGL files.

The EGL project wizards each create one source folder named **EGLSource**.

EGL build path

The list of projects that are searched for any part that is not found in the current project.

The value of this property is stored in a file named `.eglp` in the project directory and is saved in the repository (if any) that you use to store EGL files.

In the following example of an `.eglp` file, **EGLSource** is a source folder in the current project, and **AnotherProject** is a project in the EGL path:

```
<?xml version="1.0" encoding="UTF-8"?>
<eglp>
  <eglpentry kind="src" path="EGLSource"/>
  <eglpentry kind="src" path="\AnotherProject"/>
</eglp>
```

The source folders for AnotherProject are determined from the .eglpath file in *that* project.

Default build descriptors

The build descriptors that allow you to generate output quickly, as described in *Generation in the workbench*.

Package

A package is a named collection of related source parts.

By convention, you achieve uniqueness in package names by making the initial part of the package name an inversion of your organization's Internet domain name. For example, the IBM® domain name is ibm.com®, and the EGL packages begin with "com.ibm". By using this convention, you gain some assurance that the names of Web programs developed by your organization will not duplicate the names of programs developed by another organization and can be installed on the same server without possibility of a name collision.

The folders of a given package are identified by the package name, which is a sequence of identifiers separated by periods (.), as in this example:

com.mycom.mypack

Each identifier corresponds to a subfolder under an EGL source folder. The directory structure for com.mycom.mypack, for example, is \com\mycom\mypack, and the source files are stored in the bottom-most folder; in this case, in mypack. If the workspace is c:\myWorkspace, if the project is new.project, and if the source folder is EGLSource, the path for that package is as follows:

c:\myWorkspace\new.project\EGLSource\com\mycom\mypack

The parts in an EGL file all belong to the same package. The file's package statement, if any, specifies the name of that package. If you do not specify a package statement, the parts are stored directly in the source folder and are said to be in the *default package*. It is recommended that you always specify a package statement because files in the default package cannot be shared by parts in other packages or projects.

Two parts with the same identifier may not be defined in the same package. *It is strongly recommended that you avoid using the same package name under different projects or different folders.*

The package for generated Java output is the same as the EGL file package in most cases.

EGL files

Each EGL file belongs to one of these categories:

Source file

An EGL source file (extension .egl) contains logic, data, and user interface parts and is written in EGL source format. Parts of the following kinds are called primary parts:

- DataTable
- FormGroup
- Library
- PageHandler

- Program

Other parts are called subparts.

An EGL source file can include zero to many subparts but can include no more than one primary part. The primary part (if any) must be at the top level of the file and must have the same name as the file.

Build file

An EGL build file (extension .eglbld) contains any number of build parts and is written in Extensible Markup Language (XML), in EGL build-file format. You can review the related DTD, which is in the following directory:

```
installationDir\wstools\eclipse\plugins\
com.ibm.etools.egl_version
```

installationDir

The WebSphere Studio installation directory, such as c:\myStudio

version

The installed version of the plugin; for example, 5.1.1

The file name (like egl_5_1.dtd) begins with the letters *egl* and an underscore.

Web service definition file

A Web service definition file (extension .eglwsd) contains a single Web service definition, as described in *EGL Web service*.

In most cases, this file is an output of your interaction with the EGL Web Services wizard. The information in this file is used at debug time, generation time, and run time.

A Web service definition file is written in XML. You can review the related DTD, which is in the following directory:

```
installationDir\wstools\eclipse\plugins\
com.ibm.etools.egl_version
```

installationDir

The WebSphere Studio installation directory, such as c:\myStudio

version

The installed version of the plugin; for example, 5.1.2

The file name (like egl_wsd_5_1.dtd) begins with the characters *egl_wsd* and an underscore.

After you add parts to files, you can use a repository to maintain a history of changes.

Recommendations

This section gives recommendations for setting up your development projects.

For build descriptors

Team projects should appoint one person as a build-descriptor developer. The tasks for that person are as follows:

- Create the build descriptors for the source-code developers
- Put those build descriptors in a project separate from the source code projects; and make that separate project available in the repository or by some other means

- Ask the source-code developers to set the property **default build descriptors** in their projects, so that the property references the appropriate build descriptors
- If a small subset of the build descriptor options (such as for user ID and password) varies from one source-code developer to the next, ask each source-code developer to do as follows:
 - Code a personal build descriptor that uses the option **nextBuildDescriptor** to point to a group build descriptor
 - Ask the source-code developers to set the property **default build descriptors** in their files, folders, or packages, so that the property references the personal build descriptor. They do not specify the property at the project level because the project-level property is under repository control, along with other project information.

For additional information, see *Build descriptor part*.

For packages

For packages, recommendations are as follows:

- Do not use the same package name in different projects or source directories
- Do not use the default package

Part assignment

For parts, many of the recommendations refer to good practices, not hard requirements. Fulfill even the optional recommendations unless you have good reason to do otherwise:

- A *requirement* is that you put JSPs in the same project as their associated page handlers.
- If a non-primary part (like a serial record part) is used only by one program, library, or page handler, place the non-primary part in the same file as the using part.
- If a part is referenced from different files in the same package, put that part in a separate file in the package.
- If a part is shared across packages in a single project, place that part in a separate package in that project.
- Put code for completely unrelated applications in different projects. The project is the unit for transferring code between your local directory structure and the repository. Design project structure so that developers can minimize the amount of code they have to have loaded into their development system.
- Name projects, packages, and files in a way that reflects the use of the parts they contain.
- If your process emphasizes code ownership by a developer, do not assign parts for different owners to the same file.
- Assign parts to packages with a clear understanding of the purpose of the package; and group those parts by the closeness of the relationship between them.

The following distinction is important:

- Moving a part from file to file in the same package does not require that you change import statements in other files.
- Moving a part from one package to another may require an import statement to be added or changed in every file that references the moved part.

Related concepts

“Build descriptor part” on page 234

“EGL Web service” on page 139
“Generation in the workbench” on page 124
“References to parts” on page 16
“Import” on page 26
“Introduction to EGL” on page 1
“Parts”

Related reference

“EGL build-file format” on page 269
“EGL source format” on page 292
“EGL statements” on page 70

Parts

An EGL file contains a set of *parts*, each of which is a unit of declaration in the overall declaration of the program. You declare parts in whatever order is convenient for you, and you name each one.

Parts are categorized as logic, data, user interface (UI), or build parts:

- *Logic parts* define a run-time sequence that you write in the EGL procedural language--
 - A *function part* is the basic unit of logic and is included in program, pageHandler, and library parts.
 - A *pageHandler part* (or *page handler*) is a specialized program that controls the interaction between the user and a Web page.
 - A *program part* is one of these types:
 - A *textUI program* interacts with the user by way of a character-based display. The display appears in a 3270 screen or a command window, not in a Web browser.
 - A *basic program* performs a task without interacting with the user in real time.

You can declare a basic or textUI program to be a *main program*, which is started by the user, by a program transfer other than a call, or directly by an operating-system process. Also, you can declare either kind of program to be a *called program*, which can be invoked only by a call.

A basic program can be called as a subroutine from another program or from a page handler and can be deployed in the context of an *EGL Web service*. For other details on the run-time deployment of main and called programs, see *Run-time configurations*.

- A *library part* is a collection of shared functions and variables and is generated and compiled independent of any program or page handler. The library resources can be made available in your program, but only when you are generating Java.
- *Data parts* define the data structures that are available to your program. In some cases a data part includes a *structure*, which is a hierarchical layout of *structure items* that each define an area of memory. Each structure item is substructured (as a word is substructured into letters) or is not divisible (as a letter is not divisible).

Some data parts can act as *typedefs* (models of format) for variables and for other data parts:

- A *record part* contains a structure, and the structure items are usually called *record items*; and a subset of record parts can access data in permanent storage.
- A *dataItem part* defines a data area that is not divisible.

A *dataTable part* is an independently generated data part containing an array of rows initialized with values specified in the part definition. Unlike the other data parts, each *dataTable part* is essentially a global variable that is available by name throughout your program.

- *UI (user interface) parts* describe the layout of data presented to the user in fixed-font screen and print forms. UI parts are of the following types:
 - A *form part* is an organization of data that is presented to the user. One kind of form part organizes the data sent to a screen in a text application, and another organizes the data sent to a printer in any kind of application. A form includes an internal structure, and the structure items are usually called *fields* or *form items*.
 - A *formGroup part* is a collection of forms and is generated as a separate output. A program can include only one form group for most uses, along with one form group for help-related output. The same form can be included in multiple form groups.

Each of the previous UI parts is essentially a global variable that is available by name throughout your program.

You create Web user interfaces with Page Designer, which builds a JSP file and associates it with an EGL page handler. The JSP file replaces the role of the UI part for applications that interact with the user by way of the Web.

- *Build parts* define a variety of processing characteristics:
 - A *build descriptor part* controls the generation process and indicates what other control parts are read during that process.
 - A *linkage options part* gives details on how a generated program transfers to and from other programs. The information in this part is used at generation time, test time, and run time.
 - A *resource associations part* relates an EGL record with the information needed to access a file on a particular target platform; the information in this part is used at generation time, test time, and run time.

Logic, data, and user interface parts are also categorized as primary parts or subparts. The primary parts are as follows:

- DataTable
- FormGroup
- Library
- PageHandler
- Program

An EGL source file can include zero to many subparts but can include no more than one primary part. The primary part (if any) must be at the top level of the file and must have the same name as the file.

Related concepts

“Build descriptor part” on page 234

“Compatibility with VisualAge Generator” on page 276

“DataItem part” on page 284

“EGL projects, packages, and files” on page 7

“Function part” on page 380
“Import” on page 26
“Introduction to EGL” on page 1

“Linkage options part” on page 439
“Program part” on page 477
“Record parts” on page 490
“References to parts” on page 16
“References to variables and constants” on page 34
“Resource associations and file types” on page 157
“Run-time configurations” on page 2
“Structure” on page 19
“Typedef” on page 20
“Web support” on page 135

Related reference

“EGL build-file format” on page 269
“EGL editor” on page 103
“EGL source format” on page 292
“EGL statements” on page 70
“Primitive types” on page 27

Record types and properties

Several EGL record types are available:

- basicRecord
- indexedRecord
- mqRecord
- relativeRecord
- serialRecord
- SQLRecord

For details on what target systems support what record types, see *Record and file-type cross-reference*. For details on how record parts are initialized, see *Data initialization*.

basicRecord

A basic record has a structure but no properties. This type of record is used for internal processing and cannot access data storage.

Only a basic record part has the data declaration property **redefines**, which (if set) identifies a record whose memory is accessed by the basic record.

In a main program, the program property **inputRecord** identifies a basic record that is initialized automatically, as described in *Data initialization*.

indexedRecord

An indexed record lets you to work with a file that is accessed by a *key value*, which identifies the logical position of a record in the file. You can read the file by invoking a **get**, **get next**, or **get previous** statement. Also, you can write to the file by invoking an **add** or **replace** statement; and you can remove a record from the file by invoking a **delete** statement.

The properties of an indexed record include these:

- **fileName** is required. For details on the meaning of your input, see *Resource associations (overview)*. For details on the valid characters, see *Naming conventions*.
- **keyItem** is required and can only be a structure item that is unique in the same record. You must use an unqualified reference to specify the key item; for example, use *myItem* rather than *myRecord.myItem*. (In an EGL statement, however, you can reference the key item as you would reference any item.)

See also *Properties that support variable-length records*.

mqRecord

An MQ record lets you access an MQSeries® message queue. For details, see *MQSeries support*.

relativeRecord

A relative record lets you to work with a data set whose records have these properties:

- Are fixed length
- Can be accessed by an integer that represents the sequential position of the record in the file

The properties of a relative record are as follows:

- **fileName** is required. For details on the meaning of your input, see *Resource associations (overview)*. For details on the valid characters, see *Naming conventions*.
- **keyItem** is required. The key item can be any of these areas of memory:
 - A structure item in the same record
 - A structure item in a record that is global to the program or is local to the function that accesses the record
 - A data item that is global to the program or is local to the function that accesses the record

You must use an unqualified reference to specify the key item. For example, use *myItem* rather than *myRecord.myItem*. (In an EGL statement, you can reference the key item as you would reference any item.) The key item must be unique in the local scope of the function that accesses the record or must be absent from local scope and unique in global scope.

The key item has these characteristics:

- Has a primitive type of BIN, DECIMAL, INT, or NUM
- Contains no decimal places
- Allows for 9 digits at most

Only the **get** and **add** statements use the key item, but the key item must be available to any function that uses the record for file access.

serialRecord

A serial record lets you access a sequentially accessed file or data set. You can read from the file by invoking a **get** statement, and a series of **get next** statements reads the file records sequentially, from the first to the last. You can write to the file by invoking an **add** statement, which places a new record at the end of the file.

Serial record properties include **fileName**, which is required. For details on the meaning of your input for that property, see *Resource associations (overview)*. For details on the valid characters, see *Naming conventions*.

See also *Properties that support variable-length records*.

sqlRecord

An SQL record provides special services when you access a relational database.

Each SQL record has the following properties:

- An entry in **tableNames** identifies an SQL table associated with the SQL record. You may reference multiple tables in a join, but restrictions ensure that you do not write to multiple tables with a single EGL statement. You may associate a given table name with a *label*, which is an optional, short name used to reference the table in an SQL statement.
- **defaultSelectCondition** is optional. The conditions become part of the WHERE clause in the default SQL statements. The WHERE clause is meaningful when an SQL record is used in an EGL **open**, **get**, or **get next** statement.
In most cases, the SQL default select condition supplements a second condition, which is based on an association between the key item values in the SQL record and the key columns of the SQL table.
- **tableNameVariables** is optional. You can specify one or more variables whose content at run time determines what database tables to access, as described in *Dynamic SQL*.
- **keyItems** is optional. Each key item can only be a structure item that is unique in the same record. You must use an unqualified reference to specify each of those items; for example, use *myItem* rather than *myRecord.myItem*. (In an EGL statement, however, you can reference a key item as you would reference any item.)

For details, see *SQL support*.

Related concepts

“Dynamic SQL” on page 180
“MQSeries support” on page 161
“Record parts” on page 490
“Resource associations and file types” on page 157
“SQL support” on page 171

Related reference

“add” on page 293
“close” on page 303

“Data initialization” on page 277
“delete” on page 307
“execute” on page 309
“get” on page 318
“get next” on page 324
“get previous” on page 328
“MQ record properties” on page 169
“Naming conventions” on page 468
“open” on page 335
“prepare” on page 339
“Properties that support variable-length records” on page 502
“Record and file type cross-reference” on page 159
“replace” on page 341
“SQL item properties” on page 57
“sysVar.terminalID” on page 629

References to parts

This section describes a set of rules that determine how EGL identifies the part to which a name refers. These rules are important in the following situations:

- One function invokes another
- A non-function part (a dataItem part, for example) refers to a validator function
- A part acts as a typedef (a model of format) in the declaration of a structure item or variable
- One part references another in a use declaration
- One build part references another

A second set of rules determine how EGL resolves variable references. For details, see *References to variables and constants*.

Basic visibility rules

In the simplest case, you define parts one after the next in a single package, without declaring one part within another. The following list omits many details, but shows a series of parts that are at the same hierarchical level:

```
Function: Function01
Function: Function02
Function: Function03
Record:   Record01
```

Parts at the same level are available to one another. Function01, for example, can invoke one or both of the other functions; and Record01 can be used as a typedef for variables in each of the three functions.

In most cases, a part *cannot* nest another part. The exceptions are as follows:

- A program, library, or pageHandler can nest functions, but even then, the inclusion must be direct; a function cannot nest another function
- A form group can nest forms

An example with nested parts is as follows:

```
Program: Program01
  Function: Function01
  Function: Function02
Function: Function03
Record:   Record01
```

Parts at the top level are available to every other part in the package. However, the nested parts (Function01 and Function02) are available only to a subset of parts in the package:

- They are available to each other.
- They are available to the nesting part and to functions that are used by the nesting part at run time. If Function01 invokes Function03, for example, Function03 can invoke Function02 because Function03 is used in Program01.

Finally, if your code includes text or print forms, a use declaration is necessary to access the form group that includes those forms. A use declaration is also desirable when accessing data tables or libraries. For additional information, see *Use declaration*.

Additional visibility rules

Most development efforts have parts that are shared across more than one package. These rules are in effect:

- Any part in the file can reference parts from other packages, so long as the accessed parts have these characteristics:
 - Are top-level parts
 - Are not declared as *private*
 - Are either in the same project as the referencing part or are in a project listed in the EGL build path of the referencing project

You can provide access in these ways:

- You can qualify the part name with the package name, in which case no import statement is needed in your source file. If a package name is *my.package*, for example, and a part name is *myPart*, you can reference the part as follows:


```
my.package.myPart
```
- You can use import statements, which provide the following benefits:
 - Import statements make it possible for you to avoid qualifying the names of the imported parts, unless you must use a package name to avoid an ambiguous reference.
 - Import statements provide a way to document what packages are used in the source code.

Part-name resolution

To resolve a part reference, EGL conducts a search that includes one to many steps. The following statements apply *at each step*:

- The search ends successfully if a uniquely named part is found
- The search ends with an error if two same-named parts are found.

These situations are possible:

- The part reference is qualified with a package name; in this case, the search always includes only one step
- The part reference is not qualified with a package name and is not a function invocation
- The part reference is not qualified with a package name and is a function invocation

Note: If one of your functions is visible to the program or page handler and has a name that is identical to the name of an EGL system function, your function is referenced rather than the system function.

Part-name resolution when the package name is specified: As noted earlier, you can specify the name of a package when referencing a part, as in the example `my.package.myPart`. The current project is considered, as are any projects listed in the EGL build path.

If the reference is from a part that is inside the same package, the following statements apply:

- The package name is valid but unnecessary
- The part name is resolved even if the part is declared as *private*

Part-name resolution (other than function invocation) when the package name is not specified: If a part references a part other than a function and does not specify a package name, the steps in the search order are as follows:

1. Search the parts nested in the same container as the one in which the referencing part is nested.

2. Search the parts that were explicitly imported in the file where the referencing part resides. The current project is considered, as are any projects listed in the EGL build path.

Each import statement in this case explicitly references a particular part in a particular package. The part named in such an *explicit-type import statement* acts as an override of the same-named part in the current package.

If you have identically named packages in two different projects, a given explicit-type import statement uses the EGL build path to do a first-found search, stopping when the required part is found. (The part must be unique to a package in a given project.) The presence of a same-named package in two different projects is not an error, but creates a confusing situation and is not recommended.

An error occurs if you have two explicit-type import statements that name the same part.

3. Search the top-level parts that are in the same package as the referencing part. The current project is considered, as are any projects listed in the EGL build path. Finding two parts of the same name causes an error.
4. Search other imported parts. The current project is considered, as are any projects listed in the EGL build path.

Each import statement in this case references all parts in a given package and is called a *wild-card import statement*.

If you have identically named packages in two different projects, a given wild-card import statement uses the EGL build path to do a first-found search, stopping when the required part is found. (The part must be unique to a package in a given project.)

If more than one wild-card import statement retrieves the same-named part, an error occurs.

Function invocation when the package name is not specified: If a part invokes a function and does not specify a package name, the steps in the search order are as follows:

1. Search the functions nested in the same container as the one in which the invoker is nested.
2. Search the functions residing in the libraries specified in the container's use declarations.
3. Continue the search only with functions that are being included in the container at generation time. (To include functions other than those nested in the same container or residing in a library, set the container property **includeReferencedFunctions** to *yes*.)

The search of the included functions occurs as follows:

- a. Search the parts that were explicitly imported in the file where the container resides. The current project is considered, as are any projects listed in the EGL build path.

Each import statement in this case explicitly references a particular part in a particular package. The part named in such an *explicit-type import statement* acts as an override of the same-named part in the current package.

If you have identically named packages in two different projects, a given explicit-type import statement uses the EGL build path to do a first-found search, stopping when the required function is found. (The function must be unique to a package in a given project.) The presence of a same-named package in two different projects is not an error, but creates a confusing situation and is not recommended.

An error occurs if you have two explicit-type import statements that name the same part.

- b. Search the top-level functions in the same package as the container. The current project is considered, as are any projects listed in the EGL build path. An error occurs if the search finds two parts of the same name.
- c. Search other imported parts. The current project is considered, as are any projects listed in the EGL build path.

Each import statement in this case references all parts in a given package and is called a *wild-card import statement*.

If you have identically named packages in two different projects, a given wild-card import statement uses the EGL build path to do a first-found search, stopping when the required part is found. (The part must be unique to a package in a given project.)

If more than one wild-card import statement retrieves the same-named part, an error occurs.

Program invocation

When a program is invoked on a **call** or **transfer** statement, the argument list of the invoker must match the parameter list of the invoked program. A mismatch of argument and parameter causes an error.

Related concepts

“EGL projects, packages, and files” on page 7

“Import” on page 26

“Introduction to EGL” on page 1

“Parts” on page 11

“References to variables and constants” on page 34

Related reference

“EGL build path and eglpath” on page 409

“EGL editor” on page 103

“EGL source format” on page 292

“Use declaration” on page 631

Structure

A *structure* is composed of a series of *structure items*, each of which describes an elemental memory location or a collection of memory locations. A structure establishes the format of a form, table, or record part and can establish the format of at least some of the data used in a pageHandler part.

An example of a structure is as follows:

```
10 workAddress;  
20 streetAddress1 CHAR(20);  
30 Line1 CHAR(10);  
30 Line2 CHAR(10);  
20 streetAddress2 CHAR(20);  
30 Line1 CHAR(10);  
30 Line2 CHAR(10);  
20 city CHAR(20);
```

You can declare all the structure items directly in the structure declaration, as in the preceding example. Alternatively, you can indicate that all or a subset of the structure is equivalent to the structure that is in another record part; for details, see Typedef.

Access to a structure item is based on a variable name, then a series of structure item names with a dot syntax. If you declare that the record *myRecord* includes the structure shown in the previous example, each of the following identifiers refers to an area of memory:

```
myRecord.workAddress
myRecord.workAddress.streetAddress1
myRecord.workAddress.streetAddress1.Line1
```

An *elementary structure item* has no subordinate structure items and describes an area of memory in either of these ways:

- By a specification of length and primitive type, as in the previous example; or
- By pointing to the declaration of a *dataItem* part, as described in *Typedef*.

As shown earlier, a structure item can have subordinate structure items. Consider the next example:

```
10 topMost;
20 next01 HEX(4);
20 next02 HEX(4);
```

When you define a superior structure item (like *topMost*), you have several options:

- If you do not assign a length or primitive type, the superior structure item is of type CHAR, and EGL calculates the length. The primitive type of *topMost* is CHAR, for example, and the length is 4.
- If you assign a primitive type but do not assign a length, EGL calculates the length based on characteristics of the subordinate structure items
- If you assign both a length and primitive type, the length must reflect the space provided for the subordinate structure items; otherwise, an error occurs

Note: The primitive type of a structure item determines the number of bytes in each unit of length; for details, see *Primitive types*.

Each elementary structure item has a series of properties, whether by default or as specified in the structure item. (The structure item may refer to a *dataItem* part that itself has properties.) For details, see *Overview of EGL properties and overrides*.

Related concepts

“DataItem part” on page 284

“Parts” on page 11

“Overview of EGL properties and overrides” on page 40

“Record parts” on page 490

“References to variables and constants” on page 34

“Typedef”

Related reference

“Data initialization” on page 277

“EGL source format” on page 292

“Items” on page 43

“Primitive types” on page 27

“SQL item properties” on page 57

Typedef

A type definition (typedef) is a part that is used as a model of format. You use the typedef mechanism for these reasons:

- To identify the characteristics of a variable

- To reuse part declarations
- To enforce formatting conventions
- To clarify the meaning of data

Often, typedefs identify an abstract grouping. You can declare a record part named *address*, for example, and divide the information into *streetAddress1*, *streetAddress2*, and *city*. If a personnel record includes the structure items *workAddress* and *homeAddress*, each of those structure items can point to the format of the record part named *address*. This use of typedef ensures that the address formats are the same.

Within the set of rules described in this page, you may point to the format of a part either when you declare another part or when you declare a variable.

When you declare a part, you are not required to use a part as a typedef, but you may want to do so, as in the examples that are shown later. Also, you are not required to use a typedef when you declare a variable that has the characteristics of a data item; instead, you can specify all characteristics of the variable, without reference to a part.

A typedef is *always* in effect when you declare a variable that is more complex than a data item. For instance, if you declare a variable named **myRecord** and point to the format of a part named **myRecordPart**, EGL models the declared variable on that part. If you point instead to the format of a part named **myRecordPart02**, the variable is called **myRecord** but has all characteristics of the part named **myRecordPart02**.

The table and sections that follow give details on typedefs in different contexts.

Entry that points to a typedef	Type of part to which the typedef can refer
function parameter or other function variable	a record part or dataItem part
program parameter	data item part, form part, record part
program variable (non-parameter)	data item part, record part
structure item	data item part, record part

Data item part as a typedef

You can use a data item part as a typedef in the following situations:

- When declaring a variable or parameter
- When declaring a structure item, which is a subunit of a record part, form part, or dataTable part

These rules apply:

- If a structure item is a parent to other structure items that are listed in the same declaration, the structure item can point only to the format of a data item part, as in this example:

```
DataItem myPart CHAR(20) end

Record myRecordPart type basicRecord
  10 mySI myPart; // myPart acts as a typedef
  20 a CHAR(10);
  20 b CHAR(10);
end
```

The previous record part is equivalent to this declaration:

```
Record myRecordPart type basicRecord
  10 mySI CHAR(20);
  20 a CHAR(10);
  20 b CHAR(10);
end
```

- You cannot use a data item part as a typedef and also specify the length or primitive type of the entity that is pointing to the typedef, as in this example:

```
DataItem myPart HEX(20) end

// NOT valid because mySI has a primitive type
// and points to the format of a part (to myPart, in this case)
Record myRecordPart type basicRecord
  10 mySI CHAR(20) myPart;
end
```

- A variable declaration that does not refer to a record part either points to the format of a data item part or has primitive characteristics. (A program parameter can refer to a form part, too.) A data item part, however, cannot point to the format of another data item part or to any other part.
- An SQL record part can use only the following types of parts as typedefs:
 - Another SQL record part
 - A dataItem part

Record part as a typedef

You can use a record part as a typedef in the following situations:

- When declaring a structure item
- When declaring a variable (including a parameter), in which case the variable reflects the typedef in these ways:
 - Format
 - Record type (for example, indexedRecord or serialRecord)
 - Property values (for example, value of the **file** property)

When you declare a structure item that points to the format of another part, you specify whether the typedef adds a level of hierarchy, as illustrated later.

These rules apply:

- A record part can be a typedef when you use a structure item to facilitate reuse--

```
Record address type basicRecord
  10 streetAddress1 CHAR(30);
  10 streetAddress2 CHAR(30);
  10 city CHAR(20);
end

Record record1 type serialRecord
{
  fileName = myFile
}
  10 person CHAR(30);
  10 homeAddress address;
end
```

The second record part is equivalent to this declaration--

```
Record record1 type serialRecord
{ fileName = myFile }
  10 person CHAR(30);
  10 homeAddress;
```



```

20 streetAddress1 CHAR(30);
20 streetAddress2 CHAR(30);
20 city CHAR(20);
end

```

If a structure item uses the previous syntax to point to the format of a structure part, EGL adds a hierarchical level to the structure part that includes the structure item. For this reason, the internal structure in the previous example has a structure-item hierarchy, with *person* at a different level from *streetAddress1*.

- In some cases, you prefer a flat arrangement in the structure; and an SQL record that is an I/O object for relational-database access *must* have such an arrangement--
 - In the previous example, if you substitute the word **embed** for a record part's structure item name (in this case, *homeAddress*) and follow that word with the name of the record part that acts as a typedef (in this case, *address*), the part declarations look like this:

```

Record address type basicRecord
  10 streetAddress1 CHAR(30);
  10 streetAddress2 CHAR(30);
  10 city CHAR(20);
end

```

```

Record record1 type serialRecord
{
  fileName = myFile
}
  10 person CHAR(30);
  10 embed address;
end

```

The internal structure of the record part is now flat:

```

Record record1 type serialRecord
{
  fileName = myFile
}
  10 person CHAR(30);
  10 streetAddress1 CHAR(30);
  10 streetAddress2 CHAR(30);
  10 city CHAR(20);
end

```

The only reason to use the word **embed** in place of a structure item name is to avoid adding a level of hierarchy. A structure item identified by the word **embed** has these restrictions:

- Can point to the format of a record part, but not to a data item part
- Cannot specify an array or include a primitive-type specification
- Next, consider the case in which a record part is a typedef when you are declaring identical structures in two records--

```

Record common type serialRecord
{
  fileName = mySerialFile
}
  10 a BIN(10);
  10 b CHAR(10);
end

```

```

Record recordA type indexedRecord
{
  fileName = myFile,
  keyItem = a
}
  embed common; // accepts the structure of common,
                // not the properties

```

```

end

Record recordB type relativeRecord
{
  fileName = myOtherFile,
  keyItem = a
}
  embed common;
end

```

The last two record parts are equivalent to these declarations--

```

Record recordA type indexedRecord
{
  fileName = myFile,
  keyItem = a
}
  10 a BIN(10);
  10 b CHAR(10);
end

Record recordB type relativeRecord
{
  fileName = myOtherFile,
  keyItem = a
}
  10 a BIN(10);
  10 b CHAR(10);
end

```

- You can use a record part multiple times as a typedef when declaring a series of structure items. This reuse makes sense, for example, if you are declaring a personnel record part that includes a home address and a work address. A basic record could provide the same format in two locations in the structure:

```

Record address type basicRecord
  10 streetAddress1 CHAR(30);
  10 streetAddress2 CHAR(30);
  10 city CHAR(20);
end

Record record1 type serialRecord
{
  fileName = myFile
}
  10 person CHAR(30);
  10 homeAddress address;
  10 workAddress address;
end

```

The record part is equivalent to this declaration:

```

Record record1 type serialRecord
{
  fileName=myFile
}
  10 person CHAR(30);
  10 homeAddress;
    20 streetAddress1 CHAR(30);
    20 streetAddress2 CHAR(30);
    20 city CHAR(20);
  10 workAddress;
    20 streetAddress1 CHAR(30);
    20 streetAddress2 CHAR(30);
    20 city CHAR(20);
end

```

- You cannot use a record part as a typedef and also specify the length or primitive type of the entity that is pointing to the typedef, as in this example:

```

Record myTypedef type basicRecord
  10 next01 HEX(20);
  10 next02 HEX(20);
end

// not valid because myFirst has a
// primitive type and points to the format of a part
Record myStruct02 type serialRecord
{
  fileName = myFile
}
  10 myFirst HEX(40) myTypedef;
end

```

Consider the following case, however:

```

Record myTypedef type basicRecord
  10 next01 HEX(20);
  10 next02 HEX(20);
end

Record myStruct02 type basicRecord
  10 myFirst myTypedef;
end

```

The second structure is equivalent to this declaration:

```

Record myStruct02 type basicRecord
  10 myFirst;
    20 next01 HEX(20);
    20 next02 HEX(20);
end

```

The primitive type of any structure item that has subordinate structure items is CHAR by default, and the length of that structure item is the number of bytes represented by the subordinate structure items, regardless of the primitive types of those structure items. For other details, see *Structure*.

- The following restrictions are in effect in relation to SQL records:
 - If an SQL record part uses another SQL record part as a typedef, each item provided by the typedef includes a four-byte prefix. If a non-SQL record uses an SQL record part as a typedef, however, no prefix is included. For background information, see *SQL record internals*.
 - An SQL record part can use only the following types of parts as typedefs:
 - Another SQL record part
 - A dataItem part
- Finally, neither a structure nor a structure item can *be* a typedef

Form as a typedef

You can use a form part as a typedef only when declaring a program parameter.

Related concepts

“DataItem part” on page 284
 “Form part” on page 366
 “Introduction to EGL” on page 1
 “Record parts” on page 490
 “Structure” on page 19

Related tasks

“Creating an EGL program part” on page 85

Related reference

"EGL statements" on page 70

"SQL record internals" on page 199

Import

An import statement identifies a set of parts that are in a specified package (for EGL source files) or in a specified set of files (for EGL build files). The file that holds an import statement can reference the imported parts as if they were in the same package as the file.

Background

If a public part resides in a package other than the current one but is not identified in an import statement, your code needs to qualify the part name (for example, `myPart`) with the package name (for example, `my.pkg`), as in this example:

```
my.pkg.myPart
```

If the part is identified in an import statement, however, your code can drop the package name. In this case, the unqualified part name (like `myPart`) is sufficient.

For a description of the circumstances in which import statements are used to resolve a part name, see *References to Parts*.

Format of the import statement

The syntax is as follows for the import statement in an EGL file:

```
import packageName.partSelection;
```

packageName

Identifies the name of a package in which to search. The name must be complete.

partSelection

Is a part name or an asterisk (*). The asterisk indicates that all parts in the package are selected.

An import statement in a build file identifies other build files whose parts can be referenced by parts in the importing file. The import statements follow the `<EGL>` tag in the build file, and each statement has the following syntax:

```
<import file=filePath.eglbuild>
```

filePath

Identifies the path and name of the file to import. If you specify a path, the following statements apply:

- The file path is in any of the source directories in the same project or in any other project that is in the EGL path
- Each qualifier is separated from the next by a virgule (/)

You may specify an asterisk (*) as the file name or as the last character of the file name. If the asterisk is used, EGL imports all the `.eglbuild` files with these characteristics:

- Are in the specified file path.
- Have names that begin with the characters that precede the asterisk. (If the asterisk has no preceding characters, all build files in the directory path are selected.)

The file extension .eglbld is optional.

Related concepts

“EGL projects, packages, and files” on page 7

“Introduction to EGL” on page 1

“Parts” on page 11

“References to parts” on page 16

Related tasks

“Editing an EGL build path” on page 91

Primitive types

Each EGL primitive type characterizes an area of memory. These primitive types are of two kinds: character or numeric--

- The character types are as follows:
 - *CHAR* refers to single-byte characters.
 - *DBCHAR* refers to double-byte characters. *dbchar* replaces *DBCS*, which was a primitive type in EGL V5.
 - *MBCHAR* refers to multibyte characters, which are a combination of single-byte and double-byte characters. *mbchar* replaces *MIX*, which was a primitive type in EGL V5.
 - *UNICODE* refers to double-byte characters that conform to the UTF-16 encoding standards developed by the Unicode Consortium.
 - *HEX* refers to hexadecimal characters.
- The numeric types are as follows:
 - *BIGINT* refers to an 8-byte area that stores an integer of as many as 18 digits. This type is equivalent to type *BIN*, length 8, no decimal places.
 - *BIN* refers to a binary number.
 - *DECIMAL* refers to packed decimal characters whose sign is represented by a hexadecimal C (for a positive number) or a hexadecimal D (for a negative number) in the right half of the rightmost byte. *DECIMAL* replaces *PACK*, which was a primitive type in EGL V5.
 - *INT* refers to an 4-byte area that stores an integer of as many as 9 digits. This type is equivalent to type *BIN*, length 4, no decimal places.
 - *NUM* refers to numeric characters whose sign is represented by a sign-specific hexadecimal value in the left half of the rightmost byte. For ASCII, that value is 3 (for a positive number) and 7 (for negative); for EBCDIC, that value is F (for a positive number) and D (for negative).
 - *NUMC* refers to numeric characters whose sign is represented by a sign-specific hexadecimal value in the left half of the rightmost byte. For ASCII, that value is 3 (for a positive number) and 7 (for negative); for EBCDIC, that value is F (for a positive number) and C (for negative).
 - *PACF* refers to packed decimal characters whose sign is represented by a hexadecimal F (for a positive number) or a hexadecimal D (for a negative number) in the right half of the rightmost byte.
 - *SMALLINT* refers to an 2-byte area that stores an integer of as many as 4 digits. This type is equivalent to type *BIN*, length 2, no decimal places.

The internal representation of an item of any of the numeric types is the same as an integer representation, even when you specify a decimal point. The representation of 12.34 is the same as that of 1234, for example.

At declaration time, you specify the primitive type that characterizes each of these values:

- The value returned by a function
- The value in an item, which is an area of memory that is referenced by name and contains a single value

Other entities also have a primitive type:

- A system item has a primitive type (usually NUM) that is specific to the item
- A character literal is of one of these types:
 - CHAR if the literal includes only single-byte characters
 - DBCHAR if the literal includes only double-byte characters from the double-byte character set
 - MBCHAR if the literal includes a combination of single-byte and double-byte characters
- Character literals of type UNICODE are not supported.

Each primitive type is described on a separate page; and additional details are available on the pages that cover assignments, logical expressions, function invocations, and the call statement.

The sections that follow cover these subjects:

- Primitive types at declaration time
- Relative efficiency of different numeric types

Primitive types at declaration time

Consider the following declarations:

```
DataItem
  myItem CHAR(4)
end
Record mySerialRecordPart type serialRecord
{
  fileName="myFile"
}
10 name CHAR(20);
10 address;
  20 street01 CHAR(20);
  20 street02 CHAR(20);
end
```

As shown, you must specify a primitive type when you declare these entities:

- A data item
- A structure item that is not substructured

You may specify the primitive type of a substructured structure item like *address*. If you fail to specify the primitive type of such a structure item but you reference the structure item in your code, the product makes these assumptions:

- The primitive type is assumed to be CHAR, even if the subordinate structure items are of a different type
- The length is assumed to be the number of bytes in the subordinate structure items

Relative efficiency of different numeric types

EGL supports the types DECIMAL, NUM, NUMC, and PACF so you can work more easily with files and databases that are used by legacy applications. It is recommended that you use items of type BIN in new development or that you use an equivalent integer type (BIGINT, INT, or SMALLINT); calculations are most efficient with items of those types. You get the greatest efficiency by using items of type BIN, length 2, and no decimal places (the equivalent of type SMALLINT).

In calculations, assignments, and comparisons, items that are of type NUM and have no decimal places are more efficient than items that are of type NUM and have decimal places.

For code generated in Java, calculations with items of types DECIMAL, NUM, NUMC, and PACF are equally efficient. For code generated in COBOL, however, these distinctions apply:

- Calculations with items of types NUM are more efficient than calculations with items of type NUMC
- Calculations with items of types DECIMAL are more efficient than calculations with items of type PACF

Related concepts

"DataItem part" on page 284

"Record parts" on page 490

"References to variables and constants" on page 34

"Structure" on page 19

Related reference

"Assignments" on page 296

"BIN and the integer types" on page 32

"call" on page 299

"CHAR"

"DBCHAR" on page 30

"DECIMAL" on page 32

"Exception handling" on page 75

"Function invocations" on page 316

"HEX" on page 30

"Items" on page 43

"Logical expressions" on page 358

"MBCHAR" on page 31

"NUM" on page 33

"NUMC" on page 33

"Numeric expressions" on page 364

"Operators and precedence" on page 39

"PACF" on page 34

"SQL item properties" on page 57

"String expressions" on page 365

"UNICODE" on page 31

Character types

CHAR

An item of type CHAR is interpreted as a series of single-byte characters. The length reflects both the number of characters and the number of bytes and ranges from 1 to 32767.

Workstation platforms like Windows® 2000 use the ASCII character set; mainframe platforms like z/OS® UNIX® System Services use the EBCDIC character set. Differences in collating sequence generally cause greater-than and less-than comparisons to have different results in the two types of environments.

Related reference

"Primitive types" on page 27

DBCHAR

An item of type DBCHAR is interpreted as a series of double-byte characters. The length reflects the number of characters and ranges from 1 to 16383. To determine the number of bytes, double the length value.

Workstation platforms like Windows 2000 use the ASCII character set; mainframe platforms like z/OS UNIX System Services use the EBCDIC character set. Differences in collating sequence generally cause greater-than and less-than comparisons to have different results in the two types of environments.

DBCS data is ideographic, as is necessary to display Chinese, Japanese, or Korean, for example. Display of such data requires a terminal device with DBCS capability.

Related reference

"Primitive types" on page 27

HEX

An item of type HEX is interpreted as a series of hexadecimal digits (0-9, a-f, and A-F), which are treated as characters. The length reflects the number of digits and ranges from 1 to 65534. To determine the number of bytes, divide by 2.

For an item of length 4, the internal bit representations of example values are as follows:

```
// hexadecimal value 04 D2
00000100 11010010
```

```
// hexadecimal value FB 2E
11111011 00101110
```

The primary use of an item of type HEX is to access a file or database field whose data type does not match another EGL primitive type.

You can assign a hexadecimal value by using a literal that is of type CHAR and that includes only characters in the range of hexadecimal digits, as in these examples:

```
myHex01 = "ab02";
```

```
myHex02 = "123E";
```

You can include a hexadecimal item as an operand in a logical expression, as in these examples:

```
if (myHex01 = "aBCd")
  myFunction01();
else
  if (myHex > myHex02)
    myFunction02();
  end
end
```

You cannot include a hexadecimal item in an arithmetic expression.

Related reference

“Primitive types” on page 27

MBCHAR

An item of type MBCHAR is interpreted as a combination of single-byte and double-byte characters. The length reflects the number of single-byte characters that the item can contain and also reflects the number of bytes. The length ranges from 1 to 32767.

Workstation platforms like Windows 2000 use the ASCII character set; mainframe platforms like z/OS UNIX System Services use the EBCDIC character set. Differences in collating sequence generally cause greater-than and less-than comparisons to have different results in the two types of environments.

On a mainframe environment, you must include space for shift-out and shift-in characters if double-byte characters are possible in the item:

- A single-byte shift-out character (hex value 0E) indicates the beginning of a series of double-byte characters
- A single-byte shift-in character (hex value 0F) indicates the end of that series

The shift-out and shift-in characters are deleted during an EBCDIC-to-ASCII data conversion and are inserted during an ASCII-to-EBCDIC data conversion. If a variable-length record is being converted, and if the current record end (as indicated by the record length) is within a structure item that is of type MBCHAR, the record length is adjusted to reflect the insertion or deletion of the shift-out and shift-in characters.

Double-byte character data is ideographic, as is necessary to display Chinese, Japanese, or Korean, for example. Display of such data requires a terminal device with double-byte character set capability.

Related reference

“Primitive types” on page 27

UNICODE

The primitive type UNICODE gives you a way to process and store text that may be in any of several human languages; however, the text must have been provided from outside your code. Literals of type UNICODE are not supported.

The following statements are true of an item of type UNICODE:

- The length reflects the number of characters and ranges from 1 to 16383. The number of bytes reserved for such an item is twice the value you specify for length.
- The item can be assigned or compared only to another item of type UNICODE.
- All comparisons compare the bit values in accordance with the order of characters in the UTF-16 encoding standard.
- When necessary, EGL pads the item with Unicode blanks.
- The system string functions treat the item as a string of individual bytes, which include the added Unicode blanks, if any. Any lengths you specify in those functions must be in terms of bytes rather than in terms of characters.
- You can store the value of the item in a file or database. If your code interacts with DB2[®] UDB, you must ensure that the code page for GRAPHIC data is UNICODE and that the column that stores the data item value is of SQL data type GRAPHIC or VARGRAPHIC.

For details on Unicode, see the web site of the Unicode Consortium (www.unicode.org).

Related reference

“Primitive types” on page 27

Numeric types

BIN and the integer types

An item of type BIN is interpreted as a binary value. The length can be 4, 9, or 18 and reflects the number of positive digits in decimal format, including any decimal places. The value -12.34, for example, fits in an item of length 4. A 4-digit number requires 2 bytes; a 9-digit number requires 4 bytes; and an 18-digit number requires 8 bytes.

For an item of length 4, the internal bit representations of example values are as follows:

```
// for decimal 1234, the hexadecimal value is 04 D2:  
00000100 11010010  
  
// for decimal -1234, the value is the 2's complement (FB 2E):  
11111011 00101110
```

It is recommended that you use items of type BIN instead of other numeric types whenever possible; for example, for arithmetic operands or results, for array subscripts, and for key items in relative records.

The following types are equivalent to type BIN:

- BIGINT is length 18, no decimal places
- INT is length 9, no decimal places
- SMALLINT is length 4, no decimal places

Related reference

“Primitive types” on page 27

DECIMAL

An item of type DECIMAL is a numeric value in which each half-byte is a hexadecimal character, and the sign is represented by a hexadecimal C (for a positive number) or a hexadecimal D (for a negative number) in the right half of the rightmost byte. The length reflects the number of digits and ranges from 1 to 32. To determine the number of bytes, add 2 to the length value, divide the sum by 2, and truncate any fraction in the result.

For an item of length 4, the internal hexadecimal representations of example values are as follows:

```
// for decimal 123  
00 12 3C  
  
// for decimal -123  
00 12 3D  
  
// for decimal 1234  
01 23 4C  
  
// for decimal -1234  
01 23 4D
```

A negative value that is read from a file or database into a field of type DECIMAL may have a hexadecimal B in place of a D; EGL accepts the value but converts the B to D.

The format of a DB2 UDB column of type DECIMAL is equivalent to the format of a DECIMAL-type host variable.

Related reference

“Primitive types” on page 27

NUM

An item of type NUM is a numeric value in which each byte is a digit in character format, and the sign is represented by a sign-specific hexadecimal value in the left half of the rightmost byte. The length reflects both the number of digits and the number of bytes. The length ranges from 1 to 32.

For an item of length 4, the internal hexadecimal representations of example values are as follows if the item is on a host environment which uses EBCDIC:

```
// for decimal 1234
F1 F2 F3 F4

// for decimal -1234
F1 F2 F3 D4
```

The internal hexadecimal representations of example values are as follows if the item is on a workstation environment like Windows 2000, which uses ASCII:

```
// for decimal 1234
31 32 33 34

// for decimal -1234
31 32 33 74
```

Related reference

“Primitive types” on page 27

NUMC

An item of type NUMC is a numeric value in which each byte is a digit in character format, and the sign is represented by a sign-specific hexadecimal value in the left half of the rightmost byte. The length reflects both the number of digits and the number of bytes and ranges from 1 to 32.

For an item of length 4, the internal hexadecimal representations of example values are as follows if the item is on a host environment which uses EBCDIC:

```
// for decimal 1234
F1 F2 F3 C4

// for decimal -1234
F1 F2 F3 D4
```

The internal hexadecimal representations of example values are as follows if the item is on a workstation environment like Windows 2000, which uses ASCII:

```
// for decimal 1234
31 32 33 34

// for decimal -1234
31 32 33 74
```

Related reference

“Primitive types” on page 27

PACF

An item of type PACF is a numeric value in which each half-byte is a hexadecimal character, and the sign is represented by a hexadecimal F (for a positive number) or a hexadecimal D (for a negative number) in the right half of the rightmost byte. The length reflects the number of digits and ranges from 1 to 32. To determine the number of bytes, add 2 to the length value, divide the sum by 2, and truncate any fraction in the result.

For an item of length 4, the internal hexadecimal representations of example values are as follows:

```
// for decimal 123
00 12 3F

// for decimal -123
00 12 3D

// for decimal 1234
01 23 4F

// for decimal -1234
01 23 4D
```

A negative value that is read from a file or database into a field of type PACF may have a hexadecimal B in place of D; EGL accepts the value but converts the B to D.

Related reference

“Primitive types” on page 27

References to variables and constants

Each data declaration (whether variable or constant) is a named area of memory that is either global to a program or local to a function--

- You can base a variable on one of several primitive types, as in this example:

```
myItem CHAR(10);
```

For details, see *Primitive types*.

- You can base a variable on a dataItem part or record part, as in this example:

```
myRecord myRecordPart;
```

For details, see *Typedef*.

- You can compose a program parameter as described previously for variables; alternatively, you can base it on a form part, as in this example:

```
myForm myFormPart;
```

The form must be accessible through a form group that is identified in one of the program's use declarations. A form accessed as a parameter cannot be displayed to the user, but can provide access to field values that are passed from another program.

- Some parts are generated independent of the program and act as variables that are global to the program:
 - If a part is of type dataTable, library, or pageHandler and is visible to the program, the part is treated as a program-global variable. For referencing convenience, you can include a dataTable or library part in one of the program's use declarations.

- If a part is of type `formGroup`, is visible to the program, and is included in one of the program's use declarations, the form parts that are referenced in that `formGroup` are also treated as program-global variables. All the fields in those forms are also available as program-global variables.
- Several rules concern the entities that are within parts that are available to the program:
 - If a `dataTable` part is a program-global variable, the fields in that part are also global to the program.
 - If a library part is a program-global variable, the following entities in that part are also global to the program:
 - Public functions
 - Public variables
 - Public constants

For details on visibility, see *References to parts*. See also *Use declaration*.

- You can compose a function parameter on the basis of a primitive type, a `dataItem` part, or a record part; alternatively, you can base a function parameter on a *loose primitive type*, as in this example:

```
myNumber Number;
```

In addition, a function parameter can be a dynamic array of records or data items. For details, see *Function part in EGL source format*.

- Finally, you declare a constant by associating an identifier with a number or quoted string. That initial value cannot be changed at run time, as in these examples:

```
myNumber 10.10;
myString "Great software!";
```

Program variables and constants

When you declare a variable or constant in a program part, the memory is available throughout the program; in particular, the variable or constant is available in any function that the program invokes directly or indirectly.

Program variables are of two kinds:

- *Program parameters* are available only in a called program and refer to global memory that initially contains data received from another program. At run time, if the called program changes data that was passed by way of a variable, the storage area available to the caller is changed, too.

For a description of how data is provided when a *main* program is invoked, see *Program part*.

- *Other program variables* allocate program-global memory, as do *program constants*.

Each of the MQSeries options records that you specify in an MQ record property must be a program-global variable. For details, see *Options records for MQ records*.

A record identified in an I/O operation is called an *I/O object*.

Function variables

When you declare a variable in a function part, the variable is available only in the function. If `Function01` declares `Var01` and invokes `Function02`, for example, `Function01` can reference the variable but `Function02` cannot.

Function variables are of two kinds:

- *Function parameters* refer to memory that initially contains data received from an invoking function. At run time, if the invoked function changes data that was passed by way of a variable, the storage area available to the invoker is changed, too.
- *Other function variables* and *function constants* allocate local memory and refer to that memory.

For details, see *Function invocations*.

Arrays and items

In some contexts the logic part references a record as a whole, but in other contexts the logic part references an *array* or *item*:

- An array is a named, allocated area of memory that contains a series of values that have identical formats. For details on how to reference an array element, see *Arrays*.
- An item is a named area of memory that contains a single value. Such an area may be described by any of the following entities:
 - A data item
 - A structure item, so long as the structure item does not represent an array
 - An array element, so long as the array element does not represent an array
 An item (if not a data item) may have subordinate structure items, and any of those subordinates may represent an array.

Rules for referencing variables

The following rules are in effect:

- If you are referencing an item that is described by a structure item, you can avoid ambiguity about the area of memory being referenced. Consider the following part declaration, for example:

```
Record myRecordPart type serialRecord
{
  fileName = myFile
}
10 myTop;
20 myNext;
30 myAlmost;
40 myItem CHAR(10);
40 myItem02 CHAR(10);
end
```

Assume that a function or program uses the record part *myRecordPart* as a *typedef* when declaring a variable named *myRecordVar*.

If you want to refer to an item that has the characteristics of a specific structure item, you can include the following symbols in order:

- The name of the variable that points to the format of the part; in this case, *myRecordVar*
- A period (.)
- A list of structure items superior to the item, with a period separating each structure item from the next; for example, *myTop.myNext.myAlmost*
- The item name preceded by a period; for example, *.myItem*

A valid reference to *myItem* in *myRecordVar* is as follows:

```
myRecordVar.myTop.myNext.myAlmost.myItem
```

That reference is considered to be *fully qualified*.

- If you want to refer to an item that has the characteristics of a structure item whose name is unique within a structure, you can specify the variable name, followed by a period, followed by the item name. Valid references for the earlier example include this symbol:

```
myRecordVar.myItem02
```

That references are considered to be *partially qualified*.

You cannot partially qualify an item name in any other way. You cannot include only some of the structure item names that are between the variable name and the item name, for example, nor can you eliminate the variable name while keeping any of the names of structure items that are superior to the item. The following references are *not* valid for the earlier example:

```
// NOT valid
myRecordVar.myNext.myItem
myRecordVar.myAlmost.myItem
myNext.myItem
myAlmost.myItem
```

- You can refer to an item without preceding the name with any qualifiers. Valid references for the earlier example include these symbols:

```
myItem02
myOtherItem
```

Those references are considered to be *unqualified*.

- You must qualify any reference to a structure item to the extent necessary to avoid ambiguity. It is recommended that you use fully qualified names whenever possible. You must use an unqualified reference, however, when you are specifying a record *property* that refers to a key item, variable length item, or number of occurs item. (In an EGL statement, you can reference those special structure items as you would reference any structure item.)
- Unqualified and partially qualified references can be valid only if you set the property **allowUnqualifiedItemReferences** to *yes*. That property is a characteristic of programs, page handlers, and libraries, and the default value is *no*.
- The name of a structure item can be an asterisk (*) if the related memory area is a *filler*, which is an area whose name is of no importance. You cannot include an asterisk in a reference. Consider this example:

```
record myRecordPart type serialRecord
{
  fileName = myFile
}
10 person;
20 *;
30 streetAddress1 CHAR(30);
30 streetAddress2 CHAR(30);
30 nation CHAR(20);
end
```

If you use that part as a typeDef when declaring the variable *myRecordVar*, you can refer to *myRecordVar.nation* or *nation*, but the following references are not valid:

```
// NOT valid
myRecordVar.*.streetAddress1
myRecordVar.*.streetAddress2
myRecordVar.*.nation
```

- When EGL tries to resolve a reference, names of local variables are searched first, then names of structure items in the records used for I/O in the same function, then names of other local structure items, then names that are program-global.

Consider the case in which a function declares both a data item called *nation* and a variable that points to the following basic record:

```
record myRecordPart
  10 myTop;
  20 myNext;
  30 nation CHAR(20);
end
```

An unqualified reference to *nation* refers to the data item, not to the structure item.

- A name search shows no preference for program-global data items over program-global structure items. Consider the case in which a program declares both a data item called *nation* and a variable that points to the format of the following basic record:

```
record myRecordPart
  10 myTop;
  20 myNext;
  30 nation CHAR(20);
end
```

An unqualified reference to *nation* fails because *nation* could refer either to the data item or to the structure item. You can reference the structure item, but only by qualifying the reference.

- The fully qualified name of a program variable must be unique among program variables; and the name of a function variable must be unique among function variables.

For additional rules, see *Arrays* and *Use declaration*.

This

Consider the following cases:

- The name of a function variable is identical to the name of a program-global data area in the program or page handler; or
- A library is referenced in your program's use declaration and has a data area whose name is the same as a program-global data area in the program or page handler.

To refer to the program-level data area in either of those cases, qualify the name of interest with the keyword *this*, as in the following example:

```
this.myProgramVariable
```

Related concepts

"Function part" on page 380
"References to parts" on page 16
"Parts" on page 11
"Program part" on page 477
"Structure" on page 19
"Typedef" on page 20

Related reference

"Arrays" on page 64
"Function invocations" on page 316
"Function part in EGL source format" on page 381
"Options records for MQ records" on page 169
"Primitive types" on page 27
"Use declaration" on page 631

Operators and precedence

The next table lists the EGL operators in order of decreasing precedence. Except for the unary plus (+), minus (-), and not (!), each operator works with two operands.

Operators (separated by commas)	Type of operator	Meaning
+, -	Numeric, unary	Unary plus (+) or minus (-) is a sign before an operand or parenthesized expression, not an operator between two expressions.
, /, %	Numeric	Multiplication () and integer division (/) are of equal precedence. The division of integers retains a fractional value, if any; for example, 7/5 yields 1.4. % is the <i>remainder</i> operator, which resolves to the modulus when the first of two operands or numeric expressions is divided by the second; for example, 7%5 yields 2. If you need to ensure that arithmetic results are consistent between EGL output in Java and COBOL, avoid using the remainder operator.
+, -	Numeric	Addition (+) and subtraction (-) are of equal precedence.
=	Numeric or string	= is the <i>assignment</i> operator, which copies a numeric or character value from an expression or operand into an operand.
!	Logical, unary	! is the <i>not</i> operator, which resolves to a Boolean value (true or false) opposite to the value of a logical expression that immediately follows. That subsequent expression must be in parentheses.
=, !=, <, >, <=, >=, in, is, not	Logical for comparison	The logical operators used for comparison are of equal precedence and are described in the page on logical expressions. Each operator resolves to true or false.
&&	Logical	&& is the <i>and</i> operator, which means "both must be true." The operator resolves to true if the logical expression that precedes the operator is true and if the logical expression that follows the operator is true; otherwise, && resolves to false.
	Logical	is the <i>or</i> operator, which means "one or the other or both." The operator resolves to true if the logical expression that precedes the operator is true or if the logical expression that follows the operator is true or if both are true; otherwise resolves to false.

You may override the usual precedence (also called the *order of operations*) by using parentheses to separate one expression from another. Operations that have the same precedence in an expression are evaluated in left-to-right order.

The equal sign (=) is used in two ways:

- As a operator for logical comparison, as described in the page on logical expressions
- As an assignment operator, which copies numeric or string values to an operand on the left side of an assignment statement; for example:

```
myItem01 = "Fascinating";
```

Related reference

“Items” on page 43

“in” on page 416

“Logical expressions” on page 358

“Numeric expressions” on page 364

“Primitive types” on page 27

“String expressions” on page 365

Overview of EGL properties and overrides

With few exceptions, each logic, data, and user-interface part in EGL has a set of properties, whether assigned or by default. The feature is used widely in EGL:

- Properties are in effect when a record part or data item part is used as a typedef (a model of format). The typedef might be for use by another part, by a variable, or by a structure item that is used in a part or in a variable. The part, structure item, or variable that uses another part as a typedef can override the properties that are in the typedef.
- Properties are also in effect for parts of type dataTable, form, formGroup, library, program, and pageHandler.

No overrides occur for form-level properties. If a property like **validationBypassKeys** is specified in a form, for example, the value in the form is in effect at run time. If a form-level property is not specified in the form, however, the situation is as follows:

- EGL run time uses the value in the program’s use declaration
- If no value is specified in the program’s use declaration, EGL run time uses the value (if any) in the form group

Item properties

Each dataItem part and elementary structure item has a diverse set of properties, whether assigned or by default. You also may assign properties to a structure item that has subordinate structure items, but those properties are ignored unless the description of the property says otherwise.

Item properties may be meaningful only in a particular context; for example, when an item interacts with a database table. The properties may have no effect in other contexts, however, as when an item presents data to the screen. The benefit of that lack of effect is that you can re-use the same part declarations. With some exceptions, EGL allows you to maintain values for every item property, in every item.

Most often, you re-use part declarations to create consistency in presentation or validation. You might want several on-screen fields to have the same color, for example, or to restrict input to the same range of numeric values.

With the exception of a series of item properties that are likely to be set only in a form, each item property is in one of these categories:

Field presentation

Specifies characteristics that are meaningful when a field is displayed in either of these ways:

- As a field in a printable output, when the destination is a printer or a print file; or
- As a field in an on-screen output, when the destination is a 3270 screen or a command window, but not a Web browser.

The field-presentation properties have no effect on the data that is returned to the program; they are solely for output.

Properties include **color** and **highlight**.

For details, see *Field-presentation properties*.

Formatting

Specifies characteristics that are meaningful when data is presented as a printable output, is presented to an on-screen display, or is returned from an on-screen display. The display can be a 3270 screen, a command window, or a Web browser.

The formatting properties affect data at output or at both input and output. Properties include **align**, which determines how data is placed in a field.

Any formatting property can be specified on a data item.

For details, see *Formatting properties*.

page item

Specifies characteristics that are meaningful when an item is declared in a PageHandler part.

Properties include **value**, which indicates the initial content of a field when you are working in Page Designer.

For details, see *Page item properties*.

SQL item

Specifies characteristics that are meaningful when an item is used in an SQL record part. Properties include **key**, which indicates whether the specified field corresponds to a key column in the related database table.

For details, see *SQL item properties*.

Validation

Specifies characteristics that restrict what is accepted from the user who is working at a 3270 screen or command window.

Properties include **range**, which specifies a high and low value for numeric input.

Any formatting property can be specified on a data item.

For details, see *Validation properties*.

Other form-specific properties are described in *Form part in EGL source format*.

Declaration properties

Each item, record, or static array has the property **initialized**, which indicates whether the memory area is assigned an initial value. The property itself has the value *yes* or *no*.

Each dynamic array has the property **maxSize**, which indicates the maximum number of elements in the array. For details, see *Arrays*.

Syntax for specifying properties and their overrides

Properties and their overrides are treated consistently in the EGL syntax:

- Each set begins with a left curly bracket (**{**), includes either an entry or a list of entries that are separated by commas, and ends with a right curly bracket (**}**)

- A property whose valid values are *yes* and *no* (**isReadOnly**, for example) can be expressed in any of three ways:

```
isReadOnly
isReadOnly = yes
isReadOnly = no
```

If you include the name of such a property without explicitly stating a value, the value is *yes*. If you include neither the property name nor the value, however, the default varies by property. The default for **isReadOnly**, for example, is *no*.

The next example shows an SQL record part declaration, which includes two record properties:

```
Record myRecordPart type SQLRecord
{ tableNames = myTable,
  keyItems = myKey }
myKey CHAR(10);
myOtherKey CHAR(10);
myContent CHAR(60);
end
```

The next example shows a variable declaration that uses the previous part as a typedef and that overrides one of the two record properties:

```
myRecord myRecordPart {keyItems = myOtherKey};
```

The next example shows the same variable declaration, which also overrides a property of a record item:

```
myRecord myRecordPart {keyItems = myOtherKey,
  myOtherKey {isReadOnly}}
```

Syntax for overriding item properties in a record

In a record or record-part declaration, you can override the properties of record items. The syntax is in the following format, with each level of information nested within the previous level:

```
item name1 { item name2 { properties for the nested item } }
```

Properties may be nested as deeply as the structure of the record is nested, and the names of intermediate levels must be supplied either by qualification or by additional nesting.

The following example shows a basic record part for use in a page handler:

```
Record myRecordPart01 type basicRecord
10 item1;
15 item2;
20 item3 char(4) {displayUse = button, action = "label1"};
10 item4;
15 item5;
20 item6 char(4) {displayUse = button, action = "label2"};
end
```

Alternatively, that record part can be defined as follows:

```
Record myRecordPart01 type basicRecord

// nesting properties by multiple levels of nesting
{ item1 { item2 { item3
  {displayUse = button, action = "label1"}}},

// nesting properties by qualifying the item names
  item4.item5.item6 {displayUse = button, action = "label2"}
}
```

```

10 item1;
15 item2;
20 item3 char(4);
10 item4;
15 item5;
20 item6 char(4);
end

```

Finally, the next example shows a variable declaration where the properties for item3 and item6 are overridden:

```

myRecord myRecordPart01

// nesting properties by multiple levels of nesting
{ item1 { item2 { item3 { action = "label3" } } } },

// nesting properties by qualifying the item names
item4.item5.item6 { action = "label6" } }

```

Related concepts

“PageHandler part” on page 469

“Page Designer support for EGL” on page 135

“Typedef” on page 20

Related reference

“Arrays” on page 64

“Field-presentation properties” on page 45

“Formatting properties” on page 47

“Form part in EGL source format” on page 370

“Data initialization” on page 277

“Page item properties” on page 53

“SQL item properties” on page 57

“Validation properties” on page 59

Items

An item is a named area of memory that contains a single value. Such an area may be described by any of the following entities:

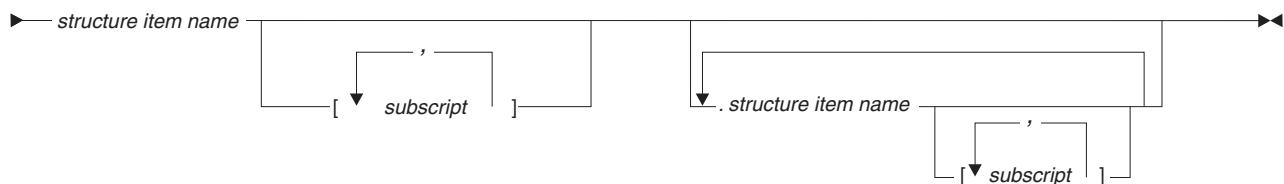
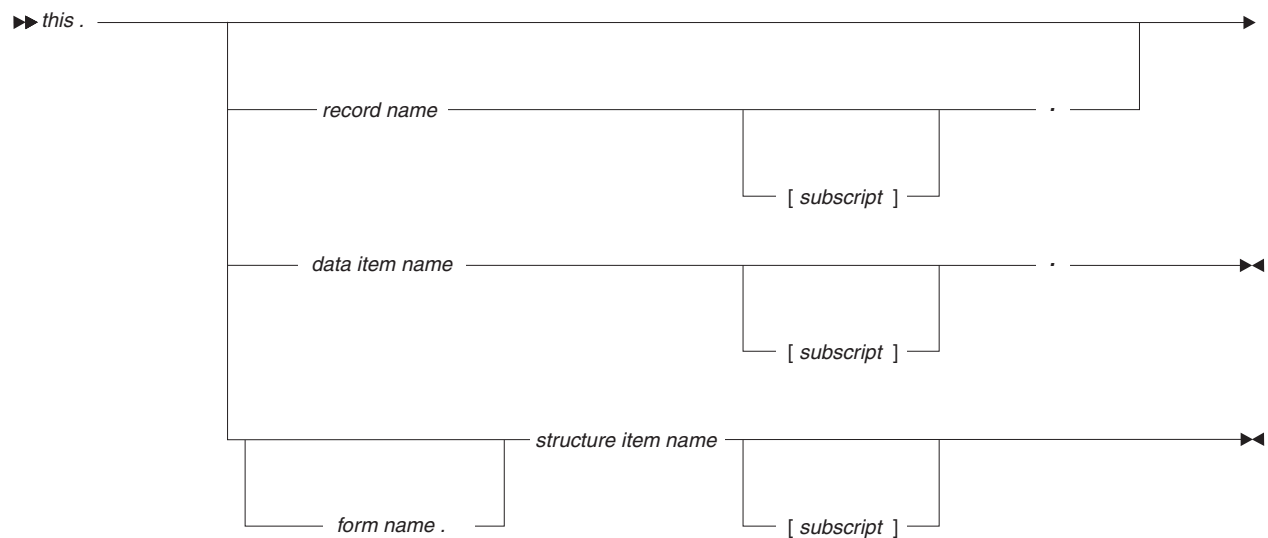
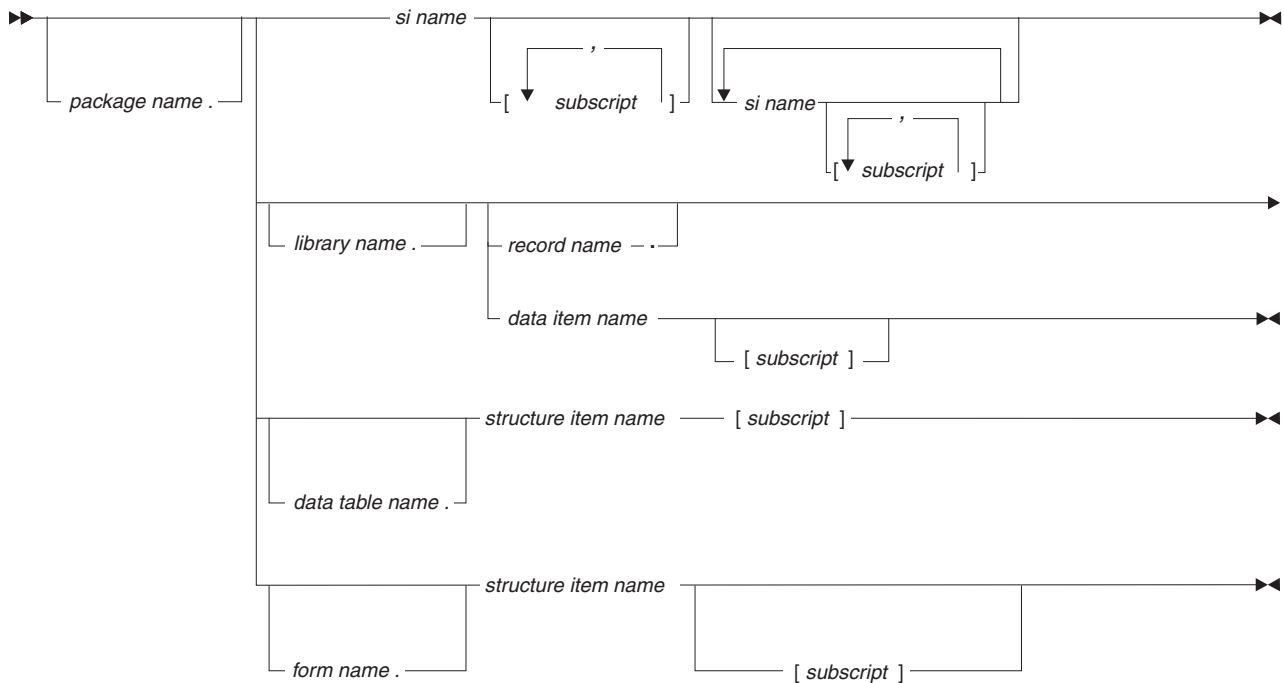
- A data item
- A structure item, so long as the structure item does not represent an array
- An element of an array, so long as the array element does not represent an array

An item may have subordinate structure items, and any of those subordinates may represent an array.

An item is distinct from any of these entities:

- A system variable
- A literal
- An array

The following two syntax diagrams tell how to refer to an item.



record name, data item name, form name

The name of a data item, record, or form.

structure item name

The name of a structure item in a record, form, or dataTable.

subscript

An integer, or an item (or system variable) that resolves to an integer. The value of a subscript is an index that refers to a specific element in an array.

An item used as a subscript of an array cannot itself be an array element. In each of the following examples, `myItemB[1]` is both a subscript and an array element; as a result, the following syntax is *not* valid:

```
/* the next syntax is not valid */
myItemA[myItemB[1]]

// This next syntax is not valid; but the
// reason (if you are working in VisualAge Generator
// compatibility mode) is that myItemB is myItemB[1],
// the first element of a one-dimensional array
myItemA[myItemB]
```

For further details on referencing an item, see *References to variables and constants*. See also Arrays.

Related concepts

“Compatibility with VisualAge Generator” on page 276
“DataItem part” on page 284
“Record parts” on page 490
“References to variables and constants” on page 34
“Structure” on page 19

Related tasks

“Syntax diagram” on page 506

Related reference

“Arrays” on page 64
“Data initialization” on page 277
“Items” on page 43
“Logical expressions” on page 358
“Numeric expressions” on page 364
“Operators and precedence” on page 39
“Primitive types” on page 27
“String expressions” on page 365

Field-presentation properties

The EGL field-presentation properties specify characteristics that are meaningful when a field is displayed in an on-screen output, when the destination is a command window, but not a Web browser.

The properties are as follows:

- “Color” on page 46
- “Highlight” on page 46
- “Intensity” on page 46
- “Outline” on page 47

In addition, the following properties are meaningful when the field is displayed in a printable output, when the destination is a printer or a print file:

- **Highlight** property (but only for *underline* and *noHighlight*, and only for Java output)
- **Outline** property, which is appropriate only for devices that support double-byte characters

The field-presentation properties have no effect on data that is returned to the program from a text form; they are solely for output.

Color: The **color** property specifies the color of a field in a text form. You can select any of these:

- black
- blue
- `defaultColor` (the default)
- green
- pink
- red
- turquoise
- white
- yellow

If you assign the value *defaultColor*, other conditions determine the displayed color, as shown in the next table.

Are all fields on the form assigned the value <i>defaultColor</i> ?	Value of <i>protect</i>	Value of <i>intensity</i>	Displayed color for a field assigned the value <i>defaultColor</i>
yes	<i>yes</i> or <i>skip</i>	not <i>bold</i>	blue
yes	<i>yes</i> or <i>skip</i>	<i>bold</i>	white
yes	<i>no</i>	not <i>bold</i>	green
yes	<i>no</i>	<i>bold</i>	red
no	any value	not <i>bold</i>	green
no	any value	<i>bold</i>	white

Highlight: The **highlight** property specifies the special effect (if any) with which to display the field. Valid values are as follows:

blink

Causes the text to blink repeatedly. This value is available only for COBOL output (but not in the EGL Debugger), and support varies by emulator.

noHighLight (the default)

Indicates that no special effect is to occur; specifically, no blink, reverse, or underline. This value and *underline* are the only ones available for Java output.

reverse

Reverses the text and background colors, so that (for example) if the display has a dark background with light letters, the background becomes light and the text becomes dark. This value is available only for COBOL output.

underline

Places an underline at the bottom of the field. This value and *noHighLight* are the only ones available for Java output.

Intensity: The **intensity** property specifies the strength of the displayed font. Valid values are as follows:

bold

Causes the text to appear in boldface.

invisible

Removes any indication that the field is on the form.

normalIntensity (the default)

Sets the field to be visible, without boldface.

Outline: The **outline** property lets you draw lines at the edges of fields on any device that supports double-byte characters. Valid values are as follows:

box

Draw lines to create a box around the field content

noOutline (the default)

Draw no lines

In addition, you can specify any or all of the components of a box. In this case, place parentheses around one or more values, with each value separated from the next by a comma, as in this example:

```
outline = (left, over, right, under)
```

The partial values are as follows:

left

Draw a vertical line at the left edge of the field

over

Draw a horizontal line at the top edge of the field

right

Draw a vertical line at the right edge of the field

under

Draw a horizontal line at the bottom edge of the field

The content of each form field is preceded by an attribute byte. Be aware that you cannot place an attribute byte in the last column of a form and expect an outline value to appear in the next column, which is beyond the form's edge. (The field does not wrap to the next line.) Similarly, you cannot place an attribute byte in the first column of a form and expect the outline value to appear in that column; the outline value can appear only in the next column.

Related concepts

"Overview of EGL properties and overrides" on page 40

Formatting properties

The formatting properties specify characteristics that are meaningful when data is presented on a form or a Web browser. The properties affect data at output or at both input and output.

The properties are as follows:

- "align" on page 48
- "boolean" on page 48
- "currency" on page 48
- "fillCharacter" on page 49
- "dateFormat" on page 49
- "numericSeparator" on page 51
- "sign" on page 51

- “timeFormat” on page 52
- “upperCase” on page 53
- “zeroFormat” on page 53

align: The **align** property specifies the position of data in a variable field when the length of the data is smaller than the length of the field. This property is useful on text and print forms.

On output, character and numeric data are affected by this property. On input, character data is affected by this property, but numeric data is always right-justified.

Valid values are as follows:

left

Place the data at the left of the field, as is the default for character data. Initial spaces are stripped and placed at the end of the field.

none

Do not justify the data. This setting is valid only for character data.

right

Place the data at the right of the field, as is the default for numeric data. Trailing spaces are stripped and placed at the beginning of the field. This setting is required for numeric data that has a decimal position or sign.

boolean: The **boolean** property indicates that the field represents a Boolean value. The property restricts the valid field values and is useful in text and print forms and in page handlers, for input or output.

On a Web page associated with an EGL page handler, a boolean item is represented by a checkbox. On a form, the situation is as follows:

- The value of a numeric field is 0 (for false) or 1 (for true).
- The value of a character field is represented by a word or subset of a word that is national-language dependent. In English, for example, a boolean field of three or more characters has the value *yes* (for true) or *no* (for false), and a one-character boolean field value has the truncated value *y* or *n*.

In Java programs, the specific character values for *yes* and *no* are determined by the locale.

currency: The **currency** property indicates whether to include a currency symbol before the value in the text field, with the exact position of the symbol determined by the **zero format** property. This property is useful in text and print forms and in page handlers. Values are as follows:

No (the default)

Do not use a currency symbol.

Yes

Use the default currency symbol. In Java code, the default is determined by the machine locale. In COBOL code, the default is determined by the national language option.

Currency

Use the specified currency symbol.

fillCharacter: The **fillCharacter** property indicates what character fills unused positions in a text or print form or in page-handler data. In addition, the property changes the effect of *set field full*, as described in *set*. The effect of this property is only at output.

The default is a space for numbers and a 0 for hex items. The default for character types depends on the medium:

- In text or print forms, the default is null (binary zero)
- For page-handler data, the default is blank for data of type CHAR or MBCHAR

In page handlers, the value of **fillCharacter** must be a space (as is the default) for items of type DBCHAR or UNICODE.

dateFormat: The **dateFormat** property identifies the format for dates that are displayed on a form or maintained in a page handler. This property affects both input and output, but is not used in the following cases:

- The item has decimal places, a currency symbol, a numeric separator, or a sign; or
- The data type for the data item is DBCHAR, MBCHAR, or HEX; or
- The item is not long enough to contain the mask. (For other details, see *Length considerations for times*.)

Valid values are as follows:

"pattern"

The value of *pattern* consists of a set of characters, as described in the next table.

Characters	Meaning
<	If this character is in the first position of the mask, leading zeros are dropped from the item at output.
dd	Day of month as a 2-digit number.
ddd	Day of the year as a 3-digit number.
mm	Month as a 2-digit number.
yy	Year as a 2-digit number.
yyyy	Year as a 4-digit number.
Any other single-byte character	A separator character. Separator characters can be omitted when a date is entered by the user or maintained in the page handler.

iso

The pattern "yyyy-mm-dd", which is the date format specified by the International Standards Organization (ISO).

usa

The pattern "mm/dd/yyyy", which is the IBM USA standard date format.

eur

The pattern "dd.mm.yyyy", which is the IBM European standard date format.

jis The pattern "yyyy-mm-dd", which is the Japanese Industrial Standard date format.

locale

If specified for a page-handler item, this choice uses the date format given in the run-time Java locale. If specified for a form item, this choice is equivalent to selecting `systemGregorian`.

systemGregorian

An 8- or 10-character pattern that includes DD (for numeric day), MM (for numeric month), and YY or YYYY (for numeric year), with characters other than D, M, Y, or digits used as separators.

For COBOL programs, the system administrator for EGL run-time services sets the format at installation.

For Java programs, the format is in this Java run-time property:

```
vgj.datemask.gregorian.long.NLS
```

NLS

The NLS (national language support) code that is specified in the Java run-time property **vgj.nls.code**. The code is one of those listed in `targetNLS`. Uppercase English (code ENP) is not supported.

For additional details on **vgj.nls.code**, see *Java run-time properties (details)*.

systemJulian

A 6- or 8-character pattern that includes DDD (for numeric day) and YY or YYYY (for numeric year), with characters other than D, Y, or digits used as separators.

For COBOL programs, the system administrator for EGL run-time services sets the format at installation.

For Java programs, the format is in this Java run-time property:

```
vgj.datemask.julian.long.NLS
```

NLS

The NLS (national language support) code that is specified in the Java run-time property **vgj.nls.code**. The code is one of those listed in `targetNLS`. Uppercase English (code ENP) is not supported.

For additional details on **vgj.nls.code**, see *Java run-time properties (details)*.

Internal date formats: When the user enters valid data, the date is converted from the format specified for the field to an internal format, as is used for subsequent validation.

The internal format for a character date is the same as the system default format and includes separator characters.

For a numeric date, the internal formats are as follows:

- For a Gregorian short date, 00YYMMDD
- For a Gregorian long date, 00YYYYMMDD
- For a Julian short date, 0YYDDD
- For a Julian long date, 0YYYYDDD

Length considerations for dates: In a form, the field length must match the length of the date mask you specify. In a page-handler item, the rules are as follows:

- The item length must be sufficient for the date mask you specify but can be longer

- In the case of a numeric item, the separator characters are excluded from the length calculation.

Examples are in the next table.

Format type	Example	Length of form field	Minimum length of page-handler item (character type)	Valid length of page-handler item (numeric type)
Short Gregorian	YY/MM/DD	8	8	6
Long Gregorian	YYYY/MM/DD	10	10	8
Short Julian	DDD-YY	6	6	5
Long Julian	DDD-YYYY	8	8	7

I/O considerations for dates: Data entered into a variable field is checked to ensure that the date was entered in the format specified. The user does not need to enter the leading zeros for days and months, but can specify (for example) 8/5/1996 instead of 08/05/1996. The user who omits the separator characters, however, must enter all leading zeros.

A date stored in internal format is not recognized as a date, but simply as data. If an 8-character date field is moved to a character item of length 10, for example, EGL pads the destination field with blanks. A 2-digit year is not converted to a 4-digit year.

When a date field is presented on a form, however, the date is converted from its internal format, as appropriate.

numericSeparator: The **numericSeparator** property indicates whether to place a character in a number that has an integer component of more than 3 digits. If the numeric separator is a comma, for example, one thousand is shown as *1,000* and one million is shown as *1,000,000*. Values are as follows:

no (the default)

Do not use a numeric separator.

yes

Use a numeric separator.

In Java code, the default is determined by the machine locale. In COBOL code, the default is determined by the national language option.

sign: The **sign** property indicates the position in which a positive (+) or negative (-) sign is displayed when a number is placed in the field, whether from user input or from the program. Values are as follows:

none

A sign is not displayed.

leading

The default: a sign is displayed to the left of the first digit in the number, with the exact position of the sign determined by the *zero format* property (described later).

trailing

A sign is displayed immediately to the right of the last digit in the number.

timeFormat: The **timeFormat** property identifies the format for times that are displayed on a form or maintained in a page handler. This property affects both input and output, but is not used in the following cases:

- The item has decimal places, a currency symbol, a numeric separator, or a sign; or
- The data type for the data item is DBCHAR, MBCHAR, or HEX; or
- The item is not long enough to contain the mask. (For other details, see *Length considerations for times*.)

Valid values are as follows:

"pattern"

The value of *pattern* consists of a set of characters, as described in the next table.

Characters	Meaning
<	If this character is in the first position of the mask, leading zeros are dropped from the item at output.
hh	Hours of the day as a 2-digit number that ranges from 1 to 12 (if the pattern includes the characters <i>am</i>) or from 0 to 23.
mm	Minutes as a 2-digit number.
ss	Seconds as a 2-digit number.
am	AM or PM; if specified, the hours range from 1 to 12.
Any other single-byte character	A separator character. Separator characters can be omitted when a time is entered by the user or maintained in the page handler.

iso

The pattern "hh.mm.ss", which is the time format specified by the International Standards Organization (ISO).

usa

The pattern "hh:mm AM", which is the IBM USA standard time format.

eur

The pattern "hh.mm.ss", which is the IBM European standard time format.

jis The pattern "hh:mm:ss", which is the Japanese Industrial Standard time format.

Length considerations for times: In a form, the field length must match the length of the time mask you specify. In a page-handler item, the rules are as follows:

- The item length must be sufficient for the time mask you specify but can be longer
- In the case of a numeric item, the separator characters are excluded from the length calculation.

I/O considerations for times: Data entered into a variable field is checked to ensure that the time was entered in the format specified. The user does not need to enter the leading zeros for hours, minutes, and second, but can specify (for example) 8:15 instead of 08:15. The user who omits the separator characters, however, must enter all leading zeros.

A date stored in internal format is not recognized as a time, but simply as data. If a 6-character time field is moved to a character item of length 10, for example, EGL pads the destination field with blanks. When the 6-character value is presented on a form, however, the time is converted from its internal format, as appropriate.

upperCase: The **upperCase** property indicates whether to set alphabetic characters to upper case in the user's single-byte character input.

This property is useful in forms and in page handlers.

The values of **upperCase** are as follows:

No (the default)

Do not set the user's input to upper case.

Yes

Set the user's input to upper case.

zeroFormat: The **zeroFormat** property specifies how zero values are displayed in numeric fields. This property is affected by the **numeric separator**, **currency**, and **fillCharacter** properties. The values of **zeroFormat** are as follows:

Yes

A zero value is displayed as the number zero, which can be expressed with decimal points (0.00 is an example, if the item is defined with two decimal places) and with currency symbols and character separators (\$000,000.00 is an example, depending on the values of the **currency** and **numericSeparator** properties). The following rules apply when the value of the property **zeroFormat** is *yes*:

- If the *fill character* (the value of the **fillCharacter** property) is 0, the data is formatted with the character 0
- If the fill character is a null, the data is left-justified
- If the fill character is a blank, the data is right-justified
- If the fill character is an asterisk (*), asterisks are used as the left-side fillers instead of blanks

No

A zero value is displayed as a series of the fill character.

Related reference

"Java run-time properties (details)" on page 423

"set" on page 345

Page item properties

The page item properties specify characteristics that are meaningful when an item or array is declared in a PageHandler part.

The properties are as follows:

- "action" on page 54
- "displayName" on page 54
- "displayUse" on page 54
- "newWindow" on page 55
- "numElementsItem" on page 55
- "selectFromList" on page 55
- "selectType" on page 56

- “validationOrder” on page 56
- “value” on page 57

In the descriptions that follow, a *bind property* is a property whose value is in effect when a page designer places the item (or a record that includes the item) on the Web page. The page designer can override the value, which is a default.

action: The **action** property is a bind property, which means that the assigned value is used as a default when you are working in Page Designer. The property is meaningful only if the value of the item property **displayUse** is *button* or *hyperlink*.

The value of **action** refers to the code that is invoked when the user clicks the button or hypertext link.

The format for setting **action** is as follows:

action = *actionValue*

actionValue

One of the following:

- The name of an event-handling function in the page handler
- A label that maps to a Web resource (for example, to a JSP) and that corresponds to a from-outcome attribute of a navigation-rule entry in the JSF Application Configuration Resource file
- The name of a method in a Java bean, in which case these rules apply:
 - The format is the bean name followed by a period and a method name
 - The bean name must relate to one of the managed bean-name entries in the JSF Application Configuration Resource file

If you do not specify a value for **action**, the user’s click of the item has the following effect:

- If the value of the property **displayUse** is *button*, validation occurs, after which JSF re-displays the same Web page.
- If the value of the property **displayUse** is *hyperlink*, no validation occurs, but JSF re-displays the same Web page.

displayName: The **displayName** property is a bind property, which means that the assigned value is used as a default when you are working in Page Designer. The property specifies the label that is displayed next to the item.

The format is as follows:

displayName = *literal*

literal

A quoted string.

displayUse: The **displayUse** property is a bind property, which means that the assigned value is used as a default when you are working in Page Designer. The property associates the item with a user-interface control.

The format is as follows:

displayUse = *displayUse*

displayUse

One of these values:

- **button** means that the control has a button command tag.

- **secret** means that the data is not visible to the user; this value is appropriate for passwords.
- **hyperlink** means that the control has a hyperlink command tag if the action property is the name of an event-handling function; the control has a link tag if the action property is a label. No validation occurs and no input data is returned when the user clicks the link.
- **input** means that the control accepts user input; the control also may display a value provided by page handler.
- **table** means that the data is within a table tag.
- **output** means that the control is an output field that is visible.

newWindow: The property **newWindow** indicates whether to use a new browser window when the EGL run time presents a Web page in response to the activity identified in the **action** property.

The format is as follows:

newWindow = *yesOrNo*

yesOrNo

One of these values:

- **No** is the default and means that the current browser window is used to display the page.
- **Yes** means that a new browser window is used.

The property **newWindow** is meaningful only when the property **action** is specified; otherwise, the current browser window is used to display the next page.

numElementsItem: The property **numElementsItem** is specified on a static array variable or record item and identifies the name of another item in the page handler. At run time, the page handler code sets that item to indicate to the JSP how many array elements to display. This property is only used for output.

The property is meaningful for either of these:

- Static arrays of records or data items
- Structure items that have an occurs value greater than 1

The format is as follows:

numElementsItem = *itemName*

itemName

Name of the page-handler item that contains the number of array elements to display.

The property **numElementsItem** is not valid for dynamic arrays because each dynamic array includes an indicator of how many elements are in use. For details, see *Arrays*.

selectFromList: The property **selectFromList** is a bind property, which means that the assigned value is used as a default when you are working in Page Designer.

The property refers to an array or table column from which the user selects one or more values; and the property indicates that the array or item being declared will receive data in response to the user's selection from that list.

If the user is allowed to select multiple values, the property is specified for an array. Otherwise, the property is specified for an item.

The values received from the user must correspond to one of these types:

- The content of the array element or table column that the user selected; or
- An array or table index, which is an integer that identifies which element or column was selected. The index ranges from 1 to the number of elements available.

The property **selectType** indicates the type of value to receive, whether the content selected by the user or an index into an array or column.

The format of **selectFromList** is as follows:

```
selectFromList = listName
```

listName

One of these values, which must be qualified as necessary to fully resolve the reference:

- An array of any kind; or
- A table column such as myTable.myColumn.

For details on resolving names, see *References to parts*.

selectType: The property **selectType** is a bind property, which means that the assigned value is used as a default when you are working in Page Designer.

The property indicates the kind of value that is retrieved into the item being declared.

The format is as follows:

```
selectType = selectType
```

selectType

One of these values:

- **index** is the default and means that the array or item being declared will receive indexes in response to a user selection. In this case, the item must be of a numeric type.
- **value** means that the array or item being declared will receive the user's selection value in response to a user selection. In this case, the item can be of any type.

For background information, see the property **selectFromList**.

validationOrder: The value of the property **validationOrder** is an integer that indicates when the item's validator function runs in relation to any other item's validator function. The property is important if the validation of one item depends on the previous validation of another.

Validation occurs first for any items for which you specified a value for the property **validationOrder**, and the items with the lowest-numbered values are validated first. Validation then occurs for any items for which you did not specify a value for **validationOrder**, and the order of validation in this case is the order in which the items are defined in the page handler.

The format of the property is as follows:

```
validationOrder = integer
```

integer

A literal integer that indicates when the item is validated in relation to other items.

value: The property **value** is a bind property, which means that the assigned value is used as a default when you are working in Page Designer. The property specifies a character string that is displayed as the item content when a Web page is displayed, before the user changes the value.

The format is as follows:

value = *literal*

literal

A quoted string.

Related concepts

"PageHandler part" on page 469

"References to parts" on page 16

Related reference

"Arrays" on page 64

SQL item properties

The SQL item properties specify characteristics that are meaningful when an item is used in a record of type SQLRecord. You do not need to specify any of the SQL item properties, however, as default values are available.

The properties are as follows:

- "column = *columnName*"
- "isNullable = yes, isNullable = no" on page 58
- "isReadOnly = no, isReadOnly = yes" on page 58
- "sqlDataCode = *code*" on page 58
- "sqlVar = no, sqlVar = yes" on page 59

column = *columnName*: The property **column** refers to the name of the database table column that is associated with the item. The default is the name of the item. The column and related item affect the default SQL statements, as described in *SQL support*.

For *columnName*, substitute a quoted string; a variable of a character type; or a concatenation, as in this example:

```
column = "Column" + "01"
```

A special syntax applies if a column name is one of the following SQL reserved words:

- CALL
- COLUMNS
- FROM
- GROUP
- HAVING
- INSERT
- ORDER
- SELECT

- SET™
- UPDATE
- VALUES
- WHERE

As shown in the following example, each of those names must be embedded in a doubled pair of quote marks, and each of the internal quote marks must be preceded with an escape character (\):

```
column = "\"SELECT\""
```

(A similar situation applies if you use of those reserved words as a table name.)

isNullable = yes, isNullable = no: The property **isNullable** indicates whether the item can be set to null, as is appropriate if the table column associated with the item can be set to NULL. Valid values are *yes* (the default) and *no*.

For a given item in an SQL record, the following features are available only if **isNullable** is set to *yes*:

- Your program can accept a NULL value from the database into the item.
- Your program can use a **set** statement to null the item, as described in *set*. The effect is also to initialize the item, as described in *Data initialization*.
- Your program can use an **if** statement to test whether the item is set to null.
- Your COBOL program can use an **if** statement to test whether the data received from the database was truncated. This feature is available in your Java program regardless of the value of **isNullable**.

isReadOnly = no, isReadOnly = yes: The property **isReadOnly** indicates whether the item and related column should be omitted from the default SQL statements that write to the database or include a FOR UPDATE OF clause. The default value is *no*; but EGL treats the structure item as "read only" in these situations:

- The property **key** of the SQL record indicates that the column that is associated with the structure item is a key column; or
- The SQL record part is associated with more than one table; or
- The SQL column name is an expression.

sqlDataCode = code: The value of property **sqlDataCode** is a number that identifies the SQL data type that is associated with the record item. The data code is used by the database management system when you access that system at declaration time, validation time, or generated-program run time.

The property **sqlDataCode** is available only if you have set up your environment for VisualAge Generator compatibility. For details, see *Compatibility with VisualAge Generator*.

The default value depends on the primitive type and length of the record item, as shown in the next table. For other details, see *SQL data codes*.

EGL primitive type	Length	SQL data code
BIN	4	501
	9	497

EGL primitive type	Length	SQL data code
CHAR	<=254	453
	>254 and <=4000	449
	>4000	457
DBCHAR	<=127	469
	>127 and <=2000	465
	>2000	473
DECIMAL	any	485
HEX	any	481
UNICODE	<=127	469
	>127 and <=2000	465
	>2000	473

sqlVar = no, sqlVar = yes: The value of property **sqlVar** indicates whether trailing blanks and nulls in a character field are truncated before the EGL run time writes the data to an SQL database. This property has no effect on non-character data.

Specify *yes* if the corresponding SQL table column is a varchar or vargraphic SQL data type.

Related concepts

“Compatibility with VisualAge Generator” on page 276

“Record types and properties” on page 13

“SQL support” on page 171

“Structure” on page 19

“Typedef” on page 20

Related tasks

“Retrieving SQL table data” on page 191

Related reference

“add” on page 293

“close” on page 303

“Data initialization” on page 277

“delete” on page 307

“execute” on page 309

“get” on page 318

“get next” on page 324

“open” on page 335

“prepare” on page 339

“Primitive types” on page 27

“Record and file type cross-reference” on page 159

“replace” on page 341

“set” on page 345

“SQL data codes and EGL host variables” on page 200

“sysVar.terminalID” on page 629

“VAGCompatibility” on page 268

Validation properties

The validation properties restrict what is accepted when the user enters data in a text form.

The properties are as follows:

- “fill”
- “inputRequired”
- “inputRequiredMsgKey”
- “isDecimalDigit”
- “isHexDigit” on page 61
- “minimumInput” on page 61
- “minimumInputMsgKey” on page 61
- “needsSOSI” on page 61
- “range” on page 62
- “rangeMsgKey” on page 62
- “typeChkMsgKey” on page 62
- “validator” on page 62
- “validatorMsgKey” on page 63
- “validatorTable” on page 63
- “validatorTableMsgKey” on page 64

fill: The **fill** property indicates whether the user is required to enter data in each field position. Valid values are *no* (the default) and *yes*.

inputRequired: The **inputRequired** property indicates whether the user is required to place data in the field. Valid values are *no* (the default) and *yes*.

If the user does not place data in the field when the property value is *yes*, EGL run time displays a message, as described in relation to the field property **inputRequiredMsgKey**.

inputRequiredMsgKey: The property **inputRequiredMsgKey** identifies the message that is displayed if the field property **inputRequired** is set to *yes* and the user fails to place data into the field.

The *message table* (the data table that contains the message) is identified in the program property **msgTablePrefix**. For details on the data-table name, see *DataTable part in EGL source format*.

The value of **inputRequiredMsgKey** is a string or literal that matches an entry of the first column in the message table.

If a numeric key is used with a message table that expects a character key, the number is converted to a character string. If a string literal is used with a message table that expects a numeric key, the value in the string must be a signed or unsigned integer.

isDecimalDigit: The **isDecimalDigit** property determines whether to check that the input value includes only decimal digits, which are as follows:

0123456789

Valid values are *no* (the default) and *yes*.

This property applies only to character fields.

isHexDigit: The **isHexDigit** property determines whether to check that the input value includes only hexadecimal digits, which are as follows:

0123456789abcdefABCDEF

Valid values are *no* (the default) and *yes*.

This property applies only to character fields.

minimumInput: The **minimumInput** property indicates the minimum number of characters that the user is required to place in the field, if the user places any data in the field. The default is 0.

If the user places fewer than the minimum number of characters, EGL run time displays a message, as described in relation to the field property

minimumInputMsgKey.

minimumInputMsgKey: The property **minimumInputMsgKey** identifies the message that is displayed if the user acts as follows:

- Places data in the field; and
- Places fewer characters than the value specified in the property **minimumInputRequired**.

The *message table* (the table that contains the message) is identified in the program property **msgTablePrefix**. For details on the table name, see *DataTable part in EGL source format*.

The value of **minimumInputMsgKey** is a string or literal that matches an entry of the first column in the message table.

If a numeric key is used with a message table that expects a character key, the number is converted to a character string. If a string literal is used with a message table that expects a numeric key, the value in the string must be a signed or unsigned integer.

needsSOSI: The **needsSO/SI** property is used only for a *multibyte field* (a field of type MBCHAR) and indicates whether EGL does a special check when the user enters data of type MBCHAR on an ASCII device. Valid values are *yes* (the default) and *no*. The check determines whether the input can be converted properly to the host SO/SI format.

The property is useful because, during conversion, trailing blanks are deleted from the end of a multibyte string to allow for insertion of SO/SI delimiters around each substring of double-byte characters. For a proper conversion, the form field must have at least two blanks for each double-byte string in the multibyte value.

If **needsSO/SI** is set to *no*, the user can fill the input field, in which case the conversion truncates data without warning.

If **needsSO/SI** is set to *yes*, however, the result is as follows when the user enters multibyte data:

- The value is accepted as is because enough blanks are provided; or
- The value is truncated, and the user receives a warning message.

Set **needsSOSI** to *yes* if the user's ASCII input of multibyte data may be used on a z/OS or iSeries system.

range: The **range** property indicates the lowest and highest values that valid for user input. The property is only used for numeric fields. The format of the property is as follows:

`range = (lowestValue, highestValue)`

lowestValue

The lowest value that is valid.

highestValue

The highest value that is valid.

If the user's input is outside the specified range, EGL run time displays a message, as described in relation to the field property **rangeMsgKey**.

This property applies only to numeric fields.

rangeMsgKey: The property **rangeMsgKey** identifies the message that is displayed if the field property **range** is set and the user places out-of-range data into the field.

The *message table* (the table that contains the message) is identified in the program property **msgTablePrefix**. For details on the table name, see *DataTable part in EGL source format*.

The value of **rangeMsgKey** is a string or literal that matches an entry of the first column in the message table.

If a numeric key is used with a message table that expects a character key, the number is converted to a character string. If a string literal is used with a message table that expects a numeric key, the value in the string must be a signed or unsigned integer.

This property applies only to numeric fields.

typeChkMsgKey: The property **typeChkMsgKey** identifies the message that is displayed if the input data is not appropriate for the field type.

The *message table* (the table that contains the message) is identified in the program property **msgTablePrefix**. For details on the table name, see *DataTable part in EGL source format*.

The value of **typeChkMsgKey** is a string or literal that matches an entry of the first column in the message table.

If a numeric key is used with a message table that expects a character key, the number is converted to a character string. If a string literal is used with a message table that expects a numeric key, the value in the string must be a signed or unsigned integer.

validator: The **validator** property identifies a validator function, which is logic that runs after the EGL run time does the elementary validation checks, if any. Those checks are described in relation to the following properties:

- **inputRequired**
- **isHexDigit**
- **isNumericDigit**
- **minimumInput**

- needsSOSI
- range

The elementary checks precede use of the validator table (as described in relation to the **validatorTable** property), and all checks precede use of the **validator** property. This order of events is important because the validator function can do cross-field checking, and such checking often requires valid field values.

The value of **validator** is a validator function that you write. You code that function with no parameters and such that, if the function detects an error, it requests the re-display of the form by invoking `sysLib.validationFailed`.

If validation fails when you specify one of the two system functions, the displayed message is based on the value of the property **validatorMsgKey**. If validation fails when you specify a validator function of your own, however, the function does not use **validatorMsgKey**, but displays a message by invoking `sysLib.validationFailed`.

validatorMsgKey: The property **validatorMsgKey** identifies a message that is displayed in the following case:

- The **validator** property indicates use of `sysLib.verifyChkDigitMod10` or `sysLib.verifyChkDigitMod11`; and
- The specified function indicates that the user's input is in error.

The *message table* (the table that contains the message) is identified in the program property **msgTablePrefix**. For details on the table name, see *DataTable part in EGL source format*.

The value of **validatorMsgKey** is a string or literal that matches an entry of the first column in the message table.

If a numeric key is used with a message table that expects a character key, the number is converted to a character string. If a string literal is used with a message table that expects a numeric key, the value in the string must be a signed or unsigned integer.

validatorTable: The **validatorTable** property identifies a *validator table*, which is a *dataTable* part that acts as the basis of a comparison with user input. Use of a validator table occurs after the EGL run time does the elementary validation checks, if any. Those elementary checks are described in relation to the following properties:

- inputRequired
- isHexDigit
- isNumericDigit
- minimumInput
- needsSOSI
- range

All checks precede use of the **validator** property, which specifies a validation function that does cross-value validation.

You can specify a validator table that is of any of the following types, as described in *DataTable part in EGL source format*:

matchInvalidTable

Indicates that the user's input must be different from any value in the first column of the data table.

matchValidTable

Indicates that the user's input must match a value in the first column of the data table.

rangeChkTable

Indicates that the user's input must match a value that is between the values in the first and second column of at least one data-table row. (The range is inclusive; the user's input is also valid if it matches a value in the first or second column of any row.)

If validation fails, the displayed message is based on the value of the property **validatorTableMsgKey**.

validatorTableMsgKey: The property **validatorTableMsgKey** identifies the message that is displayed if the user provides data that does not correspond to the requirements of the *validator table*, which is the table specified in the property **validatorTable**.

The *message table* (the table that contains the message) is identified in the program property **msgTablePrefix**. For details on the message-table name, see *DataTable part in EGL source format*.

The value of **validatorTableMsgKey** is a string or literal that matches an entry of the first column in the message table.

If a numeric key is used with a message table that expects a character key, the number is converted to a character string. If a string literal is used with a message table that expects a numeric key, the value in the string must be a signed or unsigned integer.

Related concepts

"Text forms" on page 367

Related reference

"DataTable part in EGL source format" on page 287

"sysLib.validationFailed" on page 529

"sysLib.verifyChkDigitMod10" on page 619

"sysLib.verifyChkDigitMod11" on page 620

Arrays

EGL supports the following kinds of arrays:

- Static arrays of structure items; for details, see "Arrays of structure items" on page 65
- Static arrays of records or data items; for details, see "Static arrays of records and data items" on page 67
- Dynamic arrays of records or data items; for details see "Dynamic arrays of records and data items" on page 68

In all cases, the following rules apply:

- The maximum number of supported dimensions is seven.
- The program ends if an array subscript is out of range.

Arrays of structure items

You declare an array of structure items when you specify that a structure item in a record or form part has an occurs value greater than one, as in the following example:

```
Record myRecordPart
  10 mySI CHAR(1) [3];
end
```

If a record called *myRecord* is based on that part, the symbol *myRecord.mySi* refers to a one-dimensional array of three elements, each a character.

Usage of an array

You can reference an entire array of structure items (for example, *myRecord.mySi*) in these contexts:

- As the second operand used by an *in* operator. The operator tests whether a given value is contained in the array.
- As the parameter in the function **sysLib.size**. That function returns the occurs value of the structure item.

An array element that is not itself an array is an item like any other, and you can reference that item in various ways; for example, in an assignment statement or as an argument in a function invocation.

One-dimensional array

You can refer to an element of a one-dimensional array like *myRecord.mySi* by using the name of the array followed by a bracketed subscript. The subscript is either an integer or an item that resolves to an integer; for example, you can refer to the second element of the example array as *myStruct.mySi[2]*. The subscript can vary from 1 to the occurs value of the structure item, and a run-time error occurs if the subscript is outside of that range.

If you use the name of a structure-item array in a context that requires an item but do not specify a bracketed subscript, EGL assumes that you are referring to the first element of the array, but only if you are in VisualAge Generator compatibility mode. It is recommended that you identify each element explicitly. If you are not in VisualAge Generator compatibility mode, you are required to identify each element explicitly.

The next examples show how to refer to elements in a one-dimensional array. In those examples, *valueOne* resolves to 1 and *valueTwo* resolves to 2:

```
// these refer to the first of three elements:
myRecord.mySi[valueOne]
myRecord.mySi           // not recommended; and valid
                        // only if VisualAge Generator
                        // compatibility is in effect

// this refers to the second element:
myRecord.mySi[valueTwo]
```

A one-dimensional array may be substructured, as in this example:

```
record myRecord01Part
  10 name[3];
  20 firstOne CHAR(20);
  20 midOne CHAR(20);
  20 lastOne CHAR(20);
end
```

If a record called *myRecord01* is based on the previous part, the symbol *myRecord01.name* refers to a one-dimensional array of three elements, each of which has 60 characters, and the length of *myRecord01* is 180.

You may refer to each element in *myRecord01.name* without reference to the substructure; for example, *myRecord01.name[2]* refers to the second element. You also may refer to a substructure within an element. If uniqueness rules are satisfied, for example, you can reference the last 20 characters of the second element in any of the following ways:

```
myRecord01.name.lastOne[2]
myRecord01.lastOne[2]
lastOne[2]
```

The last two are valid only if the program, page-handler, or library property **allowUnqualifiedItemReferences** is set to *yes*.

For details on the different kinds of references, see *References to variables and constants*.

Multidimensional array

If a structure item with an occurs value greater than one is substructured and if a subordinate structure item also has an occurs value greater than one, the subordinate structure item declares an array with an additional dimension.

Let's consider another record part:

```
record myRecord02Part
  10 siTop[3];
  20 siNext CHAR(20)[2];
end
```

If a record called *myRecord02* is based on that part, each element of the one-dimensional array *myRecord02.siTop* is itself a one-dimensional array. For example, you can refer to the second of the three subordinate one-dimensional arrays as *myRecord02.siTop[2]*. The structure item *siNext* declares a two-dimensional array, and you can refer to an element of that array by a syntax like this:

```
// row 1, column 2
myRecord02.siTop.siNext[1,2]
```

To clarify what area of memory is being referenced, let's consider how data in a multidimensional array is stored. In the current example, *myRecord02* constitutes 120 bytes. The referenced area is divided into a one-dimensional array of three elements, each 40 bytes:

```
siTop[1]    siTop[2]    siTop[3]
```

Each element of the one-dimensional array is further subdivided into an array of two elements, each 20 bytes, in the same area of memory:

```
siNext[1,1] siNext[1,2] siNext[2,1] siNext[2,2] siNext[3,1] siNext[3,2]
```

A two-dimensional array is stored in row-major order. One implication is that if you initialize an array in a double while loop, you get better performance by processing the columns in one row before processing the columns in a second:

```
// i, j, myTop0ccurs, and myNext0ccurs are data items;
// myRecord02 is a record; and
// sysLib.size() returns the occurs value of a structure item.
i = 1;
j = 1;
myTop0ccurs = sysLib.size(myRecord02.siTop);
```

```

myNextOccurs = sysLib.size(myRecord02.siTop.siNext);
while (i <= myTopOccurs)
  while (j <= myNextOccurs)
    myRecord02.siTop.siNext[i,j] = "abc";
    j = j + 1;
  end
  i = i + 1;
end

```

You must specify a value for each dimension of a multidimensional array. The reference *myRecord02.siTop.siNext[1]*, for example, is not valid for a 2-dimensional array.

An example declaration of a 3-dimensional array is as follows:

```

record myRecord03Part
  10 siTop[3];
  20 siNext[2];
  30 siLast CHAR(20)[5];
end

```

If a record called *myRecord03* is based on that part and if uniqueness rules are satisfied, you can reference the last element in the array in any of the following ways:

```

myRecord03.siTop.siNext.siLast[3,2,5]
myRecord03.siLast[3,2,5]
siLast[3,2,5]

```

In general terms, you can reference an element of a multidimensional array by placing subscripts at the end of the reference:

1. Specify a list that begins with the name of the variable and continues with the names of increasingly subordinate structure items, with each name separated from the next by a period (for example, *myRecord03.siTop.siNext.siLast*).
2. Add a bracketed set of subscripts, with each subscript separated from the next by a comma. The first subscript refers to the first dimension, the second subscript refers to the second dimension, and so forth. Each subscript can vary from 1 to the occurs value of the related structure item, and a run-time error occurs if a subscript resolves to a number outside of that range.

For details on the different ways to reference a structure item, see *References to variables and constants*.

Static arrays of records and data items

You can declare a static array of records or data items by specifying a size value in brackets, as in the following examples:

```

myDataItem01 CHAR(30)[5];           // an array of 5 items
myDataItem02 myDataItemPart[6];     // an array of 6 items
myRecord myRecordPart[7];           // an array of 7 records

```

The size value must be an integer literal.

A static array may be used locally within a given function, may be a program-global variable, and may even be declared in a library and used by many programs, libraries, and/or page handlers. The array cannot be passed as an argument to another function (although individual elements can be passed). Also, the array cannot act as a parameter.

The number of dimensions in a static array of records is the highest number of array dimensions in the record structure, plus one dimension for the record array itself. Assume, for example, that your code can access the following part at development time:

```
record myRecordPart
  siTop[3];
  siNext CHAR(10)[2];
end
```

The following statement declares an array of three dimensions:

```
myRecord myRecordPart[5];
```

In the previous example, these references are valid:

```
myRecord.siTop.siNext[1,1,2]
siNext[1,1,2]
```

The in-memory arrangement of elements of the static array are as described for structure-item arrays.

The system variable `sysLib.size(arrayName)` contains the number of elements in the first dimension of a static array of records or data items. (Substitute the array name for *arrayName* and note that the name may be qualified by a package name, a library name, or both.)

Dynamic arrays of records and data items

You can declare an array of records or data items without specifying the number of elements, and in this case the number varies at run time as your code invokes system functions that append, insert, or delete elements or that remove the array. This *dynamic array* is useful for a variety of tasks but is not available in forms.

The syntax for declaring a dynamic array includes brackets with no size value, as in the following examples:

```
// an array of 5 items or less
myDataItem01 CHAR(30)[] { maxSize=5 };

// an array of 6 items or less
myDataItem02 myDataItemPart[] { maxSize=6 };

// an array of the maximum number of items
// (the maximum depends on the generated language)
myRecord myRecordPart[];
```

The rules for referencing the elements of a dynamic array are equivalent to the rules for referencing a static array. If your code refers to a dynamic array of records or data items and does not specify brackets or a subscript, the reference is to the array as a whole.

An out-of-memory situation is treated as a catastrophic error and ends the program.

Functions and variables for processing dynamic arrays

A set of functions and read-only variables are available for each dynamic array. In the following descriptions, substitute the array name for *arrayName* and note that the name may be qualified by a package name, a library name, or both:

arrayName.appendAll(appendArray)

This function does as follows:

- Appends to the array that is referenced by *arrayName*, adding a copy of the array that is referenced by *appendArray*
- Increments the array size by the number of added elements
- Assigns an appropriate index value to each of the appended elements

The elements of *appendArray* must be of the same type as the elements of *arrayName*.

arrayName.appendElement(content)

This function places an element to the end of the specified array and increments the size by one. For *content*, you can substitute a variable of the appropriate type; alternatively, you can specify a literal that is assigned to an element created during the operation. The process copies data; if you assign a variable, that variable is still available for comparison or other purposes.

The rules for assigning a literal are as specified in *Assignments*.

arrayName.insertElement(content, index)

This function does as follows:

- Places an element in front of the element that is now at the specified location in the array
- Increments the array size by one
- Increments the index of each element that resides after the inserted element

content is the new content (a constant or variable of the appropriate type for the array), and *index* is an integer literal or a numeric variable that indicates the location of the new element.

If *index* is one greater than the number of elements in the array, the function creates a new element at the end of the array and increments the array size by one.

arrayName.removeAll()

This function removes the elements of the array from memory. The array can be used, but its size is zero.

arrayName.removeElement(index)

This function removes the element at the specified location, decrements the array size by one, and decrements the index of each element that resides after the removed element.

index is an integer literal or a numeric variable that indicates the location of the element to be removed.

sysLib.maximumSize

As described in *sysLib.maximumSize*.

sysLib.size

As described in *sysLib.size*.

Use of dynamic arrays as arguments and parameters

A dynamic array can be passed as an argument to an EGL function. The related parameter must be defined as a dynamic array of the same type as the argument; and for a data item, the type must include the same length and decimal places, if any.

A dynamic array cannot be passed as an argument to another program.

An example of a function that uses a dynamic array as a parameter is as follows:

```
Function getAll (employees Employee[])
;
end
```

At run time, the maximum size for a parameter is the maximum size declared for the corresponding argument. The function or called program can change the size of the array, and the change is in effect in the invoking code.

SQL processing and dynamic arrays

EGL lets you use a dynamic array to access rows of a relational database. For details on reading multiple rows, see *get*. For details on adding multiple rows, see *add*.

Related concepts

“Compatibility with VisualAge Generator” on page 276
 “References to variables and constants” on page 34

Related reference

“add” on page 293
 “Assignments” on page 296
 “EGL system limits” on page 356
 “get” on page 318
 “in” on page 416
 “sysLib.size” on page 617

EGL statements

Each EGL function is composed of zero to many EGL statements of the following kinds:

- A *variable declaration* or *constant declaration* provides access to a named area of memory. The value of a variable can be changed at run time; the value of a constant cannot. Either kind of declaration can be anywhere in a function except in a *block*, as described later.
- A *function invocation* directs processing to a function, as in this example:

```
myFunction(myInput);
```

 Recursive calls are valid only if you are generating for Java.
- An *assignment statement* can copy any of the following values into a variable:
 - Data from a constant or variable
 - A literal
 - A value returned from a function invocation
 - The result of an arithmetic calculation
 - The result of a string concatenation

Examples of assignment statements are as follows:

```
myItem = 15;
myItem = readFile(myKeyValue);
myItem = bigValue - 32;
record1.message = "Operation " + "successful!";
```

- A *keyword statement* provides additional functionality such as file access. Each of these statements is named for the keyword that begins the statement; for example:

```
add record1;    // an add statement
return (0);     // a return statement
```
- A *null statement* is a semicolon that has no effect but may be useful as a placeholder, as in this example:


```

if (myItem = 5)
;           // a null statement
else
    myFunction(myItem);
end

```

Non-null EGL statements have the following characteristics:

- A statement can reference named memory areas, which are of these kinds:
 - Form
 - PageHandler
 - Record
 - DataTable
 - Item (a category that includes data items, as well as structure items in records, forms, and tables)
 - Array (a memory area based on a structure item that has an occurs value greater than 1)
- A statement can include these kinds of expressions--
 - A *logical expression* resolves to true or false
 - A *numeric expression* resolves to a number, which may be signed and include a decimal point
 - A *string expression* resolves to a series of characters, which may include single-byte characters, double-byte characters, or a combination of the two
- A statement either ends with a semicolon or with a *block*, which is a series of zero or more subordinate statements that act as a unit. Block-containing statements are terminated with an end delimiter, as in this example:

```

if (record2.status= "Y")
    record1.total = record1.total + 1;
    record1.message = "Operation successful!";
else
    record1.message = "Operation failed!";
end

```

A semicolon after an end delimiter is not an error but is treated as a null statement.

Names in statements and throughout EGL are case-*insensitive*; *record1* is identical to *RECORD1*, for example, and both *add* and *ADD* refer to the same keyword.

System words are a set of words that provide special functionality:

- A *system function* runs code and may return a value; for example:
 - **sysLib.minimum(arg1, arg2)** returns the minimum of two numbers
 - **strLib.strLen(arg1)** returns the length of a character string

The qualifier (**mathLib****strLib** or **sysLib**) is necessary only if your program has a function of the same name.
- A *system variable* provides a value without invoking a function; for example:
 - **sysVar.errorCode** contains a status code after your program accesses a file and in other situations
 - **sysVar.sqlcode** contains a status code after your program accesses a relational database

The qualifiers **sysVar** is necessary only if your program has a variable of the same name.

A line in a function can have more than one statement. It is recommended that you include no more than one statement per line, however, because you can use the EGL Debugger to set a breakpoint only at the first statement on a line.

See also *Comments*.

Related concepts

"EGL projects, packages, and files" on page 7

"Function part" on page 380

"Parts" on page 11

Related reference

"add" on page 293

"Assignments" on page 296

"call" on page 299

"case" on page 302

"close" on page 303

"Comments" on page 305

"Data initialization" on page 277

"delete" on page 307

"EGL reserved words" on page 290

"execute" on page 309

"Function invocations" on page 316

"get" on page 318

"get next" on page 324

"get previous" on page 328

"if, else" on page 332

"Keywords"

"Logical expressions" on page 358

"Numeric expressions" on page 364

"open" on page 335

"prepare" on page 339

"replace" on page 341

"set" on page 345

"String expressions" on page 365

"System words in alphabetical order" on page 508

"sysVar.terminalID" on page 629

"while" on page 356

Keywords

Keyword	Purpose
"add" on page 293	<i>add</i> places a record in a file, message queue, or database; or places a set of records in a database.
"call" on page 299	<i>call</i> transfers control to another program and optionally passes a series of values. Control returns to the caller when the called program ends. If the called program changes any data that was passed by way of a variable, the storage area available to the caller is changed, too.
"case" on page 302	<i>case</i> marks the start of multiple sets of statements, where at most only one of those sets is run. The <i>case</i> statement is equivalent to a C or Java <i>switch</i> statement that has a break at the end of each case clause.
"close" on page 303	<i>close</i> disconnects a printer; or closes the file or message queue associated with a given record; or, in the case of an SQL record, closes the cursor that was opened by an EGL <i>open</i> or <i>get</i> statement.

Keyword	Purpose
“converse” on page 306	<i>converse</i> presents a text form in a text application.
“delete” on page 307	<i>delete</i> removes either a record from a file or a row from a database.
“display” on page 309	<i>display</i> adds a text form to a run-time buffer but does not present data to the screen.
“execute” on page 309	<i>execute</i> lets you write one or more SQL statements; in particular, SQL data-definition statements (of type CREATE TABLE, for example) and data-manipulation statements (of type INSERT or UPDATE, for example).
“exit” on page 313	<i>exit</i> leaves the specified block, which by default is the block that immediately contains the exit statement.
“forward” on page 315	<i>forward</i> displays a Web page with variable information. This statement is invoked from a page handler.
“get” on page 318	<i>get</i> retrieves a single file record or database row and provides an option that lets you replace or delete the stored data later in your code. In addition, this statement allows you to retrieve a set of database rows and place each succeeding row into the next SQL record in a dynamic array. The <i>get</i> statement is sometimes identified as <i>get by key value</i> and is distinct from <i>get next</i> and <i>get previous</i> .
“get next” on page 324	<i>get next</i> reads the next record from a file or message queue, or the next row from a database.
“get previous” on page 328	<i>get previous</i> reads the previous record in the file that is associated with a specified EGL indexed record.
“goTo” on page 332	<i>goTo</i> causes processing to continue at a specified label, which must be in the same function as the statement and outside of a block.
“if, else” on page 332	<i>if</i> marks the start of a set of statements (if any) that run only if a logical expression resolves to true. The optional keyword <i>else</i> marks the start of an alternative set of statements (if any) that run only if the logical expression resolves to false. The keyword <i>end</i> marks the close of the <i>if</i> statement.
“move” on page 333	<i>move</i> copies data, in most cases from the named items in one structure to the same-named items in another.
“open” on page 335	<i>open</i> selects a set of rows from a relational database for later retrieval with <i>get next</i> statements. The <i>open</i> statement may operate on a cursor or on a called procedure.
“prepare” on page 339	<i>prepare</i> specifies an SQL PREPARE statement, which optionally includes details that are known only at run time. You run the prepared SQL statement by running an EGL <i>execute</i> statement or (if the SQL statement returns a result set) by running an EGL <i>open</i> or <i>get</i> statement.
“print” on page 341	<i>print</i> adds a print form to a run-time buffer.
“replace” on page 341	<i>replace</i> puts a changed record into a file or database.
“return” on page 344	<i>return</i> exits from a function and optionally returns a value to the invoking function.
“set” on page 345	<i>set</i> has various effects on records, text forms, and items.
“show” on page 353	<i>show</i> presents a text form from a main program along with any other forms buffered using the <i>display</i> statement; ends the current program and optionally forwards the input data from the user and state data from the current program to the program that handles the input from the user.

Keyword	Purpose
"transfer" on page 354	<i>transfer</i> gives control from one main program to another, ends the transferring program, and optionally passes a record whose data is accepted into the receiving program's <i>input record</i> . You cannot use a <i>transfer</i> statement in a called program.
"try" on page 355	<i>try</i> indicates that the program continues running if an input/output (I/O) statement, a system-function invocation, or a <i>call</i> statement results in an error and is within the <i>try</i> statement. If an exception occurs, processing resumes at the first statement in the <i>onException</i> block (if any), or at the first statement following the end of the <i>try</i> statement. A hard I/O error, however, is handled only if the system variable <i>sysVar.handleHardIOErrors</i> is set to 1; otherwise, the program displays a message (if possible) and ends.
"while" on page 356	<i>while</i> marks the start of a set of statements that run in a loop. The first run occurs only if a logical expression resolves to true, and each subsequent iteration depends on the same test. The keyword <i>end</i> marks the close of the <i>while</i> statement.

Related reference

"EGL statements" on page 70

Transfer of control across programs

EGL provides several ways to switch control from one program to another:

- The **call** statement gives control to another program and optionally passes a series of values. Control returns to the caller when the called program ends. If the called program changes any data that was passed as a variable, the content of the variable is changed in the caller.

The call does not commit databases or other recoverable resources, although an automatic server-side commit may occur.

You may specify characteristics of the call by setting a *callLink* element of the linkage options part. For details, see *call* and *callLink element*. For details on the automatic server-side commit, see *luwControl* in *callLink element*.

- The **transfer** statement gives control from one main program to another, ends the transferring program, and optionally passes a record whose data is accepted into the receiving program's *input record*. You cannot use a **transfer** statement in a called program.

Your program can transfer control by a statement of the form *transfer to a transaction* or by a statement of the form *transfer to a program*:

- A transfer to a transaction acts as follows--
 - In a program that runs as a Java main text or main batch program, the behavior depends on the setting of build descriptor option *synchOnTrxTransfer*--
 - If the value of *synchOnTrxTransfer* is YES, the transfer statement commits recoverable resources, closes files, closes cursors, and starts a program in the same run unit.
 - If the value of *synchOnTrxTransfer* is NO (the default), the transfer statement also starts a program in the same run unit, but does not close or commit resources, which are available to the invoked program.
 - In a Web program or page handler, a transfer to a transaction is not valid.
- A *transfer to a program* does not commit or rollback recoverable resources, but closes files, releases locks, and starts a program in the same run unit.

When you are transferring control from EGL-generated Java code, the linkage options part does not affect the characteristics of either kind of transfer. When the transferring program is in COBOL, however, the following statements apply:

- The linkage options part has no effect on transfer to a transaction
- You can set a linkage options part, **transferLink** element to affect the characteristics of transfer to a program

For details, see *transfer* and *transferLink* element.

- The system function **sysLib.startTransaction** starts a run unit asynchronously. The operation does not end the transferring program and does not affect the databases, files, and locks in the transferring program. You have the option to pass data into the *input record*, which is an area in the receiving program.

If your program invokes **sysLib.startTransaction**, you must generate the program with a linkage options part, **asynchLink** element. For details, see *sysLib.startTransaction* and *asynchLink* element.

- The EGL **show** statement ends the current main program in a text application and shows data to the user by way of a form. After the user submits the form, the **show** statement optionally forwards control to a second main program, which receives data received from the user as well as data that was passed without change from the originating program.

The **show** statement is affected by the settings in the linkage options part, **transferLink** element.

For details, see *show*.

- Finally, the **forward** statement is invoked from a page handler or from a program that runs in the context of a Web application. The statement acts as follows:
 1. Commits recoverable resources, closes files, and releases locks
 2. Forwards control
 3. Ends the code

The target in this case is another program or a Web page. For details, see *forward*.

Related reference

“asynchLink element” on page 441

“call” on page 299

“callLink element” on page 443

“forward” on page 315

“luwControl in callLink element” on page 450

“show” on page 353

“sysLib.startTransaction” on page 617

“transfer” on page 354

“transferToProgram element” on page 459

Exception handling

An error may occur when an EGL-generated program acts as follows:

- Accesses a file, queue, or database
- Calls another program
- Invokes a function
- Performs an assignment, comparison, or calculation

try blocks

An EGL *try block* is a series of zero to many EGL statements within the delimiters **try** and **end**. An example is as follows:

```
if (userRequest = "A")
  try
    add record1;
  onException
    myErrorHandler(12);
  end
end
```

In general, a try block allows your program to continue processing even if an error occurs.

The try block may include an *onException clause*, as shown earlier. That clause is invoked if one of the earlier statements in the try block fails; but in the absence of an *onException clause*, an error in a try block causes invocation of the first statement that immediately follows the try block.

The previous details on try blocks must be qualified. First, a try block affects processing only for errors in the following kinds of EGL statements:

- An I/O statement
- A system function
- A call statement

Processing of numeric overflows is not affected by the presence of a try block. For details on those kinds of error, see *sysVar.handleOverflow*.

Second, a try block has no effect on errors inside a user function (or program) that is invoked from within the try block. In the next example, if a statement fails in function myABC, the program ends immediately with an error message unless function myABC itself handles the error:

```
if (userRequest = "B")
  try
    myVariable = myABC();
  onException
    myErrorHandler(12);
  end
end
```

Third, the program ends immediately and with an error message in the following cases:

- An error of a kind that is covered specifically by a try block occurs outside of a try block; or
- One of the following cases applies, even within a try block--
 - A user-written function fails at invocation or on return to the invoker; or
 - The system variable **sysVar.handleHardIOErrors** is set to zero when a file or MQSeries I/O statement ends with a hard error (as described later); or

The following cases are also of interest:

- A COBOL run unit ends if a value is divided by zero, although a Java program handles that situation as a numeric overflow
- A COBOL run unit or Java program ends if a non-numeric character is assigned to a numeric variable, although a COBOL program gains some protection if you generate with build descriptor option **spacesZero**

Note: To support the migration of programs written in VisualAge Generator and EGL 5.0, the variable `sysVar.handleSysLibErrors` (previously called `ezereply`) allows you to process some errors that occur outside of a try block. Avoid use of that variable, which is available only if you are working in VisualAge Generator compatibility mode.

Error-related system variables

EGL provides error-related system variables that are set in a try block either in response to successful events or in response to non-terminating errors. The values in those variables are available in the try block and in code that runs subsequent to the try block, and the values in most cases are restored after a converse, if any.

The EGL run time does not change the value of any error-related variables when statements run outside of a try block. Your program, however, may assign a value to an error-related variable outside of a try block.

The system variable `sysVar.exceptionCode` is given a value in various situations, and in all those situations one or more additional variables are also set, depending on the nature of the program's interaction with the run-time environment:

- The system variables `sysVar.exceptionCode` and `sysVar.errorCode` are both given values after any of the following kinds of statements run in a try block:
 - A **call** statement
 - An I/O statement that operates on an indexed, MQ, relative, or serial file
 - An invocation of almost any system function
- The system variables `sysVar.exceptionCode`, `sysVar.errorCode`, `sysVar.mqConditionCode`, and `sysVar.mqReasonCode` are all given values after an I/O statement in a try block operates on an MQ record
- The system variable `sysVar.exceptionCode` is given a value after a relational database is accessed from a statement in a try block. Values are also assigned to variables in the SQL communication area (SQLCA); for details, see *sysVar.sqlca*.

If a non-terminating error occurs in a try block, the value of `sysVar.exceptionCode` is equivalent to the numeric component of the EGL error message that would be presented to the user if the error occurred outside of the try block. The values of the situation-specific variables like `sysVar.errorCode` and `sysVar.mqConditionCode`, however, are provided by the run-time system. In the absence of an error, the value of `sysVar.exceptionCode` and at least one of the situation-specific variables is the same: a string of eight zeroes.

An error code is assigned to `sysVar.exceptionCode` and `sysVar.errorCode` in the case of a non-terminating numeric overflow, as described in *sysVar.handleOverflow*; but a successful arithmetic calculation does not affect any of the error-related system variables.

Error-related system variables are also not affected by the invocation of a function other than a system function, and `sysVar.errorCode` (the variable affected by most system functions) is not affected by errors in these:

- `sysLib.calculateChkDigitMod10`
- `sysLib.calculateChkDigitMod11`
- `strLib.concatenate`
- `strLib.concatenateWithSeparator`
- `sysLib.connectionService`
- `sysLib.connect`
- `sysLib.convert`

- **sysLib.disconnect**
- **sysLib.disconnectAll**
- **sysLib.purge**
- **sysLib.queryCurrentDatabase**
- **strLib.setBlankTerminator**
- **sysLib.setCurrentDatabase**
- **strLib.strLen**
- **sysLib.verifyChkDigitMod10**
- **sysLib.verifyChkDigitMod11**
- **sysLib.wait**

When an error value is assigned to **sysVar.exceptionCode**, the system variable **sysVar.exceptionMsg** is assigned the text of the related EGL error message, and the system variable **sysVar.exceptionMsgCount** is assigned the number of bytes in the error message, excluding trailing blanks and nulls. When the string of eight zeroes is assigned to **sysVar.exceptionCode**, **sysVar.exceptionMsg** is assigned blanks and **sysVar.exceptionMsgCount** is set to zero.

I/O statements

In relation to I/O statements, an error can be hard or soft:

- A soft error is any of these--
 - No record was found during an I/O operation on an SQL database table
 - One of the following problems occurs in an I/O operation on an indexed, relative, or serial file:
 - Duplicate record (when the external data store allows insertion of a duplicate)
 - No record found
 - End of file
- A hard error is any other problem; for example--
 - Duplicate record (when the external data store prohibits insertion of a duplicate)
 - File not found
 - Communication links are not available during remote access of a data set

If a soft I/O error occurs in a try block, the generated program continues running. If a hard I/O error occurs in a try block, the consequence depends on the value of an error-related system variable:

- During access of a file, relational database, or MQSeries message queue, the following rules apply--
 - If **sysVar.handleHardIOErrors** is set to 1, the program continues running
 - If **sysVar.handleHardIOErrors** is set to 0, the program presents an error message, if possible, and ends

If either a hard or soft I/O error occurs outside of a try block, the generated program presents an error message, if possible, and ends.

If you are accessing DB2 directly (not through JDBC), the sqlcode for a hard error is 304, 802, or less than 0.

Error identification

You can determine what kind of error occurred in a try block by including a **case** or **if** statement inside or outside the try block, and in that statement you can test

the value of various system variables. If you are responding to an I/O error and if your statement uses an EGL record, however, it is recommended that you use an elementary logical expression. Two formats of the expression are available:

recordName **is** *IOerrorValue*

recordName **not** *IOerrorValue*

recordName

Name of the record used in the I/O operation

IOerrorValue

One of several I/O error values that are constant across database management systems

If you don't use the logical expressions with I/O error values and then change database management systems, you may need to modify and regenerate your program. In particular, if you are using JDBC, it is recommended that you use the I/O error values to test for errors rather than the value of **sysVar.sqlcode** or **sysVar.sqlState** or the equivalent values in **sysVar.sqlca**. Those values are dependent on the underlying database implementation when JDBC is in use.

Related concepts

"Compatibility with VisualAge Generator" on page 276

Related reference

"EGL Java run-time error codes" on page 635

"Exception handling and status (system words)" on page 526

"sysVar.handleHardIOErrors" on page 532

"sysVar.handleOverflow" on page 532

"sysVar.overflowIndicator" on page 534

"sysVar.handleSysLibErrors" on page 533

"sysVar.sqlca" on page 551

"sysVar.sqlcode" on page 552

"sysVar.sqlState" on page 555

"sysVar.mqConditionCode" on page 534

"sysVar.errorCode" on page 530

"I/O error values" on page 418

"Logical expressions" on page 358

"EGL statements" on page 70

System words

System words are a set of words that provide special functionality:

- A *system function* runs code and may return a value; for example:
 - **sysLib.minimum**(*arg1*, *arg2*) returns the minimum of two numbers
 - **sysLib.setLocale**(*language*, *country*) does not return a value but dynamically changes the Java locale used in a Web-based application
- A *system variable* provides a value without invoking a function; for example:
 - **sysVar.errorCode** contains a status code after your program accesses a file and in other situations
 - **sysVar.sqlcode** contains a status code after your program accesses a relational database

For details, see *System words in alphabetical order* or see one of the other lists.

Related reference

- "Data conversion (system words)" on page 518
- "Date and time (system words)" on page 521
- "EGL statements" on page 70
- "Exception handling and status (system words)" on page 526
- "File and database (system words)" on page 537
- "Java access (system words)" on page 557
- "Mathematical (system words)" on page 580
- "String handling (system words)" on page 597
- "System words in alphabetical order" on page 508

Getting started

Setting EGL preferences

Set the basic EGL preferences as follows:

1. Click **Window > Preferences**.
2. When a list is displayed, click **EGL** to display the EGL screen.
3. Select or clear the check box for **VisualAge Generator Compatibility**. Your choice affects what options are available at development time, as described in *Compatibility with VisualAge Generator*.
4. In the **Encoding** list box, select the character-encoding set that will be used when you create new EGL build (.eglbld) files. The setting has no effect on existing build files. The default value is UTF-8.
5. In the **User ID** text box, specify the user ID for accessing the remote build machine, if any. The build descriptor option **destUserID** takes precedence, and both that option and the preference value take precedence over the master build descriptor option **destUserID**.
6. In the **Password** text box, specify the password for accessing the remote build machine, if any. The build descriptor option **destPassword** takes precedence, and both that option and the preference value take precedence over the master build descriptor option **destPassword**.
7. Click **Apply**.

To set other EGL preferences, see the list of related tasks at the bottom of this page. When you are finished setting preferences, click **OK**.

Related concepts

"Build" on page 121

"Compatibility with VisualAge Generator" on page 276

Related tasks

"Setting the default build descriptors" on page 237

"Setting preferences for the EGL debugger" on page 114

"Setting preferences for source styles" on page 105

"Setting preferences for SQL database connections" on page 188

"Setting preferences for SQL retrieve" on page 190

Switching to the EGL or EGL Web perspective

To open the EGL or EGL Web perspective, do as follows:

1. In the Workbench, click **Window > Open Perspective > Other**. The Select Perspective dialog is displayed.
2. Click **EGL** or **EGL Web**.
3. Click **OK**. The requested perspective is now open.

On the Workbench shortcut bar, you can click the **EGL Perspective** or **EGL Web Perspective** button to switch to the requested perspective.

Creating a project to work with EGL

For an overview of how to organize your work, see *EGL projects, packages, and files*.

To set up a new project, do as follows:

1. In the Workbench, do either of the following steps:

- Click **File > New > Project**; or
- Right-click, then click **New > Project**.

The New Project wizard opens.

2. In the left pane, click **EGL**. In the right pane, click **EGL Web Project** (if you wish to interact with users at Web browsers) or **EGL Project** (in other cases). Click **Next** and follow a process:

- “Creating an EGL Web project”; or
- “Creating an EGL project.”

Creating an EGL Web project

The **New EGL Web project** wizard is displayed. To create an EGL Web project, do as follows:

1. In the **Project name** field, type a name for the project. By default, the project is placed in your workspace; but you can click **Browse** and choose a different location.
2. Select how to specify a build descriptor, which is the part that directs processing at generation time:
 - **Create new project build descriptor(s) automatically** means that EGL provides build descriptors and writes them to a build file (extension .eglbld) that has the same name as the project.

To specify some of the values in those build descriptors, click **Options**. To change those values later, change the build file that is created for you.
For further details, see *Specifying database options at project creation*.
 - **Use build descriptor specified in EGL preference** means that EGL points to a build descriptor that you created and identified as an EGL preference.
 - **Select existing build descriptor** allows you to specify a build descriptor from those that are available in your workspace.
3. If you requested that a build descriptor be created automatically, you can place a value in the **JNDI Name for SQL Connection** field. The effect is to assign the name to which the default data source is bound in the JNDI registry at debug or generation time. (An example value is `java:comp/env/jdbc/MyDB`.) Your selection assigns a value to the build descriptor option `sqlJNDIName`. If the **JNDI Name for SQL Connection** field is already populated, the value was obtained from a Workbench preference, as described in *Setting preferences for SQL database connections*.
4. In most cases, click **Finish**. To do additional customization (as is possible for any Web project), check **Configure Advanced Options** and click **Next**. Configure J2EE settings, then click **Next**. Select feature settings, then click **Next**. Select a page template, then click **Finish**.

Creating an EGL project

The **New EGL project** wizard is displayed. To create an EGL project, do as follows:

1. In the **Project name** field, type a name for the project. By default, the project is placed in your workspace; but you can click **Browse** and choose a different location.

2. In Target Runtime Platform, click the radio button for **Java** or **COBOL**.
3. Select how to specify a build descriptor, which is the part that directs processing at generation time:
 - **Create new project build descriptor(s) automatically** means that EGL provides build descriptors and writes them to a build file (extension .eglbld) that has the same name as the project.
To specify some of the values in those build descriptors, click **Options**. To change those values later, change the build file that is created for you.
For further details, see *Specifying database options at project creation*.
 - **Use build descriptor specified in EGL preference** means that EGL points to a build descriptor that you created and identified as an EGL preference.
 - **Select existing build descriptor** allows you to specify a build descriptor from those that are available in your workspace.
4. In most cases, click **Finish**. If you click **Next**, however, you can specify other source folders and projects to reference from the project you are creating. When you have finished selecting other source folders and projects, click **Finish**.

Related concepts

“Build descriptor part” on page 234

“EGL projects, packages, and files” on page 7

Related tasks

“Specifying database options at project creation”

“Setting preferences for SQL database connections” on page 188

Related reference

“sqlJNDIName” on page 264

Specifying database options at project creation

To assign option values in the build descriptor that is created automatically by EGL, work at the **Project Build Options** dialog. For details on how to display the dialog, see *Creating a project to work with EGL*.

To accept the database-connection information that was specified in preferences, click the check box.

In relation to Java output, the next table shows each on-screen label and the related build descriptor option.

Label	Build descriptor option
Database type	dbms
Database JDBC driver	sqlJDBCClass
Database name	sqlJNDIName (for J2EE output) or sqlDB (for non-J2EE output)

In relation to COBOL output, the next table shows each on-screen label and the related build descriptor option .

Label	Build descriptor option
system	system
output directory	genDirectory

Label	Build descriptor option
host machine TCP/IP name	destHost
host machine port number	destPort
host userID	destUserID
host password	destPassword
host SQL database	sqlDB
host DB2 userID	sqlID
host DB2 password	sqlPassword

Related concepts

“Build descriptor part” on page 234

Related tasks

“Creating a project to work with EGL” on page 82

Related reference

“Build descriptor options” on page 237

Related reference

“Symbolic parameters” on page 272

Creating an EGL source folder

Once you create a project in the workbench, you can create one or more folders within that project to contain your EGL files.

To create a folder for grouping EGL files, do as follows:

1. In the workbench, click **File > New > EGL Source Folder**.
2. Select the project that will contain the EGL folder. In the Folder name field, type the name of the EGL folder, for example myFolder.
3. Click the **Finish** button.

Related concepts

“EGL projects, packages, and files” on page 7

“Introduction to EGL” on page 1

Related tasks

“Creating a project to work with EGL” on page 82

Related reference

“Creating an EGL source file” on page 85

“Naming conventions” on page 468

Creating an EGL package

An EGL package is a named collection of related source parts. To create an EGL package, do as follows:

1. Identify a project or folder to contain the package. You must create a project or folder if you do not already have one.
2. In the workbench, click **File > New > EGL Package**.

3. Select the project or folder that will contain the EGL package. The Source Folder field may be pre-populated depending on the current selection in the Project Navigator.
4. In the Package Name field, type the name of the EGL package. See *EGL projects, packages, and files* for details on package naming conventions.
5. Click the **Finish** button.

Related concepts

"EGL projects, packages, and files" on page 7

"Introduction to EGL" on page 1

Related tasks

"Creating an EGL source folder" on page 84

"Creating a project to work with EGL" on page 82

"Switching to the EGL or EGL Web perspective" on page 81

Related reference

"Creating an EGL source file"

Creating an EGL source file

To create an EGL source file, do as follows:

1. Identify a project or folder to contain the file. You must create a project or folder if you do not already have one.
2. In the workbench, click **File > New > EGL Source File**.
3. Select the project or folder that will contain the EGL file. Select the package that will contain the EGL file. In the EGL Source File Name field, type the name of the EGL file, for example myEGLFile.
4. Click **Finish** to create the file. An extension (.egl) is automatically appended to the end of the file name. The EGL file appears in the Project Navigator view and automatically opens in the default EGL editor.

Related concepts

"EGL projects, packages, and files" on page 7

"Introduction to EGL" on page 1

Related tasks

"Creating an EGL source folder" on page 84

"Creating a project to work with EGL" on page 82

Creating an EGL program part

An EGL program part is the main logical unit used to generate a COBOL program, a Java program, a Java wrapper, or an Enterprise JavaBean session bean. For more information, see *Program part*.

A program part is automatically added to a program file and named appropriately when you create the file in the workbench. Program file specifications allow only one program part per file, and require a program name that matches the file name.

To create a program file with a program part, do as follows:

1. Identify a project or folder to contain the file. You must create a project or folder if you do not already have one.
2. In the workbench, click **File > New > Program**.

3. Select the project or folder that will contain the EGL file, then select a package. Since the program name will be identical to the file name, choose a file name that adheres to EGL part name conventions. In the EGL Source File Name field, type the name of the EGL file, for example myEGLprg. Select an EGL program type (for details, see *Basic program in EGL source format* , *TextUI program in EGL source format*). If the program part is a main program, click to remove the check mark from Create as called program.
4. Click the **Finish** button.

Related concepts

"EGL projects, packages, and files" on page 7
"Introduction to EGL" on page 1
"Program part" on page 477

Related tasks

"Creating an EGL source folder" on page 84
"Creating a project to work with EGL" on page 82
"Switching to the EGL or EGL Web perspective" on page 81

Related reference

"Basic program in EGL source format" on page 479
"Creating an EGL source file" on page 85
"Naming conventions" on page 468
"Text UI program in EGL source format" on page 481

Creating an EGL dataTable part

An EGL dataTable part associates a data structure with an array of initial values for the structure. To create an EGL dataTable part, do as follows:

1. Identify a project or folder to contain the file. You must create a project or folder if you do not already have one.
2. In the workbench, click **File > New > Data Table**.
3. Select the project or folder that will contain the EGL file, then select a package. Since the dataTable name will be identical to the file name, choose a file name that adheres to EGL part name conventions. In the EGL Source File Name field, type the name of the EGL file, for example myDataTable. Select a dataTable sub-type (for details, see *Data Table part in EGL source format*).
4. Click the **Finish** button.

Related concepts

"DataTable part" on page 285
"EGL projects, packages, and files" on page 7
"Introduction to EGL" on page 1

Related tasks

"Creating an EGL source folder" on page 84
"Creating a project to work with EGL" on page 82
"Switching to the EGL or EGL Web perspective" on page 81

Related reference

"Creating an EGL source file" on page 85
"DataTable part in EGL source format" on page 287
"Naming conventions" on page 468

Creating an EGL library part

An EGL library part contains a set of functions, variables, and constants that can be used by programs, page handlers, or other libraries. To create an EGL library part, do as follows:

1. Identify a project or folder to contain the file. You must create a project or folder if you do not already have one.
2. In the workbench, click **File > New > Library**.
3. Select the project or folder that will contain the EGL file, then select a package. Since the library name will be identical to the file name, choose a file name that adheres to EGL part name conventions. In the EGL Source File Name field, type the name of the EGL file, for example myLibrary.
4. Click the **Finish** button.

Related concepts

"EGL projects, packages, and files" on page 7

"Introduction to EGL" on page 1

"Library part" on page 433

Related tasks

"Creating an EGL source folder" on page 84

"Creating a project to work with EGL" on page 82

"Switching to the EGL or EGL Web perspective" on page 81

Related reference

"Creating an EGL source file" on page 85

"Library part in EGL source format" on page 434

"Naming conventions" on page 468

Opening a part

You can use the Open Part dialog to quickly open the source file that contains an EGL part. To locate a part using the Open Part dialog, do as follows:

1. In the workbench, click the **Open Part** button on the toolbar or select **Open Part** from the Navigation menu. The Open Part dialog is displayed.
2. Type the name of the part you want to locate; or to display a list of parts with names that match a specific pattern of characters, embed wildcard symbols within the name. A question mark (?) acts as a substitute for any one character and an asterisks (*) fills in for a string of characters. For example, typing *myForm?Group* will locate parts named myForm1Group and myForm2Group, but not myForm10Group. Typing *myForm*Group* will locate parts named myForm1Group, myForm2Group, and myForm10Group. As you type the name, qualifying parts display in the Matching parts section of the Open Part dialog.
3. From the list of parts, select the part you want to open. The Qualifier section of the Open Part dialog displays the path containing the folder, project, package, and source file that hold the selected part. In the event that multiple parts have the same name, you can select a specific part by clicking on the path of the file you want to open.
4. Click **OK**. The source file containing the part you selected opens in the EGL editor and the part name is highlighted.

Related concepts

"EGL projects, packages, and files" on page 7 "Parts" on page 11

Related tasks

“Creating an EGL source file” on page 85

“Locating an EGL source file in the Project Navigator”

Related reference

“EGL editor” on page 103

Locating an EGL source file in the Project Navigator

If you are editing an EGL source file, you can quickly locate the file in the Project Navigator view. The Show in Project Navigator context menu option does the following:

- Opens the Project Navigator view, if it is not already open
- Expands the Project Navigator tree nodes needed to locate the source file
- Highlights the source file

To locate an EGL source file in the Project Navigator, do as follows:

1. Right-click within the editor area of an open EGL source file. A context menu displays.
2. Select **Show in Project Navigator** from the context menu.

Related tasks

“Creating an EGL source file” on page 85

“Opening a part” on page 87

Related reference

“EGL editor” on page 103

Creating a build file

To create a build file, do as follows:

1. Identify a project or folder to contain the file. You must create a project or folder if you do not already have one. The project should be an EGL or EGL Web project.
2. In the workbench, click **File > New > EGL Build File**.
3. Select the project or folder that will contain the EGL build file. In the File name field, type the name of the EGL build file, for example MyEGLbuildParts. The extension .eglbld is required for the file name. An extension is automatically appended to the end of the file name if no extension or an invalid extension is specified.
4. Click **Finish** to create the build file with no EGL build part declaration. The build file appears in the Project Navigator view and automatically opens in the EGL build parts editor.
5. To add an EGL build part before creating the build file, click **Next**. Select the type of build part to add, then click **Next**. Type a name and a description for the build part, then click **Finish**. The build file appears in the Project Navigator view and automatically opens in the EGL build parts editor.

Related concepts

“EGL projects, packages, and files” on page 7

“Introduction to EGL” on page 1

Related tasks

"Adding an import statement to a build file"

"Creating a build file with a build descriptor part"

"Creating a build file with a linkage options part" on page 90

"Creating a build file with a resource associations part" on page 90

"Creating a project to work with EGL" on page 82

Adding an import statement to a build file

Import statements allow EGL build files to reference parts in other build files. See *Import* for more information on the import feature.

To add an import statement to an EGL build file, do as follows:

1. Open an EGL build file with the EGL build parts editor. If you do not have a file open, do this in the Project Navigator:
 - a. Right-click on the build file in the Project Navigator
 - b. Select **Open With > EGL Build Parts Editor**
2. Click the **Imports** tab in the build parts editor.
3. Click the **Add** button.
4. Type or select the name of the file or folder to import, then click **OK**.

Related concepts

"Import" on page 26

Creating a build file with a build descriptor part

To create a build file with a build descriptor part:

1. Identify a project or folder to contain the file. You must create a project or folder if you do not already have one.
2. In the workbench, click **File > New > Other > EGL > EGL Build File**. Click **Next**.
3. Select the project or folder that will contain the EGL build file. In the File name field, type the name of the EGL build file, for example MyEGLbuildParts. The extension .eglbld is required for the file name. An extension is automatically appended to the end of the file name if no extension or an invalid extension is specified.
4. Click **Next**.
5. Click **Build Descriptor**, then click **Next**.
6. Choose a name for your build descriptor that adheres to EGL part name conventions. In the Name field, type the name of your build descriptor.
7. You can optionally create a chain of build descriptors, so that the first in the chain is processed before the second, and the second before the third. For details, see *Adding a build descriptor part*.
8. In the Description field, type a description of your build part.
9. Click **Finish**.
10. To add an import statement to the EGL build file, see *Adding an import statement to a build file*.

Related concepts

"EGL projects, packages, and files" on page 7

"Introduction to EGL" on page 1

Related tasks

- "Adding a build descriptor part" on page 92
- "Adding an import statement to a build file" on page 89
- "Creating an EGL source folder" on page 84
- "Creating a project to work with EGL" on page 82
- "Switching to the EGL or EGL Web perspective" on page 81

Related reference

- "Naming conventions" on page 468

Creating a build file with a linkage options part

To create a build file with a linkage options part:

1. Identify a project or folder to contain the file. You must create a project or folder if you do not already have one.
2. In the workbench, click **File > New > Other > EGL > EGL Build File**. Click **Next**.
3. Select the project or folder that will contain the EGL build file. In the File name field, type the name of the EGL build file, for example MyEGLbuildParts. The extension .eglbld is required for the file name. An extension is automatically appended to the end of the file name if no extension or an invalid extension is specified.
4. Click **Next**.
5. Click **Linkage Options**, then click **Next**.
6. Choose a name for your linkage options part that adheres to EGL part name conventions. In the Name field, type the name of your linkage options part.
7. In the Description field, type a description of your part.
8. Click **Finish**.
9. To add an import statement to the EGL build file, see *Adding an import statement to a build file*.

Related concepts

- "EGL projects, packages, and files" on page 7
- "Introduction to EGL" on page 1

Related tasks

- "Adding an import statement to a build file" on page 89
- "Creating an EGL source folder" on page 84
- "Creating a project to work with EGL" on page 82
- "Switching to the EGL or EGL Web perspective" on page 81

Related reference

- "Naming conventions" on page 468

Creating a build file with a resource associations part

To create a build file with a resource associations part:

1. Identify a project or folder to contain the file. You must create a project or folder if you do not already have one.
2. In the workbench, click **File > New > Other > EGL > EGL Build File**. Click **Next**.
3. Select the project or folder that will contain the EGL build file. In the File name field, type the name of the EGL build file, for example MyEGLbuildParts. The

extension .eglbld is required for the file name. An extension is automatically appended to the end of the file name if no extension or an invalid extension is specified.

4. Click **Next**.
5. Click **Resource Associations**, then click **Next**.
6. Choose a name for your resource associations part that adheres to EGL part name conventions. In the Name field, type the name of your resource associations part.
7. In the Description field, type a description of your part.
8. Click **Finish**.
9. To add an import statement to the EGL build file, see *Adding an import statement to a build file*.

Related concepts

"EGL projects, packages, and files" on page 7
"Introduction to EGL" on page 1

Related tasks

"Adding an import statement to a build file" on page 89
"Creating an EGL source folder" on page 84
"Creating a project to work with EGL" on page 82
"Switching to the EGL or EGL Web perspective" on page 81

Related reference

"Naming conventions" on page 468

Editing an EGL build path

For overview material, see these topics:

- *References to parts*
- *EGL build path and egldpath*

To include projects in the EGL project path, follow these steps:

1. In the Project Navigator, right-click on a project that you want to link to other projects, then click **Properties**.
2. Select the **EGL Build Path** properties page.
3. A list of all other projects in your workspace is displayed in the **Projects** tab. Click the check box beside each project you want to reference.
4. To put the projects in a different order or to export any of them, click the **Order and Export** tab and do as follows--
 - To change the position of a project in the build-path order, select the project and click the **Up** and **Down** buttons.
 - To export a project, select the related check box. To handle all the projects at once, click the **Select All** or **Deselect All** button.
5. Click **OK**.

Related concepts

"EGL projects, packages, and files" on page 7
"References to parts" on page 16
"Import" on page 26
"Parts" on page 11

Related reference

“EGL build path and eglpath” on page 409
“References to parts” on page 16
“Import” on page 26
“Parts” on page 11

Working with build descriptor parts

Adding a build descriptor part

A build descriptor part controls the generation process. It contains option names and their related values, and those option-and-value pairs specify how to generate and prepare EGL output. Some options specify other control parts, such as a resource association part, that are in the generation process. You can add a build descriptor part to an EGL build file. See *Build descriptor part* for more information. To add a build descriptor part, do as follows:

1. When you are working in the EGL or EGL Web perspective, open an EGL build file with the EGL build parts editor. If you do not have a file open, do as follows in the Project Navigator:
 - a. Right-click on the EGL build file
 - b. Select **Open With > EGL Build Parts Editor**.
2. By default, the Outline view is open in the EGL perspective. If you are working in the EGL Web perspective, open the Outline view by selecting **Show View > Outline** from the Window menu. In the Outline view, right-click on the build file, then click **Add Part**.
3. Click the **Build Descriptor** radio button, then click **Next**.
4. Choose a name for your build descriptor that adheres to EGL part name conventions. In the Name field, type the name of your build descriptor.
5. In the Description field, type a description of your build part.
6. Click **Finish**. The build descriptor is declared in the EGL build file and the build descriptor general options are displayed in the EGL build parts editor.
7. You can optionally create a chain of build descriptors, so that the first in the chain is processed before the second, and the second before the third. If you want to begin or continue a chain of build descriptors, specify the next build descriptor in the **nextBuildDescriptor** option field of the Options list. To populate the **nextBuildDescriptor** option field, do as follows:
 - a. Using the scroll bar on the Options list, scroll down until the **nextBuildDescriptor** option is in view.
 - b. If the nextBuildDescriptor row is not highlighted, click once to select the row.
 - c. Click the Value field once to put the field into edit mode.
 - d. You can type the name of the next build descriptor in the Value field or select an existing build descriptor from the drop-down list.

Related concepts

“Build descriptor part” on page 234

Related tasks

“Switching to the EGL or EGL Web perspective” on page 81

Related reference

“EGL build-file format” on page 269
“Naming conventions” on page 468

Editing general build descriptor options

A build descriptor part controls the generation process. To edit the general build descriptor options and symbolic parameters, do as follows:

1. When you are working in the EGL or EGL Web perspective, open an EGL build file with the EGL build parts editor. If you do not have a file open, do as follows in the Project Navigator:
 - a. Right-click on the EGL build file
 - b. Select **Open With > EGL Build Parts Editor**
2. By default, the Outline view is open in the EGL perspective. If you are working in the EGL Web perspective, open the Outline view by selecting **Show View > Outline** from the Window menu. In the Outline view, right-click on a build descriptor and select **Open**. There are two buttons in the upper right corner of the editor view. Make sure that the Show General Build Descriptor Options button (the first of the two buttons) is pressed. The EGL build parts editor displays the general build descriptor options for the current part definition.
3. You can optionally create a chain of build descriptors, so that the first in the chain is processed before the second, and the second before the third. If you want to begin or continue a chain of build descriptors, specify the next build descriptor in the **nextBuildDescriptor** field. If the nextBuildDescriptor row is not highlighted, click once to select the row, then click the Value field once to put the field into edit mode. You can type the name of the next build descriptor in the Value field or select an existing build descriptor from the drop-down list.
4. To specify the generation and preparation of EGL output, select a grouping of option-and-value pairs from the **Build option filter** drop-down list. If the option you want to define is not highlighted, click once to select the row, then click the Value field once to put the field into edit mode. You can type the option value, or if a drop-down list is available, select an existing value. If you want to limit your view of option-and-value pairs to the ones you have defined, click the Show only options specified check box.

Related concepts

"Build descriptor part" on page 234

Related tasks

"Editing Java run-time properties in a build descriptor"

"Switching to the EGL or EGL Web perspective" on page 81

Related reference

"EGL build-file format" on page 269

"Symbolic parameters" on page 272

Editing Java run-time properties in a build descriptor

When you are editing a build descriptor part, you can assign values to the following Java run-time properties, which are detailed in *Java run-time properties (details)*:

- `vgj.jdbc.database.SN`
- `vgj.datemask.gregorian.long.locale`
- `vgj.datemask.gregorian.short.locale`
- `vgj.datemask.julian.long.locale`
- `vgj.datemask.julian.short.locale`

To edit the properties, do as follows:

1. When you are working in the EGL or EGL Web perspective, open an EGL build file with the EGL build parts editor. If you do not have a file open, do as follows in the Project Navigator:
 - a. Right-click on the EGL build file
 - b. Select **Open With > EGL Build Parts Editor**
2. By default, the Outline view is open in the EGL perspective. If you are working in the EGL Web perspective, open the Outline view by selecting **Show View > Outline** from the Window menu. In the Outline view, right-click on a build descriptor and select **Open**. The EGL part editor displays the general build descriptor options for the current part definition.
3. Click the **Show Java Run-time Properties** button on the editor toolbar.
4. To add the Java run-time property `vgj.jdbc.database.SN`, do this:
 - a. In the screen area that is titled "Database mappings for connect", click the **Add** button
 - b. Type a "Server name" that you use when coding the system word `sysLib.connectionService`; this value is substituted for *SN* in the name of the generated property
 - c. If the row in the Database mappings for connect list is not highlighted, click once to select the row, then click the JNDI name or URL field once to put the field into edit mode. Type a value whose meaning is different for J2EE connections as compared with non-J2EE connections:
 - In relation to J2EE connections (as is needed in a production environment), the value is the name to which the datasource is bound in the JNDI registry; for example, `jdbc/MyDB`
 - In relation to a standard JDBC connection (as may be used for debugging), the value is the connection URL; for example, `jdbc:db2:MyDB`
5. To assign the date masks used when you code either `sysVar.currentFormattedDate` (for a Gregorian date) or `sysVar.currentFormattedJulianDate` (for a Julian date); or EGL validates a page item or a text-form field that has a length of 10 or more and a **dateFormat** property of *systemGregorian* or *systemJulian*, do this:
 - a. In the screen area that is titled "Date Masks", click the **Add** button
 - b. In the Locale column, select one of the codes in the listbox; the selected value is substituted for *locale* in the date-mask properties listed earlier. Only one of your entries is used at run time: the entry for which the value of *locale* matches the value of the Java run-time property `vgj.nls.code`
 - c. If the row in the Date Masks list is not highlighted, click once to select the row, then click the Long Gregorian Mask field once to put the field into edit mode. Either select a mask from the listbox or type a mask; characters other than D, Y, or digits can be used as separators, and the default value is specific to the locale
 - d. If the row in the Date Masks list is not highlighted, click once to select the row, then click the Long Julian Mask field once to put the field into edit mode. Either select a mask from the listbox or type a mask; characters other than D, Y, or digits can be used as separators, and the default value is specific to the locale
6. To assign the date masks used when EGL validates a page item or a text-form field that has a length less than 10 and a **dateFormat** property of *systemGregorian* or *systemJulian*, do this:
 - a. In the screen area that is titled "Date Masks", click the **Add** button

- b. In the Locale column, select one of the codes in the listbox; the selected value is substituted for *locale* in the date-mask properties listed earlier. Only one of your entries is used at run time: the entry for which the value of *locale* matches the value of the Java run-time property `vgj.nls.code`
 - c. If the row in the Date Masks list is not highlighted, click once to select the row, then click the Short Gregorian Mask field once to put the field into edit mode. Either select a mask from the listbox or type a mask; characters other than D, Y, or digits can be used as separators, and the default value is specific to the locale
 - d. If the row in the Date Masks list is not highlighted, click once to select the row, then click the Short Julian Mask field once to put the field into edit mode. Either select a mask from the listbox or type a mask; characters other than D, Y, or digits can be used as separators, and the default value is specific to the locale
7. To remove an assignment, click on it, then click the **Remove** button.

Related concepts

"Build descriptor part" on page 234

"Java run-time properties" on page 421

Related tasks

"Editing general build descriptor options" on page 93

"Switching to the EGL or EGL Web perspective" on page 81

Related reference

"EGL build-file format" on page 269

"Java run-time properties (details)" on page 423

"`sysLib.connectionService`" on page 545

"`sysVar.currentFormattedDate`" on page 522

"`sysVar.currentFormattedJulianDate`" on page 523

Working with linkage options parts

Adding a linkage options part

A linkage options part describes how a generated EGL program implements calls and transfers and how the program accesses files. To add this type of part, do as follows:

1. When you are working in the EGL or EGL Web perspective, open an EGL build file with the EGL build parts editor.
If you do not have a file open, do as follows in the Project Navigator:
 - a. Right-click on the EGL file
 - b. Select **Open With > EGL Build Parts Editor**
2. By default, the Outline view is open in the EGL perspective. If you are working in the EGL Web perspective, open the Outline view by selecting **Show View > Outline** from the Window menu. In the Outline view, right-click on the build file, then click **Add Part**.
3. Click **Linkage Options**, then click **Next**.
4. Choose a name for your linkage options part that adheres to EGL part name conventions. In the Name field, type the name of your linkage options part.
5. In the Description field, type a description of your part.
6. Click **Finish**. The linkage options part is added to the EGL file and the linkage options part page is opened in the EGL build parts editor.

Related concepts

“Linkage options part” on page 439

Related tasks

“Editing the asynchLink element of a linkage options part” on page 97

“Editing the callLink element of a linkage options part”

“Editing the transfer-related elements of a linkage options part” on page 98

“Switching to the EGL or EGL Web perspective” on page 81

Related reference

“EGL build-file format” on page 269

“Naming conventions” on page 468

Editing the callLink element of a linkage options part

A linkage options part describes how a generated EGL program implements calls and transfers and how the program accesses files. To edit the part's callLink element, do as follows:

1. When you are working in the EGL or EGL Web perspective, open an EGL build file with the EGL build parts editor.
If you do not have a file open, do as follows in the Project Navigator:
 - a. Right-click on the EGL build file
 - b. Select **Open With > EGL Build Parts Editor**
2. By default, the Outline view is open in the EGL perspective. If you are working in the EGL Web perspective, open the Outline view by selecting **Show View > Outline** from the Window menu. In the Outline view, right-click a linkage options part and click **Open**. The EGL part editor displays the current part declaration.
3. Click the Show CallLink Elements button on the editor toolbar.
4. To add a new CallLink element, click **Add** or press the Insert key, and type the Program Name (pgmName) or select a program name from the Program Name drop-down list.
5. To change the default call type associated with your program name, you can either:
 - Select the corresponding row in the CallLink elements list, then click the Type field (localCall, remoteCall, ejbCall) once to put the field into edit mode. Select the new call type from the Type drop-down list.
 - In the Properties of selected callLink elements list, click the type property once to put the Value field associated with that property into edit mode. Select the new call type from the Value drop-down list.
6. Other properties associated with your program name are listed in the Properties of selected callLink elements list based on the call type. To change the value of one of these properties, select the program name. In the Properties of selected callLink elements list, click the property you want to define once to put the Value field associated with that property into edit mode. Define the new value by selecting an option in the Value drop-down list, or by typing the new value in the Value field. For some properties, you can only select an option in a drop-down list. For other properties, you can only type a value in the Value field.
7. Modify the CallLink elements list as needed:
 - To reposition a callLink element, select an element and click either **Move Up** or **Move Down**.

- To remove a callLink element, select the element and click **Remove** or press the Delete key.

Related concepts

"Linkage options part" on page 439

Related tasks

"Adding a linkage options part" on page 95

"Switching to the EGL or EGL Web perspective" on page 81

Related reference

"asynchLink element" on page 441

"callLink element" on page 443

"EGL build-file format" on page 269

"Linkage properties file (details)" on page 431

"transferToProgram element" on page 459

Editing the asynchLink element of a linkage options part

A linkage options part describes how a generated EGL program implements calls and transfers and how it accesses files. To edit the part's asynchLink element, do as follows:

1. When you are working in the EGL or EGL Web perspective, open an EGL build file with the EGL build parts editor.
If you do not have a file open, do as follows in the Project Navigator:
 - a. Right-click on the EGL build file
 - b. Select **Open With > EGL Build Parts Editor**
2. By default, the Outline view is open in the EGL perspective. If you are working in the EGL Web perspective, open the Outline view by selecting **Show View > Outline** from the Window menu. In the Outline view, right-click a linkage options part and click **Open**. The EGL build parts editor displays the current part declaration.
3. Click the Show AsynchLink Elements button on the editor toolbar.
4. To add a new AsynchLink element, click **Add** or press the Insert key, and type the Record Name (recordName) or select a record name from the Record Name drop-down list.
5. To change the default linkage type associated with your record name, you can either:
 - Select the corresponding row in the AsynchLink Elements list, then click the Type field (localAsynch, remoteAsynch) once to put the field into edit mode. Select the new linkage type from the Type drop-down list.
 - In the Properties of selected asynchLink elements list, click the type property once to put the Value field associated with that property into edit mode. Select the new linkage type from the Value drop-down list.
6. Other properties associated with your record name are listed in the Properties of selected asynchLink elements list based on the linkage type. To change the value of one of these properties, select the record name. In the Properties of selected asynchLink elements list, click the property you want to define once to put the Value field associated with that property into edit mode. Define the new value by selecting an option in the Value drop-down list, or by typing the new value in the Value field. For some properties, you can only select an option in a drop-down list. For other properties, you can only type a value in the Value field.

7. Modify the asynchLink elements list as needed:
 - To reposition an asynchLink element, select an element and click either **Move Up** or **Move Down**.
 - To remove an asynchLink element, select an element and click **Remove** or press the Delete key.

Related concepts

"Linkage options part" on page 439

Related tasks

"Adding a linkage options part" on page 95

"Switching to the EGL or EGL Web perspective" on page 81

Related reference

"asynchLink element" on page 441

"EGL build-file format" on page 269

"Linkage properties file (details)" on page 431

"sysLib.startTransaction" on page 617

Editing the transfer-related elements of a linkage options part

A linkage options part describes how a generated EGL program implements calls and transfers and how the program accesses files. To edit the part's transfer-related elements, do as follows:

1. When you are working in the EGL or EGL Web perspective, open an EGL build file with the EGL build parts editor.

If you do not have a file open, do as follows in the Project Navigator:

 - a. Right-click on the EGL build file
 - b. Select **Open With > EGL Build Parts Editor**
2. By default, the Outline view is open in the EGL perspective. If you are working in the EGL Web perspective, open the Outline view by selecting **Show View > Outline** from the Window menu. In the Outline view, right-click a linkage options part and click **Open**. The EGL build parts editor displays the current part declaration.
3. Click the Show TransferLink Elements button on the editor toolbar. The Transfer to Program and Transfer to Transaction lists display.
4. To edit the Transfer to Program list, do as follows:
 - a. At the bottom of the Transfer to Program list, click **Add** or press the Insert key, and type the From Program (fromPgm) name or select a program name from the From Program name drop-down list.
 - b. To edit the To Program (toPgm) name, select the corresponding row in the Transfer to Program list, then click the To Program field once to put the field into edit mode. Type the program name or select a program name from the To Program drop-down list.
 - c. If an alias name is needed, select the corresponding row in the Transfer to Program list, then click the Alias field once to put the field into edit mode. Type the alias name.
 - d. To change the default linkage type associated with your program name, select the corresponding row in the Transfer to Program list, then click the Link Type (linkType) field once to put the field into edit mode. Select the new linkage type from the Link Type drop-down list.
 - e. Modify the Transfer to Program list as needed:

- To reposition a transferToProgram element, select an element and click either **Move Up** or **Move Down**.
 - To remove a transferToProgram element, select an element and click **Remove** or press the Delete key.
5. To edit the Transfer to Transaction list, do as follows:
 - a. At the bottom of the Transfer to Transaction list, click **Add** or press the Insert key, and type the To Program (toPgm) name or select a program name from the To Program name drop-down list.
 - b. If an alias name is needed, select the corresponding row in the Transfer to Transaction list, then click the Alias field once to put the field into edit mode. Type the alias name.
 - c. To edit the Externally Defined property associated with your program name, select the corresponding row in the Transfer to Transaction list, then click the Externally Defined field once to put the field into edit mode. Select the externally defined property from the Externally Defined property drop-down list.
 - d. Modify the Transfer to Transaction list as needed:
 - To reposition a transferToTransaction element, select an element and click either **Move Up** or **Move Down**.
 - To remove a transferToTransaction element, select an element and click **Remove** or press the Delete key.

Related concepts

"Linkage options part" on page 439

Related tasks

"Adding a linkage options part" on page 95

"Switching to the EGL or EGL Web perspective" on page 81

Related reference

"EGL build-file format" on page 269

"transferToProgram element" on page 459

"transferToProgram element" on page 459

Working with resource association parts

Adding a resource associations part

A resource associations part relates a file name with a system resource name on the target platform where you intend to deploy the generated code. You can add a resource associations part to an EGL build file. For details, see *Resource associations and file types*. To add a resource associations part, do the following:

1. When you are working in the EGL or EGL Web perspective, open an EGL build file with the EGL build parts editor. If you do not have a file open, do as follows in the Project Navigator:
 - a. Right-click on the EGL build file
 - b. Select **Open With > EGL Build Parts Editor**
2. By default, the Outline view is open in the EGL perspective. If you are working in the EGL Web perspective, open the Outline view by selecting **Show View > Outline** from the Window menu. In the Outline view, right-click on the build file, then click **Add Part**.
3. Click **Resource Associations**, then click **Next**.

4. Choose a name for your resource associations part that adheres to EGL part name conventions. In the Name field, type the name of your resource associations part.
5. In the Description field, type a description of your part.
6. Click **Finish**. The resource associations part is added to the EGL build file and the resource associations part page is opened in the EGL build parts editor.

Related concepts

"Build descriptor part" on page 234

"Resource associations and file types" on page 157

Related tasks

"Switching to the EGL or EGL Web perspective" on page 81

Related reference

"EGL build-file format" on page 269

"Naming conventions" on page 468

Editing a resource associations part

A resource associations part relates a file name with a system resource name on the target platform where you intend to deploy the generated code.

To edit a resource associations part, do as follows:

1. When you are working in the EGL or EGL Web perspective, open an EGL build file with the EGL build parts editor. If you do not have a file open, do as follows in the Project Navigator:
 - a. Right-click on the EGL build file
 - b. Select **Open With > EGL Build Parts Editor**
2. By default, the Outline view is open in the EGL perspective. If you are working in the EGL Web perspective, open the Outline view by selecting **Show View > Outline** from the Window menu. In the Outline view, right-click a resource associations part and click **Open**. The editor displays the current part definition.
3. To add a new Association element to the part, click **Add Association** or press the Insert key, and type the logical file name or select a logical file name.
4. To change the default system name associated with your logical file name, you can either:
 - Select the corresponding row in the Association elements list, then click the name once to put the field into edit mode. Select the new system name from the System drop-down list.
 - In the Properties of selected system entries list, click the system property once to put the Value field associated with that property into edit mode. Select the new system name from the Value drop-down list.
5. To change the default file type associated with your logical file name, you can either:
 - Select the row in the Association elements list that corresponds to your logical file name, then click the name once to put the field into edit mode. Select the new file type from the File Type drop-down list.
 - Select the row in the Association elements list that corresponds to your logical file name. In the Properties of selected system entries list, click the fileType property once to put the Value field associated with that property into edit mode. Select the file type from the Value drop-down list.

6. Modify the resource associations as needed.

- To associate more than one system and set of related properties with a logical file name, select the row in the Association elements list that corresponds to your logical file name. At the bottom of the Association elements list, click **Add System**. The added row is now selected and available for editing.
- To remove a system and related properties from an associated logical file name, select the row in the Association elements list that corresponds to your logical file name. At the bottom of the Association elements list, click **Remove** or press the Delete key.
- To remove a logical file name and any associated systems, select the row in the Association elements list that corresponds to your logical file name. At the bottom of the Association elements list, click **Remove** or press the Delete key.

Related concepts

“Build descriptor part” on page 234

“Resource associations and file types” on page 157

Related tasks

“Switching to the EGL or EGL Web perspective” on page 81

Related reference

“EGL build-file format” on page 269

EGL editor

To change an EGL file (extension .egl), work in the EGL source editor, which guides you with content assist.

To change an EGL build file (extension .eglbld), follow one of these procedures:

- Creating a build file
-
- Adding a build descriptor part
- Adding a linkage options part
-
- Adding a resource associations part

Related concepts

"EGL projects, packages, and files" on page 7

"Parts" on page 11

Related reference

"Content assist"

Content assist

The EGL editor provides *content assist*, which proposes information that you can add to your source file. With a keystroke or two, you can complete the name of a part, variable, or function or can place a *template* (the outline of a part) into your source file.

The keystroke that activates content assist is **Ctrl + Space**.

Related tasks

"Using the EGL templates with content assist"

Using the EGL templates with content assist

To practice content assist, do as follows:

1. Open a new EGL file.
2. On an available line, type **P** (for page handler or program) and press **Ctrl + Space**.
3. When a pop-up is displayed, click an icon for the part to customize. Do either of these steps:
 - Press **Enter** to select the first icon in the list; or
 - Use the arrow keys to select another icon (for a program) and press **Enter**.The editor places a part template in your file.
4. Customize the part.

When the template is displayed, the editor highlights the first area where you need to type information; in this case, specify the part name. After you type, press **Tab** to highlight the next area where you need to type.

You can use the **Tab** key repeatedly, and this use of the key is available until you reach the end of the file or until you change your in-file position in any other way.

5. To insert a function into your program or page handler, type **F** (for function), then press **Ctrl + Space**. Although you can select a part template again, do as follows

- Use the arrow keys or your mouse to scroll to the end of the list
- Press **Enter** or click the word *Function*; note that the absence of an icon means that you are selecting a string rather than a part template

The ability to select a string is more useful in other contexts, such as when you want to type a variable name quickly.

6. With the cursor at the end of the word *Function*, press **Ctrl + Space** and click an icon from the list.

The editor places the function template in your file.

7. Customize the part.
8. As you develop your code, periodically press **Ctrl + Space** to understand the range of services that are provided.

Related reference

“Content assist” on page 103

“Function part in EGL source format” on page 381

“PageHandler part in EGL source format” on page 472

“Program part in EGL source format” on page 478

Setting preferences for the EGL editor

To specify the EGL editor preferences, do as follows:

1. Click **Window > Preferences**.
2. When a list is displayed, expand **EGL** and click **Editor**.
3. To display line numbers when you review an EGL file, select the **Show line numbers** check box. To clear the line numbers, clear the check box. The file itself is not affected.
4. To show red underlines wherever errors are found in the source code, select the **Annotate errors in text** check box. To clear those underlines, clear the check box. The file itself is not affected.
5. To show a red error indicator in the right margin of the editor (overview ruler) whenever an error is found in the source code, select the **Annotate errors in overview ruler** check box. Clicking on the error indicator will take you to the location of the error in the source code. To clear the error indicator, clear the check box. The file itself is not affected.
6. To specify source styles, follow the process described in *Setting preferences for source styles*.
7. To add, remove, and customize templates for use in content assist, follow the process described in *Setting preferences for templates*.

Related tasks

“Setting preferences for source styles” on page 105

“Setting preferences for templates” on page 105

Related reference

“Content assist” on page 103

Setting preferences for source styles

You can change how EGL code is displayed in the EGL editor:

1. Click **Window > Preferences**
2. When a list of preferences is displayed, expand **EGL** and **Editor**, then click **Source Styles**.
3. To select the color that you want to appear behind the source type, click the **Custom** radio button in the Background color box. Click the button next to the Custom label. A color palette displays. Select a color, then click **OK**.
4. In the Foreground box, select a type of text, then click the **Color** button. A color palette displays. Select a color, then click **OK**.
5. Select the **Bold** check box if you want to make the type bold.
6. To save your changes, click **Apply** or (if you are finished setting preferences) click **OK**.

Related tasks

"Setting EGL preferences" on page 81

Setting preferences for templates

Do as follows to add, remove, or customize the templates that are displayed when you request content assist in the EGL editor:

1. Click **Window > Preferences**.
2. When a list of preferences is displayed, expand **EGL** and **Editor**, then click **Templates**. A list of templates is displayed.

Note: As in other applications on Windows 2000/NT/XP, you can click an entry to select it; can use **Ctrl-click** to select or unselect an entry without affecting other selections; and can use **Shift-click** to select a set of entries that are contiguous to the entry you last clicked.

3. To make a template available in the EGL editor, select the check box to the left of a template name. To make all the listed templates available, click **Enable All**. Similarly, to make a template unavailable, clear the related check box; and to make all the listed templates unavailable, click **Disable All**.
4. To create a new template, do as follows--
 - a. Click **New**
 - b. When the New Template dialog is displayed, specify both a name and a description because a template is guaranteed to be displayed in a content-assist list only if the combination of name and description are unique across all templates.

Note: If the first word used in the template is an EGL keyword (such as Function), the template is available when you request content assist in the EGL editor, but only when the on-screen cursor is at a place where the word is valid. Similarly, if you type a prefix, then request content assist, all templates beginning with that prefix are available provided the on-screen cursor is in a position where that template is syntactically allowed. For example, type "fun" to request function templates. If you do not type either a prefix or the full first word, you will not see any templates when you request content assist.

- c. In the **Pattern** field, type the template itself:
 - Type any text that you wish to display

- To place a preexisting variable at the on-screen cursor position, click **Insert Variable**, then double-click a variable. When you insert the template in the EGL editor, each of those variables resolves to the appropriate value.

- To create a custom variable, type a dollar sign (\$) followed by a left brace ({}), a string, and a right brace ({}), as in this example:

`${variable}`

You may find it easier to insert a preexisting variable and change the name for your own use.

When you insert a custom template in the EGL editor, each variable is underlined to indicate that a value is required.

- To complete the task, click **OK** and, at the templates screen, click **Apply**.
5. To review an existing template, click on the listed entry and review the Preview box.
 6. To edit an existing template, click on the listed entry, then click **Edit**. Interact with the Edit Template dialog as you did with the New Template dialog.
 7. To remove an existing template, click on the listed entry, then click **Remove**. To remove multiple templates, use the Windows 2000/NT/XP convention for selecting multiple list entries, then click **Remove**.
 8. To import a template from an XML file, click **Import** at the right of the template list and follow the browse mechanism to specify the location of the file.
 9. To export a template to an XML file, click **Export** at the right of the template list and follow the browse mechanism to specify the location of the new file. To export multiple templates, use the Windows 2000/NT/XP mechanism for selecting multiple list entries, then click **Export**.
 10. To export all the listed templates to an XML file, click **Export All** and follow the browse mechanism to specify the location of the file.
 11. To save your changes, click **Apply**. To return to the template list that was in effect at installation time, click **Restore Defaults**.

Related reference

“Content assist” on page 103

EGL debugger

When you are in the Workbench, the EGL debugger lets you debug EGL code without requiring that you first generate output. These categories are in effect:

- To debug page handlers, as well as programs used in a J2EE context, you can use the local WebSphere Application Server test environment in debug mode--
 - You must use that environment for all code that runs under J2EE in a Web application.
 - You may use that environment for programs that run in a batch application under J2EE.
- To debug other code (batch applications that do not run under J2EE; or text applications), use a launch configuration that is outside of the WebSphere Test Environment. In this case, you can start the debug session with a few keystrokes.

If you are working on a batch program that you intend to deploy in a J2EE context, you can use the launch configuration to debug the program in a non-J2EE context. Although your setup is simpler, you need to adjust some values:

- You need to set the value of the build descriptor option J2EE to NO when you use the launch configuration.
- Also, you need to adjust Java property values to conform to differences in accessing a relational database--
 - For J2EE you specify a string like *jdbc/MyDB*, which is the name to which a data source is bound in the JNDI registry. You specify that string in these ways:
 - By setting the build descriptor option *sqlJNDIName*; or
 - By placing a value in the EGL SQL Database Connections preference page, in the Connection JNDI Name field; for details, see *Setting preferences for SQL database connections*.
 - For non-J2EE you specify a connection URL like *jdbc:db2:MyDB*. You specify that string in these ways:
 - By setting the build descriptor option *sqlDB*; or
 - By placing a value in EGL SQL Database Connections preference page, in the field Connection URL; for details, see *Setting preferences for SQL database connections*.

A later section describes the interaction of build descriptors and EGL preferences.

Debugger mode

The debugger has two modes: Java and COBOL, as determined by the build descriptor option **system**. If no build descriptor is in use or if you set the system type to DEBUG as a debug preference, the mode is Java.

The mode controls how the debugger acts in situations where the EGL run-time behavior differs for Java and COBOL output.

Debugger commands

You use the following commands to interact with the EGL debugger:

Add breakpoint

Identifies a line at which processing pauses. When code execution pauses, you can examine variable values as well as the status of files and screens.

Breakpoints are remembered from one debugging session to the next, unless you remove the breakpoint.

You cannot set a breakpoint at a blank line or at a comment line.

Disable breakpoint

Inactivates a breakpoint but does not remove it.

Enable breakpoint

Activates a breakpoint that was previously disabled.

Remove breakpoint

Clears the breakpoint so that processing no longer automatically pauses at the line.

Remove all breakpoints

Clears every breakpoint.

Run

Runs the code until the next breakpoint or until the run unit ends. (In any case the debugger stops at the first statement in the main function.)

Run to line

Runs all statements up to (but not including) the statement on a specified line.

Step into

Runs the next EGL statement and pauses.

The following list indicates what happens if you issue the command **step into** for a particular statement type:

call

Stops at the first statement of a called program if the called program runs in the EGL debugger. Stops at the next statement in the current program if the called program runs outside of the EGL debugger.

The EGL debugger searches for the receiving program in every project in the workbench.

converse

Waits for user input. That input causes processing to stop at the next running statement, which may be in a validator function.

forward

If the code forwards to a page handler, the debugger waits for user input and stops at the next running statement, which may be in a validator function.

If the code forwards to a program, the debugger stops at the first statement in that program.

function invocation

Stops at the first statement in the function.

sysLib.java and related functions

Stops at the next Java statement, so you can debug the Java code that is made available by the Java access functions.

show, transfer

Stops at the first statement of the program that receives control. The target program is EGL source that runs in the EGL debugger and is not EGL-generated code.

After either a **show** statement or a **transfer** statement of the form *transfer to a transaction*, the behavior of the EGL debugger depends on the debugger mode:

- In Java mode, the EGL debugger switches to the build descriptor for the new program or (if no such build descriptor is in use) prompts the user for a new build descriptor. The new program can have a different set of properties from the program that ran previously.
- In COBOL mode, the build descriptor for the previous program remains in use, and the new program cannot have a different set of properties.

The EGL debugger searches for the receiving program in every project in the workbench.

Step over

Runs the next EGL statement and pauses, but does not stop within functions that are invoked from the current function.

The following list indicates what happens if you issue the command **step over** for a particular statement type:

converse

Waits for user input, then skips any validation function (unless a breakpoint is in effect). Stops at the statement that follows the **converse** statement.

forward

If the code forwards to a page handler, the debugger waits for user input and stops at the next running statement, but not in a validator function, unless a breakpoint is in effect.

If the code forwards to a program, the debugger stops at the first statement in that program.

show, transfer

Stops at the first statement of the program that receives control. The target program is EGL source that runs in the EGL debugger and is not EGL-generated code.

After either a **show** statement or a **transfer** statement of the form *transfer to a transaction*, the behavior of the EGL debugger depends on the debugger mode:

- In Java mode, the EGL debugger switches to the build descriptor for the new program or (if no such build descriptor is in use) prompts the user for a new build descriptor. The new program can have a different set of properties from the program that ran previously.
- In COBOL mode, the build descriptor for the previous program remains in use, and the new program cannot have a different set of properties.

The EGL debugger searches for the receiving program in every project in the workbench.

Step return

Runs the statements needed to return to an invoking program or function; then, pauses at the statement that receives control in that program or function.

An exception is in effect if you issue the command **step return** in a validator function. In that case, the behavior is identical to that of a **step into** command, which primarily means that the EGL debugger runs the next statement and pauses.

The EGL debugger treats the following EGL statements as if they were null operators:

- **sysLib.audit**
- **sysLib.purge**
- **sysLib.startTransaction**

You can add a breakpoint at these statements, for example, but a **step into** command merely continues to the subsequent statement, with no other effect.

Finally, if you issue the command **step into** or **step over** for a statement that is the last one running in the function (and if that statement is not **return**, **exit program**, or **exit stack**), processing pauses in the function itself so that you can review variables that are local to the function. To continue the debug session in this case, issue another command.

Use of build descriptors

A build descriptor helps to determine aspects of the debugging environment. The EGL debugger selects the build descriptor in accordance with the following rules:

- If you specified a debug build descriptor for your program or page handler, the EGL debugger uses that build descriptor. For details on how to establish the debug build descriptor, see *Setting the default build descriptors*.
- If you did not specify a debug build descriptor, the EGL debugger prompts you to select from a list of your build descriptors or to accept the value **None**. If you accept the value **None**, the EGL debugger constructs a build descriptor for use during the debugging session; and a preference determines whether VisualAge Generator compatibility is in effect.
- If you specified either **None** or a build descriptor that lacks some of the required database-connection information, the EGL debugger gets the connection information by reviewing your preferences. For details on how to set those preferences, see *Setting preferences for SQL database connections*.

If you are debugging a program that is intended for use in a text or batch application in a Java environment, and if that program issues a **transfer** statement that switches control to a program that is also intended for use in a different run unit in a Java environment, the EGL debugger uses a build descriptor that is assigned to the receiving program. The choice of build descriptor is based on the rules described earlier.

If you are debugging a program that is called by another program, the EGL debugger uses the build descriptor that is assigned to the called program. The choice of build descriptor is based on the rules described above, except that if you do not specify a build descriptor, the debugger does not prompt you for a build descriptor when the called program is invoked; instead, the build descriptor for the calling program remains in use.

Note: You must use a different build descriptor for the caller and the called program if one of those programs (but not both) takes advantage of

VisualAge Generator compatibility. The generation-time status of VisualAge compatibility is determined by the value of build descriptor option **VAGCompatibility**.

A build descriptor or resource association part that you use for debugging code may be different from the one that you use for generating code. For example, if you intend to access a VSAM file from a program that is written for a COBOL environment, you are likely to reference a resource association part in the build descriptor. The resource association part must refer to the run-time target system and must refer to a file type that is appropriate for the target system. The difference between the two situations is as follows:

- At generation time, the resource association part indicates the file's system name that is used in the target environment
- At debug time, the system name must reflect another naming convention, as appropriate when you access a remote VSAM file from an EGL-generated Java program on Windows 2000/NT/XP; for details on that naming convention, see *VSAM support*

SQL-database access

To determine the user ID and password to use for accessing an SQL database, the EGL debugger considers the following sources in order until the information is found or every source is considered:

1. The build descriptor used at debug time; specifically, the build descriptor options `sqlID` and `sqlPassword`.
2. The SQL preferences page, as described in *Setting preferences for SQL database connections*; at that page, you also specify other connection information.
3. An interactive dialog that is displayed at connection time. Such a dialog is displayed only if you select the checkbox **Prompt for SQL user ID and password when needed**.

call statement

As noted earlier, the EGL debugger responds to a **transfer** or **show** statement by interpreting EGL source code. The EGL debugger responds to a **call** statement, however, by reviewing the linkage options part specified in the build descriptor, if any. If the referenced linkage options part includes a **callLink** element for the call, the result is as follows:

- If the **callLink** property **remoteComType** is set to **DEBUG**, the EGL debugger interprets EGL source code. The debugger finds the source by referencing the **callLink** properties **package** and **location**.
- If the **callLink** property **remoteComType** is not set to **DEBUG**, the debugger invokes EGL-generated code and uses the information in the linkage options part as if the debugger were running an EGL-generated Java program, even if the debugger is running in COBOL mode.

In the absence of linkage information, the EGL debugger responds to a **call** statement by interpreting EGL source code. Linkage information is unavailable in these cases:

- No build descriptor is used; or
- A build descriptor is used, but no linkage options part is specified in that build descriptor; or

- A linkage options part is specified in the build descriptor, but the referenced part does not have a **callLink** element that references the called program.

If the debugger runs EGL source code, you can run statements in that program by issuing the **step into** command from the caller. If the debugger calls generated code, however, the debugger runs the entire program; the **step into** command works like the **step over** command.

System type used at debug time

A value for system type is available in `sysVar.systemType`. Also, a second value is available in `sysLib.getVAGSysType` if you requested development-time compatibility with VisualAge Generator).

The value in **sysLib.systemType** is the same as the value of the build descriptor option **system**, except that the value is **DEBUG** in either of two cases:

- You select the preference **Set systemType to DEBUG**, as mentioned in *Setting preferences for the EGL debugger*; or
- You specified **NONE** as the build descriptor to use during the debugging session, regardless of the value of that preference.

The system function **sysLib.getVAGSysType** returns the VisualAge Generator equivalent of the value in **sysLib.systemType**; for details, see the table in *sysLib.getVAGSysType*.

EGL debugger port

The EGL debugger uses a port to establish communication with the Eclipse workbench. The default port number is 8345. If another application is using that port or if that port is blocked by a firewall, set a different value as described in *Setting preferences for the EGL debugger*.

If a value other than 8345 is specified as the EGL debugger port and if an EGL program will be debugged on the J2EE server, you must edit the server configuration:

1. Go to the Environment tab, System Properties section
2. Click Add
3. For Name, type `com.ibm.debug.egl.port`
4. For Value, type the port number

Invoking the EGL debugger from generated code

You can invoke the EGL debugger from an EGL-generated Java program or wrapper so you can use the EGL debugger when you work on a partly deployed application. The program needs a call statement that you associate with a linkage options part, **callLink** element. Similarly, you must associate the wrapper with a **callLink** element. In either case, the element must specify property `removeComType` as **DEBUG**.

Different rules apply, depending on whether or not the program to be debugged runs in J2EE:

- When the called program does not run in J2EE, its caller may be running anywhere, including a remote system.

Before the call occurs, you must start a listener program that runs in Eclipse. A listener is started using an EGL Listener launch configuration that has only one configurable setting, a port number. The default port number is 8346.

To specify a different port number, do as follows:

1. At the Run menu, click Debug
2. When the Debug dialog is displayed, select EGL Listener
3. Click New

You must specify a port if multiple EGL Listeners run at the same time, because each EGL Listener requires its own port. You also must specify a port if another application is using port 8346 or if a firewall prevents use of port 8346.

The listener port is not the same as the EGL debugger port, which is specified as an EGL preference.

- When the program to be debugged is to be run in J2EE, it must run in the same J2EE server as its caller. The EGL Debugger jars must have been added to the server, and the server must be running in debug mode.

Recommendations

As you prepare to work with the EGL debugger, consider these recommendations (most of which assume that `sysVar.systemType` is set to `DEBUG` when you are debugging the code):

- If you are retrieving a date from a database but expect the runtime code to retrieve that date in a format other than the ISO format, write a function to convert the date, but invoke the function only when the system type is `DEBUG`. The ISO format is `yyyy-mm-dd`, which is the only one available to the debugger.
- To specify external Java classes for use when the debugger runs, modify the class path, as described in *Setting preferences for the EGL debugger*. You might need extra classes, for example, to support MQSeries, JDBC drivers, or Java access functions.
- When you are debugging a page-handler function that was invoked by JSF (rather than by another EGL function), use Run to leave the function rather than Step Over, Step Into, or Step Return. Using any of the three Step commands takes you to the generated Java code of the page handler, which is not useful when you are debugging EGL. If you use one of the Step commands, use Run to leave the generated Java code and display the Web page in a browser.
- If you are using the SQL option `WITH HOLD` (or the EGL equivalent), you need to know that the option `WITH HOLD` is unavailable for EGL-generated Java or in the EGL debugger. You may be able to work around the limitation, in part by placing commit statements inside a conditional statement that is invoked only at run time, as in the following example:

```
if (systemType not debug)
    sysLib.commit();
end
```

If EGL programs are debugged on the J2EE server or by an EGL Listener, the server or EGL Listener must be configured to indicate the number for the EGL debugger port:

- To configure a J2EE server, edit the server configuration--
 1. Go to the Environment tab, System Properties section
 2. Click **Add**
 3. For **Name**, type `com.ibm.debug.egl.port`
 4. For **Value**, type the new port number

- To configure an EGL Listener, edit the EGL Listener launch configuration--
 1. Go to the Arguments tab
 2. In the **VM Arguments** field, type this:

```
-Dcom.ibm.debug.egl.port=portNumber
```

portNumber

The new port number

Related concepts

"Compatibility with VisualAge Generator" on page 276

"Java access functions" on page 556

"VSAM support" on page 202

Related tasks

"Setting preferences for SQL database connections" on page 188

"Setting preferences for the EGL debugger"

"Setting the default build descriptors" on page 237

Related reference

"remoteComType in callLink element" on page 455

"sqlDB" on page 262

"sqlID" on page 263

"sqlJNDIName" on page 264

"sqlPassword" on page 264

"sysLib.getVAGSysType" on page 613

"sysVar.systemType" on page 628

Setting preferences for the EGL debugger

To set preferences for the EGL debugger, follow these steps:

1. Click **Window > Preferences**.
2. When a list is displayed, expand **EGL** and click **Debug**.
3. Clear or select the check box labeled **Prompt for SQL user ID and password when needed**.

For details on your choice, see *EGL debugger*.

4. Clear or select the check box labeled **Set systemType to DEBUG**.

For details on your choice, see *EGL debugger*.

5. Set the initial values for sysVar.terminalID, sysVar.sessionID, and sysVar.userID. If you do not specify values, each defaults to your user ID on Windows 2000/NT/XP or Linux.
6. Set the EGL Debugger Port value. The default is 8345.

7. To specify external Java classes for use when the debugger runs, modify the class path. You might need extra classes, for example, to support MQSeries, JDBC drivers, or Java access functions.

The class path additions are not visible to the WebSphere Application Server test environment; but you can add to that environment's classpath by working on the Environment tab of the server configuration.

Use the buttons to the right of the Class Path Order box:

- To add a project, JAR file, directory, or variable, click the appropriate button: **Add Project**, **Add JARs**, **Add Directory**, or **Add Variable**.
- To remove an entry, select it and click **Remove**.

- To move an entry in a list of two or more entries, select the entry and click **Move Up** or **Move Down**.
8. To restore the default settings, click **Restore Defaults**.
 9. To save your changes, click **Apply** or (if you are finished setting preferences) click **OK**.

Related concepts

"Compatibility with VisualAge Generator" on page 276
 "EGL debugger" on page 107

Related tasks

"Setting preferences for SQL database connections" on page 188

Related reference

"sysVar.sessionID" on page 627
 "sysVar.terminalID" on page 629
 "sysVar.userID" on page 630

Starting a non-J2EE program in the EGL debugger

To start debugging an EGL text program or non-J2EE basic program in an EGL debugging session, a launch configuration is required. A launch configuration defines a program's file location and specifies how the program should be launched. You can let the EGL application create the launch configuration (implicit creation), or you can create one yourself (see *Creating a launch configuration in the EGL debugger*).

To launch a program using an implicitly created launch configuration, do as follows:

1. In the Project Navigator view, right-click the EGL source file you want to launch. Alternatively, if the EGL source file is open in the EGL editor, you can right-click on the program in the Outline view.
2. A context menu displays.
3. Click **Debug EGL Program**. A launch configuration is created, and the program is launched in the EGL debugger.

To view the implicitly created launch configuration, do as follows:

1. Click the arrow next to the Debug button on the toolbar. A context menu displays.
2. Click **Debug**. The Debug dialog displays. The name of the launch configuration is displayed in the Name field. Implicitly created launch configurations are named according to the project and source file names.

Note: You can also display the Debug dialog by clicking **Debug** from the Run menu.

Related concepts

"EGL debugger" on page 107

Related tasks

"Creating a launch configuration in the EGL debugger" on page 116
 "Stepping through a program in the EGL debugger" on page 118
 "Using breakpoints in the EGL debugger" on page 118
 "Viewing variables in the EGL debugger" on page 119

Creating a launch configuration in the EGL debugger

To start debugging an EGL text program or non-J2EE basic program in an EGL debugging session, a launch configuration is required. A launch configuration defines how a program should be launched. You can create a launch configuration (explicit creation), or you can let the EGL application create one for you (see *Starting a non-J2EE program in the EGL debugger*).

To start a program using an explicitly created launch configuration, do as follows:

1. Click the arrow next to the Debug button on the toolbar, then click **Debug**, or select **Debug** from the Run menu.
2. The Debug dialog displays.
3. Click **EGL Program** in the Configurations list, then click **New**.
4. If you did not have an EGL source file highlighted in the Project Navigator view, the launch configuration is named *New_configuration*. If you had an EGL source file highlighted in the Project Navigator view, the launch configuration has the same name as the EGL source file. If you want to change the name of the launch configuration, type the new name in the Name field.
5. If the name in the Project field of the Load tab is not correct, click **Browse**. A list of projects displays. Click a project, then click **OK**.
6. If the name in the EGL program source file field is not correct or the field is empty, click **Search**. A list of EGL source files displays. Click a source file, then click **OK**.
7. If you made changes to any of the fields on the Debug dialog, click **Apply** to save the launch configuration settings.
8. Click **Debug** to launch the program in the EGL debugger.

Note: If you have not yet used **Apply** to save the launch configuration settings, clicking **Revert** will remove all changes that you have made.

Related concepts

"EGL debugger" on page 107

Related tasks

"Starting a non-J2EE program in the EGL debugger" on page 115

"Stepping through a program in the EGL debugger" on page 118

"Using breakpoints in the EGL debugger" on page 118

"Viewing variables in the EGL debugger" on page 119

Preparing a server for EGL Web debugging

To debug EGL Web programs that run in the WebSphere Application Server, you must prepare the server for debugging. The preparation step must be done once per server and does not need to be done again, even if the workbench is shut down. To prepare a server for debugging, do as follows:

1. Make sure the server is stopped.
2. In the Server view, right-click on the server. A context menu displays.
3. Select **Add EGL Debugger Jar Files**.
4. If you want to debug the generated Java instead of EGL, right-click on the server again and select **Remove EGL Debugger Jar Files**.

Note: The server cannot be running when the EGL debugger jar files are added or removed.

Related concepts

"EGL debugger" on page 107

Related tasks

"Starting an EGL Web debugging session"

"Starting a server for EGL Web debugging"

"Stepping through a program in the EGL debugger" on page 118

"Using breakpoints in the EGL debugger" on page 118

"Viewing variables in the EGL debugger" on page 119

Starting a server for EGL Web debugging

To debug EGL Web programs, the server must be prepared for EGL Web debugging (see *Preparing a server for EGL Web debugging*). To start a server for EGL Web debugging, do as follows:

1. In the Server view, right-click on the server. A context menu displays.
2. Select **Debug**. Once the server is started, the status message displayed in the Server view is *Started in debug mode*.

Related concepts

"EGL debugger" on page 107

Related tasks

"Preparing a server for EGL Web debugging" on page 116

"Starting an EGL Web debugging session"

"Stepping through a program in the EGL debugger" on page 118

"Using breakpoints in the EGL debugger" on page 118

"Viewing variables in the EGL debugger" on page 119

Starting an EGL Web debugging session

To debug EGL Web programs, the server must be prepared for EGL debugging (see *Preparing a server for EGL debugging*) and started in debug mode (see *Starting a server for EGL debugging*). To start an EGL Web debugging session, do as follows:

1. In the Project Navigator view, right-click the JSP file you want to launch. A context menu displays.
2. Click **Debug on Server**.
3. If you have not set a server as a project default, the Select Server dialog displays. Click **Use an existing server**, then select a server that is prepared for EGL Web debugging.
4. An EGL Web debugging session starts.

Related concepts

"EGL debugger" on page 107

Related tasks

"Preparing a server for EGL Web debugging" on page 116

"Starting a server for EGL Web debugging"

"Stepping through a program in the EGL debugger" on page 118

"Using breakpoints in the EGL debugger" on page 118

"Viewing variables in the EGL debugger" on page 119

Using breakpoints in the EGL debugger

Breakpoints are used to pause execution of a program. You can manage breakpoints inside or outside of an EGL debugging session. Keep the following in mind when working with breakpoints:

- A blue marker in the left margin of the Source view indicates that a breakpoint is set and enabled.
- A white marker in the left margin of the Source view indicates that a breakpoint is set but disabled.
- The absence of a marker in the left margin indicates that a breakpoint is not set.

Add or remove a breakpoint

Add or remove a single breakpoint in an EGL source file by doing one of the following:

- Position the cursor at the breakpoint line in the left margin of the Source view and double-click.
- Position the cursor at the breakpoint line in the left margin of the Source view and right-click. A context menu displays. Click the appropriate menu item.

Disable or enable a breakpoint

Disable or enable a single breakpoint in an EGL source file by doing the following:

1. In the Breakpoint view, right-click on the breakpoint. A context menu displays.
2. Click the appropriate menu item.

Remove all breakpoints

Remove all breakpoints from an EGL source file by doing the following:

1. Right-click on any of the breakpoints displayed in the Breakpoints view. A context menu displays.
2. Click **Remove All**.

Related concepts

“EGL debugger” on page 107

Related tasks

“Creating a launch configuration in the EGL debugger” on page 116

“Starting a non-J2EE program in the EGL debugger” on page 115

“Stepping through a program in the EGL debugger”

“Viewing variables in the EGL debugger” on page 119

Stepping through a program in the EGL debugger

As explained in *EGL debugger*, the EGL debugger provides the following commands to control execution of a program during a debugging session:

Resume

Runs the code until the next breakpoint or until the end of the program.

Run to Line

Allows you to select an executable line in the Source view and run the code to that line.

Step Into

Runs the next EGL statement and pauses. The program stops at the first statement of a called function.

Step Over

Runs the next EGL statement and pauses, but does not stop within functions that are invoked from the current function.

Step Return

Returns to an invoking program or function.

With the exception of Run to Line, each of the commands can be accessed in the following ways:

- Click the appropriate button on the toolbar of the Debug view; or
- Click the appropriate menu item on the Run menu; or
- Right-click a highlighted thread in the Debug view, then click the appropriate menu item.

To use Run to Line, do as follows when the program is paused:

1. Position the cursor in the left margin of the Source view at an executable line, then right-click. A context menu displays.
2. Click **Run to Line**.

When using Run to Line, keep in mind the following:

- The operation is not available from the Debug view or the Run menu
- Run to Line stops at enabled breakpoints

Related concepts

"EGL debugger" on page 107

Related tasks

"Creating a launch configuration in the EGL debugger" on page 116

"Starting a non-J2EE program in the EGL debugger" on page 115

"Using breakpoints in the EGL debugger" on page 118

"Viewing variables in the EGL debugger"

Viewing variables in the EGL debugger

Whenever a program is paused, you can view the current values of the program's variables.

To view a program's variables, do as follows:

1. In the Variables view, expand the parts in the navigator to see their variables.
2. To display the variables' types, click the **Show Type Names** button on the toolbar.
3. To display the details of a variable in a separate pane, click on the variable, then click the **Show Detail** button on the toolbar.

Related concepts

"EGL debugger" on page 107

Related tasks

"Creating a launch configuration in the EGL debugger" on page 116

"Starting a non-J2EE program in the EGL debugger" on page 115

"Stepping through a program in the EGL debugger" on page 118
"Using breakpoints in the EGL debugger" on page 118

Build

When you are working in an EGL or EGL Web project, the word *build* does not (in general) refer to code generation.

The following menu options have a distinct meaning:

Build project

Builds a subset of the project--

1. Validates all EGL files that have changed in the project since the last build
2. Generates page handlers that were changed since the prior page-handler generation
3. Compiles any Java source that changed since the last compile

The menu option **Build Project** is available only if have not set the Workbench preference **Perform build automatically on resource modification**. If you *have* set that preference, the actions described earlier occur whenever you save an EGL file.

Build all

Conducts the same actions as **Build project**, but for every open project in the workspace.

Rebuild project

Acts as follows--

1. Validates all the EGL files in the project
2. Generates all page handlers in the project
3. Compiles any Java source that changed since the last compile

Rebuild all

Conducts the same actions as **Rebuild project**, but for every open project in the workspace.

When you generate code into a project (as is possible only for Java output), a Java compile occurs locally in the following situations:

- When you build or rebuild the project; or
- When you generate the source files; but only if you checked the workbench preference **Perform build automatically on resource modification**.

When you generate code into a directory, EGL optionally creates a *build plan*, which is an XML file that includes the following details:

- The location of any files that will be transferred to another machine;
- Other information needed for the transfer, which occurs by way of TCP/IP; and
- A Java compile statement (if appropriate).

In the case of COBOL generation for iSeries, the iSeries build server calls a build script (FDAPREP) that resides in the iSeries library QEGL/QREXSRC. That build script specifies how to prepare output, and a system administrator must customize that build script, as described in the document *EGL Server Guide for iSeries*, on your installation CD.

Preparation of generated output on a remote platform requires that a build server be running on that platform.

You may wish to create a build plan and to invoke that plan at a later time. For details, see *Invoking a build plan after generation*.

Related concepts

"Build descriptor part" on page 234

"Build plan" on page 392

"Build server" on page 275

"Development process" on page 1

Related tasks

"Creating a build file" on page 88

"Invoking a build plan after generation" on page 123

Related reference

"Build descriptor options" on page 237

"Build scripts delivered with EGL" on page 271

Building EGL output

To build EGL output , complete the following steps:

Steps to build output

Java programs or wrappers	COBOL programs
Generate Java source code into a project or directory: <ul style="list-style-type: none">• If you generate into a project (as is recommended) and if your Eclipse preferences are set to build automatically on resource modification, the workbench prepares the output.• If you generate into a directory, the generator's distributed build function prepares the output.	Generate into a directory. This step produces COBOL source code. COBOL preparation occurs on a remote host system.
Prepare the generated output. This step is done automatically unless you set build descriptor option buildPlan or prep to no.	Prepare the generated output. This step compiles and links the source code to produce executable code.

Related concepts

"Build" on page 121

"Development process" on page 1

"EGL projects, packages, and files" on page 7

"Generated output" on page 386

"Generation" on page 123

Related tasks

"Creating an EGL source file" on page 85

"Generating into a directory" on page 220

"Generating into a project" on page 214

"Setting up the J2EE run-time environment for EGL-generated code" on page 206

Related reference

"Generated output (reference)" on page 387

Invoking a build plan after generation

You may wish to create a build plan and to invoke that plan at a later time. This case might occur, for example, if a network failure prevents you from preparing code on a remote machine at generation time.

To invoke a build plan in this case, complete the following steps:

1. Make sure that `eglbatchgen.jar` is in your Java classpath, as happens automatically on the machine where you install EGL. The jar file is located in the `\bin` subdirectory of the WebSphere Studio installation directory. For example:
`c:\myStudio\bin\eglbatchgen.jar`
2. Similarly, make sure that your `PATH` variable includes the `\bin` subdirectory of the WebSphere Studio installation directory. For example:
`c:\myStudio\bin`
3. From a command line, enter the following command:
`java com.ibm.etools.egl.distributedbuild.BuildPlanLauncher bp`
bp The fully qualified path of the build plan file. For details on the name of the generated file, see *Generated output (reference)*.

Related concepts

“Build plan” on page 392
“Generation”

Related tasks

“Building EGL output” on page 122

Related reference

“Build descriptor options” on page 237
“Generated output (reference)” on page 387

Generation

Generation is the creation of output from EGL parts.

You can generate output in the Workbench, from the Workbench batch interface, or from the EGL Software Development Kit (EGL SDK). The EGL SDK provides a batch interface for file-based generation that is independent of the Workbench.

Generation uses the saved versions of your EGL files.

Related concepts

“Development process” on page 1
“Generated output” on page 386

Related tasks

“Generating for COBOL” on page 132
“Generating Java wrappers” on page 131

Related reference

“Build descriptor options” on page 237
“Generated output (reference)” on page 387

Generation in the workbench

To generate output in the Workbench, do the following:

- Load parts to generate, along with any parts that are referenced during generation.
- Select parts to generate. If you invoke the generation process for a file, folder, package, or project, EGL creates output for every *primary part* (every program, page handler, form group, data table, or library) that is in the container you selected.
- Initiate generation.
- Monitor progress.

To make generation easier, it is recommended that you first select a default build descriptor from these types:

- Debug build descriptor (as appropriate when you are using the EGL debugger)
- Target system build descriptor (as used for generating parts and deploying them in a run-time environment)

For details on how to select a default build descriptor, see *Setting the default build descriptors*.

You can identify each of two build descriptors (debug and target system) in these ways:

- As a property at the file, folder, package, and project level
- As a Workbench preference

A lower-level build descriptor of a particular kind takes precedence over any higher-level build descriptor of the same kind. For example, a target system build descriptor that was assigned to the current package takes precedence over a target system build descriptor that was assigned to the project. However, a master build descriptor takes precedence over all others, as described in *Build descriptor part*.

The following precedence rules are in effect for every file that contains a primary part:

- A property that is specific to the file takes precedence over all others
- The related folder property takes precedence over the package, project, or Workbench property
- The related package property takes precedence over the project or Workbench property
- The related project property takes precedence over the Workbench property
- The Workbench property is used if no others are specified

For details on initiating generation, see *Generating in the workbench*.

Related concepts

“Build descriptor part” on page 234

“Development process” on page 1

“Generated output” on page 386

Related tasks

“Generating in the workbench” on page 125

“Setting the default build descriptors” on page 237

Related reference

“Generated output (reference)” on page 387

Generating in the workbench

Generating in the workbench is accomplished with either the generation wizard or the generation menu item. When you select the generation menu item, EGL uses the default build descriptor. If you have not selected a default build descriptor, use the generation wizard. For details on selecting the default build descriptor, see *Setting the default build descriptors*.

To generate in the workbench by invoking the generation wizard, do as follows:

1. Right-click on a resource name (project, folder, or file) in the Project Navigator.
2. Select the **Generate With Wizard...** option.

The generation wizard includes four pages:

1. The first page displays a list of parts to generate based on the selection that the user made to start the generation process. You must select at least one part from the list before you can continue to the next page. The interface provides buttons to let you select or deselect all of the parts in the list.
2. The second page lets you choose a build descriptor or build descriptors to be used to generate the parts selected on the first page. You have two options:
 - Choose from a drop-down list of all the build descriptors that are in the workspace, and use that build descriptor to generate all of the parts.
 - Select a build descriptor for each of the parts selected on the first page. You use a table to select the build descriptors for each part. The first column of the table displays the part names; the second column displays a drop-down list of build descriptors for each part.
3. The third page lets you set user IDs and passwords for both the destination machine and the SQL database used in the generation process, if IDs and passwords are necessary. These user IDs and passwords, if any, override the ones listed in the specified build descriptor for each part being generated. You may want to set user IDs and passwords on this page rather than the build descriptor to avoid keeping sensitive information in persistent storage.
4. The fourth page lets you create a command file that you can use for generating an EGL program outside of the workbench. You can reference the command file in the workbench batch interface (by using the command EGLCMD) or in the EGL SDK (by using the command EGLSDK).

To create a command file, do as follows:

- a. Select the Create a Command File checkbox
 - b. Specify the name of the output file, either by typing the fully qualified path or by clicking **Browse** and using the standard Windows procedure for selecting a file
 - c. Select or clear the Automatically Insert EGL Path (eglpth) check box to specify whether you want to include the EGL project path in the command file, as the initial value for eglpth; for details, see *EGL command file*
 - d. Select a radio button to indicate whether to avoid generating output when creating the command file
5. Click **Finish**.

To generate in the workbench using the generation menu item, do one of the following sets of steps:

1. Select one or more resource names (project, folder, or file) in the Project Navigator. To select multiple resource names, hold down the **Ctrl** key as you click.
2. Right-click, then select the **Generate** menu option.

or

1. Double-click on a resource name (project, folder, or file) in the Project Navigator. The file opens in the EGL editor.
2. Right-click inside the editor pane, then select **Generate**.

Related concepts

"Generation in the workbench" on page 124

Related tasks

"Setting the default build descriptors" on page 237

Related reference

"EGLCMD" on page 407

"EGLSDK" on page 410

"Generated output (reference)" on page 387

Generation Results view

The Generation Results view shows you code-preparation messages that are the result of generation performed in the workbench. These messages may be errors, warnings, or informational messages. This view is available only when generating from the Workbench. The format is as follows:

msgid message

msgid

Is the message identifier. For example, IWN.VAL.4610.e is the message ID for Enterprise Developer validation error number 4610.

message

Is the text of the message.

Generation results are displayed in the view by primary part (program, page handler, form group, data table, library), with a different tab for each part. The results can be a combination of validation results and generation results.

You can open this view at any time, but it displays data only after you generate output.

Related concepts

"Development process" on page 1

"Generated output" on page 386

"Generation" on page 123

Related reference

"Generated output (reference)" on page 387

Generation from the workbench batch interface

The workbench batch interface is a feature that lets you generate EGL output from a batch environment that can access the workbench. The workbench does not need to be running. The generation of EGL code can access only projects and EGL parts that were previously loaded into a workspace.

To invoke the interface, use the batch command EGLCMD, which references both a workspace and an EGL command file.

Related concepts

“Development process” on page 1

“Generated output” on page 386

Related tasks

“Generating from the workbench batch interface”

Related reference

“EGLCMD” on page 407

Generating from the workbench batch interface

To generate from the Workbench batch interface, do as follows:

1. Make sure that your Java classpath provides access to these jar files--
 - startup.jar, which is in the WebSphere Studio installation directory (like c:\myStudio), in subdirectory eclipse
 - eglutil.jar, which is in the WebSphere Studio installation directory (like c:\myStudio), in the following subdirectory:
wstools\eclipse\plugins\
com.ibm.etools.egl.utilities_*version*\runtime
version
The installed version of the plugin; for example, 5.0.1
2. Make sure that a workspace contains the projects and EGL parts that are required for generation.
3. Develop an EGL command file.
4. Invoke the command EGLCMD, possibly in a larger batch job that generates, runs, and tests the code. You specify the workspace of interest when invoking EGLCMD.

Related concepts

“Generation from the workbench batch interface” on page 126

Related reference

“EGLCMD” on page 407

Generation from the EGL SDK

The EGL software development kit (SDK) is a feature that lets you generate output in a batch environment, even when you lack access to the following aspects of WebSphere Studio:

- The graphical user interface
- The details on how projects are organized

You can use the EGL SDK to trigger generation from a software configuration management (SCM) tool such as Rational® ClearCase®, perhaps as part of a batch job that is run after normal working hours.

To invoke the EGL SDK, you use the command EGLSDK in a batch file or at a command prompt. The command invocation itself can take either of two forms:

- It can specify an EGL file and build descriptor. In this case, if you want to cause multiple generations you write multiple commands.

- Alternatively, the invocation can reference an EGL command file that includes the information necessary to cause one or more generations.

However you organize your work, you can specify a value for *eglp*path, which is a list of directories that are searched when the EGL SDK uses an import statement to resolve a part reference. Also, you must specify the build descriptor option **genDirectory** instead of **genProject**.

The prerequisites and process for using EGLSDK are described in *Generating from the EGL SDK*. For details on the command invocation, see *EGLSDK*.

Related concepts

“Development process” on page 1

“Generated output” on page 386

Related tasks

“Generating from the EGL SDK”

Related reference

“genDirectory” on page 251

“EGLCMD” on page 407

“EGL build path and eglpath” on page 409

“EGLSDK” on page 410

Generating from the EGL SDK

To generate from the EGL SDK, do as follows:

1. Make sure that Java 1.3.1 (or a higher level) is on the machine where you will generate code. An appropriate level of Java code is installed automatically on the machine where you install EGL. The Java levels on the generation and target machines must be compatible.

2. Make sure that eglbatchgen.jar is in your Java classpath, as happens automatically on the machine where you install EGL. The jar file is in the WebSphere Studio installation directory (like c:\myStudio), in the following subdirectory:

```
wstools\eclipse\plugins\
com.ibm.etools.egl.batchgeneration_
version
```

The installed version of the plugin; for example, 5.1.2

3. If you wish to enable COBOL generation for iSeries, make sure that the run-time jar file eglwdsc.jar is in your class path. The jar file is in the WebSphere Studio installation directory (like c:\myStudio), in the following subdirectory:

```
wstools\eclipse\plugins\
com.ibm.etools.egl.wdsc_
version
```

The installed version of the plugin; for example, 5.1.2

4. Make sure that the EGL SDK can access the EGL files that are required for generation
5. Optionally, develop an EGL command file
6. Invoke the command EGLSDK, possibly in a larger batch job that generates, runs, and tests the code

Related concepts

“Generation from the EGL SDK” on page 127

“EGL projects, packages, and files” on page 7

Related reference

“EGLCMD” on page 407

“EGL build path and eglpath” on page 409

“EGLSDK” on page 410

Generating for Java

Choosing options for Java generation

Build descriptor options are set in build descriptor parts. To choose build descriptor options for Java generation, start the EGL editor and edit the build descriptor part.

When you begin editing a build descriptor part from the GUI, the EGL editor contains a pane listing all EGL build descriptor options. To limit the display to options that are applicable to a program generated in Java, select a category from the Build option filter drop-down menu.

Select each option you want, and set its value. The value can be literal, symbolic, or a combination of literal and symbolic. You can define symbolic parameters in the EGL part editor; for details, see *Editing general build descriptor options*.

Two build descriptor options—**genDirectory** and **destDirectory**—let you use a symbolic parameter for the value or a portion of the value. For example, for the value of **genDirectory** you can specify `C:\genout\%EZEENV%`. Then if you generate for a Windows environment, the actual generation directory is `C:\genout\WIN`.

You do not need to specify all the options listed. If you do not specify a value for a build descriptor option, the default for the option is used when the option is applicable in the generation context.

If you have specified a master build descriptor, the option values in that build descriptor override the values in all other build descriptors. When you generate, the master and generation build descriptors can chain to other build descriptors as described in *Build descriptor part*.

Related concepts

“Build descriptor part” on page 234

Related tasks

“Editing general build descriptor options” on page 93

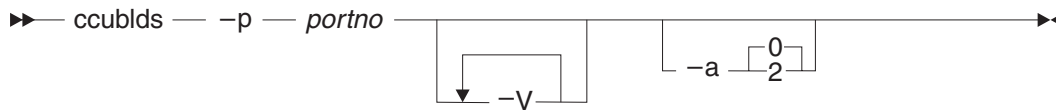
“Generating Java wrappers” on page 131

Related reference

“Build descriptor options” on page 237

Starting a build server on AIX, Linux, or Windows 2000/NT/XP

To start a remote build server on AIX®, Linux, or Windows 2000/NT/XP, enter the `ccublds` command in a Command Prompt window. The syntax is as follows:



where

- p Specifies the port number (*portno*) that the server listens to, to communicate with the clients.
- V Specifies the verbosity level of the server. You may specify this parameter up to three times (maximum verbosity).
- a Specifies the authentication mode:
 - 0 The server performs builds requested by any client. This mode is recommended only in an environment where security is not a concern.
 - 2 The server requires the client to provide a valid user ID and password before accepting a build. The user ID and password are first configured by the owner of the host machine where the build server runs. You do the configuration by using the Security Manager described below.

Setting the language of messages returned from the build server: The build server on Windows returns messages in any of the languages listed in the next table, and the default is English.

Language	Code
Brazilian Portugese	ptb
Chinese, simplified	chs
Chinese, traditional	cht
English, USA	enu
French	fra
German	deu
Italian	ita
Japanese	jpn
Korean	kor
Spanish	esp

To specify a language other than English, make sure that before you start the build server, the environment variable `CCU_CATALOG` is set to a non-English message catalog. The needed value is in the following format (on a single line):

```

installDir\wstools\eclipse\plugins
\com.ibm.etools.egl.distributedbuild\executables
\ccu.cat.xxx

```

installDir

Directory in which WebSphere Studio is installed

xxx

The language code being supported by the build server; one of the codes listed in the previous table

Security Manager: The Security Manager is a server program that the build server uses to authenticate clients that send build requests.

Setting the environment for the Security Manager: The Security Manager uses the following Windows environment variables:

CCUSEC_PORT

Sets the number of the port to which the Security Manager listens. The default value is 22825.

CCUSEC_CONFIG

Sets the path name of the file in which configuration data is saved. The default is C:\temp\ccuconfig.bin. If this file is not found, the Security Manager creates it.

CCU_TRACE

Initiates tracing of the Security Manager for diagnostics purposes, if this variable is set to *.

Starting the Security Manager: To start the Security Manager, issue the the following command:

```
java com.ibm.etools.egl.distributedbuild.security.CcuSecManager
```

Configuring the Security Manager: To configure the Security Manager, use the Configuration Tool, which has a graphical interface. You can run the tool by issuing the following command:

```
java com.ibm.etools.egl.distributedbuild.security.CCUconfig
```

When Configuration Tool is running, select the **Server Items** tab. Using the button 'Add...', To add the user that you want the build server to support, click the **Add ...** button. You must define a password for the user ID. You can define the following restrictions and privileges for the user:

- The locations, that is, the values of the -la parameter to ccubldc command, that this user can specify. Different locations are separated by semicolons.
- The name of the build script that this user can specify. (The EGL build plan only uses the javac command as a build script.)
- Whether or not this user can send build scripts from client, that is, use the -ft parameter of ccubldc command. (The EGL generator does not use the -ft parameter. You would specify this parameter if they were using the build for purposes other than preparing Java-generation outputs.)

These definitions are kept in persistent storage, in the file specified by CCUSEC_CONFIG, and are remembered across sessions.

Related concepts

"Build script" on page 271

"Build server" on page 275

Related tasks

"Syntax diagram" on page 506

Generating Java wrappers

You can generate Java wrapper classes when you generate the related program. For details on how to set up the build descriptor, see *Java wrapper*.

Related concepts

"Generation" on page 123

"Java wrapper" on page 395

Related tasks

- "Building EGL output" on page 122
- "Generating into a directory" on page 220
- "Generating into a project" on page 214

Related reference

- "Build descriptor options" on page 237
- "Java wrapper classes" on page 395
- "Output of Java wrapper generation" on page 389

Generating for COBOL

The steps required to create a COBOL load module are as follows:

1. Create one or more EGL source files and one or more EGL build files
2. Generate COBOL source code.
3. Prepare the COBOL code by compiling and linking it. This happens automatically unless you set build descriptor option **buildPlan** or **prep** to *no*.

The build descriptor option **distLibrary** specifies the iSeries library into which the code is placed.

Related concepts

- "COBOL program" on page 393
- "EGL projects, packages, and files" on page 7
- "Generated output" on page 386
- "Generation" on page 123

Related tasks

- "Building EGL output" on page 122

Related reference

- "Build scripts delivered with EGL" on page 271
- "destLibrary" on page 248
- "Generated output (reference)" on page 387
- "Output of COBOL generation" on page 388

Choosing options for COBOL generation

Build descriptor options are set in build descriptor parts. To choose build descriptor options for COBOL generation, start the EGL editor and edit the build descriptor part.

When you begin editing a build descriptor part from the GUI, the EGL editor contains a pane listing all EGL build descriptor options. To limit the display to options that are applicable to a particular kind of generated output, select a category from the Build option filter drop-down menu. The categories with the word *Basic* contain build descriptor options that are used most frequently, while the categories with the word *All* contain every option possible for the specified output. The word *iseriesc* refers to EGL-generated COBOL programs that run on iSeries.

Select each option you want and set its value. The value can be literal, symbolic, or a combination of literal and symbolic. You can define symbolic parameters in the EGL part editor, as described in *Editing general build descriptor options*.

Two build descriptor options—**genDirectory** and **destDirectory**—let you use a symbolic parameter for the value or a portion of the value. For example, for the value of **genDirectory** you can specify `C:\genout\%EZEENV%`. Then if you generate for a Windows environment, the actual generation directory is `C:\genout\WIN`.

You do not need to specify all the options listed. If you do not specify a value for a build descriptor option, the default for the option is used when the option is applicable in the generation context.

If you have specified a master build descriptor, the option values in that build descriptor override the values in all other build descriptors. When you generate, the master and generation build descriptors can chain to other build descriptors as described in *Build descriptor part*.

Related concepts

“Build descriptor part” on page 234

Related tasks

“Choosing options for Java generation” on page 129

“Editing general build descriptor options” on page 93

Related reference

“Build descriptor options” on page 237

Web applications

Web support

EGL provides support for Web-based applications in the following ways:

- You can develop a *page handler*, which is a logic part whose functions are each invoked by a specific user action at a Web page. Generation of this logic part also can provide a JavaServer Faces JSP for customization.
- You can provide functionality that is common to several Web pages, as when each page in an application displays a button that the user clicks to log out. The user's click in this case can invoke an EGL program, which acts as a common subroutine.
- You may want to retrieve an item of information (such as a stock price) or cause another action (such as building an output string to send by e-mail) when the functionality is useful across multiple applications. In this case, build an EGL Web service, which is a set of operations that can be invoked by many Internet-based clients. For an overview, see *EGL Web service*.
- Finally, when you work in the WebSphere Page Designer, you can customize JavaServer Faces JSPs and can affect page handlers, as described in *Page Designer Support for EGL*.

Related concepts

"EGL Web service" on page 139

"PageHandler part" on page 469

Related reference

"Page Designer support for EGL"

Page Designer support for EGL

When you work in the WebSphere Studio Page Designer, you can drag controls from the palette to a JSP, then use the Attributes view to set control-specific characteristics such as color and to set up *bindings*, which are relationships between controls and either data or logic. You also can do work that is specific to EGL:

- Create an EGL page handler when you make a new JavaServer Faces JSP. (The page-handler creation occurs automatically if you are working in an EGL Web project.)
- Create EGL data items and records and place them in an existing page handler.
- Bind JSP data controls to page-handler items.
- Bind buttons and hypertext links to event handlers.

Binding controls to data areas in the page handler

Most controls on the JSP have a one-to-one correspondence with data. A text box, for example, shows the content of the EGL item to which the text box is bound. An input text box also updates the EGL item if the user changes the data.

A more complex situation occurs when you are specifying a check box group, list box, radio button group, or combo box. In those cases, you need two different kinds of bindings:

- One is to bind the control to the text that you want to display to the user. An example is the text of an item in a list box.
- One is to bind the control to a page-handler data area that receives a value to indicate the user's choice. You might create a data item, for example, to receive the numeric index of a user-selected list-box item.

In the Attributes view, you can follow either of two procedures to bind the control to the text that the user sees:

- You can use **Add Text Item** to indicate that the control is associated with a single character string, which may be specified explicitly or by identifying a page-handler item
- You can use **Add Computed Items** to indicate that the control is associated with a list of character strings, which may be specified explicitly or by identifying a page-handler area such as a data table or an array of character items

Alternatively, you can bind a single-select control (combo box, single-select list box, or radio button group) to an array of character items by dragging the array from the Page Data view to the control.

To bind a control to a data area that will receive a value indicating the user's choice, you can work in either the Page Data view or the Attributes view. The procedure is the same as when you are binding any control, even a simple text box.

If the value can be only one of two alternatives, you can bind the control to an EGL item for which the item property **boolean** is set to *yes*. The control populates the item with one of two values:

- For a character item, the value is **Y** (for yes) or **N** (for no)
- For a numeric item, the value is **1** (for yes) or **0** (for no)

When a check box is displayed, the status (whether checked or not) is dependent on the value in the bound item.

For details on the properties that can be applied to data items in the page handler, see *Page item properties*.

Note: If you have experience with JavaServer Faces programming, you can manually bind controls to data areas in the page handler using the Page Designer Source view. Although EGL is not case sensitive, variable names in the JavaServer Faces source file must be the same case as the EGL variable declaration. Additionally, the case of the EGL variable must not be changed after it is bound to a control or a JavaServer Faces error will occur.

Binding controls to functions

After you drag a button or hypertext link to the page surface, you can bind that control to an existing event handler or to an event handler that the Page Designer creates:

- You can bind the control to an existing event handler in any of these ways--
 - By dragging the event handler from the Actions node in the Page Data view to the control
 - By opening the control in the Quick Edit view
 - By right-clicking on the control and selecting **Edit JFS Command Event**

- You can cause Page Designer to create a new event handler either when you open the control in the Quick Edit view or when you right-click on the control and select **Edit JFS Command Event**

If the Page Designer creates an event handler in the page handler and gives you access to that page-handler function, the name of the function is the tool-assigned button ID plus the string "Action". If the name is not unique to the page handler, the Page Designer appends a number to the function name.

Related concepts

"PageHandler part" on page 469

Related tasks

"Associating an EGL data item with a JSP" on page 138

"Associating an EGL primitive type with a JSP" on page 138

"Associating an EGL record with a JSP" on page 139

"Using the Quick Edit view"

Related reference

"PageHandler part in EGL source format" on page 472

"Page item properties" on page 53

Using the Quick Edit view

The Quick Edit view allows you to maintain EGL page handler code for JSP server events without opening the page handler file. To use the Quick Edit view, do as follows:

1. Open a JSP file in the Page Designer. If you do not have a file open, double-click on the JSP file in the Project Navigator. The JSP opens in the Page Designer. Click the **Design** tab to access the Design view.
2. Right-click in the Page Designer, then select **Quick Edit**. The Quick Edit view opens.
3. Follow these steps to maintain page handler functions for command components:
 - a. Select a command component in the JSP.
 - b. If the command component already has a page handler function associated with it, the function displays in the script editor (right pane) of the Quick Edit view. Any changes you make to the code are reflected in the page handler.
 - c. To create a page handler function for the selected command component, click **Command** in the event pane (left pane) of the Quick Edit view, then click in the script editor (right pane) of the Quick Edit view. The function displays. Type the page handler code for the function.
4. Follow these steps to maintain the onPageLoad function:
 - a. Click inside the JSP.
 - b. Click **onPageLoad** in the event pane (left pane) of the Quick Edit view.
 - c. The onPageLoad function displays in the script editor (right pane) of the Quick Edit view. Any changes you make to the code are reflected in the page handler.

Related concepts

"PageHandler part" on page 469

Related reference

“PageHandler part in EGL source format” on page 472

Associating an EGL primitive type with a JSP

To associate an EGL primitive type with a JSP, do as follows:

1. Open a JSP file in the Page Designer. If you do not have a JSP file open, double-click on the JSP file in the Project Navigator. The JSP opens in the Page Designer. Click the **Design** tab to access the Design view.
2. From the **Window** menu, select **Show View > Palette**.
3. In the Palette view, click the **EGL** drawer to display the EGL data object types.
4. Drag **Primitive Type** from the palette to the JSP. The Create a field dialog is displayed.
5. Type a name for the field, then enter a field type and length.
6. If the field is an array, enter the array properties.
7. Click **OK**. The new field is visible in the Page Data view.
8. Drag the field from the Page Data view to the JSP. The Insert Control dialog is displayed. (For arrays, the Insert List Control dialog is displayed.)
9. Select the type of button controls needed for the field. Specify the columns to display and how to display them. Click **Options** to change the properties of the button controls.
10. Click **Finish**.

Related reference

“Primitive types” on page 27

Associating an EGL data item with a JSP

To associate an EGL data item with a JSP, do as follows:

1. Open a JSP file in the Page Designer. If you do not have a JSP file open, double-click on the JSP file in the Project Navigator. The JSP opens in the Page Designer. Click the **Design** tab to access the Design view.
2. From the **Window** menu, select **Show View > Palette**.
3. In the Palette view, click the **EGL** drawer to display the EGL data object types.
4. Drag **Data item** from the palette to the JSP. The Select a Data Item dialog is displayed.
5. Select a data item from the list.
6. The name for the field defaults to the name of the data item; however, you can change the name of the field.
7. If the field is an array, enter the array properties.
8. Click **OK**. The new field is visible in the Page Data view.
9. Drag the field from the Page Data view to the JSP. The Insert Control dialog is displayed. (For arrays, the Insert List Control dialog is displayed.)
10. Select the type of control needed for the field. Specify the columns to display and how to display them. Click **Options** to change the properties of the button controls.
11. Click **Finish**.

Related concepts

“DataItem part” on page 284

Associating an EGL record with a JSP

To associate an EGL record with a JSP, do as follows:

1. Open a JSP file in the Page Designer. If you do not have a JSP file open, double-click on the JSP file in the Project Navigator. The JSP opens in the Page Designer. Click the **Design** tab to access the Design view.
2. From the **Window** menu, select **Show View > Palette**.
3. In the Palette view, click the **EGL** drawer to display the EGL data object types.
4. Drag **Record** from the palette to the JSP. The Select a Record dialog is displayed.
5. Select a record from the list.
6. The name for the field defaults to the name of the record; however, you can change the name of the field.
7. If the field is an array, enter the array properties.
8. Click **OK**. The new field is visible in the Page Data view.
9. Drag the field from the Page Data view to the JSP. The Insert Record dialog is displayed.
10. Select the type of control needed for the field. Specify the columns to display and how to display them. Click **Options** to change the properties of the button controls.
11. Click **Finish**.

Related concepts

"Record parts" on page 490

EGL Web service

A *Web service* is a set of operations that can be invoked by a program interface over the Internet. These operations are accessible to a variety of *Web service clients* that run on the same or different platforms. A client can be a JSP, servlet, or Java application; or an executable written in any of several languages such as C++, Perl, Visual Basic, or JavaScript™.

A Web service provides no user interface.

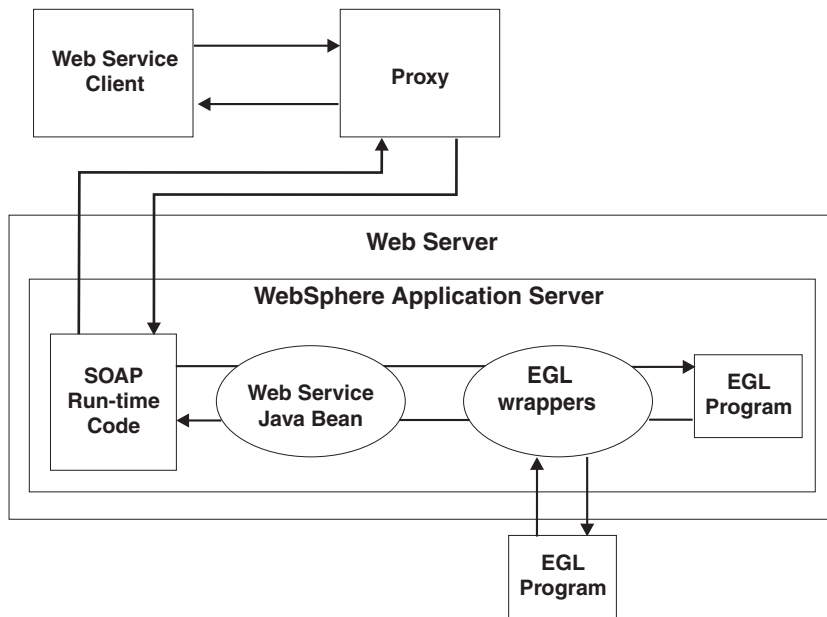
An *EGL Web service* is composed primarily of the following components:

- A *Web service Java bean*, which is the entry point to the Web service. This Java object interacts with Web service clients and invokes the operations as requested.
- One or more EGL-generated programs, which provide the operations. These programs may reside on different platforms.

Additional components are described in the next section.

Run-time events

The next diagram shows the relationship of objects at run time.



Run-time events are usually as follows:

- To access a particular operation (as provided by a particular EGL-generated program), a Web service client invokes a method of a local proxy. The proxy is usually the output of a tool and can be based on any computer language. The proxy is hereafter assumed to be an output of the EGL Web Service Wizard and to be based on Java.
- The Java proxy has the same methods as the Web service Java bean and includes the code necessary for these purposes:
 - To transform the invocation data into XML
 - To submit an XML message to the SOAP run-time code that resides on the same machine as the Web service Java bean.
- The SOAP run-time code invokes the appropriate method of the Web service Java bean, deserializing the client-side data from SOAP data types to Java data types.
- The Web service Java bean accesses an EGL-generated program by instantiating and invoking an EGL Java wrapper. This use of Java wrappers is not specific to Web service processing and is described in *Java wrapper classes*.
- The parameter data is returned to the Web service Java bean. If the invoked EGL program returns two or more parameters, the Web service Java bean instantiates and populates a *Return Value Java bean* so that the (potentially changed) parameter data can be returned to the SOAP run time in a single object.
- The Web service Java bean returns control to the SOAP run-time code, which does these tasks:
 - Serializes the server-side data from Java data types to SOAP data types
 - Sends the serialized data to the client-side proxy, which returns control to the Web service client

A deployment consideration

If an EGL-generated program in a Web service includes an EGL record or structure as a parameter, the implications are as follows:

- The Java proxy must serialize data when invoking the remote Web service and must deserialize data when control returns. To do those tasks, the Java proxy requires access to either of these kinds of software for each record or structure parameter:
 - A helper class, which is created when the EGL Web Service Wizard creates the Java proxy; or
 - A BeanInfo class and a related parameter wrapper class, which the EGL Web Service Wizard can create when generating the Java wrapper for the EGL-generated program.

A BeanInfo class provides a standard way to discover the public data and methods of another Java class.
- For each EGL structure or record parameter, the SOAP run-time code must access a BeanInfo class and a related parameter wrapper class. The purpose in this case is to deserialize the data being submitted to the EGL Web service and to serialize the data returned from the EGL Web service.

If the Java proxy uses a combination of BeanInfo and parameter wrapper class for a given parameter, those classes are copies of the classes that are used by the SOAP run-time code.

If the Web service client runs on the same machine as the Web service Java bean, the complexities of the technology can result in run-time errors unless you follow a simple rule: *Always maintain the Java proxy in a package different from that of the Web service Java bean.* You can easily fulfill this rule because the EGL Web Service Wizard allows you to create a client project.

EGL Web Service Wizard

The task of creating an EGL Web service is separate from the task of writing EGL programs. When you create the Web service Java bean, you can reference any EGL program. The script in the EGL program may be incomplete, but the program parameters must be fully specified.

The EGL Web Service Wizard guides you through the steps needed to create and use a Web service:

1. Collects the name and operations of the Web service; and (optionally) creates a Web service definition file to retain that information
2. Creates the Web service Java bean, as well as any wrappers, BeanInfo classes, and Return Value Java beans used at run time
3. Creates Web services description language (WSDL) that specifies both the operations that a Web service client can access and the parameters required to invoke those operations
4. Optionally, creates a Java proxy based on the WSDL
5. Optionally, creates a sample Web service client
6. Optionally, publishes the Web service to a Universal Description, Discovery, and Integration (UDDI) registry
7. Optionally, launches the WebSphere Test Environment so you can confirm that the Web service works correctly

The name of the Web service becomes the name of the Web service Java bean. Also, the name of each operation become the name of a method in the Java bean.

The following restrictions apply:

- The Web service name cannot be the same as an operation name
- Neither a Web service name nor an operation name can include a currency symbol or hyphen or be a Java reserved word; for other details, see *Naming conventions*

Web service definition file

The EGL Web service wizard lets you produce a Web service definition file, but does not require that you do so. (You can create this kind of file outside of the wizard, too, either by hand or as described in *Creating an EGL Web service definition file*. In particular, if you want to associate multiple operations with a single EGL program, you must create the file outside of the wizard.)

When you invoke the EGL Web Service Wizard, you can use an existing Web service definition file as input and so can avoid re-stating the following information, which is provided in the file:

- The name and package name of the Web service under development
- A set of attributes that identifies each operation--
 - The name by which a Web service client invokes the operation
 - The path (within the specified package) for an EGL program that provides the operation
 - The parameters of the EGL program, including an indication of whether a given parameter is for input, for output, or (as is the default) for both

The following Web service definition file associates each of three operations with the appropriate EGL program:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE EGLWebServiceDefinition PUBLIC "-//IBM//DTD EGLWSD 5.0//EN" "">
<EGLWebServiceDefinition name="TimeAndDate"
  package="com.ibm.timedate.service">

  <operation name="GetDate"
    programName="GetDate"
    programPath="/TimeDateWeb/EGL Source/GetDate.egl"/>
    <parameter name="DateRecord" input="NO" output="YES" />
  </operation>

  <operation name="GetTime"
    programName="GetTime"
    programPath="/TimeDateWeb/EGL Source/GetTime.egl"/>
    <parameter name="Time" input="NO" output="YES"/>
  </operation>

  <operation name="GetTimeAndDate"
    programName="GetTimeAndDate"
    programPath="/TimeDateWeb/EGL Source/GetTimeAndDate.egl"/>
    <parameter name="Time" input="NO" output="YES"/>
    <parameter name="DateRecord" input="NO" output="YES" />
  </operation>

</EGLWebServiceDefinition>
```

The name of the Web service definition file is the name of the Web service followed by the extension .eglwsd.

Java wrappers and BeanInfo classes

The EGL Web Service Wizard allows you to select either of two ways to generate Java wrappers and (if appropriate) BeanInfo classes. (The BeanInfo classes are necessary only if the EGL-generated program receives data as an EGL record or structure.)

Your choices are as follows:

- Specify a linkage options part to indicate how each Java wrapper in the Web service accesses an EGL program. The Wizard handles various details, and you do not need to reference a build descriptor. BeanInfo classes are generated if appropriate.
- Alternatively, you can specify a set of build descriptors, one for each Java wrapper.

The following rules apply for each build descriptor you specify (if any):

- A linkage options part must be referenced in the build descriptor option linkage
- BeanInfo classes are generated (if necessary) only if you set the build descriptor option `enableJavaWrapperGen` to YES

Test choices

You can use the EGL Web service wizard to create any of three Web service clients for use in testing the Web service:

- Web service sample JSP
- Web tools Java bean JSP
- Universal Test Client

You must use the Universal Test Client if a parameter in any of the EGL programs includes either of these:

- An array
- Data of type HEX

Related concepts

"Java wrapper" on page 395

"Linkage options part" on page 439

Help topics *Application testing and publishing*, *Tools for Web services development*, and *Web services overview*

Related tasks

"Adding a build descriptor part" on page 92

"Adding a linkage options part" on page 95

"Creating an EGL Web service definition file" on page 147

"Creating an EGL Web service"

"Editing an EGL Web service definition file" on page 148

Related reference

"enableJavaWrapperGen" on page 250

"Java wrapper classes" on page 395

"linkage" on page 256

"Naming conventions" on page 468

Creating an EGL Web service

An *EGL Web service* is a set of operations with these characteristics:

- Is invoked by a program interface over the Internet

- Is implemented by one or more EGL-generated programs

For a description of the components of a Web service and the uses of a Web service definition file, see *EGL Web service*.

Before you can create an EGL Web service, you must have access to one or more EGL files (extension .egl) that each contains a program. Any program may be incomplete, but the parameters must be fully specified.

To create an EGL Web service, do as follows:

1. Click **File > New > Other**; and at the New screen, click **Web Services**, then **Web Service**, then **Next**
2. Work through the pages of the EGL Web service wizard:
 - “Web Services page”
 - “Service Deployment Configuration page”
 - “Create an EGL Web Service page” on page 145
 - “EGL Web Service Definition page” on page 145
 - “Web Service Mapping page” on page 146
 - “Web Service Java Bean Identity and Selection pages” on page 146

For background information, review material on Web services, including these help topics:

- *The Server Tools*
- *Tools for Web services development*
- *Universal Test Client*

Web Services page

Do as follows at the Web Services page:

1. In the list box for **Web service type**, select **EGL Web Service**
2. If you want to test the Web service in the WebSphere Test Environment, select the check box for **Start Web service in Web project**
3. If you intend to publish the Web service to a Universal Description, Discovery, and Integration (UDDI) directory, select the check box for **Launch the Web Services Explorer**
4. To generate a Java proxy for client-side access of the EGL Web service, select the check box for **Generate a proxy**; you need the proxy to test the Web service in the WebSphere Test Environment.
5. If you want to test the proxy, you also need to select the check box for **Test the generated proxy**
6. If you want to avoid confirmation messages when writing to existing files, select the check box for **Overwrite files without warning**
7. If you want to avoid manual setup of file-system folders, select the check box for **Create folders when necessary**
8. If you want to avoid confirmation messages when checking files out of a repository, select the check box for **Check out files without warning**
9. Click **Next**

Service Deployment Configuration page

Do as follows at the Service Deployment Configuration page:

1. Use the default values in this situation--
 - You intend to use the IBM SOAP run time

- You are creating the EGL Web service for use in the test environment of WebSphere Application Server Version 5.1

If you are not using the defaults, click on the **Edit** button in the Server-Side Deployment Selection box. Select values for the run-time protocol and for the web application server.

2. In the Service Web project field, specify the Web project where you want to store the Web service Java bean. Select a service project EAR.
3. If you selected **Generate a proxy** on the Web services page, and you are not using the defaults, click on the **Edit** button in the Client-Side Environment Selection box. Select values for the run-time protocol and for the web application server. Select a client type, client project, and project EAR.
4. Click **Next**.

Create an EGL Web Service page

Do as follows at the EGL Web Service page:

1. Click the radio button that indicates one of two alternatives:
 - a. You are creating a new EGL Web service without using a Web service definition file as the basis of your work; or
 - b. You are using a Web service definition file, either to create a Web service from scratch or to update a bean and proxy. An update is necessary, for example, if you change the parameters of an EGL program used in the Web service.
2. Specify the name of the package that will contain the EGL Web service Java bean. If you do not name the package, the Wizard uses the default package, with these results:
 - The Java bean will include no package statement.
 - In the Web project, the Java bean source file will reside directly in the Java Source directory; and the Java bean class file will reside directly in the WEB-INF/classes directory.
3. Click **Next**.

EGL Web Service Definition page

Your tasks at the EGL Web Service Definition page vary as follows:

- If you are using a Web service definition file, select the file from a pulldown list and click **Next**.
- Alternatively, if you are creating a new EGL Web service without using a Web service definition file, your tasks are these--
 1. Specify the name of the Web service. The Web service name cannot include a currency symbol or hyphen and cannot be the same as an operation name; for other details, see *Naming conventions*.
 2. Select the check box next to each EGL program that you want to use in the Web service.
 3. If you want to create an EGL Web service definition file, select the check box toward the bottom of the page, then identify a folder for the file:
 - Type the path (relative to the workspace); or
 - Click the **Browse** button and, when a list is displayed, expand folders as needed and double-click a folder name.

The file name is the Web service name plus the extension .eglwsd.

4. Click **Next**.

Web Service Mapping page

At the Web Service Mapping page, you map the previously selected EGL programs to Web service operations. For each entry in the list at the top of the page, do as follows:

1. Click the program name.
2. Either type an operation name into the **Operation Name** text box or accept the displayed value. You are required to specify an operation name explicitly if the program name is invalid as an operation name; for details, see *Naming conventions*.
3. In the Operation Input Parameters area, select the check box for each EGL program parameter that you want to expose as an input parameter for the Web service operation.
4. In the Operation Output Parameters area, select the check box for each EGL program parameter that you want to expose as an output parameter for the Web service operation.
5. Click **Next**.

Web Service Java Bean Identity and Selection pages

Do as follows at the Web Service Java Bean Selection page:

1. Enter the name of the Java bean to be used to create the Web service. Click the **Browse classes** button to view the bean class of the selected Java bean. Click **Browse files** to select a Java bean from the project.
2. Check the box if you want to use an existing service endpoint interface (SEI), then enter the location of the service endpoint interface, browse the file system to locate it, or select from the classes included with WebSphere Studio.
3. Click **Next**.

Do as follows at the Web Service Java Bean Identity page:

1. In the **Web Service URI** text box, type the Uniform Resource Identifier (URI) that provides Internet access to your Web service.
The default base `http://tempuri.org/` comes from the WSDL specification of the World Wide Web consortium. You can use that base when testing the Web service in the WebSphere Test Environment. When you deploy the Web service, however, the URI should start with your organization's Web address or with another unique namespace.
2. Select the check box for **Use static methods**; you do not need the Web service Java bean to be instantiated at run time.
When you select **Use static methods**, the value in the **Scope** text box has no effect.
3. If you wish to deploy the Web service with Secure Digital Signature security in the WebSphere Application Server, select the check box for **Use secure SOAP**. For details on security, see SOAP Security Extensions: Digital Signature (www.w3.org/TR/2001/NOTE-SOAP-dsig-20010206).
The first Web service in a project determines whether a secure SOAP runtime is in use for any Web services in the project. Secure and non-secure Web services cannot coexist, but you can change the security status for the project in this way:
 - a. Open the project file `web.xml` in the Web Deployment Descriptor editor or in an XML or text editor
 - b. Remove the `<servlet>` elements for `rpcrouter` and `messengerouter`:

```

<servlet>
  <servlet-name>rpcrouter</servlet-name>
  .
  .
  .
</servlet>

<servlet>
  <servlet-name>messagerouter</servlet-name>
  .
  .
  .
</servlet>

```

- c. Re-run the EGL Web services wizard and choose a different setting for **Use secure SOAP**

Also, if you are using the wizard to create a proxy, be aware that when you select or clear the check box for the security status of the Web service, the wizard assigns the same default value for the security status of the proxy. The two security statuses should be the same.

4. In the **Web Service URI** text box, type (or accept the existing value for) the path of the ISD file, which is the service-specific deployment descriptor used by the SOAP run time. The path is relative to the workspace, and the default is appropriate in most cases.
5. In each of the WSDL-related text boxes, type (or accept the existing value for) the path of the document or folder specified. The path is relative to the workspace, and the default is appropriate in most cases.
6. If the defaults on subsequent pages in the wizard are appropriate (as is true in most cases), click **Finish**. If you wish to explore those pages, click **Next** as necessary, then click **Finish**.

Related concepts

"EGL Web service" on page 139

Help topics *Application testing and publishing*, *Tools for Web services development*, and *Web services overview*

"Java wrapper" on page 395

"Linkage options part" on page 439

SOAP Security Extensions: Digital Signature

(www.w3.org/TR/2001/NOTE-SOAP-dsig-20010206)

Related tasks

"Adding a build descriptor part" on page 92

"Adding a linkage options part" on page 95

"Creating an EGL Web service definition file"

Related reference

"Java wrapper classes" on page 395

"linkage" on page 256

"Naming conventions" on page 468

Creating an EGL Web service definition file

For details on the uses of a Web service definition file, see EGL Web service. To create such a file, do as follows:

1. Click **File > New > Other**; and at the New screen, click **EGL**, then **EGL Web Service Definition**, then **Next**.
2. The Select Container page is displayed.
3. Identify the file:

- a. Assign a folder name, either by typing a folder name in the first text box or by expanding entries until the folder name of interest is in that text box.
- b. Type a file name in the last text box.
- c. Click **Finish**.

The name of the Web service definition file is the name of the Web service, followed by the extension .eglwsd. The Web service name cannot be the same as an operation name and cannot include a currency symbol or hyphen; for other details, see *Naming conventions*.

4. Work in the Web Service Operations grid to relate one or more operations to each EGL program:
 - a. Click **Add**.
 - b. In the second column of the active row, click the down arrow to select an entry from a drop-down list. The path is relative to the workspace.
 - c. To accept the program name as the operation name, do nothing more in the active row; otherwise, type the operation name in the first column of the active row.
 You are required to specify an operation name explicitly if the program name is invalid as an operation name; for details, see *Naming conventions*.
 - d. If you want to delete a row, click into that row and click **Remove**.
 - e. To save the file, press **Ctrl-S**.

Related concepts

"EGL Web service" on page 139

Related tasks

"Creating an EGL Web service" on page 143

"Editing an EGL Web service definition file"

Related reference

"Naming conventions" on page 468

Editing an EGL Web service definition file

For details on the uses of a Web service definition file, see *EGL Web service*.

To edit such a file, do as follows:

1. In the Project Navigator view, right-click on the file name.
2. In the context menu, click **Open With > EGL Web Services Part Editor**.
3. A Web Service Operations grid is displayed, with each row relating an operation name to an EGL program.
4. If you want to delete a row, click into that row and click **Remove**.
5. If you want to add a row, take these steps--
 - a. Click **Add**.
 - b. In the second column of the active row, type the path of an EGL program file or click the down arrow to select an entry from a drop-down list; in either case, the path is relative to the workspace.
 - c. To accept the program name as the operation name, do nothing more in the active row; otherwise, type the operation name in the first column of the active row.

You are required to specify an operation name explicitly if the program name is invalid as an operation name; for details, see *Naming conventions*.

6. To save the file, press **Ctrl-S**.

Related concepts

"EGL Web service" on page 139

Related tasks

"Creating an EGL Web service definition file" on page 147

"Creating an EGL Web service" on page 143

Related reference

"Naming conventions" on page 468

Text applications

Segmentation in text applications

Segmentation concerns how a program interacts with its environment before issuing a **converse** statement.

By default a program that presents text forms is *non-segmented*, which means that the program behaves as if it were always in memory and providing a service to only one user. The following rules are in effect before a non-segmented program issues a **converse** statement:

- Databases and other recoverable resources are not committed
- Locks are not released
- File and database positions are retained
- Single-user EGL tables are not refreshed; their values are the same before and after the converse
- Similarly, system variables are not refreshed

A called program is always non-segmented.

A non-segmented program can be easier to code. For example, you do not need to reacquire a lock on an SQL row after a **converse**. Disadvantages include the fact that SQL rows are held during user think time, a behavior that leads to performance problems for other users who need to access the same SQL row.

Two techniques are available for releasing or refreshing resources before a converse in a non-segmented program:

- You can set the system variable **sysVar.commitOnConverse** to 1. Results are as follows before a converse:
 - Databases and other recoverable resources are committed
 - Locks are released
 - File and database positions are not retained, except when the database open statement includes the `withHold` option, as is available only for COBOL programs

The setting of **sysVar.commitOnConverse** never affects system variables or EGL tables.

- A second technique for handling converse is to set the *segmented* property of the text program to *yes*, either by changing a program property at development time or by setting the system variable **sysVar.segmentedMode** to 1 at run time. Segmentation causes the following results before a converse:
 - Databases and other recoverable resources are committed
 - Locks are released
 - File and database positions are not retained, even when the database open statement includes the `withHold` option
 - Single-user EGL tables are refreshed; their values become the same as when the program began
 - System variables are refreshed; their values become the same as when the program began, except for a subset of variables whose values are saved *across segments*

The behavior of a segmented program is unaffected by the value of the system variable `sysVar.commitOnConverse`.

Related concepts

“Program part” on page 477

Modified data tag and modified property

Each item on a text form has a *modified data tag*, which is a status value that indicates whether the user is considered to have changed the form item when the form was last presented.

As described later, an item’s modified data tag is distinct from the item’s **modified** property, which is set in the program and which pre-sets the value of the modified data tag.

Interacting with the user

In most cases, the modified data tag is pre-set to *no* when the program presents the form to the user; then, if the user changes the data in the form item, the modified data tag is set to *yes*, and your program logic can do as follows:

- Use a data table or function to validate the modified data (as occurs automatically when the modified data tag for the item is *yes*)
- Detect that the user modified the item (for example, by using a conditional statement of the type *if item modified*)

The user sets the modified data tag by typing a character in the item or by deleting a character. The modified data tag stays set, even if the user, before submitting the form, returns the field content to the value that was presented.

When a form is re-displayed due to an error, the form is still processing the same converse statement. As a result, any fields that were modified on the converse have the modified data tag set to *yes* when the form is re-displayed. For example, if data is entered into a field that has a validator function, the function can invoke the `sysLib.validationFailed` function to set an error message and cause the form to re-display. In this case, when an action key is pressed, the validator function will execute again because the field’s modified data tag is still set to *yes*.

Setting the modified property

You may want your program to do a task regardless of whether the user modified a particular field; for example:

- You may want to force the validation of a password field even if the user did not enter data into that field
- You may specify a validation function for a critical field (even for a protected field) so that the program always does a particular *cross-field validation*, which means that your program logic validates a group of fields and considers how one field’s value affects the validity of another.

To handle the previous cases, you can set the **modified** property for a particular item either in your program logic or in the form declaration:

- In the logic that precedes the form presentation, include a statement of the type *set item modified*. The result is that when the form is presented, the modified data tag for the item is pre-set to *yes*.

- In the form declaration, set the **modified** property of the item to *yes*. In this case, the following rules apply:
 - When the form is presented for the first time, the modified data tag for the item is pre-set to *yes*.
 - If any of the following situations occurs before the form is presented, the modified data tag is pre-set to *yes* when the form is presented:
 - The code runs a statement of the type *set item initial*, which reassigns the original content and property values for the item; or
 - The code runs a statement of the type *set item initialAttributes*, which reassigns the original property values (but not content) for each item on the form; or
 - The code runs a statement of the type *set form initial*, which reassigns the original content and property values for each item on the form; or
 - The code runs a statement of the type *set form initialAttributes*, which reassigns the original property values (but not content) for each item on the form

The *set* statements affect the value of the **modified** property, not the current setting of the modified data tag. A test of the type *if item modified* is based on the modified data tag value that was in effect when the form data was last returned to your program. If you try to test the modified data tag for an item before your logic presents the form *for the first time*, an error occurs at run time.

If you need to detect whether the user (rather than the program) modified an item, make sure that the value of the modified data tag for the item is pre-set to *no*:

- If the **modified** property of the item is set to *no* in the form declaration, do not use a statement of the type *set item modified*. In the absence of that statement, the **modified** property is automatically set to *no* prior to each form presentation.
- If the **modified** property of the item is set to *yes* in the form declaration, use a statement of the type *set item normal* in the logic that precedes form presentation. That statement sets the **modified** property to *no* and (as a secondary result) presents the item as unprotected, with normal intensity.

Testing whether the form is modified

The form as a whole is considered to be modified if the modified data tag is set to *yes* for any of the variable form items. If you test the modified status of a form that was not yet presented to the user, the test result is FALSE.

Examples

Assume the following settings in the form *form01*:

- The **modified** property for the field *item01* is set to *no*
- The **modified** property for the field *item02* is set to *yes*

The following logic shows the result of various tests:

```
// tests false because a converse statement
// was not run for the form
if (form01 is modified)
;
end

// causes a run-time error because a converse
// statement was not run for the form
if (item01 is modified)
;
;
```

```

end

// assume that the user modifies both items
converse form01;

// tests true
if (item01 is modified)
;
end

// tests true
if (item02 is modified)
;
end

// sets the modified property to no
// at the next converse statement for the form
set item01 initialAttributes;

// sets the modified property to yes
// at the next converse statement for the form
set item02 initialAttributes;

// tests true
// (the previous set statement takes effect only
// at the next converse statement for the form
if (item01 is modified)
;
end

// assume that the user does not modify either item
converse form01;

// tests false because the program set the modified
// data tag to no, and the user entered no data
if (item01 is modified)
;
end

// tests true because the program set the modified
// data tag to yes
if (item02 is modified)
;
end

// assume that the user does not modify either item
converse form01;

// tests false
if (item01 is modified)
;
end

// tests false because the presentation was not
// the first, and the program did not reset the
// item properties to their initial values
if (item02 is modified)
;
end

```

Creating an EGL formGroup part

An EGL formGroup part defines a collection of text and print forms. To create an EGL formGroup part, do as follows:

1. Identify a project or folder to contain the file. You must create a project or folder if you do not already have one.
2. In the workbench, click **File > New > Form Group**.
3. Select the project or folder that will contain the EGL file, then select a package. Since the formGroup name will be identical to the file name, choose a file name that adheres to EGL part name conventions. In the EGL Source File Name field, type the name of the EGL file, for example myFormGroup.
4. Click the **Finish** button.

Related concepts

"EGL projects, packages, and files" on page 7

"FormGroup part" on page 376

"Introduction to EGL" on page 1

Related tasks

"Creating an EGL source folder" on page 84

"Creating a project to work with EGL" on page 82

"Switching to the EGL or EGL Web perspective" on page 81

Related reference

"Creating an EGL source file" on page 85

"Form part in EGL source format" on page 370

"Naming conventions" on page 468

Files and databases

Resource associations and file types

An EGL record that accesses an external file, printer, or queue has a logical file or queue name. (In the case of a printer, the logical file name is *printer* for most run-time systems.) The name can be no more than 8 characters and is meaningful only as a way of relating the record to a *system name*, which the target system uses to access a physical file, printer, or queue.

In relation to files or queues, the file or queue name is a default for the system name. In relation to printers, a default exists only for COBOL output.

Instead of accepting a default, you can take one or both of these actions:

- At generation time, you control the generation process with a build descriptor that in turn references a specific resource associations part. The resource associations part relates the file name with a system name on the target platform where you intend to deploy the generated code.
- At run time (in most cases) you can change the value in the record-specific variable `resourceAssociation` (for files or queues) or in the system variable `sysVar.printerAssociation` (for print output). Your purpose is to override the system name that you specified either by default or by specifying a resource associations part.

The resource associations part does not apply to these record types:

- `basicRecord`, because basic records do not interact with data stores
- `SQLRecord`, because SQL records interact with relational databases

Resource associations part

The resource associations part is a set of association elements, each of which has these characteristics:

- Is specific to a logical file or queue name
- Has a set of entries, each specific to a target system; each entry identifies the file type on the target platform, along with the system name and in some cases additional information

You can think of an association element as a set of properties and values in a hierarchical relationship, as in the following example:

```
// an association element
property: fileName
value:    myFile01

// an entry, with multiple properties
property: system
value:    aix
property: fileType
value:    spool
property: systemName
value:    employee

// a second entry
property: system
value:    win
```

```
property: fileType
value:      seqws
property: systemName
value:      c:\myProduct\myFile.txt
```

In this example, the file name `myFile01` is related to these files:

- *employee* on AIX
- *myFile.txt* on Windows 2000/NT/XP

The file name must be a valid name, an asterisk, or the beginning of a valid name followed by an asterisk. The asterisk is the wild-card equivalent of one or more characters and provides a way to identify a set of names. An association element that includes the following value for a file name, for example, pertains to any file name that begins with the letters *myFile*:

```
myFile*
```

If multiple elements are valid for a file name used in your program, EGL uses the first element that applies. A series of association elements, for example, might be characterized by the following values for file name, in order:

```
myFile
myFile*
*
```

Consider the element associated with the last value, where the value of `myFile` is only an asterisk. Such an element could apply to any file; but in relation to a particular file, the last element applies only if the previous elements do not. If your program references `myFile01`, for instance, the linkage specified in the second element supersedes the third element to define how the reference is handled.

At generation time, EGL selects a particular association element, along with the first entry that is appropriate. An entry is appropriate in either of two cases:

- A match exists between the target system for which you are generating, on the one hand, and the **system** property, on the other; or
- The **system** property has the following value:
any

If you are generating for AIX, for example, EGL uses the first entry that refers to `aix` or to **any**.

File types

A file type determines what properties are necessary for a given entry in an association element. The next table describes the EGL file types.

Record types and VSAM

Each of three types of records is appropriate for accessing a VSAM data set, but only if the file type in the association element for the record is `ibmcobol`, `vsam`, or `vsamrs`:

- If the record is of type `indexedRecord`, the VSAM data set is a Key Sequenced Data Set with a primary or alternate index
- If the record is of type `relativeRecord`, the VSAM data set is a Relative Record Data Set
- If the record is of type `serialRecord`, the VSAM data set is an Entry Sequenced Data Set

For further details

For further details on resource associations, see these topics:

- *Record and file type cross-reference*
- *Association elements*

Related concepts

"MQSeries support" on page 161
"Parts" on page 11
"Record types and properties" on page 13
"Record parts" on page 490
"VSAM support" on page 202

Related reference

"Association elements" on page 231
"Record and file type cross-reference"
"recordName.resourceAssociation" on page 539
"resourceAssociations" on page 259
"system" on page 266
"sysVar.printerAssociation" on page 623

Record and file type cross-reference

The next table shows the association of record type and file type, by target platform.

Related concepts

"Record types and properties" on page 13
"Resource associations and file types" on page 157

Related reference

"resourceAssociations" on page 259

Logical unit of work

When you change resources that are categorized as *non-recoverable* (such as serial files on Windows 2000), your work is relatively permanent; neither your code nor EGL run-time services can simply rescind the changes. When you change resources that are categorized as *recoverable* (such as relational databases), your code or EGL run-time services either can commit the changes to make the work permanent or can rollback the changes to return to content that was in effect when changes were last committed.

Recoverable resources are as follows:

- Relational databases
- CICS queues and files that are configured to be recoverable
- MQSeries message queues, unless your MQSeries record specifies otherwise, as described in *MQSeries support*

A *logical unit of work* identifies input operations that are either committed or rolled back as a group. A unit of work begins when your code changes a recoverable resource; and ends when the first of these events occurs:

- Your code invokes the system function **sysLib.commit** or **sysLib.rollback** to commit or roll back the changes

- EGL run-time services performs a rollback in response to a hard error that is not handled in your code; in this case, all the programs in the run unit are removed from memory
- An implicit commit occurs, as happens in the following cases--
 - A program issues a **show** statement.
 - The top-level program in a run unit ends successfully, as described in *Run unit*.
 - A Web page is displayed, as when a page handler issues a **forward** statement.
 - A program issues a **converse** statement and any of the following applies:
 - You are not in VisualAge Generator compatibility mode, and the program is a segmented program
 - **sysVar.commitOnConverse** is set to 1
 - You are in VisualAge Generator compatibility mode, and **sysVar.segmentedMode** is set to 1

Unit of work for Java

In a Java run unit, the details are as follows:

- When any of the Java programs ends with a hard error, the effect is equivalent to performing rollbacks, closing cursors, and releasing locks.
- When the run unit ends successfully, EGL performs a commit, closes cursors, and releases locks.
- You can use multiple connections to read from multiple databases, but you should update only one database in a unit of work because only a one-phase commit is available. For related information, see *sysLib.connectionService*.
- When an EGL-generated program is accessed by way of an EGL-generated EJB session bean, transaction control may be affected by a transaction attribute (also called the container transaction type), which is in the deployment descriptor of the EJB session bean. The transaction attribute affects transaction control only when the linkage options part, callLink element, property **remoteComType** for the call is direct, as described in *remoteComType in callLink element*.

The EJB session bean is generated with transaction attribute REQUIRED, but you can change the value at deployment time. For details on the implications of the transaction attribute, see your Java documentation.

Related concepts

“MQSeries support” on page 161

“Run unit” on page 504

“SQL support” on page 171

Related tasks

“Setting up a J2EE JDBC connection” on page 209

“Understanding how a standard JDBC connection is made” on page 208

Related reference

“Default database” on page 198

“sysLib.commit” on page 541

“sysLib.connectionService” on page 545

“sysLib.rollback” on page 550

“Java wrapper classes” on page 395

“luwControl in callLink element” on page 450

“remoteComType in callLink element” on page 455

“sqlDB” on page 262

MQSeries support

EGL supports access of MQSeries message queues on any of the target platforms. You can provide such access in either of the following ways:

- Use MQSeries-related EGL keywords like **add** and **get next** on an MQ record; in this case, EGL hides details of MQSeries so you can focus on the business problem your code is addressing
- Invoke EGL functions that call MQSeries commands directly, in which case some commands are available that are not supported by the EGL keywords

You can mix the two approaches in a given program. For most purposes, however, you use one or the other approach exclusively.

Regardless of your approach, you can control various run-time conditions by customizing *options records*, which are global basic records that EGL run-time services passes on calls to MQSeries. When you declare an options record as a program variable, you can use an EGL-installed options record part as a typedef; or you can copy the installed part into your own EGL file, customize the part, and use the customized part as a typedef.

Your approach determines how EGL run-time services makes the options records available to MQSeries:

- If you are working with the EGL **add** and **get next** statements, you identify the options records when you specify properties of an MQ record. If you do not identify a particular options record, EGL uses a default.
- If you are invoking the EGL functions that call MQSeries directly, you use options records as arguments when you invoke the functions. Defaults are not available in this case.

For details on options records and on the values that are passed to MQSeries by default, see *Options records for MQ records*. For details on MQSeries itself, refer to these documents:

- *An Introduction to Messaging and Queueing* (GC33-0805-01)
- *MQSeries MQI Technical Reference* (SC33-0850)
- *MQSeries Application Programming Guide* (SC33-0807-10)
- *MQSeries Application Programming Reference* (SC33-1673-06)

Connections

You connect to a queue manager (called the *connecting queue manager*) the first time you invoke a statement from the following list:

- An EGL **add** or **get next** statement that accesses a message queue
- An invocation of the EGL function MQCONN or MQCONNX

You can access only one connecting queue manager at a time; however, you can access multiple queues that are under the control of the connecting queue manager. If you wish to connect directly to a queue manager other than the current connecting queue manager, you must disconnect from the first by invoking MQDISC, then connect to the second queue manager by invoking **add**, **get next**, MQCONN, or MQCONNX.

You can also access queues that are under the control of a *remote queue manager*, which is a queue manager with which the connecting queue manager can interact. Access between the two queue managers is possible only if MQSeries itself is configured to allow for that access.

Access to the connecting queue manager is terminated when you invoke MQDISC or when your code ends.

Include message in transaction

You can embed queue-access statements in a unit of work so that all your changes to the queues are committed or rolled back at a single processing point. If a statement is in a unit of work, the following is true:

- An EGL **get next** statement (or an EGL MQGET invocation) removes a message only when a commit occurs
- The message placed on a queue by an EGL **add** statement (or an EGL MQPUT invocation) is visible outside the unit of work only when a commit occurs

When queue-access statements are not in a unit of work, each change to a message queue is committed immediately.

An MQSeries-related EGL **add** or **get next** statement is embedded in a unit of work if the property **includeMsgInTransaction** is in effect for the MQ record. The generated code includes these options:

- For MQGET, MQGMO_SYNCPOINT
- For MQPUT, MQPMO_SYNCPOINT

If you do not specify the property **includeMsgInTransaction** for an MQ record, the queue-access statements run outside of a unit of work. The generated code includes these options:

- For MQGET, MQGMO_NO_SYNCPOINT
- For MQPUT, MQPMO_NO_SYNCPOINT

When your code ends a unit of work, EGL commits or rolls back *all* recoverable resources being accessed by your program, including databases, message queues, and recoverable files. This outcome occurs whether you use the system functions (**sysLib.commit**, **sysLib.rollback**) or the EGL calls to MQSeries (MQCMIT, MQBACK); the appropriate EGL system function is invoked in either case.

A rollback occurs if an EGL program terminates early because of an error detected by EGL run-time services.

Customization

If you wish to customize your interaction with MQSeries rather than relying on the default processing of **add** and **get next** statements, you need to review the information in this section.

EGL dataTable part

A set of EGL dataTable parts is available to help you interact with MQSeries. Each part allows EGL-supplied functions to retrieve values from memory-based lists at run time. The next section includes details on how data tables are deployed.

Making customization possible

To make customization possible, you must bring a variety of installed EGL files into your project *without changing them in any way*. The files are as follows:

records.egl

Contains basic record parts that can be used as typedefs for the options records that are used in your program; also includes structure parts that are used by those records and that give you the flexibility to develop record parts of your own

functions.egl

Contains two sets of functions:

- MQSeries command functions, which access MQSeries directly
- Initialization functions, which let you place initial values in the options records that are used in your program

mqrcode.egl, mqrc.egl, mqvalue.egl

Contains a set of EGL dataTable parts that are used by the command and initialization functions

Your tasks are as follows:

1. Using the process for importing files into the workbench, bring those files into an EGL project. The files reside in the following directory:

```
installationDir\wstools\eclipse\plugins\  
com.ibm.etools.egl.generators_version\MqReusableParts
```

installationDir

The WebSphere Studio installation directory, such as c:\myStudio

version

The latest version of the plugin; for example, 5.1.2

2. To make the parts more easily available to your program, write one or more EGL import statements in the file that contains your program. If the files to be imported reside in a project other than the one in which you are developing code, make sure that your project references the other project.

For details, see *Import*.

3. In your program, declare global variables:
 - Declare MQRC, MQRCODE, and MQVALUE, each of which must use as a typedef the dataTable part that has the same name as the variable.
 - For each options record that you wish to pass to MQSeries, declare a basic record that uses an options record part as a typedef. For details on each part, see *Options records for MQ records*.
4. In your function, initialize the options records that you intend to pass to MQSeries. You can do this easily by invoking the imported EGL initialization function for a given options record. The name of each function is the name of the part that is used as a typedef for the record, followed by _INIT. An example is MQGMO_INIT.
5. Set values in the options records. In many cases you set a value by assigning an EGL symbol that represents a constant, each of which is based on a symbol described in the MQSeries documentation. You can specify multiple EGL symbols by summing individual ones, as in this example:

```
MQGMO.GETOPTIONS = MQGMO_LOCK  
                  + MQGMO_ACCEPT_TRUNCATED_MSG  
                  + MQGMO_BROWSE_FIRST
```

MQSeries-related EGL keywords

When you work with the MQSeries-related EGL keywords like *add* and *scan*, you define an MQ record for each message queue you wish to access. The record layout is the format of the message.

The next table lists the keywords.

Keyword	Purpose
add	<p>Places the content of an MQ record at the end of the specified queue.</p> <p>The EGL add statement invokes as many as three MQSeries commands:</p> <ul style="list-style-type: none"> • MQCONN connects the generated code to a queue manager and is invoked when no connection is active. • MQOPEN establishes a connection to a queue and is invoked when a connection is active but the queue is not open. • MQPUT puts the record in the queue and is always invoked unless an error occurred in an earlier MQSeries call. <p>After adding an MQ record, you must close a message queue before reading an MQ record from the same queue.</p>
close	<p>Relinquishes access to the message queue that is associated with an MQ record.</p> <p>The EGL close statement invokes the MQSeries MQCLOSE command, which also is invoked automatically when your program ends.</p> <p>You should close the message queue after an add or scan if another program requires access to the queue. The close is particularly appropriate if your program runs for a long time and no longer needs access.</p>
scan	<p>Reads the first message in a queue into a message queue record and (by default) removes the message from the queue.</p> <p>The EGL scan statement invokes as many as three MQSeries commands:</p> <ul style="list-style-type: none"> • MQCONN connects the generated code to a queue manager and is invoked when no connection is active. • MQOPEN establishes a connection to a queue and is invoked when a connection is active but the queue is not open. • MQGET removes the record from the queue and is always invoked unless an error occurred in an earlier MQSeries call. <p>After reading an MQ record, you must close the queue before adding an MQ record to the same queue.</p>

Manager and queue specification

When you work with the MQSeries-related EGL keywords, you identify a queue in the following situations:

- At declaration time, you specify a logical queue name, and you do so by setting the **queueName** property of the MQ record part. That logical queue name acts as a default for the queue name accessed at run time; but in most cases the name is meaningful only as way of associating the MQ record with a physical queue. The logical queue name can be no more than 8 characters.
- At generation time, you control the generation process with a buildDescriptor part that in turn can reference a resource associations part. The resource associations part associates the queue name with the name of a physical queue.
- At run time, your code can change the value in the record-specific variable **record.resourceAssociation** to override any queue name you specified at declaration or generation time.

The name of the physical queue has the following format:

queueManagerName:physicalQueueName

queueManagerName

Name of the queue manager; if this name is omitted, the colon is omitted, too

physicalQueueName

Name of the physical queue, as known to the specified queue manager

The first time that you issue an add or scan statement against a message queue record, a connecting queue manager must be specified, whether by default or otherwise. In the simplest case, you do not specify a connecting queue manager at all, but rely on a default value in the MQSeries configuration.

The record-specific variable **record.resourceAssociation** always contains at least the name of the message queue for a given MQ record.

Remote message queues

If you want to access a queue that is controlled by a remote queue manager, you must do the following:

- Issue the EGL close statement to relinquish access to the queue now in use
- Set the record-specific variable **record.resourceAssociation** to ensure later access of the remote queue

You set **record.resourceAssociation** in one of two ways, depending on how the queue-manager relationships are established in MQSeries:

- If the connecting queue manager has a local definition of the remote queue, set **record.resourceAssociation** as follows:
 - Accept the same value for the connecting queue manager (either by specifying the name of the connecting queue manager or by specifying no name; in the latter case, omit the colon).
 - Specify the name of the local definition of the remote queue.

Your next use of the *add* or *scan* statement issues an MQOPEN to establish access to the remote queue.

- Alternatively, set **record.resourceAssociation** with the name of the remote queue manager, along with the name of the remote queue. The connecting queue manager does not change in this case. Your next use of the *add* or *scan* statement issues MQOPEN and uses the connection already in place.

Related concepts

“Direct MQSeries calls”

“MQSeries support” on page 161

Related reference

“MQ record properties” on page 169

“Options records for MQ records” on page 169

Direct MQSeries calls

You can use a set of installed EGL functions that mediate between your code and MQSeries, as described in *MQSeries support*.

The next table lists the available functions and identifies the required arguments. MQBACK (MQSTATE), for example, indicates that when you invoke MQBACK you pass an argument that is based on the MQSTATE record part. The record parts are described later.

MQSeries-related EGL function invocation	Effect
MQBACK (MQSTATE)	Invokes the system function sysLib.rollback to rollback a logical unit of work. The rollback affects <i>all</i> recoverable resources being accessed by your program, including databases, message queues, and recoverable files.
MQBEGIN (MQSTATE, MQBO)	Begins a logical unit of work.
MQCHECK_COMPLETION (MQSTATE)	Sets the mqdescription field of the record that is based on MQSTATE. The setting is based on the last-returned reason code. The function MQCHECK_COMPLETION is called automatically from the EGL functions MQBEGIN, MQCLOSE, MQCONN, MQCONNEX, MQDISC, MQGET, MQINQ, MQOPEN, MQPUT, MQPUT1, and MQSET.
MQCLOSE (MQSTATE)	Closes the message queue to which MQSTATE.hobj refers.
MQCMIT (MQSTATE)	Invokes the system function sysLib.commit to commit a logical unit of work. The commit affects <i>all</i> recoverable resources being accessed by your program, including databases, message queues, and recoverable files.
MQCONN (MQSTATE, qManagerName)	Connects to a queue manager, which is identified by <i>qManagerName</i> , a string of up to 48 characters. MQSeries sets the connection handle (MQSTATE.hconn) for use in subsequent calls. Note: Your code can be connected to one queue manager at a time.
MQCONNEX(MQSTATE, qManagerName, MQCNO)	Connects to a queue manager with options that control the way that the call works. The queue manager is identified by <i>qManagerName</i> , a string of up to 48 characters. MQSeries sets the connection handle (MQSTATE.hconn) for use in subsequent calls.
MQDISC (MQSTATE)	Disconnects from a queue manager.
MQGET(MQSTATE, MQMD, MQGMO, BUFFER)	Reads and removes a message from the queue. The buffer cannot be more than 32767 bytes, but that restriction does not apply if you are using the EGL get next statement.
MQINQ(MQSTATE, MQATTRIBUTES)	Requests attributes of a queue.
MQNOOP()	Used only by EGL.
MQOPEN(MQSTATE, MQOD)	Opens a message queue. MQSeries sets the queue handle (MQSTATE.hobj) for use in subsequent calls.
MQPUT(MQSTATE, MQMD, MQPMO, BUFFER)	Adds a message to the queue. The buffer cannot be more than 32767 bytes, but that restriction does not apply if you are using the EGL add statement.
MQPUT1(MQSTATE, MQOD, MQMD, MQPMO, BUFFER)	Opens a queue, writes a single message, and closes the queue.
MQSET(MQSTATE, MQATTRIBUTES)	Sets attributes of a queue.

The next table lists the options records that are used as arguments when you invoke the MQSeries-related EGL functions. Also listed is the initialization function that should be invoked for a given argument.

Your first step is to initialize the argument that is based on the MQSTATE record part. In the following example (as in the table that follows), the argument name is assumed to be the same as the name of the record part:

```
MQSTATE_INIT(MQSTATE);
```

Argument (the record part name)	Initialization function	Description	For Java or COBOL output?
MQATTRIBUTES	none	Arrays of attributes and attribute selectors, plus other information used in the command MQINQ or MQSET	Either
MQBO	MQBO_INIT (MQBO)	Begin options	Either
MQCNO	MQCNO_INIT (MQCNO)	Connect options	Either
MQDH	MQDH_INIT (MQDH)	Distribution header	COBOL only
MQDLH	MQDLH_INIT (MQDLH)	Dead-letter header	COBOL only
MQGMO	MQGMO_INIT (MQGMO)	Get-message options	Either
MQIIH	MQIIH_INIT (MQIIH)	IMS [™] information header; describes information that is required at the start of an MQSeries message sent to IMS	Either; however, MQSeries documentation indicates that use of this header is not supported on Windows 2000/NT/XP
MQINTATTRS	none	Arrays of integer attributes for use in the command MQINQ or MQSET	Either
MQMD	MQMD_INIT (MQMD, MQSTATE)	Message descriptor (MQSeries version 2)	Either
MQMD1	MQMD1_INIT (MQMD1, MQSTATE)	Message descriptor (MQSeries version 1)	COBOL only
MQMDE	MQMDE_INIT (MQMDE, MQSTATE)	Message descriptor extension	Supported for COBOL; but for Java, use only the fields that are in MQSeries version 2
MQOD	MQOD_INIT (MQOD)	Object descriptor	Either
MQOO	MQOO_INIT (MQOO)	Open options	Either

Argument (the record part name)	Initialization function	Description	For Java or COBOL output?
MQOR	MQOR_INIT (MQOR)	Object record	COBOL only
MQPMO	MQPMO_INIT (MQPMO)	Put-message options	Either
MQRMH	MQRMH_INIT (MQRMH, MQSTATE)	Message reference header	COBOL only
MQRR	MQRR (MQRR)	Response record	COBOL only
MQSELECTORS	none	An array of attribute selectors, used only if you wish to access MQSeries without use of EGL functions	Either
MQSTATE	MQSTATE_INIT (MQSTATE)	A collection of arguments that are each used in one or more calls to MQSeries; for example, when you connect with the EGL function MQCONN or MQCONNEX, MQSeries sets the connection handle (MQSTATE.hconn) for use in subsequent calls	Either
MQTM	MQTM_INIT (MQTM)	Trigger message	COBOL only
MQTMC2	MQTMC2_INIT (MQTMC2)	Trigger message 2 (character format)	COBOL only
MQXQH	MQXQH_INIT (MQXQH, MQSTATE)	Transmission queue header	Either

As shown, the supported arguments are more numerous when you generate in COBOL than when you generate in Java.

Note: The record parts each contain only one structure item, and the structure item uses a structure part as a typeDef. This setup gives you maximum flexibility. You can create your own record parts that are each composed of a series of structure parts.

The name of each structure part is the name of the record part followed by `_S`; the record part MQGMO, for example, uses a structure part named MQGMO_S.

Related concepts

"MQSeries-related EGL keywords" on page 163

"MQSeries support" on page 161

"Record parts" on page 490

"Typedef" on page 20

Related reference

“get next” on page 324

“sysLib.commit” on page 541

“sysLib.rollback” on page 550

MQ record properties

This page describes these MQ record properties:

- Queue name
- Include message in transaction
- Open input queue for exclusive use

For details on the other properties, see these pages:

- Options records for MQ records
- Properties that support variable-length records

Queue name

Queue name is required and refers to the logical queue name, which can be no more than 8 characters. For details on the meaning of your input, see *MQSeries-related EGL keywords*.

Include message in transaction

Include message in transaction, if set, embeds each of the record-specific messages in a transaction, and your code can commit or roll back that transaction.

For details on the implications of your choice, see *MQSeries support*.

Open input queue for exclusive use

If you set *Open input queue for exclusive use*, your code has the exclusive ability to read from the message queue; otherwise, other programs can read from the queue. This property is equivalent to the MQSeries option MQOO_INPUT_EXCLUSIVE.

Related concepts

“MQSeries-related EGL keywords” on page 163

“MQSeries support” on page 161

“Record types and properties” on page 13

Related reference

“Options records for MQ records”

“Properties that support variable-length records” on page 502

Options records for MQ records

Each MQ record is associated with five *options records*, which EGL uses as arguments in the hidden calls to MQSeries:

- Get options record (MQGMO)
- Put options record (MQPMO)
- Open options record (MQOO: a record with one structure item)
- Message descriptor record (MQMD)
- Queue descriptor record (MQOD)

When you specify an options record as a property of an MQ record, you are referring to a variable that uses a working storage record part (like MQOD) as a typeDef. The part resides in an EGL file that is provided with the product, as described in *MQSeries support*. Instead of using the record part as is, you can copy it into your own EGL file and customize the part.

If you do not indicate that a given options record is in use, EGL builds a default record and assigns values, as described in the following sections. The default options records are not available, however, when you access MQSeries without using MQ records.

Get options record

You can create a get options record based on the MQSeries Get Message Options (MQGMO), which is an argument on MQSeries MQGET calls. If you do not declare a get options record, EGL automatically builds a default named MQGMO, and your generated program does the following:

- Initializes the get options record with the values listed at the beginning of *Data initialization*
- Sets OPTIONS to either MQGMO_SYNCPOINT or MQGMO_NO_SYNCPOINT, depending on whether you set the MQ record property *Include message in transaction*

Put options record

You can create a put options record based on the MQSeries Put Message Options (MQPMO), which is an argument on MQSeries MQPUT calls. If you do not declare a put options record, EGL automatically builds a default named MQPMO, and your generated program does the following:

- Initializes the put options record with the values listed at the beginning of *Data initialization*
- Sets OPTIONS to either MQPMO_SYNCPOINT or MQPMO_NO_SYNCPOINT, depending on whether you set the MQ record property *Include message in transaction*

Open options record

The content of the open options record determines the value of the Options parameter that is used in calls to the MQSeries command MQOPEN or MQCLOSE. The open options record part (MQOO) is available, but if you do not declare a record based on that part, EGL automatically builds a default named MQOO as follows:

- On an MQOPEN that is invoked because of an EGL add statement, the generated program sets MQOO.OPTIONS to this:
MQOO_OUTPUT + MQOO_FAIL_IF QUIESCING
- On an MQOPEN that is invoked because of an EGL scan statement, the generated program sets MQOO.OPTIONS to the following when the message queue record property option *Open input queue for exclusive use* is in effect:
MQOO_INPUT_EXCLUSIVE + MQOO_FAIL_IF QUIESCING
- On an MQOPEN that is invoked because of an EGL scan statement, the generated program sets MQOO.OPTIONS to the following when the message queue record property option *Open input queue for exclusive use* is not in effect:
MQOO_INPUT_SHARED + MQOO_FAIL_IF QUIESCING
- On an MQCLOSE that is invoked because of an EGL close statement, the generated program sets MQOO.OPTIONS to the following:
MQCO_NONE

Message descriptor record

You can create a message descriptor record based on the MQSeries Message Descriptor (MQMD), which is a parameter on MQGET and MQPUT calls. If you do not declare a message descriptor record, EGL automatically builds a default named MQMD and initializes that record with the values listed in *Data initialization*.

Queue descriptor record

You can create a queue descriptor record based on the MQSeries Object Descriptor (MQOD), which is an argument on MQSeries MQOPEN and MQCLOSE calls. If you do not declare a queue descriptor record, EGL automatically builds a default named MQOD, and your generated program does the following:

- Initializes the queue descriptor record with the values listed at the beginning of *Data initialization*
- Sets OBJECTTYPE in that record to MQOT_Q
- Sets OBJECTMGRNAME to the queue manager name specified in the system word *record.resourceAssociation*; but if *record.resourceAssociation* does not reference the queue manager name, OBJECTMGRNAME has no value
- Sets OBJECTNAME to the queue name in *record.resourceAssociation*

Related concepts

“Direct MQSeries calls” on page 165

“MQSeries-related EGL keywords” on page 163

“MQSeries support” on page 161

Related reference

“Data initialization” on page 277

“recordName.resourceAssociation” on page 539

“MQ record properties” on page 169

SQL support

As shown in the next table, EGL-generated code can access a relational database on any of the target systems.

Target System	Support for access of relational databases
AIX, iSeries, Linux, Windows 2000/NT/XP, UNIX System Services	JDBC provides access to DB2 UDB, Oracle, or Informix [®]

As you work on a program, you can code SQL statements as you would when coding programs in most other languages. To ease your way, EGL provides SQL statement templates for you to fill.

Alternatively, you can use an SQL record as the I/O object when you code an EGL statement. Using the record in this way means that you access a database either by customizing an SQL statement provided to you or by relying on a default that removes the need to code SQL.

In either case, be aware of these aspects of EGL support for SQL:

- If you want to test for a null in a particular table column, you must receive the column value into an SQL record, into a record item that is declared as nullable. For details, see *Testing for and setting NULL* described later.
- In the overview sections that follow (and in keeping with SQL terminology), each item that is referenced in an SQL statement is called a *host variable*. The word *host* refers to the language that embeds the SQL statement; in this case, to the EGL procedural language. A host variable in an SQL statement is preceded by a colon, as in this example:

```

select empnum, empname
from employee
where empnum >= :myRecord.empnum
for update of empname

```

EGL statements and SQL

The next table lists the EGL keywords that you can use to access a relational database. Included in this table is an outline of the SQL statements that correspond to each keyword. When you code an EGL **add** statement, for example, you generate an SQL INSERT statement.

You use the EGL **open** and **get next** statements in many business applications. Those statements help you to declare, open, and process a *cursor*, which is a run-time entity that acts as follows:

- Returns a *result set*, which a list of rows that fulfill your search criteria
- Points to a specific row in the result set

When your output is Java code, you can use the EGL **open** statement to call a stored procedure. That procedure is composed of logic that is written outside of EGL, is stored in the database management system, and also returns a result set. (Regardless of your output language, you can use the EGL **execute** statement to call a stored procedure.)

Later sections give details on processing a result set.

If you intend to code SQL statements explicitly, you use the EGL **execute** statement and possibly the EGL **prepare** statement.

Keyword/Purpose	Outline of SQL statements	Can you modify the SQL?
add Places a row in a database; or (if you use a dynamic array of SQL records), places a set of rows based on the content of successive elements of the array.	INSERT row (as occurs repeatedly, if you specify a dynamic array).	Yes
close Releases unprocessed rows.	CLOSE cursor.	No
delete Deletes a row from a database.	DELETE row. The row was selected in either of two ways: <ul style="list-style-type: none"> • When you invoked a get statement with the forUpdate option (as appropriate when you wish to select the first of several rows that have the same key value) • When you invoked an open statement with the forUpdate option and then a get next statement (as appropriate when you wish to select a set of rows and to process the retrieved data in a loop) 	No

Keyword/Purpose	Outline of SQL statements	Can you modify the SQL?
<p>get (also called get by key value)</p> <p>Reads a single row from a database; or (if you use a dynamic array of SQL records), reads successive rows into successive elements in the array.</p>	<p>SELECT row, but only if you set the option <code>singleRow</code>. Otherwise, the following rules apply:</p> <ul style="list-style-type: none"> EGL converts a get statement to this: <ul style="list-style-type: none"> DECLARE cursor with SELECT or (if you set the <code>forUpdate</code> option) with SELECT FOR UPDATE. OPEN cursor. FETCH row. If you did not specify the option <code>forUpdate</code>, EGL also closes the cursor. The <code>singleRow</code> and <code>forUpdate</code> options are not supported with dynamic arrays; in that case, EGL run time declares and opens a cursor, fetches a series of rows, and closes the cursor. 	Yes
<p>get next</p> <p>Reads the next row in a database that was selected by an open statement.</p>	<p>EGL converts a get next statement to an SQL FETCH statement.</p>	Yes, but only to set the INTO clause
<p>execute</p> <p>Lets you run an SQL data-definition statement (of type CREATE TABLE, for example); or a data-manipulation statement (of type INSERT or UPDATE, for example); or a prepared SQL statement that does not begin with a SELECT clause.</p>	<p>The SQL statement you write is made available to the database management system.</p> <p>The primary use of execute is to code a single SQL statement that is fully formatted at generation time, as in this example--</p> <pre>try execute #sql{ // no space after "#sql" delete from EMPLOYEE where department = :myRecord.department }; onException myErrorHandler(10); end</pre> <p>A fully formatted SQL statement may include host variables in the WHERE clause.</p>	Yes
<p>open</p> <p>Selects a set of rows from a relational database for later retrieval with get next statements.</p>	<p>EGL converts an open statement to a CALL statement (for accessing a stored procedure) or to these statements:</p> <ul style="list-style-type: none"> DECLARE cursor with SELECT or with SELECT FOR UPDATE. OPEN cursor. 	Yes

Keyword/Purpose	Outline of SQL statements	Can you modify the SQL?
<p>prepare</p> <p>Specifies an SQL PREPARE statement, which optionally includes details that are known only at run time; you run the prepared SQL statement by running an EGL execute statement or (if the SQL statement begins with SELECT) by running an EGL open or get statement.</p>	<p>EGL converts a prepare statement to an SQL PREPARE statement, which is always constructed at run time. In the following example of an EGL prepare statement, each parameter marker (?) is resolved by the USING clause in the subsequent execute statement:</p> <pre>myString = "insert into myTable " + "(empnum, empname) " + "value ?, ?"; try prepare myStatement from myString; onException // exit the program myErrorHandler(12); end try execute myStatement using :myRecord.empnum, :myRecord.empname; onException myErrorHandler(15); end</pre>	Yes
<p>replace</p> <p>Puts a changed row back into a database.</p>	<p>UPDATE row. The row was selected in either of two ways:</p> <ul style="list-style-type: none"> • When you invoked a get statement with the forUpdate option (as appropriate when you wish to select the first of several rows that have the same key value); or • When you invoked an open statement with the forUpdate option and then a get next statement (as appropriate when you wish to select a set of rows and to process the retrieved data in a loop). 	Yes

Note: Under no circumstances can you update multiple database tables by coding a single EGL statement.

Result-set processing

A common way to update a series of rows is as follows:

1. Declare and open a cursor by running an EGL **open** statement with the option forUpdate; that option causes the selected rows to be locked for subsequent update or deletion
2. Fetch a row by running an EGL **get next** statement
3. Do the following in a loop:
 - a. Change the data in the host variables into which you retrieved data
 - b. Update the row by running an EGL **replace** statement
 - c. Fetch another row by running an EGL **get next** statement
4. Commit changes by running the EGL function **commit**.

The statements that open the cursor and that act on the rows of that cursor are related to each other by a result-set identifier, which must be unique across all result-set identifiers, program variables, and program parameters within the program. You specify that identifier in the **open** statement that opens the cursor, and you reference the same identifier in the **get next**, **delete**, and **replace** statements that affect an individual row, as well as on the **close** statement that closes the cursor. For additional details, see *resultSetID*.

The following code shows how to update a series of rows when you are coding the SQL yourself:

```

handleHardIOErrors = 1;

try
  open selectEmp forUpdate with
  #sql{          // no space after "#sql"
    select empname
    from EMPLOYEE
    where empnum >= :myRecord.empnum
    for update of empname
  };

onException
  myErrorHandler(8);    // exits program
end

try
  get next from selectEmp into :myRecord.empname;
onException
  if (sysVar.sqlcode != 100)
    myErrorHandler(8); // exit the program
  end
end

while (sysVar.sqlcode != 100)
  myRecord.empname = myRecord.empname + " " + "III";

  try
    execute
    #sql{
      update EMPLOYEE
      set empname = :empname
      where current of selectEmp
    };
  onException
    myErrorHandler(10); // exits program
  end

  try
    get next from selectEmp into :myRecord.empname;
  onException
    if (sysVar.sqlcode != 100)
      myErrorHandler(8); // exits program
    end
  end
end // end while; cursor is closed automatically
// when the last row in the result set is read

sysLib.commit;

```

If you wish to avoid some of the complexity in the previous example, consider SQL records. Their use allows you to streamline your code and to use I/O error values that do not vary across database management systems. The next example is equivalent to the previous one but uses an SQL record called emp:

```

handleHardIOErrors = 1;

try
  open selectEmp forUpdate for emp;
onException
  myErrorHandler(8);    // exits program
end

try
  get next emp;
onException
  if (sysVar.sqlcode not noRecordFound)
    myErrorHandler(8); // exit the program
  end
end

while (sysVar.sqlcode not noRecordFound)
  myRecord.empname = myRecord.empname + " " + "III";

  try
    replace emp;
  onException
    myErrorHandler(10); // exits program
  end

  try
    get next emp;
  on exception
    if (sysVar.sqlcode not noRecordFound)
      myErrorHandler(8); // exits program
    end
  end
end // end while; cursor is closed automatically
// when the last row in the result set is read

sysLib.commit;

```

Later sections describe SQL records.

SQL records and their uses

An SQL record is a variable that is based on an SQL record part. This type of record allows you to interact with a relational database as though you were accessing a file. If the variable EMP is based on an SQL record part that references the database table EMPLOYEE, for example, you can use EMP in an EGL **add** statement:

```
add EMP;
```

In this case, EGL inserts the data from EMP into EMPLOYEE. The SQL record also includes state information so that after the EGL statement runs, you can test the SQL record to perform tasks conditionally, in accordance with the I/O error value that resulted from database access:

```

handleHardIOErrors = 1;

try
  add EMP;
onException
  if (EMP is unique)    // if a table row
                      // had the same key
    myErrorHandler(8);
  end
end

```

An SQL record like EMP allows you to interact with a relational database as follows:

- Declare an SQL record part and the related SQL record
- Define a set of EGL statements that each use the SQL record as an I/O object
- Accept the default behavior of the EGL statements or make the SQL changes that are appropriate for your business logic

Declaring an SQL record part and the related record

You declare an SQL record part and associate each of the record items with a column in a relational table or view. You can let EGL make this association automatically by way of the EGL editor's retrieve feature, as described later in *Database access at declaration time*.

The structure in each SQL record part must be *flat* (without hierarchy), and none of the record items can be an array.

After you declare an SQL record part, you declare an SQL record that is based on that part. The SQL record must be declared as a program variable (global to the program), not as a program parameter or function variable.

Defining the SQL-related EGL statements

You can define a set of EGL statements that each use the SQL record as the I/O object in the statement. For each statement, EGL provides an *implicit SQL statement*, which is not in the source but is implied by the combination of SQL record and EGL statement. In the case of an EGL **add** statement, for example, an implicit SQL INSERT statement places the value of a given record item into the associated table column. If your SQL record includes a record item for which no table column was assigned, EGL forms the implicit SQL statement on the assumption that the name of the record item is identical to the name of the column.

Using implicit SELECT statements: When you define an EGL statement that uses an SQL record and that generates either an SQL SELECT statement or a cursor declaration, EGL provides an implicit SQL SELECT statement. (That statement is embedded in the cursor declaration, if any.) For example, you might declare a variable that is named EMP and is based on the following record part:

```
Record Employee sqlRecord
  tableNames = EMPLOYEE
  keyItems = empnum
  empnum decimal(6,0) isReadOnly=yes;
  empname char(40);
end
```

Then, you might code a **get** statement:

```
get EMP;
```

The implicit SQL SELECT statement is as follows:

```
SELECT empnum, empname
FROM   EMPLOYEE
WHERE  empnum = :empnum
```

EGL also places an INTO clause into the standalone SELECT statement (if no cursor declaration is involved) or into the FETCH statement associated with the cursor. The INTO clause lists the host variables that receive values from the columns listed in the first clause of the SELECT statement:

```
INTO :empnum, :empname
```

The implicit SELECT statement reads each column value into the corresponding host variable; references the tables specified in the SQL record; and has a search criterion (a WHERE clause) that depends on *a combination of two factors*:

- The value you specified for the record property **defaultSelectCondition**; and
- A relationship (such as an equality) between two sets of values:
 - Names of the columns that constitute the table key
 - Values of the host variables that constitute the record key

A special situation is in effect if you read data into a dynamic array of SQL records, as is possible with the **get** statement:

- A cursor is open, successive rows from the database are read into successive elements of the array, the result set is freed, and the cursor is closed.
- If you do not specify an SQL statement, the search criterion depends on the record property **defaultSelectCondition**, but also depends on a relationship (specifically, a greater-than-or-equal-to relationship) between the following sets of values:
 - Names of columns, as specified indirectly when you specify items in the EGL statement
 - Values of those items

Any host variables specified in the property **defaultSelectCondition** must be outside the SQL record that is the basis of the dynamic array.

For details on the implicit SELECT statement, which vary by keyword, see *get* and *open*.

Using SQL records with cursors: When you are using SQL records, you can relate cursor-processing statements by using the same SQL record in several EGL statements, as you can by using a result-set identifier. However, any cross-statement relationship that is indicated by a result-set identifier takes precedence over a relationship indicated by the SQL record; and in some cases you must specify a resultSetID.

In addition, only one cursor can be open for a particular SQL record. If an EGL statement opens a cursor when another cursor is open for the same SQL record, the generated code automatically closes the first cursor.

Customizing the SQL statements

Given an EGL statement that uses an SQL record as the I/O object, you can progress in either of two ways:

- You can accept the implicit SQL statement. In this case, changes made to the SQL record part affect the SQL statements used at run time. If you later indicate that a different record item is to be used as the key of the SQL record, for example, EGL changes the implicit SELECT statement used in any cursor declaration that is based on that SQL record part.
- You can choose instead to make the SQL statement explicit. In this case, the details of that SQL statement are isolated from the SQL record part, and any subsequent changes made to the SQL record part have no effect on the SQL statement that is used at run time.

If you remove an explicit SQL statement from the source, the implicit SQL statement (if any) is again available at generation time.

Testing for and setting NULL

EGL internally maintains a null indicator for each host variable that has the following characteristics:

- Is in an SQL record
- Is declared to be **nullable**

Do not code host variables for null indicators in your SQL statements, as you might in some languages. To test for null in a nullable host variable, use an EGL **if** statement. You also can test for retrieval of a truncated value, but only when a null indicator is available.

You can null an SQL table column in either of two ways:

- Use an EGL **set** statement to null a nullable host variable, then write the related SQL record to the database; or
- Use the appropriate SQL syntax, either by writing an SQL statement from scratch or by customizing an SQL statement that is associated with the EGL **add** or **replace** statement

For additional details on null processing, see *SQL item properties*.

Database access at declaration time

You receive the following benefits from accessing (at declaration time) a database that has similar characteristics to the database that your code will access at run time:

- If you access a database table or view that is equivalent to a table or view associated with an SQL record, you can use the retrieve feature of the EGL part editor to create or overwrite the record items. The retrieve feature accesses information stored in the database management system so that the number and data characteristics of the created items reflect the number and characteristics of the table columns. After you invoke the retrieve feature, you can rename record items, delete record items, and make other changes to the SQL record.
- Your access of an appropriately structured database at declaration time helps to ensure that your SQL statements will be valid in relation to an equivalent database at run time.

The retrieve feature creates record items that each have the same name (or almost the same name) as the related table column.

You cannot retrieve a view that is defined with the DB2 condition **WITH CHECK OPTIONS**.

For further details on using the retrieve feature, see *Retrieving SQL table data*. For details on naming, see *Setting preferences for SQL retrieve*.

To access a database at declaration time, specify connection information in a preferences page, as described in *Setting preferences for SQL database connections*.

Related concepts

“Dynamic SQL” on page 180
“Logical unit of work” on page 159
“resultSetID” on page 181

Related tasks

- "Retrieving SQL table data" on page 191
- "Setting preferences for SQL database connections" on page 188
- "Setting preferences for SQL retrieve" on page 190

Related reference

- "add" on page 293
- "close" on page 303
- "Database authorization and table names" on page 197
- "Default database" on page 198
- "delete" on page 307
- "execute" on page 309
- "File and database (system words)" on page 537
- "get" on page 318
- "get next" on page 324
- "Informix and EGL" on page 199
- "open" on page 335
- "prepare" on page 339
- "replace" on page 341
- "SQL data codes and EGL host variables" on page 200
- "SQL item properties" on page 57
- "SQL record internals" on page 199
- "SQL record part in EGL source format" on page 498
- "Testing for and setting NULL" on page 179

Dynamic SQL

The SQL statement associated with an EGL statement can be specified statically, with every detail in place at generation time. When dynamic SQL is in effect, however, the SQL statement is built at run time, each time that the EGL statement is invoked.

Use of dynamic SQL decreases the speed of run-time processing, but lets you vary a database operation in response to a run-time value:

- For a database query, you may want to vary the selection criteria, how data is aggregated, or the order in which rows are returned; those details are controlled by the WHERE, HAVING, GROUP BY, and ORDER BY clauses. In this case, you can use the prepare statement.
- For many kinds of operations, you may want a run-time value to determine which table to access. You can accomplish dynamic specification of a table in either of two ways:
 - Use the prepare statement; or
 - Use an SQL record and specify a value for the property **tableNameVariables**, as described in *SQL record part in EGL source format*.

Related concepts

- "SQL support" on page 171

Related reference

- "Database authorization and table names" on page 197
- "prepare" on page 339
- "SQL record part in EGL source format" on page 498

resultSetID

The result-set identifier is in the EGL syntax and is used when you are accessing a relational database and need to relate the following kinds of statements:

- First, an **open** or **get** statement that selects a result set, or an **open** statement that calls a stored procedure that returns a result set
- Second, the statements that access the result set

If you are using an SQL record as the I/O object, the record name is sufficient to relate one kind of statement to another, unless you modify the SQL statements associated with the record to retrieve different sets of columns for update in different statements. In this case, use a result-set identifier to identify the result set associated with an EGL **replace** statement.

Related concepts

"SQL support" on page 171

Related reference

"replace" on page 341

"open" on page 335

"get" on page 318

SQL examples

You can access an SQL data base in any of these ways:

- By hand-coding an SQL statement whose format is known at generation time.
- By using an SQL record as the I/O object of an EGL statement, when the format of the SQL statement is known at generation time--
 - If you place an explicit SQL statement in the EGL source, that SQL statement is used at run time;
 - Otherwise, an implicit SQL statement is used at run time.
- By coding an EGL **prepare** statement, which generates an SQL PREPARE statement that in turn creates an SQL statement at run time.

In every case, you can use an SQL record as a memory area and to provide a simple way to test for successful operation. The examples in this section assume that a record part is declared in an EGL file and that a record based on the part was declared in a program in that file:

- The SQL record part is as follows--

```
Record Employee type sqlRecord
{
    tableNames = employee,
    keyItems = empnum,
    defaultSelectCondition =
        #sql{ // no space between #sql and the brace
            aTableColumn = 4 -- start each SQL comment
                               -- with a double hyphen
        }
}

empnum decimal(6,0) {isReadOnly=yes};
empname char(40);
end
```

- The SQL record is as follows--

```
emp Employee;
```

For further details on SQL records and implicit statements, see SQL support.

Coding SQL statements

To prepare to code SQL statements, declare variables:

```
empnum decimal(6,0);
empname char(40);
```

Adding a row to an SQL table: To prepare to add a row, assign values to variables:

```
empnum = 1;
empname = "John";
```

To add the row, associate an EGL **execute** statement with an SQL INSERT statement as follows:

```
try
  execute
    #sql{
      insert into employee (empnum, empname)
      values (:empnum, :empname)
    };
onException
  myErrorHandler(8);
end
```

Reading rows from an SQL table: To prepare to read rows from an SQL table, identify a record key:

```
empnum = 1;
```

To get rows, code a series of EGL statements:

- To select a result set, run an EGL open statement--

```
open selectEmp
with #sql{
  select empnum, empname
  from employee
  where empnum >= :empnum
  for update of empname
}
into empnum, empname;
```

- To access the next row of the result set, run an EGL get next statement:

```
get next from selectEmp;
```

If you did not specify the into clause in the open statement, you need to specify the into clause in the get next statement; and if you specified the into clause in both places, the clause in the get next statement takes precedence:

```
get next from selectEmp
into empnum, empname;
```

The cursor is closed automatically when the last record is read from the result set.

The following code updates SQL rows:

```
handleHardIOErrors = 1;

try
  open selectEmp
  with #sql{
    select empnum, empname
    from employee
    where empnum >= :empnum
    for update of empname
  }
  into empnum, empname;
```



```

onException
  myErrorHandler(8);    // exits program
end

try
  get next from selectEmp;
onException
  if (sqlcode != 100)
    myErrorHandler(8); // exits program
  end
end

while (sqlcode != 100)
  empname = empname + " " + "III";

  try
    execute
      #sql{
        update employee
        set empname = :empname
        where current of selectEmp
      }
  onException
    myErrorHandler(10); // exits program
  end

  try
    get next from selectEmp;
  onException
    if (sqlcode != 100)
      myErrorHandler(8); // exits program
    end
  end
end // end while; cursor is closed automatically
// when the last row in the result set is read

```

Using SQL records with implicit SQL statements

To begin using EGL SQL records, declare an SQL record part:

```

Record Employee type sqlRecord
{
  tableNames = employee,
  keyItems = empnum,
  defaultSelectCondition =
    #sql{
      aTableColumn = 4 -- start each SQL comment
                      -- with a double hyphen
    }
}

empnum decimal(6,0) {isReadOnly=yes};
empname char(40);
end

```

Declare a record that is based on the record part:

```
emp Employee;
```

Adding a row to an SQL table: To prepare to add a row to an SQL table, place values in the EGL record:

```

emp.empnum = 1;
emp.empname = "John";

```

Add an employee to the table by specifying the EGL add statement:

```

try
  add emp;
onException
  myErrorHandler(8);
end

```

Reading rows from an SQL table: To prepare to read rows from an SQL table, identify a record key:

```
emp.empnum = 1;
```

Get a single row in either of these ways:

- Specify the EGL get statement in a way that generates a series of statements (DECLARE cursor, OPEN cursor, FETCH row, and in the absence of forUpdate, CLOSE cursor):

```

try
  get emp;
onException
  myErrorHandler(8);
end

```

- Specify the EGL get statement in a way that generates a single SELECT statement:

```

try
  get emp singleRow;
onException
  myErrorHandler(8);
end

```

Process multiple rows by using the EGL open and get next statements:

```

handleHardIOErrors = 1;

try
  open selectEmp forUpdate for emp;
onException
  myErrorHandler(8);  // exits program
end

try
  get next emp;
onException
  if (emp not noRecordFound)
    myErrorHandler(8); // exit the program
  end
end

while (emp not noRecordFound)
  myRecord.empname = myRecord.empname + " " + "III";

  try
    replace emp;
onException
  myErrorHandler(10); // exits program
end

  try
    get next emp;
onException
  if (emp not noRecordFound)
    myErrorHandler(8); // exits program
  end
end

```

```

end // end while; cursor is closed automatically
    // when the last row in the result set is read

sysLib.commit();

```

Using SQL records with explicit SQL statements

Before using SQL records with explicit SQL statements, you declare an SQL record part. This part is different from the previous one, in the syntax for SQL item properties and in the use of a calculated value:

```

Record Employee type sqlRecord
{
    tableNameVariables = empTable,
        // use of a table-name variable
        // means that the table is specified
        // at run time
    keyItems = empnum
}
empnum decimal(6,0) { isReadOnly = yes };
empname char(40);

// specify properties of a calculated column
aValue decimal(6,0)
{ isReadOnly = yes,
  column = "(empnum + 1) as NEWNUM" };
end

```

Declare variables:

```

emp Employee;
empTable char(40);

```

Adding a row to an SQL table: To prepare to add a row to an SQL table, place values in the EGL record and in a table name variable:

```

emp.empnum = 1;
emp.empname = "John";
empTable = "Employee";

```

Add an employee to the table by specifying the EGL add statement and modifying the SQL statement:

```

// a colon does not precede a table name variable
try
    add emp
        with #sql{
            insert into empTable (empnum, empname)
            values (:empnum, :empname || ' ' || 'Smith')
        }

onException
    myErrorHandler(8);
end

```

Reading rows from an SQL table: To prepare to read rows from an SQL table, identify a record key:

```

emp.empnum = 1;

```

Get a single row in any of these ways:

- Specify the EGL get statement in a way that generates a series of statements (DECLARE cursor, OPEN cursor, FETCH row, CLOSE cursor):

```

try
    get emp into empname // The into clause is optional. (It
                        // cannot be in the SELECT statement.)
    with #sql{

```

```

        select empname
        from empTable
        where empnum = :empnum + 1
    }
onException
    myErrorHandler(8);
end

```

- Specify the EGL get statement in a way that generates a single SELECT statement:

```

try
    get emp singleRow // The into clause is derived
                    // from the SQL record and is based
                    // on the columns in the select clause

    with #sql{
        select empname
        from empTable
        where empnum = :empnum + 1
    }
onException
    myErrorHandler(8);
end

```

Process multiple rows by using the EGL open and get next statements:

```

try

// The into clause is derived
// from the SQL record and is based
// on the columns in the select clause
open selectEmp forUpdate for emp
    with #sql{
        select empnum, empname
        from empTable
        where empnum >= :empnum
        order by NEWNUM      -- uses the calculated value
        for update of empname
    }
onException
    myErrorHandler(8);      // exits the program
end

try
    get next emp;
onException
    myErrorHandler(9);      // exits the program
end

while (emp not noRecordFound)
    try
        replace emp
        with #sql{
            update :empTable
            set empname = :empname || ' ' || 'III'
        }

onException
    myErrorHandler(10); // exits the program
end

try
    get next emp;
onException
    myErrorHandler(9); // exits the program
end
end // end while

```

```
// no need to say "close emp;" because emp
// is closed automatically when the last
// record is read from the result set or
// (in case of an exception) when the program ends

sysLib.commit();
```

Using EGL prepare statements

You have the option to use an SQL record part when coding the EGL prepare statement. Declare the following part:

```
Record Employee type sqlRecord
{
    tableNames = employee,
    keyItems = empnum,
    defaultSelectCondition =
        #sql{
            aTableColumn = 4 -- start each SQL comment
                           -- with a double hyphen
        }
}

empnum decimal(6,0) {isReadOnly=yes};
empname char(40);
end
```

Declare variables:

```
emp Employee;
empnum02 decimal(6,0);
empname02 char(40);
myString char(120);
```

Adding a row to an SQL table: Before adding a row, assign values to variables:

```
emp.empnum = 1;
emp.empname = "John";
empnum02 = 2;
empname02 = "Jane";
```

Develop the SQL statement:

- Code the EGL prepare statement and reference an SQL record, which provides an SQL statement that you can customize:

```
prepare myPrep
from "insert into employee (empnum, empname) " +
    "values (?, ?)" for emp;

// you can use the SQL record
// to test the result of the operation
if (emp is error)
    myErrorHandler(8);
end
```

- Alternatively, code the EGL prepare statement without reference to an SQL record:

```
myString = "insert into employee (empnum, empname) " +
    "values (?, ?)";

try
    prepare addEmployee from myString;
onException
    myErrorHandler(8);
end
```

In each of the previous cases, the EGL prepare statement includes placeholders for data that will be provided by an EGL execute statement. Two examples of the execute statement are as follows:

- You can provide values from a record (SQL or otherwise):

```
execute addEmployee using emp.empnum, emp.empname;
```
- You can provide values from individual items:

```
execute addEmployee using empnum02, empname02;
```

Reading rows from an SQL table: To prepare to read rows from an SQL table, identify a record key:

```
empnum02 = 2;
```

The next example replaces multiple rows:

```
myString = "select empnum, empname from employee " +
           "where empnum >= ? for update of empname";

try
    prepare selectEmployee from myString for emp;
onException
    myErrorHandler(8);    // exits the program
end

try
    open selectEmp with selectEmployee
        using empnum02
        into emp.empnum, emp.empname;
onException
    myErrorHandler(9);    // exits the program
end

try
    get next from selectEmp;
onException
    myErrorHandler(10);   // exits the program
end

while (emp not noRecordFound)

    emp.empname = emp.empname + " " + "III";

    try
        replace emp
        with #sql{
            update employee
            set empname = :empname
        }
        from selectEmp;
onException
    myErrorHandler(11); // exits the program
end

try
    get next from selectEmp;
onException
    myErrorHandler(12); // exits the program
end
end // end while; close is automatic when last row is read
```

SQL-specific tasks

Setting preferences for SQL database connections

You use the page for SQL database connections for these reasons:

- You can enable declaration-time and debug-time access to a database that is accessed outside of J2EE.
- Also, you can set a value for the build descriptor option `sqlJNDIName`, which specifies a name to which the default datasource is bound in the JNDI registry; for example, `java:comp/env/jdbc/MyDB`. That option is included in the build descriptor that is created for you in the following situation:
 - You use the EGL Web Project Wizard, as described in *Creating a project to work with EGL*; and
 - When working in that wizard, you request that a build descriptor be created.

Do as follows:

1. Click **Window > Preferences**
2. When a list of preferences is displayed, expand **EGL**, then click **SQL Database Connections**.
3. In the **Connection URL** field, type the URL used to connect to the database through JDBC:

- For IBM DB2 APP DRIVER for Windows, the URL is `jdbc:db2:dbName` (where *dbName* is the database name)
- For the Oracle JDBC thin client-side driver, the URL varies by database location. If the database is local to your machine, the URL is `jdbc:oracle:thin:dbName` (where *dbName* is the database name). If the database is on a remote server, the URL is `jdbc:oracle:thin:@host:port:dbName` (where *host* is the host name of the database server, *port* is the port number, and *dbName* is the database name)
- For the Informix JDBC NET driver, the URL is as follows (with the lines combined into one)--

```
jdbc:informix-sqli://host:port
/dbName:informixserver=servername;
user=userName;password=passWord
```

host

Name of the machine on which the database server resides

port

Port number

dbName

Database name

serverName

Name of the database server

userName

Informix user ID

passWord

Password associated with the user ID

4. In the **Database** field, type the name of the database.
5. In the **User ID** field, type the user ID for the connection.
6. In the **Password** field, type the password for the user ID.
7. In the **Database vendor type** field, select the database product and version that you are using for your JDBC connection.
8. In the **JDBC driver** field, select the JDBC driver that you are using for your JDBC connection.

9. In the **JDBC driver class** field, type the driver class for the driver you selected. For IBM DB2 APP DRIVER for Windows, the driver class is `COM.ibm.db2.jdbc.app.DB2Driver`; for the Oracle JDBC thin client-side driver, the driver class is `oracle.jdbc.driver.OracleDriver`; and for the Informix JDBC NET driver, the driver class is `com.informix.jdbc.IfxDriver`. For other driver classes, refer to the documentation for the driver.
10. In the **class location** field, type the fully qualified filename of the *.jar or *.zip file that contains the driver class. For IBM DB2 APP DRIVER for Windows, type the fully qualified filename to the db2java.zip file; for example, `d:\sql11ib\java\db2java.zip`. For the Oracle THIN JDBC DRIVER, type the fully qualified filename to the classes12.zip file; for example, `d:\0ra81\jdbc\lib\classes12.zip`. For other driver classes, refer to the documentation for the driver.
11. In the **Connection JNDI name** field, specify the database used in J2EE. The value is the name to which the datasource is bound in the JNDI registry; for example, `java:comp/env/jdbc/MyDB`. As noted earlier, this value is assigned to the option **sqlJNDIName** in the build descriptor that is constructed automatically for a given EGL Web project.
12. If you are accessing DB2 UDB and specify a value in the **Secondary authentication ID** field, the value is used in the SET CURRENT SQLID statement used by EGL at validation time. The value is case-sensitive.

You can clear or apply preference settings:

- To restore default values, click **Restore Defaults**.
- To apply preference settings without exiting the preferences dialog, click **Apply**.
- If you are finished setting preferences, click **OK**.

Related tasks

"Creating a project to work with EGL" on page 82

"Setting EGL preferences" on page 81

Related reference

"sqlJNDIName" on page 264

Setting preferences for SQL retrieve

At EGL declaration time, you can use the SQL retrieve feature to create an SQL record from the columns of an SQL table. For an overview, see *SQL support*.

To set preferences for the SQL retrieve feature, do as follows:

1. Click **Window > Preferences**, then expand **EGL** and click **SQL Retrieve**
2. Specify rules for naming each structure item that is created by the SQL retrieve feature:
 - a. To specify the case of the structure item name, click one of the following radio buttons:
 - **Do not change case** (the default) means that the case of the structure item name is the same as the case of the related table column name
 - **Change to lower case** means that the structure item name is a lower-case version of the table column name
 - **Change to lower case and capitalize first letter after underscore** also means that the structure item name is a lower-case version of the table column name, except that a letter in the structure item name is rendered in uppercase if, in the table column name, the letter immediately follows an underscore

- b. To specify how the underscores in the table column name are reflected in the structure item name, click one of the following radio buttons:
 - **Do not change underscores** (the default) means that underscores in the table column name are included in the structure item name
 - **Remove underscores** means that underscores in the table column name are not included in the structure item name
 - **Change underscores to hyphens** means that underscores in the table column name are rendered as hyphens in the structure item name
 3. If you intend to retrieve data from a table that is part of an Informix system schema, clear the check box for **Exclude system schemas**. (In this case, "Informix" is the table owner.) In all other cases, select the check box to improve the performance of the SQL retrieve feature.
The check box is selected by default.

Related concepts

"SQL support" on page 171

Related tasks

"Retrieving SQL table data"

"Setting EGL preferences" on page 81

"Setting preferences for SQL database connections" on page 188

Related reference

"Informix and EGL" on page 199

Retrieving SQL table data

EGL provides a way to create SQL record items from the definition of an SQL table, view, or join; for an overview, see *SQL support*.

Do as follows when you are working in the EGL or EGL Web perspective:

1. Ensure that you have set SQL preferences as appropriate. For details, see and *Setting preferences for SQL retrieve*.
2. Decide where to do the task--
 - In an EGL source file, as you develop each SQL record; or
 - In the Outline view, as may be easier when you already have SQL records.
3. If you are working in the EGL source file, you can proceed in this way--
 - a. If you do not have the SQL record, create it:
 - 1) Type **R**, press Ctrl-Space, and in the content-assist list, select one of the SQL table entries (usually **SQL record with table names**).
 - 2) Type the name of the SQL record; press Tab; and type a table name, or a comma-delimited list of tables, or the alias of a view.

You also can create an SQL record by typing the minimal content, as appropriate if the name of the record is the same as the name of the table, as in this example:

```
Record myTable type sqlRecord
end
```
 - b. Right-click anywhere in the record.
 - c. In the context menu, click **SQL record > Retrieve SQL**.
4. If you are working in the Outline view, right click on the entry for the SQL record and, in the context menu, click **Retrieve SQL**.

Note: You cannot retrieve an SQL view that is defined with the DB2 condition `WITH CHECK OPTIONS`.

After you create structure items, you may want to gain a productivity benefit by creating the equivalent data item parts; see *Overview on creating data item parts from an SQL record part*.

Related concepts

“Creating data item parts from an SQL record part (overview)”
“SQL support” on page 171

Related tasks

“Creating data item parts from an SQL record part” on page 193
“Setting preferences for SQL database connections” on page 188
“Setting preferences for SQL retrieve” on page 190

Related reference

“SQL item properties” on page 57

Creating data item parts from an SQL record part (overview)

After you declare structure items in an SQL record part, you can use a special mechanism in the EGL editor to create data item parts that are equivalent to the structure items. The benefit is that you can more easily create a non-SQL record (usually a basic record) for transferring data to and from the related SQL record at run time.

Consider the following structure items:

```
10 myHostVar01 CHAR(3);  
10 myHostVar02 BIN(9,2);
```

You can request that data item parts be created:

```
DataItem myHostVar01 CHAR(3) end  
  
DataItem myHostVar02 BIN(9,2) end
```

Another effect is that the structure item declarations are rewritten:

```
10 myHostVar01 myHostVar01;  
10 myHostVar02 myHostVar02;
```

As shown in this example, each data item part is given the same name as the related structure item and acts as a typedef for the structure item. Each data item part is also available as a typedef for other structure items.

Before you can use a structure item as the basis of a data item part, the structure item must have a name, must have valid primitive characteristics, and must not point to a typedef.

Related concepts

“SQL support” on page 171

Related tasks

“Creating data item parts from an SQL record part” on page 193

Related reference

“DataItem part in EGL source format” on page 284
“SQL record part in EGL source format” on page 498

Creating data item parts from an SQL record part: After you declare structure items in an SQL record part, you can use a special mechanism in the EGL editor to create data item parts that are equivalent to the structure items. For general information, see *Overview on creating data item parts from an SQL record part*.

By default, the Outline view is open in the EGL perspective. If you are working in the EGL Web perspective, open the Outline view by selecting **Show View > Outline** from the Window menu. Do as follows in the Outline view:

1. For a given SQL record part, hold down **Ctrl** while clicking on each of the structure items of interest. To select all the structure items in a given record, click the topmost structure item, then hold down **Shift** while clicking on the bottommost structure item.
2. Right-click on the selected structure items.
3. In the context menu, click **Create Data Item**.

The data-item parts are written at the bottom of the EGL source file, and each structure item is changed to refer to the equivalent part.

Related concepts

“Creating data item parts from an SQL record part (overview)” on page 192
“SQL support” on page 171

Related tasks

“Retrieving SQL table data” on page 191

Related reference

“SQL record part in EGL source format” on page 498

Viewing the SQL SELECT statement for an SQL record

EGL provides an implicit SQL SELECT statement for a given SQL record part. To view the implicit SQL SELECT statement, do as follows:

1. Open the EGL file that contains the SQL record part. If you do not have the file open, right-click on the EGL file in the Project Navigator, then select **Open With > EGL Editor**.
2. Click inside the SQL record part, then right-click. A context menu displays.
3. Select **SQL Record > View Default Select**.
4. To validate the SQL SELECT statement against a database, click **Validate**.

Note: Before using the validate function, DB2 UDB users must set the DEFERREDPREPARE option. You can set this option interactively in the CLP (DB2 command line processor) using the **db2 update cli cfg for section COMMON using DEFERREDPREPARE 0** command. This command will put the keyword under the COMMON section. Execute the command **db2 get cli cfg for section common** to verify that the keyword is being picked up.

Related concepts

“SQL support” on page 171

Related tasks

“Validating the SQL SELECT statement for an SQL record” on page 194

Related reference

“SQL record part in EGL source format” on page 498

Validating the SQL SELECT statement for an SQL record

EGL provides an implicit SQL SELECT statement for a given SQL record part. To validate the implicit SQL SELECT statement against a database, do as follows:

1. Open the EGL file that contains the SQL record part. If you do not have the file open, right-click on the EGL file in the Project Navigator, then select **Open With > EGL Editor**.
2. Click inside the SQL record part, then right-click. A context menu displays.
3. Select **SQL Record > Validate Default Select**.

Note: Before using the validate function, DB2 UDB users must set the DEFERREDPREPARE option. You can set this option interactively in the CLP (DB2 command line processor) using the **db2 update cli cfg for section COMMON using DEFERREDPREPARE 0** command. This command will put the keyword under the COMMON section. Execute the command **db2 get cli cfg for section common** to verify that the keyword is being picked up.

Related concepts

"SQL support" on page 171

Related tasks

"Viewing the SQL SELECT statement for an SQL record" on page 193

Related reference

"SQL record part in EGL source format" on page 498

Constructing an SQL PREPARE statement

Within a function, you can construct an SQL PREPARE statement based on an SQL record part and an SQL statement coded in a literal. To construct an SQL PREPARE statement, do as follows:

1. Open an EGL file with the EGL editor. The file must contain a function and a coded SQL statement. If you do not have a file open, right-click on the EGL file in the workbench Project Navigator, then select **Open With > EGL Editor**.
2. Click inside the function at the location where the SQL PREPARE statement will reside, then right-click. A context menu displays.
3. Select **Add SQL Prepare Statement**.
4. Type a name to identify the SQL PREPARE statement. For rules, see *Naming conventions*.
5. If you have an SQL record variable defined, select it from the drop-down list. The corresponding SQL record part name displays. If you do not have an SQL record variable defined, you can type a name in the SQL record variable name field, then select an SQL record part name using the **Browse** button. You must eventually define an SQL record variable with that name in the EGL source code.
6. Select an execution statement type from the drop-down list.
7. If the execution statement is of type open, enter a result-set identifier.
8. Click **OK**. An EGL **prepare** statement and the appropriate EGL **open**, **get**, or **execute** statement is constructed inside the function.

Related concepts

"SQL support" on page 171

Related tasks

"Validating the SQL SELECT statement for an SQL record" on page 194

"Viewing the SQL SELECT statement for an SQL record" on page 193

Related reference

"Naming conventions" on page 468

"SQL record part in EGL source format" on page 498

Constructing an explicit SQL statement from an implicit one

EGL provides an implicit SQL statement for each SQL-related EGL input/output (I/O) statement. To construct an explicit SQL statement from an implicit one, do as follows:

1. Open the EGL file that contains the EGL I/O statement. If you do not have the file open, right-click on the EGL file in the Project Navigator, then select **Open With > EGL Editor**.
2. Click on the EGL I/O statement, then right-click. A context menu displays.
3. To construct an explicit SQL statement without an INTO clause, select **SQL Statement > Add**. To construct an explicit SQL statement with an INTO clause, select **SQL Statement > Add with Into**. The implicit SQL statement is appended to the EGL I/O statement making it an explicit SQL statement.

Note: The INTO clause is only valid with **open**, **get**, and **get next** statements.

Related concepts

"SQL support" on page 171

Related tasks

"Removing an SQL statement from an SQL-related EGL statement" on page 196

"Resetting an explicit SQL statement" on page 196

"Validating an implicit or explicit SQL statement"

"Viewing the implicit SQL for an SQL-related EGL statement"

Viewing the implicit SQL for an SQL-related EGL statement

EGL provides an implicit SQL statement for each SQL-related EGL input/output (I/O) statement. To view the implicit SQL for an EGL I/O statement, do as follows:

1. Open the EGL file that contains the EGL I/O statement. If you do not have the file open, right-click on the EGL file in the Project Navigator, then select **Open With > EGL Editor**.
2. Click on the EGL I/O statement, then right-click. A context menu displays.
3. Select **SQL Statement > View**.

Related concepts

"SQL support" on page 171

Related tasks

"Constructing an explicit SQL statement from an implicit one"

"Removing an SQL statement from an SQL-related EGL statement" on page 196

"Resetting an explicit SQL statement" on page 196

"Validating an implicit or explicit SQL statement"

Validating an implicit or explicit SQL statement

To validate an implicit or explicit SQL statement against a database, do as follows:

1. Open the EGL file that contains the SQL-related EGL statement or explicit SQL statement. If you do not have the file open, right-click on the EGL file in the Project Navigator, then select **Open With > EGL Editor**.
2. Click the EGL statement or SQL statement, then right-click. A context menu displays.
3. Select **SQL Statement > Validate**.

Note: Before using the validate function, DB2 UDB users must set the DEFERREDPREPARE option. You can set this option interactively in the CLP (DB2 command line processor) using the **db2 update cli cfg for section COMMON using DEFERREDPREPARE 0** command. This command will put the keyword under the COMMON section. Execute the command **db2 get cli cfg for section common** to verify that the keyword is being picked up.

Related concepts

"SQL support" on page 171

Related tasks

"Constructing an explicit SQL statement from an implicit one" on page 195

"Removing an SQL statement from an SQL-related EGL statement"

"Resetting an explicit SQL statement"

"Viewing the implicit SQL for an SQL-related EGL statement" on page 195

Removing an SQL statement from an SQL-related EGL statement

EGL provides an implicit SQL statement for each SQL-related EGL input/output (I/O) statement. An implicit SQL statement can be appended to an EGL I/O statement making it an explicit SQL statement (see *Constructing an explicit SQL statement from an implicit one*). To remove the appended SQL statement, do as follows:

1. Open the EGL file that contains the explicit SQL statement. If you do not have the file open, right-click on the EGL file in the Project Navigator, then select **Open With > EGL Editor**.
2. Click on the explicit SQL statement, then right-click. A context menu displays.
3. Select **SQL Statement > Remove**. The EGL I/O statement remains.

Related concepts

"SQL support" on page 171

Related tasks

"Constructing an explicit SQL statement from an implicit one" on page 195

"Resetting an explicit SQL statement"

"Validating an implicit or explicit SQL statement" on page 195

"Viewing the implicit SQL for an SQL-related EGL statement" on page 195

Resetting an explicit SQL statement

EGL provides an implicit SQL statement for each SQL-related EGL input/output (I/O) statement. An implicit SQL statement can be appended to an EGL I/O statement making it an explicit SQL statement. If you change the explicit SQL statement, do as follows to return to an SQL statement based on the implicit SQL:

1. Open the EGL file that contains the explicit SQL statement. If you do not have the file open, right-click on the EGL file in the Project Navigator, then select **Open With > EGL Editor**.
2. Click on the explicit SQL statement, then right-click. A context menu displays.

3. Select **SQL Statement** > **Reset**.

Related concepts

"SQL support" on page 171

Related tasks

"Constructing an explicit SQL statement from an implicit one" on page 195

"Removing an SQL statement from an SQL-related EGL statement" on page 196

"Validating an implicit or explicit SQL statement" on page 195

"Viewing the implicit SQL for an SQL-related EGL statement" on page 195

Resolving a reference to display an implicit SQL statement

Consider what happens when you specify the following EGL statement:

```
open myRecord;
```

When the EGL editor tries to create a default SQL statement, the editor attempts to find a variable named myRecord and to identify the SQL record part on which that variable is based. If the variable is unavailable at development time or if the variable is undeclared, the editor attempts to use an SQL record part named myRecord as the basis for the default SQL statement. The editor assumes that you intend to create a variable whose name is the name of the SQL record part.

If you wish to store an SQL-related function in a file that does not include the variable myRecord, you can do as follows:

1. In the program part, declare the global variable
2. Create the function as a nested function in the program part
3. Create the default SQL statement and modify it as appropriate; then, save the file
4. Move the function to the other file

After the function is moved from the program part, the record name cannot be resolved at development time, and the editor cannot display any default SQL statements that are based on that record.

Related concepts

"SQL support" on page 171

Database authorization and table names

An authorization ID is a character string that is passed to the database manager when a connection is established between the database manager and a program, whether the program prepares another program or allows end-user access to SQL tables. The character string is the user identifier that is required to check the database-access authorization held by the preparer or end user.

The source of the authorization ID depends on the system where database access occurs.

For iSeries COBOL programs, the authorization ID is the user ID under which the programs runs.

The situation for EGL-generated Java programs is as follows:

- The authorization ID is one obtained by the database manager when a connection was established between the database manager and the program:

- In relation to the default database, the authorization ID is the value specified for the Java run-time property **vgj.jdbc.default.userid**
- When you invoke the system function **sysLib.connect** or **sysLib.connectionService**, the authorization ID is the value specified for the **userID** parameter

The authorization ID may be used when you specify a table name. In that case, you can specify a table-name qualifier, in accordance with this syntax:

tableOwner.myTable

tableOwner

A qualifier that is known to the database manager and that is necessary to identify the table. The qualifier at table creation is the authorization ID of the person who created the table.

myTable

The table name.

If you do not include a table-name qualifier when you specify a table name, the table owner is resolved at run time. The qualifier is set to the authorization ID.

For more information on authorization IDs, consult your database manager documentation.

Related concepts

“Dynamic SQL” on page 180

“Java run-time properties” on page 421

“SQL support” on page 171

Related reference

“Java run-time properties (details)” on page 423

“SQL record part in EGL source format” on page 498

“sysLib.connect” on page 543

“sysLib.connectionService” on page 545

Default database

The default database is a relational database that is accessed when an SQL-related I/O statement runs in EGL-generated code and when no other database connection is current. The default database is available from the beginning of a run unit; however, you can dynamically connect to a different database for subsequent access in the same run unit, as described in *sysLib.connectionService*.

In relation to a Java run unit, the default database is specified in the optional run-time property **vgj.jdbc.default.database**, which receives a generated value from build descriptor option **sqlDB** if option **genProperties** is set to GLOBAL or PROGRAM at generation time.

In relation to an iSeries COBOL program, the idea of a database connection is not meaningful. The build descriptor option **destLibrary** identifies the library used at run time, and if you wish to reference another library, you must use the name of that other library as a qualifier. For instance, if you need to access the SQL table **SAMPLE** in library **LIBRARY02**, you would refer to the table as **LIBRARY02.SAMPLE**.

Related concepts

"Java run-time properties" on page 421

"Run unit" on page 504

"SQL support" on page 171

Related tasks

"Setting up a J2EE JDBC connection" on page 209

"Setting up the J2EE run-time environment for EGL-generated code" on page 206

"Understanding how a standard JDBC connection is made" on page 208

Related reference

"destLibrary" on page 248

"Java run-time properties (details)" on page 423

"sqlDB" on page 262

"sysLib.connectionService" on page 545

Informix and EGL

The following rules are specific to Informix databases and EGL:

- An Informix database that is accessed by EGL or by an EGL-generated program must have transactions enabled.
- If you are using the SQL retrieve feature of EGL to access data from a non-ANSI Informix database, make sure that any database column of type DECIMAL includes a scale value. Instead of defining a column as DECIMAL (4), for example, define the column as DECIMAL (4,0).
- If you intend to use the SQL retrieve feature to retrieve data from a table that is part of an Informix system schema, you must set a special preference, as described in *Setting preferences for SQL retrieve*.

Related concepts

"SQL support" on page 171

Related tasks

"Retrieving SQL table data" on page 191

"Setting preferences for SQL retrieve" on page 190

SQL record internals

You need to be aware of the internal layout of an SQL record in any of these situations:

- You use an EGL assignment statement to copy an SQL record to or from a record of a different type
- The run-time argument passed to an EGL program is an SQL record, but the program parameter is not an SQL record
- The run-time argument passed to an EGL function is an SQL record; in this case, the parameter must be a working storage record
- You receive an SQL record as a parameter in a non-EGL program

Four bytes precede each structure item in an SQL record. The first two bytes are a null indicator, and a null is interpreted as any negative value. The second two bytes are reserved for use as a length field, and you should *not* access that field.

If you are generating a COBOL program, the name of an SQL record is at the 01 level, and all structure items are at the next lowest level.

Related concepts

“Function part” on page 380

“Program part” on page 477

“SQL support” on page 171

Related reference

“Assignments” on page 296

SQL data codes and EGL host variables

The property **SQL data code** identifies the SQL data type to associate with the EGL host variable. The data code is used by the database management system at declaration time, validation time, or generated-program run time.

You may want to vary the SQL data code for a host variable that is of primitive type CHAR, DBCHAR, HEX, or UNICODE. For a host variable of one of the other primitive types, however, SQL data codes are fixed.

If EGL retrieved a column definition from the database management system, do not modify the SQL data code that was retrieved, if any.

The next sections cover these topics:

- “Variable and fixed-length columns”
- “Compatibility of SQL data types and EGL primitive types”
- “VARCHAR, VARGRAPHIC, and the related LONG data types” on page 202
- “DATE, TIME, and TIMESTAMP” on page 202

Variable and fixed-length columns

To indicate that a table column is variable length or fixed length, set the SQL data code for the corresponding host variable to the appropriate value, as shown in the next table.

EGL primitive type	SQL data type	Variable or fixed	SQL data code
CHAR	CHAR (the default)	Fixed	453
	VARCHAR, length < 255	Variable	449
	VARCHAR, length > 254	Variable	457
DBCHAR, UNICODE	GRAPHIC (the default)	Fixed	469
	VARGRAPHIC, length < 128	Variable	465
	VARGRAPHIC, length > 127	Variable	473

Note: A SQL data type may require the use of null indicators, but this requirement has no effect on how you code an EGL program. For details on nulls, see *SQL support*.

Compatibility of SQL data types and EGL primitive types

An EGL host variable and the corresponding SQL table column are compatible in any of the following situations:

- The SQL column is any form of character data, and the EGL host variable is of type CHAR with a length less than or equal to the length of the SQL column.
- The SQL column is any form of DBCHAR data, and the EGL host variable is of type DBCHAR with a length less than or equal to the length of the SQL column.

- The SQL column is any form of number and the EGL host variable is of either of these types:
 - BIN, with 2 or 4 bytes and no decimal places.
 - DECIMAL, with a maximum length of 18 digits, including decimal places. The number of digits for a DECIMAL variable should be the same for the EGL host variable and for the column.
 - SMALLINT.
 - The SQL column is of any data type, the EGL host variable is of type HEX, and the column and host variable contain the same number of bytes. No data conversion occurs during data transfer.
- EGL host variables of type HEX support access to any SQL column of a data type that does not correspond to an EGL primitive type.

If character data is read from an SQL table column into a shorter host variable, content is truncated on the right. To test for truncation, use the reserved word **trunc** in an EGL **if** statement.

If numeric data is read from an SQL table column into a shorter host variable, leading zeros are truncated on the left. If the number still does not fit into the host variable, fractional parts of the number (in decimal) are deleted on the right, with no indication of error. If the number still does not fit, a negative SQL code is returned to indicate an overflow condition.

The next table shows the EGL host variable characteristics that are assigned when the retrieve feature of the EGL editor extracts information from a database management system.

SQL data type	EGL host variable characteristics			SQL data code (SQLTYPE)
	Primitive type	Length	Number of bytes	
BIGINT	HEX	16	8	493
CHAR	CHAR	1–32767	1–32767	453
DATE	CHAR	10	10	453
DECIMAL	DECIMAL	1-18	1–10	485
DOUBLE	HEX	16	8	481
FLOAT	HEX	16	8	481
GRAPHIC	DBCHAR	1–16383	2–32766	469
INTEGER	BIN	9	4	497
LONG VARBINARY	HEX	65534	32767	481
LONG VARCHAR	CHAR	>4000	>4000	457
LONG VARGRAPHIC	DBCHAR	>2000	>4000	473
NUMERIC	DECIMAL	1-18	1–10	485
REAL	HEX	8	4	481
SMALLINT	BIN	4	2	501
TIME	CHAR	8	8	453
TIMESTAMP	CHAR	26	26	453
VARBINARY	HEX	2–65534	1–32767	481

SQL data type	EGL host variable characteristics			SQL data code (SQLTYPE)
	Primitive type	Length	Number of bytes	
VARCHAR	CHAR	≤4000	≤4000	449
VARGRAPHIC	DBCHAR	≤2000	≤4000	465

Columns with the following SQL data types cannot be accessed in a generated COBOL program because an equivalent COBOL data type does not exist:

- 460, 461: a null-terminated character string
- 476, 477: a varying-length character string, as used in Pascal

VARCHAR, VARGRAPHIC, and the related LONG data types

The definition of an SQL table column of type VARCHAR or VARGRAPHIC includes a maximum length, and the retrieve command uses that maximum to assign a length to the EGL host variable. The definition of an SQL table column of type LONG VARCHAR or VARGRAPHIC, however, does not include a maximum length, and the retrieve command uses the SQL-data-type maximum to assign a length.

DATE, TIME, and TIMESTAMP

Make sure that the format used for the EGL system default long Gregorian format is the same as the date format specified for the SQL database manager. For details on how the EGL format is set, see *sysVar.currentFormattedDate*.

You want the two formats to match so that the dates provided by the system variable *sysVar.currentFormattedDate* are in the format expected by the SQL database manager.

Related concepts

“SQL support” on page 171

Related reference

“SQL item properties” on page 57

“sysVar.currentFormattedDate” on page 522

VSAM support

EGL-generated COBOL code can access local or remote VSAM files. VSAM support for EGL-generated Java code is as follows:

- AIX-based code can access local VSAM files
- The following code can access remote VSAM files on z/OS:
 - EGL-generated Java code that runs on Windows 2000/NT/XP
 - The EGL debugger, which runs on Windows 2000/NT/XP

Access prerequisites

Access requires that you first define the VSAM file on the system where you want the file to reside. Remote access from Windows 2000/NT/XP (whether for the EGL debugger or at run time) also requires that you install Distributed File Manager (DFM) on the workstation as follows:

1. Locate the following file in your EGL installation directory:

workbench\bin\VSAMWIN.zip

2. Unzip the file into a new directory and follow the directions in the INSTALL.README file.

System name

To access a local VSAM file, specify the system name in the resource associations part and use the naming convention that is appropriate to the operating system. To access a remote VSAM file from the EGL debugger or from EGL-generated Java code, specify the system name in the following way:

`\\machineName\qualifier.fileName`

machineName

The SNA LU alias name as specified in the SNA configuration

qualifier.fileName

The VSAM data set name, including a qualifier

The naming convention is similar to the Universal Naming Convention (UNC) format. For details on UNC format, refer to the *Distributed FileManager User's Guide*, which is in the following file in your EGL installation directory:

`workbench\bin\VSAMWIN.zip`

Deploying EGL-generated Java output

Installing the EGL run-time code for Java

The EGL run-time code for generated Java applications is packaged on disk1 of the WebSphere Studio Site Developer (WSSD) install CD and on disk4 of the WebSphere Studio Application Developer (WSAD) install CD. The run-time code is found in the `redist\EGLRuntimes` directory and its subdirectories for each of the distributed platforms supported for Java applications. The supported distributed platforms are AIX, Linux (Intel[™]), iSeries, and Windows 2000/NT/XP. (See product prerequisites for supported versions.)

Jar files containing Java code common to all supported distributed platforms are contained in the `redist\EGLRuntimes` directory. Copy these jar files to each machine on which deployed EGL applications are to be run outside of a J2EE application server. (These files are already included in any Enterprise Archive (EAR) file used to deploy J2EE applications.) Include the `fda.jar` and `fdaj.jar` files in the classpath of the deployment machines.

There is a subdirectory for each supported distributed platform: AIX for AIX platforms, Linux for Linux platforms, iSeries for iSeries platforms, and Win32 for Windows 2000/NT/XP platforms. Each of the platform subdirectories has a `bin` subdirectory that contains non-Java, platform-dependent code. Copy all of the files in the `bin` directory for the deployment platform to a directory on each deployment machine, then set environment variables for that platform as follows:

AIX

Add the directory into which the non-Java code was copied to the `PATH` and `LIBPATH` environment variables

Linux

Add the directory into which the non-Java code was copied to the `PATH` and `LIBPATH` environment variables

iSeries

In `qshell`, change to the directory you just uploaded the files to and run the `setup.sh` script with the `"install"` option:

```
> setup.sh install
```

In addition, some other environment variables must be set. For information on how to set these environment variables, run the script with the `"envinfo"` option:

```
> setup.sh envinfo
```

If for some reason you delete a symlink that is created for you during install, you can recreate it with the `"link"` option:

```
> setup.sh link
```

Windows 2000/NT/XP

Add the directory into which the non-Java code was copied to the `PATH` environment variable

Related tasks

"Deploying Java applications outside of J2EE" on page 228

Setting up the J2EE run-time environment for EGL-generated code

EGL-generated Java programs and wrappers run on a J2EE 1.3 server such as WebSphere Application Server (WAS) 5.1, on the platforms listed in *Run-time configurations*.

Your primary tasks when embedding generated Java classes into a J2EE module are as follows:

1. Place output files in a project, in either of two ways:
 - Generate into a project, as is the preferred technique; or
 - Generate into a directory, then import files into a project.
2. Place a linkage properties file in the module (see *Deploying a linkage properties file*).
3. Eliminate duplicate jar files.
4. Export an enterprise archive (.ear) file, which may include Web application archive (.war) files and other .ear files; for details on the procedure, see the help pages on export.
5. Import the .ear file into the Web application server that will host your application; for details on the procedure, see the documentation for your Web application server.

You may need to fulfill these tasks as well:

- “Setting up a J2EE JDBC connection” on page 209
- “Setting up the J2EE server for CICSJ2C calls” on page 213
- “Setting up the TCP/IP listener” on page 210

Related concepts

- “Development process” on page 1
- “Java program, page handler, and library” on page 394
- “Linkage options part” on page 439
- “Linkage properties file” on page 404
- “Run-time configurations” on page 2

Related tasks

- “Deploying Java applications outside of J2EE” on page 228
- “Deploying a linkage properties file” on page 223
- “Eliminating duplicate jar files” on page 219
- “Generating deployment code for EJB projects” on page 227
- “Generating into a directory” on page 220
- “Generating into a project” on page 214
- “Providing access to non-EGL jar files” on page 217
- “Setting deployment-descriptor values” on page 207
- “Setting the JNDI name for EJB projects” on page 228
- “Setting up a J2EE JDBC connection” on page 209
- “Setting up the J2EE server for CICSJ2C calls” on page 213
- “Setting up the TCP/IP listener” on page 210
- “Understanding how a standard JDBC connection is made” on page 208
- “Updating the deployment descriptor manually” on page 216
- “Updating the J2EE environment file” on page 224

Related reference

- “Java run-time properties (details)” on page 423
- “Linkage properties file (details)” on page 431

Setting deployment-descriptor values

An important task is to place run-time values (similar to environment-variable values) into the deployment descriptor of your J2EE module. You can interact with a workbench editor listed in the next table, for example; and in any case, the editors are available if you wish to reassign a value.

Project type	Name of deployment descriptor	How to assign values
application client	application-client.xml	Use the XML editor, Design tab
EJB	ejb-jar.xml	Use the EJB editor, Beans tab
J2EE Web	web.xml	Use the web.xml editor, Environment tab

The recommended way to update the deployment descriptor is to add content automatically, as happens if all of the following conditions apply:

- You are generating a Java program or wrapper
- The build descriptor option **genProperties** is set to GLOBAL or PROGRAM
- You are generating for J2EE run time by setting **J2EE** to YES
- You set **genProject** to a valid J2EE project

EGL never deletes a property from an existing deployment descriptor, but does as follows:

- Overwrites properties that already exist
- Appends properties that do not exist

Another method of updating the deployment descriptor is to paste values from the J2EE environment file, which is an output of generation if all of the following conditions apply:

- You are generating a Java program
- The build descriptor option **genProperties** is set to GLOBAL or PROGRAM
- You are generating for J2EE run time by setting **J2EE** to YES
- You do not set **genProject** to a valid J2EE project, as when you generate into a directory instead

Before you paste entries from a J2EE environment file into the deployment descriptor of an application client or EJB project, you need to change the order of entries in the file, as described in *Updating the J2EE environment file*. You do not need to change the order of entries if you are working with a J2EE Web project.

For details on deployment descriptor properties, see *Java run-time properties (details)*.

Related concepts

“J2EE environment file” on page 394

“Program properties file” on page 404

Related tasks

“Generating into a directory” on page 220

“Generating into a project” on page 214

“Setting up the J2EE run-time environment for EGL-generated code” on page 206

"Updating the J2EE environment file" on page 224
"Updating the deployment descriptor manually" on page 216

Related reference

"genDirectory" on page 251
"genProperties" on page 254
"J2EE" on page 255
"Java run-time properties (details)" on page 423

Understanding how a standard JDBC connection is made

A standard JDBC connection is created for you at run time if you are debugging a generated Java program and if the program properties file includes the necessary values. For details on the meaning of the program properties, including details on how the values are derived, see *Java run-time properties (details)*.

The JDBC connection is based on the following kinds of information:

Connection URL

If your code tries to access a database before invoking the system function `sysLib.connect` or `sysLib.connectionService`, the connection URL is the value of property **`vgj.jdbc.default.database`**.

If your code tries to access a database in response to an invocation of the system function `sysLib.connect` or `sysLib.connectionService`, the connection URL is the value of property **`vgj.jdbc.databaseSN`**.

For details on the format of a connection URL, see *sqlValidationConnectionURL*.

User ID

If your code tries to access a database before invoking the system function `sysLib.connect` or `sysLib.connectionService`, the user ID is the value of property **`vgj.jdbc.default.userid`**.

If your code tries to access a database in response to an invocation of one of those system functions, the user ID is a value specified in the invocation.

Password

If your code tries to access a database before invoking the system function `sysLib.connect` or `sysLib.connectionService`, the password is the value of property **`vgj.jdbc.default.password`**.

If your code tries to access a database in response to an invocation of one of those system functions, the password is a value specified in the invocation. You can use a system function to avoid exposing the password in the program properties file.

JDBC driver class

The JDBC driver class is the value of property **`vgj.jdbc.drivers`**.

Related concepts

"Program properties file" on page 404

Related tasks

"Setting up a J2EE JDBC connection" on page 209
"Setting up the J2EE run-time environment for EGL-generated code" on page 206

Related reference

"`sysLib.connect`" on page 543
"`sysLib.connectionService`" on page 545

"genProperties" on page 254
"Java run-time properties (details)" on page 423
"sqlDB" on page 262
"sqlID" on page 263
"sqlPassword" on page 264
"sqlValidationConnectionURL" on page 265
"sqlJDBCClass" on page 263

Setting up a J2EE JDBC connection

If you are connecting to a relational database at run time, follow the directions in the IBM WebSphere Administrator's Console task-oriented help pages, which tell how to define a data source for use with your program. When you define a data source, assign values to the following properties:

Name

Specify a name for your data source.

JNDI name

Specify a value that matches the name to which the database is bound in the JNDI registry:

- If you are defining a data source which connects to a database that your J2EE module uses by default, make sure that the JNDI name specified in the data source definition matches the value of the **vgj.jdbc.default.database** property in the J2EE deployment descriptor used at run time
- If you are defining a data source that will be accessed when the system function `sysLib.connectionService` runs, make sure that the JNDI name specified in the data source definition matches the value of the appropriate **vgj.jdbc.database.SN** property in the J2EE deployment descriptor used at run time

Database name

Specify the name of your database, as known to the database management system

User ID

Specify the user name for connecting to the database. If the data source definition refers to the default database, the value you specify in the User ID field is overridden by any value set in the **vgj.jdbc.default.userid** property of the J2EE deployment descriptor used at run time, but only if you have specified values for both **vgj.jdbc.default.userid** and **vgj.jdbc.default.password**. Similarly, if the data source definition refers to a database that is accessed by way of the system function `sysLib.connect` or `sysLib.connectionService`, the value you specify in the User ID field is overridden by any user ID that you specify in the call to that system function, but only if the call passes both a user ID and password.

Password

Specify the password for connecting to the database. If the data source definition refers to the default database, the value you specify in the Password field is overridden by any value set in the **vgj.jdbc.default.password** property of the J2EE deployment descriptor used at run time, but only if you have specified values for both **vgj.jdbc.default.userid** and **vgj.jdbc.default.password**. Similarly, if the data source definition refers to a database that is accessed by way of the system function `sysLib.connectionService`, the value you specify in the Password field is overridden by any password that you specify in the call to that system function, but only if the call passes both a user ID and password.

You may define multiple data sources, in which case you use the system function `sysLib.connectionService` to switch between them.

For details on the meaning of the deployment descriptor properties, including details on how the generated values are derived, see *Java run-time properties (reference)*.

Related tasks

“Setting up the J2EE run-time environment for EGL-generated code” on page 206

“Understanding how a standard JDBC connection is made” on page 208

Related reference

“`sysLib.connectionService`” on page 545

“Java run-time properties (details)” on page 423

Setting up the TCP/IP listener

If you want a caller to use TCP/IP to exchange data with a called program in a J2EE application client module or with a called non-J2EE Java program, you must set up a TCP/IP listener for the called program.

TCP/IP for a called non-J2EE Java program

If you are using TCP/IP to communicate with a called non-J2EE Java program, you must configure a standalone Java program called `CSOTcpipListener` for that program. Specifically, you must do as follows:

- Make sure that the classpath used when running `CSOTcpipListener` contains `fda.jar`, `fdaj.jar`, and the directories or archives that contain the called programs; and
- Set the Java run-time property `tcpiplistener.port` to the number of the port at which `CSOTcpipListener` receives data.

You can start the standalone TCP/IP listener in either of two ways:

- To start the listener from the workbench, use the launch configuration for a Java application. In this case, you can specify the name of the properties file in the program arguments of the launch configuration. Alternatively, if you are using the file `tcpiplistener.properties` as a default, that file should not be in a folder, but should be directly under the project that you specified when you created the launch configuration.
- To start the listener from the command line, run the program as follows:

```
java CSOTcpipListener propertiesFile
```

propertiesFile

The fully qualified path to the properties file used by the TCP/IP listener. If you do not specify a properties file, the listener attempts to open the following file in the current directory:

```
tcpiplistener.properties
```

TCP/IP for a called program in a J2EE application client module

In the case of a called program in a J2EE application client module, you need to make sure that the following situation is in effect:

- An EGL-specific TCP/IP listener is the main class for the module, as specified in the manifest (.MF) file of the module
- A port is assigned to the listener, as specified in the deployment descriptor (`application-client.xml`) of the module

If you are working with projects at the level of J2EE 1.2, it is recommended that you set up an application client project that is initialized with the listener, before you generate any EGL code into that project. If you fail to follow that sequence (listener first, EGL code second) or if you are working with projects at the level of J2EE 1.3, you need to follow the procedure described in *Providing access to the listener from an existing application client project*.

Setting up an application client project that is initialized with the listener: To set up an application client project that is initialized with the listener, do as follows:

1. Click **File > Import**.
2. At the Select page, double-click **App Client JAR file**.
3. At the Application Client Import page, specify several details--
 - a. In the Application Client file field, specify the jar file that sets up access to (but does not include) the TCP/IP listener:
`installationDir\wstools\eclipse\plugins\com.ibm.etools.egl.generators_version\runtime\EGLTcpListener.jar`
installationDir
The WebSphere Studio installation directory, such as c:\myStudio
version
The latest version of the plugin; for example, 5.0.1.
The TCP/IP listener itself resides in fdaj.jar, which is placed in the application client project when you first generate EGL code into that project.
 - b. Click the **New** radio button, which follows the label **Application Client project**.
 - c. Type the name of the application client project into the **New Project Name** field; then set or unset the **Use default** check box. If you set the check box, the project is stored in a workspace directory that is named with the name of the project. If you unset the check box, specify the project name in the **New project location** field.
 - d. Specify the name of the enterprise application project that contains the application client project:
 - If you are using an existing J2EE 1.2 enterprise application project, click the **Existing** radio button, which follows the label **Enterprise application project**. In this case, specify the project name in the **Existing project name** field.
 - If you are creating a new enterprise application project, do as follows:
 - 1) Click the **New** radio button, which follows the label **Enterprise application project**.
 - 2) Type the name of the enterprise application project into the **New Project Name** field.
 - 3) Set or unset the **Use default** check box.
 - 4) If you set the check box, the project is stored in a workspace directory that is named with the name of the project. If you unset the check box, specify the project name in the **New project location** field.
4. Click **Finish**.
5. Ignore the two warning messages that refer to the jar files (fda.jar, fdaj.jar) that will be added automatically when you generate EGL output into the project.

In the application client project, the deployment descriptor property **tcpiplistener.port** is set to the number of the port at which the listener receives data. By default, that port number is 9876. To change the port number, do as follows:

1. In the Project Navigator view, expand your application client project, then `appClientModule`, then `META-INF`
2. Click **application-client.xml > Open With > Deployment Descriptor editor**
3. The deployment descriptor editor includes a source tab; click that tab and change the 9876 value, which is the content of the last tag in a grouping like this:

```
<env-entry-name>tcpiplistener.port</env-entry-name>
<env-entry-type>java.lang.Integer</env-entry-name>
<env-entry-value>9876</env-entry-value>
```

4. To save the deployment descriptor, press **Ctrl-S**.

Providing access to the listener from an existing application client project: If you generate EGL code into an application client project that was not initialized with the listener, you need to update the deployment descriptor (`application-client.xml`) and the manifest file (`MANIFEST.MF`):

1. In the Project Navigator view, expand your application client project, then `appClientModule`, then `META-INF`
2. Click **application-client.xml > Open With > Deployment Descriptor Editor**
3. The deployment descriptor editor includes a Source tab. Click that tab. In the text, immediately below the line that holds the tag `<display-name>`, add the following entries (however, if port 9876 is already in use on your machine, substitute a different number for 9876):

```
<env-entry>
  <env-entry-name>tcpiplistener.port</env-entry-name>
  <env-entry-type>java.lang.Integer</env-entry-name>
  <env-entry-value>9876</env-entry-value>
</env-entry>
```

4. To save the deployment descriptor, press **Ctrl-S**.
5. In the Project Navigator view, click **MANIFEST.MF > Open With > JAR Dependency Editor**.
6. The JAR Dependency Editor includes a Dependencies tab. Click that tab.
7. Review the Dependencies section to make sure that `fda.jar` and `fdaj.jar` are selected.
8. In the Main Class section, in the Main-Class field, type the following value or use the Browse mechanism to specify the following value:
`CS0TcpiListenerJ2EE`
9. To save the manifest file, press **Ctrl-S**.

Deploying the application client project: To start the TCP/IP listener, follow either of two procedures:

- Start the listener from the Workbench by using the launch configuration for a WebSphere application client:
 1. Switch to an EGL or J2EE perspective
 2. Click **Run > Run**
 3. At the Launch Configurations page, click either **WebSphere v5 Application Client** (as is necessary if you are working with a project at the level of J2EE 1.3) or **WebSphere v4 Application Client**

4. Select an existing configuration. Alternatively, click **New** and set up a configuration:

- a. In the Application tab, select the enterprise application project
- b. In the Arguments tab, add an argument:

`-CCjar=myJar.jar`

myJar.jar

The name of the application client jar file. This argument is only necessary when you have multiple client jar files in the ear file. In most cases, the value is the name of the application client project, followed by the extension .jar.

If you wish to confirm the relationship of project name to jar-file name, do as follows:

- 1) In the Project Navigator view, expand your enterprise application project, then META-INF
- 2) Click **application.xml > Open With > Deployment Descriptor Editor**.
- 3) The Deployment Descriptor editor includes a Module tab. Click that tab.
- 4) At the leftmost part of the page, click the jar file and see (at the rightmost part of the page) the project name associated with that jar file.

- If you have on WebSphere Application Server (WAS) installed, you can use launchClient.bat, which is in the WAS installation directory, subdirectory bin.

You can invoke launchClient as follows from a command prompt:

```
launchClient myCode.ear -CCjar=myJar.jar
```

myCode.ear

The name of the enterprise archive

myJar.jar

The name of the application client jar file, as described in relation to the Workbench procedure

For details on launchClient.bat, see the WebSphere Application Server documentation.

Related tasks

“Providing access to non-EGL jar files” on page 217

“Setting up the J2EE run-time environment for EGL-generated code” on page 206

Setting up the J2EE server for CICSJ2C calls

You must set up a ConnectionFactory in the J2EE server for each CICS transaction accessed through protocol CICSJ2C.

If a generated Java wrapper is making the CICSJ2C call, you can handle security in any of the following ways (where a wrapper-specified value overrides that of the J2EE server):

- Set the userid and password in the wrapper’s CSOCallOptions object; or
- Set the userid and password in the ConnectionFactory configuration in the J2EE server; or
- Set up the CICS region so that user authentication is not required.

When calling a program from WebSphere 390, the following restrictions apply:

- If the callLink element property **luwControl** is set to CLIENT, the call fails. The WebSphere 390 connect implementation does not support an extended unit of work.
- The setting of deployment descriptor property **cso.cicsj2c.timeout** has no effect. By default, timeouts never occur. In the EXCI options table generated by the macro DFHXCOPT, however, you can set the parameter TIMEOUT, which lets you specify the time that EXCI will wait for a DPL command (an ECI request) to complete. A setting of 0 means to wait indefinitely.

For details, see *Java Connectors for CICS: Featuring the J2EE Connector Architecture* (SG24-6401-00), which is available from web site <http://www.redbooks.ibm.com>.

Related tasks

“Setting up the J2EE run-time environment for EGL-generated code” on page 206

Generating into a project

If you are generating a Java program or wrapper, it is recommended (and in some cases required) that you set build descriptor option **genProject**, which causes generation into a project. If you are generating a COBOL program, however, you must use build descriptor option **genDirectory**, which generates output into a directory.

For details on the relationship of **genDirectory** and **genProject**, see *genProject*.

EGL provides various services for you when you generate into a project. The services vary by project type, as do your next tasks:

Application client project

When you generate into an application client project, EGL does as follows:

- Provides preparation-time access to EGL jar files (fda.jar and fdaj.jar) by adding the following entries to the project’s Java build path:

```
EGL_GENERATORS_PLUGINDIR/runtime/fda.jar
EGL_GENERATORS_PLUGINDIR/runtime/fdaj.jar
```

For details on the variable at the beginning of each entry, see *Setting the variable EGL_GENERATORS_PLUGINDIR*.

- Provides run-time access to the EGL jar files:
 - Imports the jar files into each enterprise application project that references the application client project
 - Updates the manifest in the application client project so that the jar files in an enterprise application project are available
- Puts run-time values into the deployment descriptor so that you can avoid cutting and pasting entries from a generated J2EE environment file; for an overview of this subject, see *Setting deployment-descriptor values*

Your next tasks are as follows:

1. If you are calling the generated program by way of TCP/IP, provide run-time access to a listener, as described in *Setting up the TCP/IP listener*
2. Provide access to non-EGL jar files
3. Now that you have placed output files in a project, continue setting up the J2EE run-time environment

EJB project

When you generate into an EJB project, EGL does as follows:

- Provides preparation-time access to EGL jar files (fda.jar and fdaj.jar) by adding the following entries to the project's Java build path:

```
EGL_GENERATORS_PLUGINDIR/runtime/fda.jar
EGL_GENERATORS_PLUGINDIR/runtime/fdaj.jar
```

For details on the environment variable at the beginning of each entry, see *Setting the variable EGL_GENERATORS_PLUGINDIR*.

- Provides run-time access to the EGL jar files:
 - Imports fda.jar and fdaj.jar into each enterprise application project that references the EJB project
 - Updates the manifest in the EJB project so that fda.jar and fdaj.jar in an enterprise application project are available at run time
- Assigns the JNDI name automatically so that the EGL run-time code can access the EJB code; but this step occurs only when you generate an EJB session bean.
- In most cases, puts run-time values into the deployment descriptor so that you can avoid cutting and pasting entries from a generated J2EE environment file; for an overview of this subject, see *Setting deployment-descriptor values*.

EGL does not put run-time values into the deployment descriptor if EGL cannot find the necessary session element in the deployment descriptor. This situation occurs, for example, when the Java program is generated before the wrapper or when the build descriptor option **sessionBeanID** is set to a value that is not found in the deployment descriptor. For details on session elements, see *sessionBeanID*.

Your next tasks are as follows:

1. Provide access to non-EGL jar files
2. Generate deployment code
3. Now that you have placed output files in a project, continue setting up the J2EE run-time environment

J2EE Web project

EGL does as follows:

- Provides access to EGL jar files by importing fda.jar and fdaj.jar into the project's Web Content/WEB-INF/lib folder
- Puts run-time values into the deployment descriptor so that you can avoid cutting and pasting entries from a generated J2EE environment file; for an overview of this subject, see *Setting deployment-descriptor values*

Your next tasks are as follows:

1. Providing access to non-EGL jar files
2. Now that you have placed output files in a project, continue Setting up the J2EE run-time environment

Java project

If you are generating into a non-J2EE Java project for debugging or production purposes, EGL does as follows:

- Provides access to EGL jar files (fda.jar and fdaj.jar) by adding the following entries to the project's Java build path:

```
EGL_GENERATORS_PLUGINDIR/runtime/fda.jar
EGL_GENERATORS_PLUGINDIR/runtime/fdaj.jar
```

For details on the variable at the beginning of each entry, see *Setting the variable EGL_GENERATORS_PLUGINDIR*.

- Generates a program properties file and places it in the top-level project folder, but only if the build descriptor includes the following option values:
 - **genProperties** is set to GLOBAL or PROGRAM; and
 - **J2EE** is set to NO.

At run time, values in the program properties file are used to set up a standard JDBC connection. For details, see *Understanding how a standard JDBC connection is made*.

Now that you have placed output files in a project, do as follows:

- If your program accesses a relational database, make sure that your Java build path includes the directory where the driver is installed. For DB2, for example, specify the directory that contains db2java.zip.
- If your code accesses MQSeries, provide access to non-EGL jar files
- Place a linkage properties file in the module

For details on the consequence of generating into a non-existent project, see *genProject*.

Related tasks

“Generating deployment code for EJB projects” on page 227

“Deploying a linkage properties file” on page 223

“Setting deployment-descriptor values” on page 207

“Providing access to non-EGL jar files” on page 217

“Setting the variable EGL_GENERATORS_PLUGINDIR” on page 219

“Setting up the J2EE run-time environment for EGL-generated code” on page 206

“Understanding how a standard JDBC connection is made” on page 208

Related reference

“genDirectory” on page 251

“genProject” on page 252

“sessionBeanID” on page 259

Updating the deployment descriptor manually

If you are updating a deployment descriptor from a generated J2EE environment file, do as follows:

1. Read the overview information in *Setting deployment-descriptor values*.
2. If you worked on an application client or EJB project, you must make sure that the order of the sub-elements in the generated environment entries is correct, as described in *Updating the J2EE environment file*.
3. Copy the environment entries into your project’s deployment descriptor as follows:
 - a. Make a backup copy of the deployment descriptor.
 - b. Open the J2EE environment file, which is called *programName-env.txt* file. Copy the environment entries into the clipboard.
 - c. Double-click on the deployment descriptor.
 - d. Click on the Source tab.
 - e. Paste the entries at a proper location.

For details on deployment descriptors, see *Java run-time properties (reference)*.

Related tasks

“Setting deployment-descriptor values” on page 207

“Setting up the J2EE run-time environment for EGL-generated code” on page 206
“Updating the J2EE environment file” on page 224

Related reference

“Java run-time properties (details)” on page 423

Providing access to non-EGL jar files

You may need to provide access to non-EGL jar files to debug and run your EGL-generated Java code. The process for providing access to those files varies by project type:

Application client project

Before using the interpretive debugger, reference the non-EGL jar files in the CLASSPATH variable, as described in *Setting preferences for the EGL debugger*.

Before running your code (with or without the EGL Java debugger), do as follows:

1. For each enterprise application project that references the application client project, import jar files of interest from a directory in the file system:
 - a. In the Project Navigator view, right-click an enterprise application project and click **Import**
 - b. At the Select page, click **File System**
 - c. At the File System page, specify the directory in which the jar files reside
 - d. At the right of the page, select the jar files of interest
 - e. Click **Finish**
2. Update the manifest in the application client project so that the jar files in the enterprise application project are available at run time:
 - a. In the Project Navigator view, right-click your application client project and click **Properties**
 - b. At the left of the Properties page, click **Java JAR Dependencies**
 - c. When the section called Java JAR Dependencies is displayed at the right of the page, set each check box that corresponds to a jar file of interest
 - d. Click **OK**

EJB project

Before using the interpretive debugger, reference the non-EGL jar files in the CLASSPATH variable, as described in *Setting preferences for the EGL debugger*.

Before running your code (with or without the EGL Java debugger), do as follows:

1. For each enterprise application project that references the EJB project, import jar files of interest from a directory in the file system:
 - a. In the Project Navigator view, right-click an enterprise application project and click **Import**
 - b. At the Select page, click **File System**
 - c. At the File System page, specify the directory in which the jar files reside
 - d. At the right of the page, select the jar files of interest
 - e. Click **Finish**
2. Update the manifest in the EJB project so that the jar files in the enterprise application project are available at run time:

- a. In the Project Navigator view, right-click your EJB project and click **Properties**
- b. At the left of the Properties page, click **Java JAR Dependencies**
- c. When the section called Java JAR Dependencies is displayed at the right of the page, set each check box that corresponds to a jar file of interest
- d. Click **OK**

Java project

Before running your code with the interpretive debugger, reference the non-EGL jar files in the CLASSPATH variable, as described in *Setting preferences for the EGL debugger*.

Before running your code with the EGL Java debugger, add entries to the project's Java build path:

1. In the Project Navigator view, right-click your Java project and click **Properties**
2. At the left of the Properties page, click **Java Build Path**
3. When the section called Java Build Path is displayed at the right of the page, click the Libraries tab
4. For each jar file to be added, click **Add External Jars** and use the Browse mechanism to select the file
5. To close the Properties page, click **OK**

J2EE Web project

Before using the interpretive debugger, reference the non-EGL jar files in the CLASSPATH variable, as described in *Setting preferences for the EGL debugger*.

Before running your code (with or without the EGL Java debugger), import the jar files from the file system to the following Web project folder:

Web Content/WEB-INF/lib

The import process is as follows for a set of jar files in a directory:

1. In the Project Navigator view, expand the Web project, expand **Web Content**, expand **WEB-INF**, right-click **lib**, and click **Import**
2. At the Select page, click **File System**
3. At the File System page, specify the directory in which the jar files reside
4. At the right of the page, select the jar files of interest
5. Click **Finish**

The following jar-file requirements are in effect:

- A generated Java program that accesses MQSeries in any way requires MQ Series Classes for Java; in particular, the Java program needs the following jar files (although not at preparation time):
 - com.ibm.mq.jar
 - com.ibm.mqbind.jar

If you have WebSphere MQ V5.2, the software is in IBM WebSphere MQ SupportPac™ MA88, which you can find by going to the IBM web site (www.ibm.com) and searching for MA88. Download and install the software; then you can access the jar files from the Java\lib subdirectory of the directory where you installed that software.

If you have WebSphere MQ V5.3, you can get the equivalent software by doing a custom install and selecting Java Messaging. Then you can access the jar files from the Java\lib subdirectory of the MQSeries installation directory.

- A generated Java program or wrapper that uses the protocol CICSJ2C to access CICS for z/OS requires access to connector.jar and cicsj2ee.jar, but only at run time. Those files are available to you when you install the CICS Transaction Gateway.

Note: Access of CICS is possible when the EGL Java debugger runs in J2EE. Calls to CICS are attempted but fail, however, when that debugger runs outside of J2EE or when you are using the EGL interpretive debugger, which always runs outside of J2EE.

- A generated Java program that accesses an SQL table requires a file that is installed with the database management system--

- For DB2 UDB, the file is one of the following:

```
sqllib\java\db2java.zip
sqllib\java\db2jcc.jar
```

The second of those files is available with DB2 UDB Version 8 or higher, as described in the DB2 UDB documentation.

- For Informix, the files are as follows:

```
ifxjdbc.jar
ifxjdbcx.jar
```

- For Oracle, consult the Oracle documentation.

The database file is required at run time, and can be used to validate SQL statements at preparation time.

Related tasks

“Setting preferences for the EGL debugger” on page 114

“Setting up the J2EE run-time environment for EGL-generated code” on page 206

Eliminating duplicate jar files

If you place multiple J2EE modules into a single ear file, eliminate the duplicate jar files as follows:

1. Move a copy of each duplicate jar file to the top level of the ear
2. Delete the duplicate jar files from the J2EE modules
3. Ensure that the build path for each of the affected J2EE modules points to the jar files in the ear; specifically, do as follows for each of those J2EE modules:
 - a. Right-click on the module from within the Project Navigator or J2EE view
 - b. Select **Edit Module Dependencies**
 - c. When the Module Dependencies dialog is displayed, select the jar files to access from the top level of the ear, then click **Finish**.

Related tasks

“Setting up the J2EE run-time environment for EGL-generated code” on page 206

Setting the variable EGL_GENERATORS_PLUGINDIR

The workbench classpath variable EGL_GENERATORS_PLUGINDIR contains the fully qualified path to the EGL plug-in in the Workbench. The variable is used in the Java build path when you generate an EGL program into a project of type Java, application client, or EJB.

If you encounter a classpath error that refers to EGL_GENERATORS_PLUGINDIR, the variable may not be set. The problem occurs, for example, if you check out an EGL-related project from a software configuration management system like Concurrent Versions System (CVS) before you ever work with an EGL part.

You can set the variable by creating an EGL part, by generating EGL code, or by following these steps:

1. Select **Window**, then **Preferences**
2. On the Preferences page, select **Java**, then **Classpath Variables**
3. Select **New...**
4. At the New Variable Entry page, type **EGL_GENERATORS_PLUGINDIR** and specify the following directory:

installationDir\wstools\eclipse\plugins\
com.ibm.etools.egl.generators_*version*

installationDir

The WebSphere Studio installation directory, such as c:\myStudio

version

The installed version of the plugin; for example, 5.0.1

After you set the variable, rebuild the project.

Related tasks

“Generating into a project” on page 214

Related reference

“genProject” on page 252

Generating into a directory

This page describes how to process Java code that is generated into a directory. It is recommended, however, that you avoid generating Java code into a directory; instead, follow the procedures described in *Generating into a project*.

For details on generating a COBOL program, see *Generating for COBOL*.

To generate Java code into a directory, specify the build descriptor option **genDirectory** and avoid specifying the build descriptor option **genProject**.

Your next tasks depend on the project type:

Application client project

For an application client project, do as follows:

1. Provide preparation-time access to EGL jar files by adding the following entries to the project’s Java build path:

EGL_GENERATORS_PLUGINDIR/runtime/fda.jar
EGL_GENERATORS_PLUGINDIR/runtime/fdaj.jar

For details on the variable at the beginning of each entry, see *Setting the variable EGL_GENERATORS_PLUGINDIR*.

2. Provide run-time access to fda.jar, fdaj.jar, and (if you are calling the generated program by way of TCP/IP) EGLTcipListener.jar:

- Access the jar files from the following directory:

installationDir\wstools\eclipse\plugins\
com.ibm.etools.egl.generators_*version*\runtime

installationDir

The WebSphere Studio installation directory, such as c:\myStudio

version

The installed version of the plugin; for example, 5.1.2

Copy those files into each enterprise application project that references the application client project.

- Update the manifest in the application client project so that the jar files (as stored in an enterprise application project) are available.
3. Provide access to non-EGL jar files (an optional task)
 4. Import your generated output into the project, in keeping with these rules:
 - The folder *appClientModule* must include the top-level folder of the package that contains your generated output
 - The hierarchy of folder names beneath *appClientModule* must match the name of your Java package

If you are importing generated output from package *my.trial.package*, for example, you must import that output into a folder that resides in the following location:

`appClientModule/my/trial/package`

5. If you generated a J2EE environment file, update that file
6. Update the deployment descriptor
7. Now that you have placed output files in a project, continue setting up the J2EE run-time environment

EJB project

For an EJB project, do as follows:

1. Provide preparation-time access to EGL jar files (fda.jar and fdaj.jar) by adding the following entries to the project's Java build path:

`EGL_GENERATORS_PLUGINDIR/runtime/fda.jar`
`EGL_GENERATORS_PLUGINDIR/runtime/fdaj.jar`

For details on the variable at the beginning of each entry, see *Setting the variable EGL_GENERATORS_PLUGINDIR*.

2. Provide run-time access to the EGL jar files:

- Access fda.jar and fdaj.jar from the following directory:

`installationDir\wstools\eclipse\plugins\`
`com.ibm.etools.egl.generators_version\runtime`

installationDir

The WebSphere Studio installation directory, such as `c:\myStudio`

version

The installed version of the plugin; for example, 5.1.2

Copy those files into each enterprise application project that references the EJB project.

- Update the manifest in the EJB project so that fda.jar and fdaj.jar (as stored in an enterprise application project) are available.
3. Provide access to non-EGL jar files (an optional task)
 4. Import your generated output into the project, in keeping with these rules:
 - The folder *ejbModule* must include the top-level folder of the package that contains your generated output
 - The hierarchy of folder names beneath *ejbModule* must match the name of your Java package

If you are importing generated output from package *my.trial.package*, for example, you must import that output into a folder that resides in the following location:

`ejbModule/my/trial/package`

5. If you generated a J2EE environment file, update that file.
6. Update the deployment descriptor
7. Set the JNDI name
8. Generate deployment code
9. Now that you have placed output files in a project, continue setting up the J2EE run-time environment

J2EE Web project

For a Web project, do as follows:

1. Provide access to EGL jar files by copying `fda.jar` and `fdaj.jar` into your Web project folder. To do so, import the external jars found in the following directory:

```
installationDir\wstools\eclipse\plugins\
com.ibm.etools.egl.generators_version\runtime
```

installationDir

The WebSphere Studio installation directory, such as `c:\myStudio`

version

The installed version of the plugin; for example, 5.0.1

The destination for the files is the following project folder:

```
WebContent/WEB-INF/lib
```

2. Provide access to non-EGL jar files (an option)
3. Import your generated output into the project, in keeping with these rules:
 - The folder *WebContent* must include the top-level folder of the package that contains your generated output
 - The hierarchy of folder names beneath *WebContent* must match the name of your Java package

If you are importing generated output from package *my.trial.package*, for example, you must import that output into a folder that resides in the following location:

```
WebContent/my/trial/package
```

4. Update the deployment descriptor
5. Now that you have placed output files in a project, continue setting up the J2EE run-time environment

Java project

If you want to debug in a non-J2EE environment, you need an EGL-generated program properties file, which identifies various values that are specific to the program. The file is generated if you use the following combination of build descriptor options:

- **genProperties** is set to GLOBAL or PROGRAM; and
- **J2EE** is set to NO.

At run time, values in the program properties file are used to set up a standard JDBC connection. For details, see *Understanding how a standard JDBC connection is made*.

For a Java project, your tasks are as follows:

1. Provide access to EGL jar files by adding the following entries to the project's Java build path:

```
EGL_GENERATORS_PLUGIN_DIR/runtime/fda.jar
EGL_GENERATORS_PLUGIN_DIR/runtime/fdaj.jar
```


For details on the variable at the beginning of each entry, see *Setting the variable EGL_GENERATORS_PLUGIN_DIR*.

2. If your program accesses a relational database, make sure that your Java build path includes the directory where the driver is installed. For DB2, for example, specify the directory that contains db2java.zip.
3. If your generated code accesses MQSeries, provide access to non-EGL jar files
4. Place the program properties file in the top-level project folder
5. Place a linkage properties file in the project (an optional task)

Related tasks

"Generating deployment code for EJB projects" on page 227

"Generating for COBOL" on page 132

"Generating into a project" on page 214

"Deploying a linkage properties file"

"Setting deployment-descriptor values" on page 207

"Providing access to non-EGL jar files" on page 217

"Setting the JNDI name for EJB projects" on page 228

"Setting the variable EGL_GENERATORS_PLUGIN_DIR" on page 219

"Setting up the J2EE run-time environment for EGL-generated code" on page 206

"Understanding how a standard JDBC connection is made" on page 208

"Updating the deployment descriptor manually" on page 216

"Updating the J2EE environment file" on page 224

Related reference

"genDirectory" on page 251

"genProject" on page 252

Deploying a linkage properties file

The linkage properties file must be in the same project as the Java program that uses the file. If the file is in the top-level directory of the project, set the Java run-time property `so.linkageOptions.LO` to the file name, without path information. If the file is under the top-level directory of the project, use a path that starts at the top-level directory and includes a virgule (/) for each level, even if the project is running on a Windows platform.

For J2EE projects, the top-level directory corresponds to the `appClientModule`, `ejbModule`, or `Web Content` directory of the project in which the module resides. Similarly for Java projects, the top-level directory is the project directory.

For additional details on how a linkage properties file is formatted and identified, see *Linkage properties file (reference)*.

Related concepts

"Java run-time properties" on page 421

"Linkage options part" on page 439

"Linkage properties file" on page 404

Related tasks

"Setting up the J2EE run-time environment for EGL-generated code" on page 206

"Setting deployment-descriptor values" on page 207

Related reference

"callLink element" on page 443

“Exception handling” on page 75

“Linkage properties file (details)” on page 431

Updating the J2EE environment file

The J2EE environment file contains a series of entries like the following example:

```
<env-entry>
  <env-entry-name>vgj.nls.code</env-entry-name>
  <env-entry-value>ENU</env-entry-value>
  <env-entry-type>java.lang.String</env-entry-type>
</env-entry>
```

The order of sub-elements is name, value, type. This is correct for J2EE Web projects; however, for application client and EJB projects, you need to change the order to name, type, value. For the example above, change the order of the sub-elements to:

```
<env-entry>
  <env-entry-name>vgj.nls.code</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>ENU</env-entry-value>
</env-entry>
```

This step can be avoided if you generate directly into a project instead of into a directory. When you generate into a project, EGL can determine the type of project you are using and generate the environment entries in the appropriate order.

Related tasks

“Setting up the J2EE run-time environment for EGL-generated code” on page 206

“Setting deployment-descriptor values” on page 207

Related reference

“Java run-time properties (details)” on page 423

Updating the deployment descriptor manually

If you are updating a deployment descriptor from a generated J2EE environment file, do as follows:

1. Read the overview information in *Setting deployment-descriptor values*.
2. If you worked on an application client or EJB project, you must make sure that the order of the sub-elements in the generated environment entries is correct, as described in *Updating the J2EE environment file*.
3. Copy the environment entries into your project’s deployment descriptor as follows:
 - a. Make a backup copy of the deployment descriptor.
 - b. Open the J2EE environment file, which is called *programName-env.txt* file. Copy the environment entries into the clipboard.
 - c. Double-click on the deployment descriptor.
 - d. Click on the Source tab.
 - e. Paste the entries at a proper location.

For details on deployment descriptors, see *Java run-time properties (reference)*.

Related tasks

“Setting deployment-descriptor values” on page 207

“Setting up the J2EE run-time environment for EGL-generated code” on page 206

“Updating the J2EE environment file”

Related reference

"Java run-time properties (details)" on page 423

Providing access to non-EGL jar files

You may need to provide access to non-EGL jar files to debug and run your EGL-generated Java code. The process for providing access to those files varies by project type:

Application client project

Before using the interpretive debugger, reference the non-EGL jar files in the CLASSPATH variable, as described in *Setting preferences for the EGL debugger*.

Before running your code (with or without the EGL Java debugger), do as follows:

1. For each enterprise application project that references the application client project, import jar files of interest from a directory in the file system:
 - a. In the Project Navigator view, right-click an enterprise application project and click **Import**
 - b. At the Select page, click **File System**
 - c. At the File System page, specify the directory in which the jar files reside
 - d. At the right of the page, select the jar files of interest
 - e. Click **Finish**
2. Update the manifest in the application client project so that the jar files in the enterprise application project are available at run time:
 - a. In the Project Navigator view, right-click your application client project and click **Properties**
 - b. At the left of the Properties page, click **Java JAR Dependencies**
 - c. When the section called Java JAR Dependencies is displayed at the right of the page, set each check box that corresponds to a jar file of interest
 - d. Click **OK**

EJB project

Before using the interpretive debugger, reference the non-EGL jar files in the CLASSPATH variable, as described in *Setting preferences for the EGL debugger*.

Before running your code (with or without the EGL Java debugger), do as follows:

1. For each enterprise application project that references the EJB project, import jar files of interest from a directory in the file system:
 - a. In the Project Navigator view, right-click an enterprise application project and click **Import**
 - b. At the Select page, click **File System**
 - c. At the File System page, specify the directory in which the jar files reside
 - d. At the right of the page, select the jar files of interest
 - e. Click **Finish**
2. Update the manifest in the EJB project so that the jar files in the enterprise application project are available at run time:
 - a. In the Project Navigator view, right-click your EJB project and click **Properties**
 - b. At the left of the Properties page, click **Java JAR Dependencies**

- c. When the section called Java JAR Dependencies is displayed at the right of the page, set each check box that corresponds to a jar file of interest
- d. Click **OK**

Java project

Before running your code with the interpretive debugger, reference the non-EGL jar files in the CLASSPATH variable, as described in *Setting preferences for the EGL debugger*.

Before running your code with the EGL Java debugger, add entries to the project's Java build path:

1. In the Project Navigator view, right-click your Java project and click **Properties**
2. At the left of the Properties page, click **Java Build Path**
3. When the section called Java Build Path is displayed at the right of the page, click the Libraries tab
4. For each jar file to be added, click **Add External Jars** and use the Browse mechanism to select the file
5. To close the Properties page, click **OK**

J2EE Web project

Before using the interpretive debugger, reference the non-EGL jar files in the CLASSPATH variable, as described in *Setting preferences for the EGL debugger*.

Before running your code (with or without the EGL Java debugger), import the jar files from the file system to the following Web project folder:

Web Content/WEB-INF/lib

The import process is as follows for a set of jar files in a directory:

1. In the Project Navigator view, expand the Web project, expand **Web Content**, expand **WEB-INF**, right-click **lib**, and click **Import**
2. At the Select page, click **File System**
3. At the File System page, specify the directory in which the jar files reside
4. At the right of the page, select the jar files of interest
5. Click **Finish**

The following jar-file requirements are in effect:

- A generated Java program that accesses MQSeries in any way requires MQ Series Classes for Java; in particular, the Java program needs the following jar files (although not at preparation time):
 - com.ibm.mq.jar
 - com.ibm.mqbind.jar

If you have WebSphere MQ V5.2, the software is in IBM WebSphere MQ SupportPac MA88, which you can find by going to the IBM web site (www.ibm.com) and searching for MA88. Download and install the software; then you can access the jar files from the Java\lib subdirectory of the directory where you installed that software.

If you have WebSphere MQ V5.3, you can get the equivalent software by doing a custom install and selecting Java Messaging. Then you can access the jar files from the Java\lib subdirectory of the MQSeries installation directory.

- A generated Java program or wrapper that uses the protocol CICSJ2C to access CICS for z/OS requires access to connector.jar and cicsj2ee.jar, but only at run time. Those files are available to you when you install the CICS Transaction Gateway.

Note: Access of CICS is possible when the EGL Java debugger runs in J2EE. Calls to CICS are attempted but fail, however, when that debugger runs outside of J2EE or when you are using the EGL interpretive debugger, which always runs outside of J2EE.

- A generated Java program that accesses an SQL table requires a file that is installed with the database management system--

- For DB2 UDB, the file is one of the following:

```
sqllib\java\db2java.zip  
sqllib\java\db2jcc.jar
```

The second of those files is available with DB2 UDB Version 8 or higher, as described in the DB2 UDB documentation.

- For Informix, the files are as follows:

```
ifxjdbc.jar  
ifxjdbcx.jar
```

- For Oracle, consult the Oracle documentation.

The database file is required at run time, and can be used to validate SQL statements at preparation time.

Related tasks

“Setting preferences for the EGL debugger” on page 114

“Setting up the J2EE run-time environment for EGL-generated code” on page 206

Eliminating duplicate jar files

If you place multiple J2EE modules into a single ear file, eliminate the duplicate jar files as follows:

1. Move a copy of each duplicate jar file to the top level of the ear
2. Delete the duplicate jar files from the J2EE modules
3. Ensure that the build path for each of the affected J2EE modules points to the jar files in the ear; specifically, do as follows for each of those J2EE modules:
 - a. Right-click on the module from within the Project Navigator or J2EE view
 - b. Select **Edit Module Dependencies**
 - c. When the Module Dependencies dialog is displayed, select the jar files to access from the top level of the ear, then click **Finish**.

Related tasks

“Setting up the J2EE run-time environment for EGL-generated code” on page 206

Setting up EJB projects (additional tasks)

Generating deployment code for EJB projects

After you generate into an EJB project and specify the deployment descriptor properties, you can generate the stubs and skeletons that allow for remote access of the EJB:

1. In the Project Navigator, right-click on the project name; then click **Generate > Deploy and RMIC code**
2. Follow the directions specified in the help page on Generating EJB deployment code from the workbench

Related tasks

“Setting up the J2EE run-time environment for EGL-generated code” on page 206

Setting the JNDI name for EJB projects

To set the JNDI name for an EJB project, do as follows:

1. Right click on ejb-jar.xml (the deployment descriptor) to open the context menu.
2. Use the EJB Editor to open the following file in the project--
`\ejbModule\META-INF\ejb-jar.xml`
3. Click on the Beans tab.
4. On the list, click on the name of the EJB you just generated.
5. Enter the JNDI name under WebSphere Bindings. The JNDI name must be as follows for use by the EGL run-time code:
 - First character of the program name, in upper case
 - Subsequent characters of the program name, in lower case
 - The letters EJB in upper case.

Related tasks

“Setting up the J2EE run-time environment for EGL-generated code” on page 206

Deploying Java applications outside of J2EE

To deploy a Java application outside of J2EE, do as follows:

1. Follow the procedure detailed in *Installing the EGL run-time code for Java*
2. Export the EGL-generated code into jar files
3. Export any manually written Java code into jar files
4. Include the exported files in the classpath of the target machine

Related tasks

“Installing the EGL run-time code for Java” on page 205

Setting up the UNIX curses library for EGL run time

When you deploy an EGL text program on AIX or Linux, the EGL run time tries to use the UNIX curses library. If the environment is not set up for the UNIX curses library or if that library is not supported, the EGL run time tries to use the Java Swing technology; and if that technology is also not available, the program fails.

The UNIX curses library is required when the user runs an EGL program from a terminal emulator window or a character terminal.

To enable the EGL run time to access the UNIX curses terminal library on AIX or Linux, you must fulfill several steps in the UNIX shell environment. In each of the first two steps, *installDir* refers to the run-time installation library:

1. Modify the LD_LIBRARY_PATH environment variable to include the shared object libVGJ3270CursesCanvas.so, which is provided in the run-time installation library--
`export LD_LIBRARY_PATH=$LD_LIBRARY_PATH: /installDir/bin`
2. Modify the CLASSPATH environment variable to add fda.jar and fdaj.jar--
`export CLASSPATH=$CLASSPATH:
/installDir/lib/fda.jar: /installDir/lib/fdaj.jar`

The previous information must be typed on a single line.

3. Set the TERM environment variable to the appropriate terminal setting, as in the following example:

```
export TERM=vt100
```

If terminal exceptions occur, try various terminal settings such as `xterm`, `dtterm`, or `vt220`.

4. Run your EGL Java program from the UNIX shell, as in the following example:

```
java myProgram
```

Make sure that the `CLASSPATH` environment variable identifies the directory in which your program resides.

For additional details on using the Curses library on UNIX, refer to the UNIX man pages.

Reference pages

Association elements

As described in *Resource associations*, the resource associations part is composed of association elements. Each element is specific to a file name (property “fileName” on page 232) and contains a set of entries, each with these properties:

- “system” on page 233
- “fileType” on page 232

The values of the **system** and **fileType** properties determine what additional properties are available to you from the following list:

- “commit”
- “conversionTable”
- “formFeedOnClose” on page 232
- “duplicates”
- “replace” on page 232
- “systemName” on page 233
- “text” on page 233

commit

Indicates (for an EGL-generated Java program on iSeries) whether to enable commitment control.

Select one of these values:

NO (the default)

Use of sysLib.commit or sysLib.rollback has no effect.

YES

You can use sysLib.commit and sysLib.rollback to define the end of a logical unit of work.

conversionTable

Specifies the name of the conversion table used by a generated Java program during access of an MQSeries message queue.

For additional information, see *Data conversion*.

duplicates

Specifies (for an EGL-generated COBOL program on iSeries) whether an accessed VSAM file is allowed to contain duplicate keys. Valid values are NO (the default) and YES.

The value of **duplicates** must be consistent with the use of the keyword UNIQUE in the data description specification (DDS) that describes the physical file on iSeries. If the value of **duplicates** is YES, for example, you must not specify UNIQUE.

The next table shows the consequence of an inconsistency in the two values.

DDS keyword	Value of duplicates in the association element	COBOL return code after a file open	EGL return code after a file	EGL I/O error value
UNIQUE	YES	95	00000220	format
no UNIQUE	NO	95	00000220	format

fileType

Specifies the file organization on the target system. You can select an explicit type like *seqws*. Alternatively, you can select the value *default*, which is itself the default value of the property **fileType**. Use of the default means that a file type is selected automatically:

- For a particular combination of target system and EGL record type; or
- For print output, when the file name is *printer*.

Record and file type cross-reference shows the explicit **fileType** values, as well as the value used if you select *default*.

fileName

Refers to a logical file name, as specified in one or more records. You are creating an association element that relates this name to a physical resource on one or more target systems. (For print output, specify the value *printer*.)

You can use an asterisk (*) as a global substitution character in a logical file name; however, that character is valid only as the last character. For details, see *Resource associations and file types*.

formFeedOnClose

Indicates whether a form feed is issued when the output of a print form ends. (A print form is produced when your code issues a **print** statement.)

This property is available only if the **fileName** value is *printer* in one of the following cases:

- The **system** value is *aix*, *iSeriesj*, or *linux*, and the **fileType** value is *seqws* or *spool*; or
- The **system** value is *win*, and the **fileType** value is *seqws*.

Select one of these values:

YES

A form feed occurs (the default)

NO

A form feed does not occur

replace

Specifies whether adding a record to the file replaces the file rather than appending to the file. This entry is used only in these cases:

- You are generating Java code; and
- The record is of file type **seqws**.

Select one of these values:

NO
Append to the file (the default)

YES
Replace the file

system

Specifies the target platform. Select one of the following values:

aix
AIX

iseriesj
iSeries

linux
Linux

win
Windows 2000/NT/XP

any
Any target platform; for details, see *Resource associations and file types*

systemName

Specifies the system resource name of the file or data set associated with the file name. Enclose the value in single or double quote marks if a space or any of the following characters is in the value:

% = , () /

text

Specifies whether to cause a generated Java program to do the following when accessing a file by way of a serial record:

- Append end-of-line characters during the **add** operation. On non-UNIX platforms, those characters are a carriage return and linefeed; on UNIX platforms, the only character is a linefeed.
- Remove end-of-line characters during the **get** or **get next** operation.

Select one of these values:

NO
The default is not to append or remove the end-of-line characters

YES
Make the changes, as is useful if the generated program is exchanging data with products that expect records to end with the end-of-line characters

Related concepts

"Resource associations and file types" on page 157

Related reference

"Data conversion" on page 279

"I/O error values" on page 418

"Record and file type cross-reference" on page 159

Build descriptor part

A build descriptor part controls the generation process. The part contains several kinds of information:

- *Build descriptor options* specify how to generate and prepare EGL output, and a subset of the build descriptor options can cause other build parts to be included in the generation process. For details on specific options, see *Build descriptor options*.

- *Java run-time properties* assign values to the following properties:

- `vgj.datemask.gregorian.long.locale`, which contains the date mask used in either of two cases:

- The Java code generated for the system variable `sysVar.currentFormattedDate` is invoked; or
- EGL validates a page item or text-form field that has a length of 10 or more, if the item property **dateFormat** is set to *systemGregorian*.

The meaning of *locale* is described at the end of this section.

- `vgj.datemask.gregorian.short.locale`, which contains the date mask used when EGL validates a page item or text-form field that has a length of less than 10, if the item property **dateFormat** is set to *systemGregorian*.

The meaning of *locale* is described at the end of this section.

- `vgj.datemask.julian.long.locale`, which contains the date mask used in either of two cases:

- The Java code generated for the system variable `sysVar.currentFormattedJulianDate` is invoked; or
- EGL validates a page item or text-form field that has a length of 8 or more, if the item property **dateFormat** is set to *systemJulian*.

The meaning of *locale* is described at the end of this section.

- `vgj.datemask.julian.short.locale`, which contains the date mask used when EGL validates a page item or text-form field that has a length of less than 10, if the item property **dateFormat** is set to *systemJulian*.

The meaning of *locale* is described at the end of this section.

- `vgj.jdbc.database.SN`, which identifies a database that is made available to your Java code.

You must customize the name of the property itself when you specify a substitution value for *SN*, at deployment time. The substitution value in turn must match either the server name that is included in the invocation of `sysLib.connectionService` or the database name that is included in the invocation of `sysLib.connect`.

You also must customize the name of the date-mask properties:

- In a given run unit, each property that is initially in effect has a name whose last qualifier (the *locale*) matches the value in the program property **vgj.nls.code**
- In a Web application, a different set of properties is in effect if a program sets the system variable `sysLib.setLocale`

The Java run-time properties have no effect when you generate COBOL.

Master build descriptors

Your system administrator may require that you use a *master build descriptor* to specify information that cannot be overridden and that is in effect for every generation that occurs in your installation of EGL. By a mechanism described in

Master build descriptor, the system administrator identifies that part by name, along with the EGL build file that contains the part.

If the information in the master build descriptor is not sufficient for a particular generation process or if no master build descriptor is identified, you can specify a build descriptor at generation time, along with the EGL build file that contains the generation-specific part. The generation-specific build descriptor (like the master build descriptor) must be at the top level of an EGL build file.

You can create a chain of build descriptors from the generation-specific build descriptor, so that the first in the chain is processed before the second, and the second before the third. When you define a given build descriptor, you begin a chain (or continue one) by assigning a value to the build descriptor option **nextBuildDescriptor**. Your system administrator can use the same technique to create a chain from the master build descriptor. The implication of chaining information is described later.

Any build part referenced by a build descriptor must be visible to the referencing build descriptor, in accordance with the rules described in References to parts. The build part can be a linkage options part or a resource associations part, for example, or the next build descriptor.

Precedence of options

For a given build descriptor option (or Java run-time property), the value that is initially processed at generation time stays in effect, and the overall order of precedence is as follows:

1. The master build descriptor
2. The generation-specific build descriptor, followed by the chain that extends from it
3. The chain that extends from the master build descriptor

The benefit of this scheme is convenience:

- The system administrator can specify unchanging values by setting up a master build descriptor.
- You can use a generation-specific build descriptor to assign values that are specific to a generation.
- A project manager can specify a set of defaults by customizing one or more build descriptors. In most situations of this kind, the generation-specific build descriptor points to the first build descriptor in a chain that was developed by the project manager.

Default options can be useful when your organization develops a set of programs that must be generated or prepared similarly.

- The system administrator can create a set of general defaults by establishing a chain that extends from the master build descriptor, although use of this feature is unusual.

If a given build descriptor is used more than once, only the first access of that build descriptor has an effect. Also, only the first specification of a particular option has an effect.

Example

Let's assume that the master build descriptor contains these (unrealistic) option-and-value pairs:

OptionX	02
OptionY	05

In this example, the generation-specific build descriptor (called myGen) contains these option-and-value pairs.

OptionA	20
OptionB	30
OptionC	40
OptionX	50

As identified in myGen, the next build descriptor is myNext01, which contains these:

OptionA	120
OptionD	150

As identified in myNext01, the next build descriptor is myNext02, which contains these:

OptionB	220
OptionD	260
OptionE	270

As identified in the master build descriptor, the next build descriptor is myNext99, which contains this:

OptionZ	99
---------	----

EGL accepts option values in the following order:

1. Values for options in the master build descriptor:

OptionX	02
OptionY	05

Those options override all others.

2. Values in the generation-specific build descriptor myGen:

OptionA	20
OptionB	30
OptionC	40

The value for optionX in myGen was ignored.

3. Values for other options in myNext01 and myNext02:

OptionD	150
OptionE	270

The value for optionA in myNext01 was ignored, as was the value for optionD in myNext02.

4. Values for other options in myNext99:

OptionZ	99
---------	----

Related concepts

[“Build” on page 121](#)
[“Java run-time properties” on page 421](#)
[“References to parts” on page 16](#)
[“Master build descriptor” on page 413](#)
[“Parts” on page 11](#)

Related tasks

[“Adding a build descriptor part” on page 92](#)
[“Adding a resource associations part” on page 99](#)
[“Editing general build descriptor options” on page 93](#)

“Editing Java run-time properties in a build descriptor” on page 93
“Editing a resource associations part” on page 100

Related reference

“Build descriptor options”
“Java run-time properties (details)” on page 423
“Symbolic parameters” on page 272
“sysLib.connect” on page 543
“sysLib.connectionService” on page 545
“sysLib.setLocale” on page 615
“sysVar.currentFormattedDate” on page 522
“sysVar.currentFormattedJulianDate” on page 523

Setting the default build descriptors

For an overview of the default build descriptor and the build descriptor precedence rules, see *Generation in the workbench*.

To specify a preference for build descriptors at the Workbench level, do as follows:

1. Click **Window > Preferences**.
2. When a list is displayed, expand **EGL** and click **Default Build Descriptor**.
3. Select the Debug build descriptor and the Target system build descriptor.
4. Click **Apply**, then click **OK**.

To specify a preference for build descriptors at the file, folder, package, or project level, do as follows:

1. Right-click on the level of interest (for example, on the file or folder name) and, from the context menu, click **Properties**.
2. Select **EGL Default Build Descriptors**.
3. Select the Debug build descriptor and the Target system build descriptor.
4. Click **OK**.

Related concepts

“Generation in the workbench” on page 124

Build descriptor options

The next table lists all the build descriptor options.

Build descriptor option	Build option filter(s)	Description
bidirectionalConversionTable	<ul style="list-style-type: none">• iSeriesc	Identifies a conversion table, but only when you generate a COBOL program that contains literals with Arabic or Hebrew characters
buildPlan	<ul style="list-style-type: none">• Java target• iSeriesc	Specifies whether a build plan is created
checkNumericOverflow	<ul style="list-style-type: none">• iSeriesc	Specifies whether to support numeric overflow checking in a generated COBOL program
checkType	<ul style="list-style-type: none">• iSeriesc	Specifies the degree to which EGL checks at validation time for primitive-type conflicts within structures and records

Build descriptor option	Build option filter(s)	Description
cicsj2cTimeout	<ul style="list-style-type: none"> • Debug • Java target • Java iSeries 	Assigns a value to the Java run-time property cso.cicsj2c.timeout , which specifies the number of milliseconds before a timeout occurs during a call that uses protocol CICSJ2C
clientCodeSet	<ul style="list-style-type: none"> • iSeriesc 	Specifies the coded character set name used when you generate COBOL source code on the workstation
commentLevel	<ul style="list-style-type: none"> • Java target • Java iSeries • iSeriesc 	Specifies the extent to which EGL system comments are included in output source code
currencySymbol	<ul style="list-style-type: none"> • Debug • Java target 	Specifies a currency symbol that is composed of one to three characters
dbms	<ul style="list-style-type: none"> • Debug • Java target • Java iSeries 	Specifies the type of database accessed by the generated program
debugTrace	<ul style="list-style-type: none"> • iSeriesc 	Indicates whether EGL puts trace information at the end of a generated COBOL source file
decimalSymbol	<ul style="list-style-type: none"> • Debug • Java target • Java iSeries 	Assigns a character to the Java run-time property vgj.nls.number.decimal , which indicates what character is used as a decimal symbol
destDirectory	<ul style="list-style-type: none"> • Java target 	Specifies the name of the directory that stores the output of preparation, but only when you generate Java
destHost	<ul style="list-style-type: none"> • Java target • iSeriesc 	Specifies the name or numeric TCP/IP address of the target machine where the build server resides
destLibrary	<ul style="list-style-type: none"> • iSeriesc 	Specifies the 1- to 10-character name of the iSeries library that receives the objects created during generation and contains the objects used at run time
destPassword	<ul style="list-style-type: none"> • Java target • iSeriesc 	Specifies the password that EGL uses to log on to the machine where preparation occurs
destPort	<ul style="list-style-type: none"> • Java target • iSeriesc 	Specifies the port on which a remote build server is listening for build requests

Build descriptor option	Build option filter(s)	Description
destUserID	<ul style="list-style-type: none"> • Java target • iSeriesc 	Specifies the user ID that EGL uses to log on to the machine where preparation occurs
eliminateSystemDependentCode	<ul style="list-style-type: none"> • Java target • Java iSeries • iSeriesc 	Indicates whether, at validation time, EGL ignores code that will never run in the target system.
enableJavaWrapperGen	<ul style="list-style-type: none"> • Java target • Java iSeries • iSeriesc 	Specifies whether to allow generation of Java wrapper classes
fillWithNulls	<ul style="list-style-type: none"> • iSeriesc 	Indicates whether to fill print-form fields with null characters
genDataTables	<ul style="list-style-type: none"> • Java target • Java iSeries • iSeriesc 	Indicates whether you want to generate data tables
genDDSFile	<ul style="list-style-type: none"> • iSeriesc 	Indicates whether you want to create iSeries data description specification (DDS) files from the record declarations with which your program does input or output.
genDirectory	<ul style="list-style-type: none"> • Java target • iSeriesc 	Specifies the fully qualified path of the directory into which EGL places generated output and preparation-status files
genFormGroup	<ul style="list-style-type: none"> • Java target • iSeriesc 	Indicates whether you want to generate the form group that is referenced in the use declaration of the program you are generating
genHelpFormGroup	<ul style="list-style-type: none"> • Java target • iSeriesc 	Indicates whether you want to generate the help form group that is referenced in the use declaration of the program you are generating.
genProject	<ul style="list-style-type: none"> • Java target • Java iSeries • iSeriesc 	Places the output of Java generation into a workbench project and automates tasks that are required for Java run-time setup
genProperties	<ul style="list-style-type: none"> • Java target • Java iSeries 	Specifies what kind of Java run-time properties to generate (if any) and, in some cases, whether to generate a linkage properties file

Build descriptor option	Build option filter(s)	Description
initIORecords	<ul style="list-style-type: none"> • iSeriesc 	Specifies whether the generated COBOL program initializes global records other than basic records
initNonIOData	<ul style="list-style-type: none"> • iSeriesc 	Specifies whether the generated COBOL program initializes global, basic records
J2EE	<ul style="list-style-type: none"> • Debug • Java target • Java iSeries 	Specifies whether a Java program is generated to run in a J2EE environment
leftAlign	<ul style="list-style-type: none"> • iSeriesc 	Indicates whether to left-justify the output data on print-form fields
linkage	<ul style="list-style-type: none"> • Debug • Java target • Java iSeries • iSeriesc 	Contains the name of the linkage options part that guides aspects of generation
math	<ul style="list-style-type: none"> • iSeriesc 	Specifies whether to do arithmetic calculations based on <i>CSP math</i> , which is used in some COBOL programs that were written either with IBM Cross System Product (CSP) or with VisualAge Generator
nextBuildDescriptor (see Build descriptor part)	<ul style="list-style-type: none"> • Debug • Java target • Java iSeries • iSeriesc 	Identifies the next build descriptor in chain
oneFormItemCopybook	<ul style="list-style-type: none"> • iSeriesc 	Allows your program to pass and receive text forms that are formatted as VisualAge Generator maps
positiveSignIndicator	<ul style="list-style-type: none"> • iSeriesc 	Specifies the character that the iSeries-based ILE COBOL compiler uses as the positive sign for numeric data of types DECIMAL, NUM, NUMC, and PACF
prep	<ul style="list-style-type: none"> • Java target • iSeriesc 	Specifies whether EGL begins preparation when generation completes with a return code <= 4
reservedWord	<ul style="list-style-type: none"> • iSeriesc 	Specifies a fully qualified path name for a text file that contains reserved words other than the EGL reserved words

Build descriptor option	Build option filter(s)	Description
resourceAssociations	<ul style="list-style-type: none"> • Debug • Java target • Java iSeries • iSeriesc 	Contains the name of a resource associations part, which relates record parts to files and queues on the target platforms
serverCodeSet	<ul style="list-style-type: none"> • iSeriesc 	Specifies the name of the coded character set used by the z/OS build server
sessionBeanID	<ul style="list-style-type: none"> • Java target • Java iSeries 	Identifies the name of an existing session element in the J2EE deployment descriptor
setFormItemFull	<ul style="list-style-type: none"> • iSeriesc 	Indicates whether to display asterisks (*) in every empty print field for which you specified <i>set field full</i>
spacesZero	<ul style="list-style-type: none"> • iSeriesc 	Specifies whether a generated COBOL program includes extra code to process numeric items that are filled with spaces
sqlIDB	<ul style="list-style-type: none"> • Debug • Java target • Java iSeries • iSeriesc 	Specifies the default database used by a generated program
sqlErrorTrace	<ul style="list-style-type: none"> • iSeriesc 	Specifies whether a generated COBOL program includes the code necessary to trace errors that occur during I/O operations against a relational database
sqlID	<ul style="list-style-type: none"> • Debug • Java target • Java iSeries • iSeriesc 	Specifies a user ID that is used to connect to a database during generation-time validation of SQL statements or at run time
sqlIOTrace	<ul style="list-style-type: none"> • iSeriesc 	Specifies whether a generated COBOL program includes the code necessary to trace the I/O operations done against a relational database
sqlJDBCClass	<ul style="list-style-type: none"> • Debug • Java target • Java iSeries 	Specifies a driver class that is used to connect to a database during generation-time validation of SQL statements or during a non-J2EE Java debugging session
sqlJNDIName	<ul style="list-style-type: none"> • Debug • Java target • Java iSeries 	Specifies the default database used by a generated Java program that runs in J2EE

Build descriptor option	Build option filter(s)	Description
sqlPassword	<ul style="list-style-type: none"> • Debug • Java target • Java iSeries • iSeriesc 	Specifies a password that is used to connect to a database during generation-time validation of SQL statements or at run time
sqlValidationConnectionURL	<ul style="list-style-type: none"> • Debug • Java target • Java iSeries • iSeriesc 	Specifies a URL that is used to connect to a database during generation-time validation of SQL statements
sysCodes	<ul style="list-style-type: none"> • iSeriesc 	Determines the source of the code that is placed in the system variable sysVar.errorCode in response to a file I/O error in a COBOL program
system	<ul style="list-style-type: none"> • Debug • Java target • Java iSeries • iSeriesc 	Specifies a category of generation output
targetNLS	<ul style="list-style-type: none"> • Debug • Java target • Java iSeries • iSeriesc 	Specifies the target national language code used for run-time output
templateDir	<ul style="list-style-type: none"> • iSeriesc 	Specifies the directory that contains templates used to produce run-time JCL
VAGCompatibility	<ul style="list-style-type: none"> • Debug • Java target • Java iSeries • iSeriesc 	Indicates whether the generation process allows use of special program syntax
validateMixedItems	<ul style="list-style-type: none"> • iSeriesc 	Specifies whether a generated COBOL program validates items that are of type MIX
validateOnlyIfModified	<ul style="list-style-type: none"> • iSeriesc 	Specifies whether to validate only those text-form fields for which the modified data tag is set
validateSQLStatements	<ul style="list-style-type: none"> • Java target • Java iSeries • iSeriesc 	Indicates whether SQL statements are validated against a database

Related concepts

“Build descriptor part” on page 234

“Java run-time properties” on page 421

Related tasks

“Adding a build descriptor part” on page 92

“Editing general build descriptor options” on page 93

Related reference

"Java run-time properties (details)" on page 423

bidirectionalConversionTable

The build descriptor option **bidirectionalConversionTable** identifies a conversion table, but only when you generate a COBOL program that contains literals with Arabic or Hebrew characters. For details, see *Bidirectional text conversion*.

Related reference

"Bidirectional language text" on page 282

"Build descriptor options" on page 237

"Data conversion" on page 279

buildPlan

The build descriptor option **buildPlan** specifies whether a build plan is created. Valid values are YES and NO, and the default is YES.

The build plan is placed in the directory identified by build descriptor option **genDirectory**.

A special case is in effect when you generate Java code into a project. Then, no build plan is created regardless of the setting of **buildPlan**, but preparation occurs in either of two situations:

- Whenever you rebuild the project
- Whenever you generate the source files; but only if you checked the workbench preference **Perform build automatically on resource modification**

You may wish to create a build plan and to invoke that plan at a later time. For details, see *Invoking a build plan after generation*.

Related concepts

"Build plan" on page 392

Related tasks

"Invoking a build plan after generation" on page 123

Related reference

"Build descriptor options" on page 237

checkNumericOverflow

The build descriptor option **checkNumericOverflow** specifies whether to support numeric overflow checking in a generated COBOL program. Valid values are YES (the default) and NO.

If you specify NO, the system function `sysVar.handleOverflow` is ignored. Division by zero results in an abend with a message. In other overflow conditions, the result is truncated, causing the significant digits to be lost, but the generated program gives no indication that truncation has occurred.

Setting **checkNumericOverflow** to NO may result in smaller programs with better performance.

Related reference

"Build descriptor options" on page 237

"sysVar.handleOverflow" on page 532

checkType

The build descriptor option **checkType** specifies the degree to which EGL checks at validation time for primitive-type conflicts within records. You can receive an information message, for example, if a structure item that is of type CHAR is substructured with structure items of type DECIMAL. Such conflicts can cause run-time errors.

Valid values are as follows:

NONE

Specify NONE (the default) if you do not want to check for potential conflicts in the primitive types of substructured items.

LOW

Specify LOW to check for conflicting primitive types only in the items that are subordinate to the highest level of the structure. Consider the following example:

```
10 ItemA
 15 ItemB
   20 ItemC
    30 ItemD
```

If you specify LOW, EGL will not compare ItemA to ItemB, but will compare ItemB to ItemC, and ItemC to ItemD.

ALL

Specify ALL to check for conflicting primitive types in all levels of a substructured data item.

Specifying a value other than NONE increases both the time needed for validation and the number of messages issued.

Related reference

"Build descriptor options" on page 237

cicsj2cTimeout

When you are generating Java code, the build descriptor option **cicsj2cTimeout** assigns a value to the Java run-time property **cso.cicsj2c.timeout**. That property specifies the number of milliseconds before a timeout occurs during a call that uses protocol CICSJ2C.

The default value of the run-time property is 30000, which represents 30 seconds. If the value is set to 0, no timeout occurs. The value must be greater than or equal to 0.

The property **cso.cicsj2c.timeout** has no effect on calls when the called program is running in WebSphere 390; for details, see *Setting up the J2EE server for CICSJ2C calls*. Also, the build descriptor option **cicsj2cTimeout** has no effect when you are generating COBOL code.

Related concepts

"Java run-time properties" on page 421

Related tasks

“Setting up the J2EE server for CICSJ2C calls” on page 213

Related reference

“Build descriptor options” on page 237

“Java run-time properties (details)” on page 423

clientCodeSet

The build descriptor option **clientCodeSet** specifies the name of the coded character set that is in effect when you generate COBOL source on the workstation.

The coded character set identified in **clientCodeSet** must be the one defined to the ICONV conversion service on the machine where the build server is started. The default is IBM-850, which is a standard character set for Latin-1 countries.

Related reference

“Build descriptor options” on page 237

“serverCodeSet” on page 259

commentLevel

The build descriptor option **commentLevel** specifies the extent to which EGL system comments are included in output source code.

Valid values are as follows:

- 0 Minimal comments are in the output, which includes comments on any name aliases that EGL generates
- 1 In addition to the comments included with level 0, scripting statements are placed immediately before the code that is generated to implement those statements.

The default is 1.

Raising the comment level has no effect on the size or performance of the prepared code, but increases the size of the output and the time needed to generate, transfer, and prepare the output.

Related reference

“Build descriptor options” on page 237

currencySymbol

The build descriptor option **currencySymbol** is available only for Java output and specifies a currency symbol that is composed of one to three characters. If you do not specify this option, the default value is derived from the locale of the system on which you generate output.

To specify a character that is not on your keyboard, hold down the **Alt** key and use the numeric key pad to type the character’s decimal code. The decimal code for the Euro, for example, is 0128 on Windows 2000/NT/XP.

Related concepts

“Build descriptor part” on page 234

Related reference

"Build descriptor options" on page 237

dbms

The build descriptor option **dbms** specifies the type of database accessed by the generated program. Select one of the following values:

- DB2 (the default value)
- INFORMIX
- ORACLE

Related reference

"Build descriptor options" on page 237

"Informix and EGL" on page 199

debugTrace

The build descriptor option **debugTrace** indicates whether EGL puts trace information at the end of a generated COBOL source file. The valid values are YES and NO. The default is NO.

It is recommended that you use this option only when you are providing debugging information to IBM service personnel.

Related reference

"Build descriptor options" on page 237

decimalSymbol

When you are generating Java code, the build descriptor option **decimalSymbol** assigns a character to the Java run-time property **vgj.nls.number.decimal**, which indicates what character is used as a decimal symbol. If you do not specify the build descriptor option **decimalSymbol**, the character is determined by the locale associated with the Java run-time property **vgj.nls.code**.

The build descriptor option **decimalSymbol** has no effect when you are generating COBOL code. Also, the value can be no more than one character.

Related concepts

"Java run-time properties" on page 421

Related reference

"Build descriptor options" on page 237

"Java run-time properties (details)" on page 423

destDirectory

The build descriptor option **destDirectory** specifies the directory that stores the output of preparation, but only when you generate Java. This option is meaningful only when you generate into a directory rather than into a project. A similar option for COBOL generation is **projectID**.

When you specify a fully qualified file path, all but the last directory must exist. If you specify c:\buildout on Windows 2000, for example, EGL creates the buildout directory if it does not exist. If you specify c:\interim\buildout and the interim directory does not exist, however, preparation fails.

If you specify a relative directory (such as myid/mysource on USS), the output is placed in the bottom-most directory, which is relative to the default directory, as described next.

The default value of **destDirectory** is affected by the status of build descriptor option **destHost**:

- If **destHost** is specified, the default value of **destDirectory** is the directory in which the build server was started
- If **destHost** is not specified, preparation occurs on the machine where generation occurs, and the default value of **destDirectory** is given by build descriptor option **genDirectory**

The user specified by build descriptor option **destUserID** must have the authority to write to the directory that receives the output of preparation.

You cannot use a UNIX variable (\$HOME, for example) to identify part of a directory structure on USS.

Related reference

“Build descriptor options” on page 237

“destHost”

“genProject” on page 252

destHost

The build descriptor option **destHost** specifies the name or numeric TCP/IP address of the target machine where the build server resides. No default is available.

If you are preparing a generated COBOL program, the following statements apply:

- **destHost** is required
- A build server must be started on the remote machine before generation begins

If you are preparing Java output, the following statements apply:

- **destHost** is optional
- **destHost** is meaningful only if you generate into a directory rather than into a project
- If you specify **destHost** without specifying **destDirectory**, the directory in which the build server was started is the one that receives source and preparation outputs
- If you do not specify **destHost**, preparation occurs on the machine where generation occurs; and if **destDirectory** is not specified, the directory that is specified by build descriptor option **genDirectory** is the one that receives source and preparation outputs
- The UNIX environments are case sensitive

You can type up to 64 characters for the name or TCP/IP address. If you are developing on Windows NT®, you must specify a name rather than a TCP/IP address.

Two example values for **destHost** are as follows:

abc.def.ghi.com

9.99.999.99

Related reference

"Build descriptor options" on page 237

"destDirectory" on page 246

"destPassword"

"destPort"

destLibrary

Specifies the 1- to 10-character name of the iSeries library that receives the objects created during generation and contains the objects used at run time.

The default is QGPL.

Related concepts

"Build descriptor part" on page 234

Related reference

"Build descriptor options" on page 237

destPassword

The build descriptor option **destPassword** specifies the password that EGL uses to log on to the machine where preparation occurs.

This option and the description on this page are meaningful only if you are generating into a directory rather than into a project and only if you specify a value for build descriptor option **destHost**.

The password provides access for the userid specified in build descriptor option **destUserID**. The value of the password is case sensitive for all target systems.

No default is available.

Use of **destPassword** means that a password is stored in an EGL build file. You can avoid the security risk by not setting the build descriptor option. When you start generation, you can set the password in an interactive generation dialog or on the command line.

Related reference

"Build descriptor options" on page 237

"destHost" on page 247

"destUserID" on page 249

destPort

The build descriptor option **destPort** specifies the port on which a remote build server is listening for build requests.

This option is meaningful only if you are generating into a directory rather than into a project and only if you specify a value for build descriptor option **destHost**.

No default value is available.

Related reference

"Build descriptor options" on page 237

"destHost" on page 247

destUserID

The build descriptor option **destUserID** specifies the userid that EGL uses to log on to the machine where preparation occurs.

This option and the description on this page are meaningful only if you are generating into a directory rather than into a project and only if you specify a value for build descriptor option **destHost**.

The user specified by **destUserID** must have the authority to write to the directory. The option value is case sensitive for all target systems.

No default is available.

Related reference

“Build descriptor options” on page 237

“destHost” on page 247

“destPassword” on page 248

eliminateSystemDependentCode

The build descriptor option **eliminateSystemDependentCode** indicates whether, at validation time, EGL ignores code that will never run in the target system. Valid values are *yes* (the default) and *no*. Specify *no* only if the output of the current generation will run in multiple systems.

The option **eliminateSystemDependentCode** is meaningful only in relation to the system function **sysVar.systemType**. That function does not itself affect what code is validated at generation time. For example, the following **add** statement may be validated even if you are generating for Windows:

```
if (sysVar.systemType IS AIX)
  add myRecord;
end
```

To avoid validating code that will never run in the target system, take either of the following actions:

- Set the build descriptor option **EliminateSystemDependentCode** to *yes*. In the current example, the **add** statement is not validated if you set that build descriptor option to *yes*. Be aware, however, that the generator can eliminate system-dependent code only if the logical expression (in this case, **sysVar.systemType IS AIX**) is simple enough to evaluate at generation time.
- Alternatively, move the statements that you do not want to validate to a second program; then, let the original program call the new program conditionally:

```
if (sysVar.systemType IS AIX)
  call myAddProgram myRecord;
end
```

Related concepts

“Build descriptor part” on page 234

Related reference

“Build descriptor options” on page 237

enableJavaWrapperGen

When you issue the commands to generate a program, the build descriptor option **enableJavaWrapperGen** allows you to choose from three alternatives:

YES (the default)

Generate the program and allow generation of the related Java wrapper classes and (if appropriate) the related EJB session bean

ONLY

Do not generate the program, but allow generation of the related Java wrapper classes and (if appropriate) the related EJB session bean

NO

Generate the program, but not the Java wrapper classes or the related EJB session bean, if any

Actual generation of the Java wrapper classes and EJB session bean requires appropriate settings in the linkage options part that is used at generation time. For an overview, see *Java wrapper*.

Related concepts

"Java wrapper" on page 395

Related reference

"Java wrapper classes" on page 395

fillWithNulls

The build descriptor option **fillWithNulls** is used only when you generate a form group that includes print forms. The option indicates whether to fill print-form fields with null characters. The affected fields have these characteristics:

- The item property **fillCharacter** is set to null; and
- The field is of one of these types: CHAR, DBCHAR, MBCHAR, or NUM.

Valid values are *yes* (the default) and *no*. If you specify *no*, the affected fields are filled with spaces.

Related concepts

"Build descriptor part" on page 234

Related reference

"Build descriptor options" on page 237

genDataTables

The build descriptor option **genDataTables** indicates whether you want to generate the data tables that are referenced in the program you are generating. The references are in the program's use declaration and in the program property **msgTablePrefix**.

Valid values are *yes* (the default) and *no*.

Set the value to *no* in the following case:

- The data tables referenced in the program were previously generated; and
- Those tables have not changed since they were last generated.

For other details, see *DataTable part*.

Related concepts

“Build descriptor part” on page 234

“DataTable part” on page 285

Related reference

“Build descriptor options” on page 237

“Program part in EGL source format” on page 478

“Use declaration” on page 631

genDDSFile

The build descriptor option **genDDSFile** indicates whether you want to create iSeries data description specification (DDS) files from the record declarations with which your program does input or output. Valid values are *no* (the default) and *yes*.

If you are creating DDS files and if you accept the default value of the build descriptor option **prep**, EGL uploads the DDS files to the host system.

Related concepts

“Build descriptor part” on page 234

Related reference

“Build descriptor options” on page 237

genDirectory

The build descriptor option **genDirectory** specifies the fully qualified path of the directory into which EGL places generated output and preparation-status files.

When you are generating in the workbench or from the workbench batch interface, the following rules apply:

For Java generation

You must specify either **genProject** or **genDirectory**, but an error results if you specify both. Also, you must specify **genProject** if you generate Java code for iSeries.

For COBOL generation

You must specify **genDirectory**, and in most cases EGL ignores any setting for **genProject**.

If you are generating from the EGL SDK, the following rules apply:

- You must specify **genDirectory**
- An error results if you specify **genProject**
- You cannot generate Java code for iSeries

For details on deploying Java code, see *Generating into a directory*.

Related concepts

“Generation from the EGL SDK” on page 127

“Generation from the workbench batch interface” on page 126

“Generation in the workbench” on page 124

Related tasks

“Generating into a directory” on page 220

Related reference

“Build descriptor options” on page 237

“genDirectory” on page 251

“genProject”

genFormGroup

The build descriptor option **genFormGroup** indicates whether you want to generate the form group that is referenced in the use declaration of the program you are generating. Valid values are *yes* (the default) and *no*.

The help form group, if any, is not affected by this option, but by the build descriptor option **genHelpFormGroup**.

Related concepts

“Build descriptor part” on page 234

Related reference

“Build descriptor options” on page 237

“genHelpFormGroup”

“Use declaration” on page 631

genHelpFormGroup

The build descriptor option **genHelpFormGroup** indicates whether you want to generate the help form group that is referenced in the use declaration of the program you are generating. Valid values are *yes* (the default) and *no*.

The main form group is not affected by this option, but by the build descriptor option **genFormGroup**.

Related concepts

“Build descriptor part” on page 234

Related reference

“Build descriptor options” on page 237

“genFormGroup”

“Use declaration” on page 631

genProject

The build descriptor option **genProject** places the output of Java generation into a Workbench project and automates tasks that are required for Java run-time setup. For details on that setup and on the benefits of using **genProject**, see *Generating into a project*.

To use **genProject**, specify the project name. EGL then ignores the build descriptor options **buildPlan**, **genDirectory**, and **prep**, and preparation occurs in either of two cases:

- Whenever you rebuild the project
- Whenever you generate the source files; but only if you checked the workbench preference **Perform build automatically on resource modification**

If you set the option **genProject** to the name of a project that does not exist in the workbench, EGL uses the name to create a Java project, except in these cases:

- If you are generating a page handler and specify a project different from the one that contains the related JSP and if that other project does not exist, EGL creates an EGL Web project. (However, it is recommended that you generate the page handler into the project that contains the related JSP.)
- A second exception concerns EJB processing and occurs if you are generating a Java wrapper when the linkage options part, callLink element, **type** property is ejbCall (for the call from the wrapper to the EGL-generated program). In that case, EGL uses the value of **genProject** to create an EJB project and creates a new enterprise application project (if necessary) with a name that is the same as the EJB project name plus the letters EAR.

In addition to creating a project, EGL does as follows:

- EGL creates folders in the project. The package structure begins under the top-level folder JavaSource. You may change the name JavaSource by right-clicking on the folder name and selecting **Refactor**.
- If a JRE definition is specified in the preferences page for Java (installed JREs), EGL adds the classpath variable JRE_LIB. That variable contains the path to the run-time JAR files for the JRE currently in use.

When you are generating in the Workbench or from the Workbench batch interface, the following rules apply:

For Java generation

You are not required to specify either **genProject** or **genDirectory**. If neither is specified, Java output is generated into the project that contains the EGL source file being generated.

If you are generating a page handler, and the project specified exists, then the project must be an EGL Web project. If you are generating a session EJB, and the project specified exists, then the project must be an EJB project.

For COBOL generation

You must specify **genDirectory**, and EGL ignores any setting for **genProject**.

If you are generating from the EGL SDK, the following rules apply:

- You must specify **genDirectory**
- An error results if you specify **genProject**
- You cannot generate Java code for iSeries

Related concepts

“Generation from the EGL SDK” on page 127

“Generation from the workbench batch interface” on page 126

“Generation in the workbench” on page 124

Related tasks

“Generating into a project” on page 214

Related reference

“Build descriptor options” on page 237

“buildPlan” on page 243

“genDirectory” on page 251

“prep” on page 258

“type in callLink element” on page 458

genProperties

The build descriptor option **genProperties** specifies what kind of Java run-time properties to generate (if any) and, in some cases, whether to generate a linkage properties file. This build descriptor option is meaningful only when you are generating a Java program (which can use either kind of output) or a wrapper (which can use only the linkage properties file).

Valid values are as follows:

NO (the default)

EGL does not generate run-time or linkage properties.

PROGRAM

The effects are as follows:

- If you are generating a program to run outside of J2EE, EGL generates a properties file that is specific to the program being generated. The name of that file is as follows:
pgmAlias.properties
pgmAlias
The name of the program at run time.
- The other effects occur whether you specify **PROGRAM** or **GLOBAL**:
 - If you are generating a program that runs in J2EE, EGL generates a J2EE environment file or into a deployment descriptor; for details, see *Understanding alternatives for setting deployment-descriptor values*.
 - If you are generating a Java wrapper or calling program, EGL may generate a linkage properties file; for details on the situation in which this file is generated, see *Linkage properties file (reference)*.

GLOBAL

The effects are as follows:

- If you are generating a program to run outside of J2EE, EGL generates a properties file that is used throughout the run unit but is not named for the initial program in the run unit. The name of that properties file is **vgj.properties**.

This option is especially useful when the first program of a run unit does not access a file or database but calls programs that do.

When generating the caller, you can generate a properties file named for the program, and the content might include no database-related properties. When you generate the called program, you can generate **vgj.properties**, and the content would be available for both programs.

- The other effects occur whether you specify **GLOBAL** or **PROGRAM**:
 - If you are generating a program that runs in J2EE, EGL generates a J2EE environment file or into a deployment descriptor; for details, see *Understanding alternatives for setting deployment-descriptor values*.
 - If you are generating a Java wrapper or calling program, EGL may generate a linkage properties file; for details on the situation in which this file is generated, see *Linkage properties file (reference)*.

For further details, see *Java run-time properties* and *Linkage properties file*.

Related concepts

"J2EE environment file" on page 394

"Java run-time properties" on page 421

"Linkage options part" on page 439

"Linkage properties file" on page 404

Related tasks

"Build descriptor options" on page 237

"Setting deployment-descriptor values" on page 207

Related reference

"Java run-time properties (details)" on page 423

initIORecords

The build descriptor option **initIORecords** specifies whether the generated COBOL program initializes global records that are used in I/O operations. Valid values are YES and NO. The default is YES.

The option **initIORecords** is meaningful only when you are generating a COBOL program. If you specify the **initialized** property when declaring a global record, the property takes precedence over the build descriptor option. Also, this build descriptor option has no effect on records that are not used in I/O operations.

For details on initialization, see *Data initialization*.

Related reference

"Build descriptor options" on page 237

"Data initialization" on page 277

"initNonIOData"

initNonIOData

The build descriptor option **initNonIOData** specifies whether the generated COBOL program initializes global basic records. Valid values are YES and NO. The default is YES.

The option **initNonIOData** is meaningful only when you are generating a COBOL program. If you specify the **initialized** property when declaring a global basic record, the property takes precedence over the build descriptor option.

For details on initialization, see *Data initialization*.

Related reference

"Build descriptor options" on page 237

"Data initialization" on page 277

"initIORecords"

J2EE

The build descriptor option **J2EE** specifies whether a Java program is generated to run in a J2EE environment. Valid values are as follows:

NO (the default)

Generates a program that will not run in a J2EE environment. The program connects to databases directly, and the environment is defined by a properties file.

YES

Generates a program to run in a J2EE environment. The program connects to databases using a data source, and the environment is defined by a deployment descriptor.

When you generate a page handler, J2EE is always set to YES regardless of what is specified in this option.

Related concepts

“EGL debugger” on page 107

Related reference

“Build descriptor options” on page 237

leftAlign

The build descriptor option **leftAlign** is used only when you generate a form group that includes print forms. The option indicates whether to left-justify the output data on print-form fields that have the following characteristics:

- The item property **align** is set to *left*; and
- The field is of one of these types: CHAR, DBCHAR, or MBCHAR.

Valid values are *yes* (the default) and *no*. If you do not need left alignment during output, specify *no* to give better performance and to reduce the code size.

Left alignment strips leading spaces and places them at the end of the field.

Related concepts

“Build descriptor part” on page 234

Related reference

“Build descriptor options” on page 237

linkage

The build descriptor option **linkage** contains the name of the linkage options part that guides aspects of generation. This option is not required for generation, and no default value is available.

Related concepts

“Linkage options part” on page 439

Related reference

“Build descriptor options” on page 237

“callLink element” on page 443

math

The build descriptor option **math** specifies whether to do arithmetic calculations based on *CSP math*, which is used in some COBOL programs that were written either with IBM Cross System Product (CSP) or with VisualAge Generator. You might choose CSP math if your EGL-generated program interacts with an older application. Otherwise, accept the default, which is COBOL.

This option is meaningful only if you are generating a COBOL program.

Valid values are as follows:

COBOL

Use COBOL truncation algorithms, which may provide faster performance, smaller load module size, and better accuracy.

CSPA

Truncate intermediate arithmetic values to a number of significant digits. The number is equal to the number of significant digits for the memory area that holds the final result.

Related reference

“Build descriptor options” on page 237

nextBuildDescriptor

The build descriptor option **nextBuildDescriptor** identifies the next build descriptor in chain, if any. For details, see *Build descriptor part*.

Related concepts

“Build descriptor part” on page 234

Related reference

“Build descriptor options” on page 237

oneFormItemCopybook

The build descriptor option **oneFormItemCopybook** indicates how EGL-generated COBOL code accesses the values of form-item properties. Values are as follows:

no (the default)

EGL generates a COBOL copybook into the definition of each form item, in the Data Section of the COBOL program. Access is direct, not requiring use of COBOL SET statements.

yes

EGL places a single copybook in the Linkage Section, and access is by COBOL SET statements.

If possible, accept the default value, which maximizes performance. If your program uses many forms or if the forms contain many items, however, EGL generates a large number of COBOL variable names, and the COBOL compiler symbol table can become so large that compilation fails.

If you need to avoid the compilation problem just described, set **oneFormItemCopybook** to *yes*; then, the EGL-generated code will invoke a COBOL SET statement whenever a form-item property value is accessed.

Related concepts

“Build descriptor part” on page 234

Related reference

“Build descriptor options” on page 237

positiveSignIndicator

The build descriptor option **positiveSignIndicator** specifies the character that the iSeries-based ILE COBOL compiler uses as the positive sign for numeric data of types DECIMAL, NUM, NUMC, and PACF. You can specify the value *F* or *C*.

The default value is *F*. If your code includes more occurrences of items that are of type NUMC and DECIMAL (as compared to items of type NUM and PACKF), you can improve performance by setting **positiveSignIndicator** to *C*.

Related concepts

"Build descriptor part" on page 234

Related reference

"Build descriptor options" on page 237

"Primitive types" on page 27

prep

The build descriptor option **prep** specifies whether EGL begins preparation when generation completes with a return code ≤ 4 . Valid values are YES and NO, and the default is YES.

Even if you set **prep** to NO, you can prepare code later. For details, see *Invoking a build plan after generation*.

Consider these cases:

- When you generate a COBOL program, EGL writes preparation messages to the directory specified in build descriptor option **genDirectory**, to the results file, and to additional files that are each specific to a preparation step
- When you generate Java code into a directory, EGL writes preparation messages to the directory specified in build descriptor option **genDirectory**, to the results file
- When you generate Java code into a project (option **genProject**), the option **prep** has no effect, and preparation occurs in either of two situations:
 - Whenever you rebuild the project
 - Whenever you generate the source files; but only if you checked the workbench preference **Perform build automatically on resource modification**

If you wish to customize the generated build plan, do as follows:

- Set option **prep** to NO
- Set option **buildPlan** to YES (as is the default)
- Generate the output
- Customize the build plan
- Invoke the build plan, as described in *buildPlan*

Related concepts

"Results file" on page 405

Related tasks

"Invoking a build plan after generation" on page 123

Related reference

"Build descriptor options" on page 237

"buildPlan" on page 243

"Generated output (reference)" on page 387

"genDirectory" on page 251

"genProject" on page 252

reservedWord

The build descriptor option **reservedWord** specifies a fully qualified path name for a text file that contains reserved words other than the EGL reserved words.

This option has no default and is meaningful only when you are generating a COBOL program. For details, see *COBOL reserved-word file*.

Related concepts

"COBOL reserved-word file" on page 275

Related reference

"Build descriptor options" on page 237

"EGL reserved words" on page 290

"Format of COBOL reserved-word file" on page 276

resourceAssociations

The build descriptor option **resourceAssociations** contains the name of a resource associations part, which relates record parts to files and queues on the target platforms. This option is not required for generation, and no default value is available.

Related concepts

"Resource associations and file types" on page 157

Related tasks

"Adding a resource associations part" on page 99

Related reference

"Build descriptor options" on page 237

"Association elements" on page 231

"Record and file type cross-reference" on page 159

serverCodeSet

The build descriptor option **serverCodeSet** specifies the name of the coded character set that is used by the iSeries build server. This option (along with the build descriptor option **clientCodeSet**) helps to cause a particular data conversion to occur when file content, file-path information, and environment variables are transferred from the workstation to the build server.

The coded character set specified for **serverCodeSet** must be the one defined to the ICONV conversion service on the machine where the build server is started. The default is IBM-037, which is a character set that is used for English (U.S.).

Related reference

"Build descriptor options" on page 237

"clientCodeSet" on page 245

sessionBeanID

The build descriptor option **sessionBeanID** identifies the name of an existing session element in the J2EE deployment descriptor. The environment entries are placed into the session element when you act as follows:

- Generate a program for a Java platform (by setting **system** to AIX, WIN, or USS)
- Generate into an EJB project (by setting **genProject** to an EJB project)

- Request that environment properties be generated (by setting **genProperties** to GLOBAL or PROGRAM)

The option **sessionBeanID** is useful in the following case:

1. You generate a Java wrapper, along with an EJB session bean. In the EJB project deployment descriptor (file ejb-jar.xml), EGL creates a session element, without environment entries.

Both the EJB session bean and the session element are named as follows:

ProgramnameEJBBean

Programname is the name of the run-time program that receives data by way of the EJB session bean. The first letter in the name is uppercase, the other letters are lowercase.

In this example, the name of the program is ProgramA, and the name of the session element and the EJB session bean is ProgramaEJBBean.

2. After you generate the EJB session bean, you generate the Java program itself. Because the build descriptor option **genProperties** is set to YES, EGL generates J2EE environment entries into the deployment descriptor, into the session element established in step 1.
3. You generate ProgramB, which is a Java program that is used as a helper class for ProgramA. The values of **system** and **genProject** are the same as those used in step 2; also, you generate environment entries and set **sessionBeanID** to the name of the session element.

Your use of **sessionBeanID** causes EGL to place the environment entries for the second program into the session element that was created in step 2; specifically, into the session element ProgramaEJBBean.

In the portion of the deployment descriptor that follows, EGL created the environment entries **vgj.nls.code** and **vgj.nls.number.decimal** during step 2, when ProgramA was generated; but the entry **vgj.jdbc.default.database** is used only by ProgramB and was created during step 3:

```
<ejb-jar id="ejb-jar_ID">
  <display-name>EJBTest</display-name>
  <enterprise-beans>
    <session id="ProgramaEJBBean">
      <ejb-name>ProgramaEJBBean</ejb-name>
      <home>test.ProgramaEJBHome</home>
      <remote>test.ProgramaEJB</remote>
      <ejb-class>test.ProgramaEJBBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
      <env-entry>
        <env-entry-name>vgj.nls.code</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>ENU</env-entry-value>
      </env-entry>
      <env-entry>
        <env-entry-name>vgj.nls.number.decimal</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>.</env-entry-value>
      </env-entry>
      <env-entry>
        <env-entry-name>vgj.jdbc.default.database</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>jdbc/Sample</env-entry-value>
      </env-entry>
    </session>
  </enterprise-beans>
</ejb-jar>
```

A session element must be in the deployment descriptor before you can add environment entries. Because session elements are created during Java wrapper generation, it is recommended that you generate the Java wrapper before generating the related programs.

In the following cases, you generate a program into an EJB project, but the environment entries are placed into a J2EE environment file rather than into the deployment descriptor:

- **sessionBeanID** is set, but the session element that matches the value of **sessionBeanID** is not found in the deployment descriptor; or
- **sessionBeanID** is not set, and the session element that is named for the program is not found in the deployment descriptor. This case occurs when the program is generated before the wrapper.

For EJB projects, an environment entry name (like **vgj.nls.code**) can appear only once for each session element. If an environment entry already exists, EGL updates the entry type and value instead of creating a new entry.

EGL never deletes an environment entry from a deployment descriptor.

No default value is available for **sessionBeanID**.

Related reference

“Build descriptor options” on page 237

setFormItemFull

The build descriptor option **setFormItemFull** is used only when you generate a form group that includes print forms. The option indicates whether to display asterisks (*) in every empty field for which you specified a **set** statement of the form *set field full*. Valid values are *yes* (the default) and *no*.

If you specify *no*, the **set** statement of format *set field full* is ignored for the print forms, which may result in better performance and smaller load module size, especially if the form group contains print forms with many variables. Specify *no* if the programs that use the form group do not use a **set** statement of format *set field full*.

spacesZero

The build descriptor option **spacesZero** specifies whether a generated COBOL program includes extra code to process numeric items that are filled with spaces. The specific situation concerns items that have the following characteristics:

- Were declared in EGL with primitive type NUM or NUMC
- May receive spaces, as when the item is subordinate to a structure item of type CHAR

Valid values are as follows:

NO

Do not include the extra code necessary to interpret the spaces as zeros. NO is appropriate if you are sure that no item in the program will ever receive spaces. NO is the default because the lack of code is more efficient at run time.

YES

Include extra code to avoid an abend when a numeric item that contains spaces is processed in a program or function script.

The option **spacesZero** has no effect on items that receive a combination of spaces and other characters.

Related reference

"Build descriptor options" on page 237

"CHAR" on page 29

"NUM" on page 33

"NUMC" on page 33

sqlDB

The build descriptor option **sqlDB** specifies the default database used by a generated Java program that runs outside of J2EE. The value is a connection URL; for example, jdbc:db2:MyDB.

The option **sqlDB** is case-sensitive, has no default value, and is used only when you are generating a non-J2EE Java program. The option assigns a value to the Java run-time property **vgj.jdbc.default.database**, but only if option **genProperties** is set to GLOBAL or PROGRAM.

To specify the database used for validation (in relation to Java or COBOL generation), set **sqlValidationConnectionURL**.

Related concepts

"Java run-time properties" on page 421

"SQL support" on page 171

Related reference

"Build descriptor options" on page 237

"genProperties" on page 254

"Java run-time properties (details)" on page 423

"sqlPassword" on page 264

"sqlValidationConnectionURL" on page 265

"sqlJDBCClass" on page 263

"validateSQLStatements" on page 269

sqlErrorTrace

The build descriptor option **sqlErrorTrace** specifies whether a generated COBOL program includes the code necessary to trace errors that occur during I/O operations against a relational database. The valid values are YES and NO. The default is NO.

This option is intended for use by support personnel and should be used only when a trace is requested as part of a support effort.

Related reference

"Build descriptor options" on page 237

"sqlIOTrace" on page 263

sqlID

The build descriptor option **sqlID** specifies a userid that is used to connect to a database during generation-time validation of SQL statements. You specify the database by setting **sqlValidationConnectionURL**.

When you generate a Java program, EGL also assigns the value of **sqlID** to the Java run-time property **vgj.jdbc.default.userid**. That property identifies the userid for connecting to the default database at run time, and you can specify the default database in **sqlDB**.

The option **sqlID** is case-sensitive and has no default value.

Related reference

"Build descriptor options" on page 237
"Java run-time properties (details)" on page 423
"sqlDB" on page 262
"sqlPassword" on page 264
"sqlValidationConnectionURL" on page 265
"sqlJDBCDriverClass"
"validateSQLStatements" on page 269

sqlIOTrace

The build descriptor option **sqlIOTrace** specifies whether a generated COBOL program includes the code necessary to trace the I/O operations done against a relational database. The valid values are YES and NO. The default is NO.

This option is intended for use by support personnel and should be used only when a trace is requested as part of a support effort.

Related reference

"Build descriptor options" on page 237

sqlJDBCDriverClass

The build descriptor option **sqlJDBCDriverClass** specifies a driver class for connecting to the database that EGL uses to validate SQL statements at generation time. You specify the database by setting **sqlValidationConnectionURL**. Database access is through JDBC.

In the following cases EGL also assigns the value of **sqlJDBCDriverClass** to the Java run-time property **vgj.jdbc.drivers** in the program properties file:

- **genProperties** is set to GLOBAL or PROGRAM
- **J2EE** is set to NO

No default is available for the driver class, and the format varies by driver:

- For IBM DB2 APP DRIVER for Windows, the driver class is as follows--
COM.ibm.db2.jdbc.app.DB2Driver
- For IBM DB2 NET DRIVER for Windows, the driver class is as follows--
COM.ibm.db2.jdbc.net.DB2Driver
- For IBM DB2 UNIVERAL DRIVER for Windows, the driver class is as follows (with com in lower case)--
com.ibm.db2.jcc.DB2Driver
- For the Oracle JDBC thin client-side driver, the driver class is as follows--

`oracle.jdbc.driver.OracleDriver`

- For the IBM Informix JDBC driver, the driver class is as follows--
`com.informix.jdbc.IfxDriver`

For other driver classes, refer to the documentation for the driver.

To specify more than one driver class, separate each class name from the next with a colon (:). You might do this if one Java program makes a local call to another but accesses a different database management system.

Related reference

“Build descriptor options” on page 237

“Informix and EGL” on page 199

“sqlDB” on page 262

“sqlID” on page 263

“sqlPassword”

“sqlValidationConnectionURL” on page 265

“validateSQLStatements” on page 269

sqlJNDIName

The build descriptor option **sqlJNDIName** specifies the default database used by a generated Java program that runs in J2EE. The value is the name to which the default datasource is bound in the JNDI registry; for example, `jdbc/MyDB`.

The option **sqlJNDIName** is case-sensitive, has no default value, and is used only when you are generating a Java program for J2EE. The option assigns a value to the Java run-time property **vgj.jdbc.default.database**, but only if option **genProperties** is set to `GLOBAL` or `PROGRAM`.

To specify the database used for validation (in relation to Java or COBOL generation), set **sqlValidationConnectionURL**.

Related concepts

“Java run-time properties” on page 421

“SQL support” on page 171

Related reference

“Build descriptor options” on page 237

“genProperties” on page 254

“Java run-time properties (details)” on page 423

“sqlPassword”

“sqlValidationConnectionURL” on page 265

“sqlJDBCDriverClass” on page 263

“validateSQLStatements” on page 269

sqlPassword

The build descriptor option **sqlPassword** specifies a password that is used to connect to a database during generation-time validation of SQL statements. You specify the database by setting **sqlValidationConnectionURL**.

When you generate a Java program, EGL also assigns the value of **sqlPassword** to the Java run-time property **vgj.jdbc.default.password**. That property identifies the password for connecting to the default database at run time, and you can specify the default database in **sqlDB**.

The option **sqlPassword** is case-sensitive and has no default value.

Related concepts

"Java run-time properties" on page 421

Related reference

"Build descriptor options" on page 237

"Java run-time properties (details)" on page 423

"sqlDB" on page 262

"sqlID" on page 263

"sqlValidationConnectionURL"

"sqlJDBCDriverClass" on page 263

"validateSQLStatements" on page 269

sqlValidationConnectionURL

The build descriptor option **sqlValidationConnectionURL** specifies a URL for connecting to the database that EGL uses to validate SQL statements at generation time. Database access is through JDBC.

No default is available for the URL, and the format varies by driver:

- For IBM DB2 APP DRIVER for Windows, the URL is as follows--

`jdbc:db2:dbName`

dbName

Database name

- For the Oracle JDBC thin client-side driver, the URL varies by database location. If the database is local to your machine, the URL is as follows--

`jdbc:oracle:thin:dbName`

If the database is on a remote server, the URL is as follows--

`jdbc:oracle:thin:@host:port:dbName`

host

Host name of the database server

port

Port number

dbName

Database name

- For the IBM Informix JDBC driver, the URL is as follows (with the lines combined into one)--

`jdbc:informix-sqli://host:port
/dbName:informixserver=servername;
user=userName;password=passWord`

host

Name of the machine on which the database server resides

port

Port number

dbName

Database name

serverName

Name of the database server

userName

Informix user ID

passWord

Password associated with the user ID

- For other drivers, refer to the documentation for the driver.

Related reference

“Build descriptor options” on page 237

“Informix and EGL” on page 199

“sqlDB” on page 262

“sqlID” on page 263

“sqlPassword” on page 264

“sqlJDBCClass” on page 263

“validateSQLStatements” on page 269

sysCodes

The build descriptor option **sysCodes** determines the source of the code that is placed in the system word **sysVar.errorCode** in response to a file I/O error. Values are as follows:

NO

sysVar.errorCode receives codes that are returned from EGL run-time services.

NO is the default.

YES

sysVar.errorCode receives code that are returned from the operating system.

The code is specific to the type of resource being accessed (VSAM rather than a transient data queue, for example). For details on the meaning of specific error codes, see the reference material provided for the operating system or for the subsystem (like VSAM).

For additional details, including specific error values, see *sysVar.errorCode*.

Related reference

“Build descriptor options” on page 237

“Exception handling” on page 75

“sysVar.errorCode” on page 530

system

The build descriptor option **system** specifies the target platform for generation. This option is required; no default value is available. Valid values are as follows:

AIX

Indicates that generation produces a Java program that can run on AIX

ISERIESC

Indicates that generation produces a COBOL program that can run on iSeries

ISERIESJ

Indicates that generation produces a Java program that can run on iSeries

LINUX

Indicates that generation produces a Java program that can run on Linux (with an Intel processor)

USS

Indicates that generation produces a Java program that can run on z/OS UNIX System Services

WIN

Indicates that generation produces a Java program that can run on Windows 2000/NT/XP

Related concepts

"Generated output" on page 386

"Linkage options part" on page 439

"Run-time configurations" on page 2

Related reference

"Build descriptor options" on page 237

"callLink element" on page 443

"Generated output (reference)" on page 387

"Informix and EGL" on page 199

targetNLS

The build descriptor option **targetNLS** specifies the national language code used to identify run-time messages.

The next table lists the supported languages. The code page for the language you specify must be loaded on your target platform.

Code	Languages
CHS	Simplified Chinese
CHT	Traditional Chinese
DES	Swiss German
DEU	German
ENP	Uppercase English (not supported on Windows 2000, Windows NT, and z/OS UNIX System Services)
ENU	US English
ESP	Spanish
FRA	French
ITA	Italian
JPN	Japanese
KOR	Korean
PTB	Brazilian Portuguese

EGL determines if the Java locale on the development machine is associated with one of the supported languages. If the answer is "yes," the default value of **targetNLS** is the supported language. Otherwise, **targetNLS** has no default value.

Related reference

"Build descriptor options" on page 237

templateDir

The build descriptor option **templateDir** specifies the directory containing templates that the iSeries build server uses to create the CL program identified by the symbolic parameter %EZEMBR%_R. That program is used at runtime only if it is called from a client program running on a workstation.

This option is meaningful only when you are generating a program of type *iSeries*.

Related concepts

“Build descriptor part” on page 234

“Run-time configurations” on page 2

Related reference

“Build descriptor options” on page 237

VAGCompatibility

The build descriptor option **VAGCompatibility** indicates whether the generation process allows use of special program syntax, as described in *Compatibility with VisualAge Generator*. Valid values are *no* and *yes*.

The setting of the EGL preference **VAGCompatibility** determines the default value of the build descriptor. If you are generating in the EGL SDK, no preferences are available, and the default value of **VAGCompatibility** is *no*.

Specify *yes* only if your program or page handler uses the special syntax.

Related concepts

“Build descriptor part” on page 234

“Compatibility with VisualAge Generator” on page 276

Related reference

“Build descriptor options” on page 237

validateMixedItems

The build descriptor option **validateMixedItems** specifies whether a generated COBOL program validates the integrity of DBCHAR strings when an item of type MBCHAR is assigned to an item of type MBCHAR. Valid values are YES and NO. YES is the default.

The value YES means that EGL run-time services raises an error if a DBCHAR string is truncated as a result of a MBCHAR-to-MBCHAR assignment on the mainframe. If the error occurs in an invoked function, the function returns control, and the result depends on code aspects that are described in *Exception handling*. If the error occurs in the main function, the program ends with an error message.

If your code is meant for the mainframe and assigns values to items of type MBCHAR, and if the error situation is not possible, set **validateMixedItems** to NO for better run-time performance.

Related reference

“Build descriptor options” on page 237

“CHAR” on page 29

“DBCHAR” on page 30

“Exception handling” on page 75

“MBCHAR” on page 31

validateOnlyIfModified

The build descriptor option **validateOnlyIfModified** specifies whether to validate only those text-form fields for which the modified data tag is set. Valid values are as follows:

no (the default)

Validate all variable fields.

yes

Validate only fields for which the modified data tag is set.

Related concepts

“Build descriptor part” on page 234

“Modified data tag and modified property” on page 152

Related reference

“Build descriptor options” on page 237

validateSQLStatements

The build descriptor option **validateSQLStatements** indicates whether SQL statements are validated against a database. Successful use of **validateSQLStatements** requires that you specify option **sqlValidationConnectionURL** and, in most cases, other options that begin with the letters **sql**, as listed later.

Valid values are YES and NO, and the default is NO. Validation of SQL statements increases the time required to generate your code.

When you request SQL validation, the database manager accessed from the generation platform prepares the SQL statements dynamically.

SQL statement validation has these restrictions:

- No validation is possible for SQL statements that use dynamic SQL and are based on SQL records
- The validation process may indicate errors that are found by the database manager in the generation environment but that will not be found by the database manager on the target platform
- Validation occurs only if your JDBC driver supports validation of SQL prepare statements and (in some cases) only if you have configured the driver to do such validation; for details, see the documentation for your JDBC driver

Related reference

“Build descriptor options” on page 237

“sqlID” on page 263

“sqlPassword” on page 264

“sqlValidationConnectionURL” on page 265

“sqlJDBCDriverClass” on page 263

EGL build-file format

The structure of a .eglbld file is as follows:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EGL PUBLIC "-//IBM//DTD EGL 5.1//EN" "">
<EGL>
  <!-- place your import statements here -->
  <!-- place your parts here -->
</EGL>

```

Your task is to place import statements and parts inside the <EGL> element.

You specify <import> elements to reference the file containing the next build descriptor in a chain or to reference any of the build parts referenced by a build descriptor. An example of an import statement is as follows:

```
<import file="myBldFile.egl.bld"/>
```

You declare parts from this list:

- <BuildDescriptor>
- <LinkageOptions>
- <ResourceAssociations>

A simple example is as follows:

```

<EGL>
  <import file="myBldFile.egl.bld"/>
  <BuildDescriptor name="myBuildDescriptor"
    genProject="myNextProject"
    system="WIN"
    J2EE="NO"
    genProperties="GLOBAL"
    genDataTables="YES"
    dbms="DB2"
    sqlValidationConnectionURL="jdbc:db2:SAMPLE"
    sqlJDBCClass="COM.ibm.db2.jdbc.app.DB2Driver"
    sqlDB="jdbc:db2:SAMPLE"
  </BuildDescriptor>
</EGL>

```

You can review the build-file DTD, which is in the following subdirectory:

```

installationDir\wstools\eclipse\plugins\
com.ibm.etools.egl_version\dtd

```

installationDir

The WebSphere Studio installation directory, such as c:\myStudio

version

The installed version of the plugin; for example, 5.1.2

The file name (like egl_5_1.dtd) begins with the letters *egl* and an underscore.

Related concepts

“Import” on page 26

“Parts” on page 11

Related tasks

“Creating an EGL source file” on page 85

Related reference

“EGL editor” on page 103

Build script reference

Build script

A build script is a file that is invoked by a build plan and that prepares output from generated files. Examples are as follows:

- A Java compiler or other .exe (binary) file or a .bat (text) file is available to a build server on the development system or is sent to a build server on a remote Windows 2000/NT/XP.
- A script (.scr file) or some binary code is sent to a USS build server.

You specify the address of a build machine by setting the build descriptor option **destHost**.

COBOL build script for iSeries

The build script for iSeries is a REXX program named FDAPREP and is described in the *EGL Server Guide for iSeries*, which is available in the help system.

Java build script

To prepare Java code for execution, EGL puts the javac (Java compiler) command and its parameters in the build plan and sends to the build machine the javac command and the input required by the command.

Related concepts

"Build" on page 121

"Build plan" on page 392

"Build server" on page 275

Related reference

"Build descriptor options" on page 237

"Build scripts delivered with EGL"

"destDirectory" on page 246

"destHost" on page 247

"destPassword" on page 248

"destUserID" on page 249

"Output of COBOL generation" on page 388

"Output of Java program generation" on page 388

"Output of Java wrapper generation" on page 389

Build scripts delivered with EGL

On iSeries, the EGL build server invokes the build script FDAPREP. The script normally resides in the QEGL/REXSRC file but can be copied to another location and customized.

Related concepts

"Build script"

Options required in EGL build scripts

In EGL build scripts, certain preparation options are required if you are using DB2 UDB.

Required options for DB2 precompiler

The following options are required for DB2 usage and are included in the fdaptcl build script:

- HOST(COB2)
- APOSTSQL
- QUOTE

Symbolic parameters

A symbolic parameter is a variable that is substituted in certain build descriptor options and in iSeries build scripts to affect placement and preparation of generated outputs.

Some symbolic parameters are predefined by the generator. EZEETIME, for example, contains the time at which generation occurs. For a list of these, see *Predefined symbolic parameters for EGL generation*. Users may also define their own symbolic parameters.

You can use symbolic parameters in the values for the **genDirectory** and the **destDirectory** generation options and in build scripts.

When specifying symbolic parameters with **genDirectory** and **destDirectory**, you reference the value of a symbolic parameter by delimiting the parameter name with percentage signs (%). If you want to refer to the time at which generation occurs, for example, specify %EZEETIME%.

You can also use more than one symbolic parameter to assign a value. The following symbols represent date and time, separated by a space:

```
%EZEEDATE% %EZEETIME%
```

When specifying symbolic parameters in build scripts you reference the value by preceding the symbolic parameter name with an ampersand (&) and appending a period (.).

```
&EZEEDATE.
```

For example, if **genDirectory** is set to C:\MyProject\%EZEENV%, and the build descriptor option is set to ZOSCICS, then generation outputs will be written to C:\MyProject\ZOSCICS. (The EZEENV predefined symbolic parameter is populated from the option **system**.)

You also can specify your own symbolic parameters, and for each such parameter you assign a value; **MYDIR**, for example, might contain the name of a directory. It is not valid to define the same symbolic parameter (like MYDIR) twice in the same build descriptor.

Note: User-defined symbols cannot start with the prefix EZE.

If the build descriptor you are using for generation uses the **nextBuildDescriptor** option to chain multiple build descriptors, and you define the same-named symbolic parameter in multiple build descriptors that are chained together, the value in use at generation time is determined by the precedence rules described in the page on build descriptors.

The value assigned to the symbolic parameter MYDIR in the master build descriptor, for example, takes precedence over the value assigned to MYDIR in any other build descriptor.

Both predefined and user-defined symbolic parameters are available as substitution variables in the build scripts used to prepare COBOL output. For details, see the *EGL Server Guide for iSeries*.

Predefined symbols

The substitution values for predefined symbols are automatically set at generation. You do not define values for these symbols. There are two categories of predefined symbols.

- Symbolic parameters for EGL generation
-

User-defined symbols cannot start with the prefix EZE.

Related concepts

“Build descriptor part” on page 234

“Source of additional information on EGL” on page 5

Related reference

“Build descriptor options” on page 237

“Predefined symbolic parameters for EGL generation”

Predefined symbolic parameters for EGL generation

All of the EGL generator’s symbolic parameters, whether predefined or user-defined, are passed as environment variables to the build server when you build a generated COBOL program. Environment variables passed to the build server supply values for build script substitution variables of the same name. The environment variable values override any default values defined for the substitution variables.

The next table shows the predefined symbolic parameters.

Name	Description
BUILD_SCRIPT_LIBRARY	<p>Allows overriding the name of the PDS from which the build server reads the build scripts.</p> <p>This symbolic parameter is useful when exception build processing is needed. For example, you can use a separate build script PDS if a special test system is needed with a separate database, COBOL libraries, or CICS libraries. The build scripts could have different default substitution variables or different compile options.</p> <p>An alternative approach is to start a build server on a different port and to allocate a different build script PDS.</p>
DATA	<p>A flag that specifies whether you want to allocate working storage with 24- or 31- bit addresses. The value supplied for this symbolic parameter is passed in the DATA parameter for the COBOL compiler and the z/OS linkage editor. The parameter’s value is taken from the build descriptor option data.</p>

Name	Description
EZEALIAS	The member name used to store the currently generated member in its associated PDS. If an alias property was specified for the currently generated member, the value of that property, truncated to 8 characters if necessary, is used. If no alias property is specified, then the part name, truncated to 8 characters if necessary, is used. When a form group is the current member, and the form group has print forms included, the format module name is truncated to 6 characters rather than 8, and the characters FM are appended.
EZEGDATE	The date on which a program is generated. The format is <i>mm/dd/yy</i> , where <i>mm</i> is the two-digit month, <i>dd</i> is the two-digit day of the month, and <i>yy</i> is the last two digits of the year.
EZEGMBR	The name of the program part that was specified to start generation.
EZEMBR	The name of the program that was generated. The value will be the same as the value of the MBR parameter unless you specified the alias property for the program generated, or the name of the generated program is longer than 8 characters.
EZEPID	The high-level qualifier that is used for the PDSs that receive the generated and built outputs. The parameter's value is taken from the build descriptor option projectID .
EZESQL	An indicator as to whether or not the generated part performs SQL I/O. "Y" indicates yes; "N" indicates no.
EZETIME	The time at which a program is generated. The format is <i>hh:mm:ss</i> , where <i>hh</i> is the hour, <i>mm</i> is the minute, and <i>ss</i> is the seconds portion of the time.
EZEEXTNM	The external name, if any, that is specified in the program part's alias property. This symbolic parameter is available only during generation of bind control and link edit files. It is not available when build scripts are being executed. If the external name is not specified, the name of the part is used but is truncated (if necessary) to the maximum number of characters allowed in the run-time environment.
MBR	The external name given to the generated source code. The external name is the same as the EGL program name unless an alias property was specified for the program or the program name is longer than 8 characters. If the alias property was specified, its value is placed in the MBR environment variable. Otherwise, if the program name is longer than 8 characters, the program name is truncated to 8 characters and the result is placed in the MBR environment variable.
SYSTEM	The target system for which the EGL program was generated; for example, ZOSCICS or ISERIESC. The parameter's value is taken from the build descriptor option system .

In addition to these predefined symbolic parameters, you can define your own symbolic parameters that are passed to the build server as environment variables. If the build script contains a substitution variable whose name matches the

symbolic parameter name, the build server uses the value of the symbolic parameter in the build script, in place of the substitution variable.

Related concepts

“Generation” on page 123

Build server

A build server receives requests from a client system to create executable files from source code sent from that client. A build server must be started prior to sending any requests from a build client. A build server typically services requests from multiple clients. Multiple threads may be started if concurrent build requests are received.

In a generator environment you start a build server on a machine whose operating system is the target generation system, for example, Windows 2000. The generator produces Java source code. Java code is sent to a specified build server where the Java compiler is invoked.

If you are generating Java code for Windows, you can build the Java outputs on the same machine as the machine where generation was performed. This is called a local build. In this case you do not have to start a build server. If you want to perform a local build, omit the **destHost** option from the build descriptor.

Related concepts

“Build” on page 121

“Build script” on page 271

Related tasks

“Starting a build server on AIX, Linux, or Windows 2000/NT/XP” on page 129

Related reference

“Build descriptor options” on page 237

COBOL reserved-word file

The COBOL reserved word file is a text file that contains reserved words other than EGL reserved words. When EGL generates a COBOL program and finds one of the listed words used as a function name, record name, structure name, or variable name, the name is aliased. If the name of an EGL item would cause the COBOL generator to create a COBOL variable matching a word in the file, the generator instead uses an alias that does not conflict with other variable names and does not match a word in the reserved-word file.

COBOL reserved words documented at the time the generator was developed are automatically reserved by the COBOL generator. You do not need to use a COBOL reserved-word file unless a new keyword is introduced by the COBOL compiler or unless you just have some words that you do not want used as COBOL variable names.

If you have added a new word to the COBOL reserved-word file, set the build descriptor option **reservedWord** to the fully qualified path name of that file.

Related concepts

“COBOL program” on page 393

Related reference

“Format of COBOL reserved-word file”
“How COBOL names are aliased” on page 464
“EGL reserved words” on page 290
“reservedWord” on page 259

Format of COBOL reserved-word file

A reserved-word file contains a list of reserved words that are in addition to the built-in COBOL reserved words. The format of the file is one reserved word per line. An EGL reserved-word file applies only to COBOL generation.

If you use a reserved word as the name of a program, the EGL generator exits with an error. If you use a reserved word as the name of another part, the generator aliases it.

Related concepts

“COBOL reserved-word file” on page 275

Related reference

“How COBOL names are aliased” on page 464

Compatibility with VisualAge Generator

EGL is the replacement for VisualAge Generator 4.5 and includes some syntax primarily to enable the migration of existing programs to the new development environment. This syntax is supported in the development environment if the EGL preference **VAGCompatibility** is selected or (at generation or debug time) if the build descriptor option **VAGCompatibility** is set to *yes*. The setting of the preference also establishes the default value of the build descriptor option.

The following statements apply when VisualAge Generator compatibility is in effect:

- Three otherwise invalid characters (- @ #) are valid in identifiers, although the hyphen (-) and pound sign (#) are each invalid as an initial character in any case; for details, see *Naming conventions*
- If you refer to a static, single-dimension array of structure items without specifying an index, the array index defaults to 1; for details, see *Arrays*
- The primitive types NUMC and PACF are available, as described in *Primitive types*
- If you specify an even length for an item of primitive type DECIMAL, EGL increments the length by one except when the item is used as an SQL host variable.
- The SQL item property **SQLDataCode** is available, as described in *SQL item properties*
- A set of call options are available in the call statement
- The option **externallyDefined** is in the statements show and transfer
- The following system variables are available:
 - sysVar.handleSysLibErrors
 - sysVar.segmentedMode
- The following system functions are available:
 - sysLib.getVAGSysType
 - sysLib.connectionService

- You cannot use a dynamic array as an argument to a function or to another EGL program.
- The libraries **sysMath** and **sysStr** are automatically available to your code.
- You can issue a statement of the following form:

```
display printForm
```

```
printForm
```

Name of a print form that is visible to the program.

In that case, **display** is equivalent to **print**.

- The following program properties also support VAG compatibility.
 - **AllowUnqualifiedItemReferences**
 - **IncludeReferencedFunctions**

For details, see *Program part in EGL source format*.

- If you set the text-form property **value**, the content of that property is available in the program only after the user has returned the form. For this reason, the value that you set in the program does not need to be valid for the item in the program.

Related reference

“Arrays” on page 64

“call” on page 299

“Input form” on page 486

“Input record” on page 487

“Naming conventions” on page 468

“pfKeyEquate” on page 380

“Primitive types” on page 27

“print” on page 341

“Program part in EGL source format” on page 478

“show” on page 353

“SQL item properties” on page 57

“sysLib.connectionService” on page 545

“sysLib.getVAGSysType” on page 613

“sysVar.handleSysLibErrors” on page 533

“sysVar.segmentedMode” on page 626

“transfer” on page 354

Data initialization

If an EGL-generated program initializes a record automatically (as occurs in some cases, described later), each of the lowest-level structure items is set to a value appropriate to the primitive type. Form initialization is similar, except that your form declaration can assign values that override the defaults.

Initialization also occurs in these situations:

- The **initialized** property of a variable (specifically, of an item or record or static array) is set to *yes*.
- Your logic includes certain variations of set

The next table gives details on the initialization values.

Primitive type	Initialization value
BIN and the integer types (BIGINT, INT, and SMALLINT), HEX	Binary zeros

Primitive type	Initialization value
CHAR, MBCHAR	Single-byte blanks
DBCHAR	Double-byte blanks
DECIMAL, NUM, NUMC, PACF	Numeric zeros
UNICODE	Unicode blanks (each of which is hexadecimal 0020)

In a structure, only the lowest-level structure items are considered. If a structure item of type HEX is subordinate to a structure item of type CHAR, for example, the memory area is initialized with binary zeros.

Records or items that are received as program or function parameters are never initialized automatically.

An EGL-generated Java program initializes records, whether local or global.

An EGL-generated COBOL program initializes the input record, which is identified in the program properties. Other record initialization depends on whether you set the *initialized* property for a given variable. If you do not, record initialization depends on how you set two build descriptor options at generation time:

- Setting `initNonIOData` to YES causes the generated program to initialize global basic records
- Setting `initIORecords` to YES causes the generated program to initialize other global records

In keeping with the behavior of COBOL programs in general, EGL-generated COBOL programs do *not* initialize *local* records.

If you generate a COBOL program that compares an item of type NUM with an item of type CHAR, make sure that your code initializes the items; otherwise, the comparison may cause the program to fail with an abend (an abnormal end), in which case no exception-handling code is run. A similar, COBOL-specific warning applies to structure items in local structures and records.

Related concepts

“Function part” on page 380

“DataItem part” on page 284

“Program part” on page 477

“Record parts” on page 490

“Structure” on page 19

Related reference

“EGL statements” on page 70

“`initNonIOData`” on page 255

“`initIORecords`” on page 255

“Items” on page 43

“set” on page 345

Data conversion

Because of differences in how data is interpreted in different run-time environments, your program may need to convert the data that passes from one environment to another. Data conversion occurs at COBOL preparation time and at COBOL or Java run time.

The COBOL preparation process converts file content, file-path information, and values of environment variables when transferring workstation-based files to a build server. The steps needed to establish a data conversion table in this case are described later.

Your code also uses a conversion table in the following run-time situations:

- Your EGL-generated Java code calls a program on CICS for z/OS.
In this case, you can specify the conversion table in a `callLink` element that refers to the called program. Alternatively, you can indicate (in that `callLink` element) that the system variable `sysVar.callConversionTable` identifies the conversion table at run time.
- Your EGL-generated COBOL program calls a program residing on a remote platform that supports the ASCII character set.
In this case, you also can specify the conversion table in a `callLink` element that refers to the called program. Alternatively, you can indicate (in that `callLink` element) that the system variable `sysVar.callConversionTable` identifies the conversion table at run time.
- An EGL-generated Java or COBOL program (on a platform that supports the EBCDIC character set) transfers asynchronously to a program on a platform that supports the ASCII character set, as might occur when the transferring program invokes the system function `sysLib.startTransaction`.
In this case, you can specify the conversion table in an `asynchLink` element that refers to the program to which control is transferred. Alternatively, you can indicate (in that `asynchLink` element) that the system variable `sysVar.callConversionTable` identifies the conversion table at run time.
- An EGL-generated Java program shows a text or print form that includes series of Arabic or Hebrew characters; or presents a text form that accepts a series of such characters from the user.
In these cases, you specify the bidirectional conversion table in the system variable `sysVar.formConversionTable`.

You would use run-time conversion, for example, if your code places values into one of two redefined records, each of which refers to the same area of memory as a record that is passed to another program. Assume that the characteristics of the data that you pass would be different, depending on the redefined record to which you assign values. In this case, the requirements of data conversion cannot be known at generation time.

The next sections provide the following details:

- “Data conversion when you generate a COBOL program”
- “Data conversion when the invoker is Java code” on page 280
- “Conversion algorithm” on page 281

Data conversion when you generate a COBOL program

When COBOL is generated on a workstation and prepared on an iSeries build server, conversion is handled on the build server in accordance with your

specification in build descriptor options `clientCodeSet` and `serverCodeSet`. Each of those build descriptor options must identify a code set that is defined to the ICONV conversion service on iSeries, and default settings are used in the absence of a specification.

See also “Bidirectional language text” on page 282.

Data conversion when the invoker is Java code

The following rules pertain to Java code:

- When a generated Java program or wrapper invokes a generated Java program, conversion occurs in the caller, in accordance with a set of EGL classes invoked at run-time. It is sufficient to request no conversion at all in most cases, even if the caller is accessing a remote platform that uses a code page that is different from the code page used by the invoker. You must specify a conversion table, however, in the following situation:
 - The caller is Java code and is on a machine that supports one code page
 - The called program is non-Java and is on a machine that supports another code page

The table name in this case is a symbol that indicates the kind of conversion that is required at run time.

- When a generated Java program accesses a remote MQSeries message queue, conversion occurs in the invoker, in accordance with a set of EGL classes invoked at run time. If the caller is accessing a remote platform that uses a code page that is different from the code page used by the invoker, specify a conversion table in the association element that refers to the MQSeries message queue.

The next table lists the conversion tables that can be accessed by generated Java code at run time. Each name has the format CSOcx:

- c Represents the character set supported on the invoked platform. Select one of these:
 - J for Java (if the called program is an EGL-generated Java program)
 - E for EBCDIC (if the called platform is an EGL-generated COBOL program)
- x Represents the code page number on the invoked platform. Each number is specified in the *Character Data Representation Architecture Reference and Registry*, SC09-2190. The registry identifies the coded character sets supported by the conversion tables.

Language	Platform of Invoked Program			
	UNIX	Windows 2000/NT/XP	z/OS UNIX System Services or iSeries Java	iSeries COBOL
Arabic	CSOJ1046	CSOJ1256	CSOJ420	CSOE420
Chinese, simplified	CSOJ1381	CSOJ1386	CSOJ1388	CSOE1388
Chinese, traditional	CSOJ950	CSOJ950	CSOJ1371	CSOE1371
Cyrillic	CSOJ866	CSOJ1251	CSOJ1025	CSOE1025
Danish	CSOJ850	CSOJ850	CSOJ277	CSOE277
Eastern European	CSOJ852	CSOJ1250	CSOJ870	CSOE870

Language	Platform of Invoked Program			
	UNIX	Windows 2000/NT/XP	z/OS UNIX System Services or iSeries Java	iSeries COBOL
English (UK)	CSOJ850	CSOJ1252	CSOJ285	CSOE285
English (US)	CSOJ850	CSOJ1252	CSOJ037	CSOE037
French	CSOJ850	CSOJ1252	CSOJ297	CSOE297
German	CSOJ850	CSOJ1252	CSOJ273	CSOE273
Greek	CSOJ813	CSOJ1253	CSOJ875	CSOE875
Hebrew	CSOJ856	CSOJ1255	CSOJ424	CSOE424
Japanese	CSOJ943	CSOJ943	CSOJ1390 (Katakana SBCS), CSOJ1399 (Latin SBCS)	CSOE1390 (Katakana SBCS), CSOE1399 (Latin SBCS)
Korean	CSOJ1363	CSOJ1363	CSOJ1364	CSOE1364
Portuguese	CSOJ850	CSOJ1252	CSOJ037	CSOE037
Spanish	CSOJ850	CSOJ1252	CSOJ284	CSOE284
Swedish	CSOJ850	CSOJ1252	CSOJ278	CSOE278
Swiss German	CSOJ850	CSOJ1252	CSOJ500	CSOE500
Turkish	CSOJ920	CSOJ1254	CSOJ1026	CSOE1026

If you do not specify a value for the conversion table in the linkage options part when you are calling a program from Java, the default conversion tables are those for English (US).

Conversion algorithm

Data conversion of records and structures is based on the declarations of the structure items that lack a substructure.

Data of type CHAR, DBCHAR, or MBCHAR is converted in accordance with the COBOL or Java conversion tables (for conversion that occurs in an EGL-generated invoker).

No conversion is performed for filler data items (data items that have no name) or for data items of type DECIMAL, PACF, HEX, or UNICODE.

On EBCDIC-to-ASCII conversion for MBCHAR data, the conversion routine deletes shift-out/shift-in (SO/SI) characters and inserts an equivalent number of blanks at the end of the data item. On ASCII-to-EBCDIC conversion, the conversion routine inserts SO/SI characters around double-byte strings and truncates the value at the last valid character that can fit in the field. If the MBCHAR field is in a variable length record and the current record end is in the MBCHAR field, the record length is adjusted to reflect the insertion or deletion of SO/SI characters. The record length indicates where the current record ends.

For data items of type BIN, the conversion routine reverses the byte order of the item if the caller or called platform uses Intel binary format and the other platform does not.

For data items of type NUM or NUMC items, the conversion routine converts all but the last byte using the CHAR algorithm. The sign half-byte (the first half byte of the last byte in the field) is converted according to the hexadecimal values shown in the next table.

EBCDIC for type NUM	EBCDIC for type NUMC	ASCII
F (positive sign)	C	3
D (negative sign)	D	7

Related reference

“Association elements” on page 231

“bidiConversionTable” on page 243

“Bidirectional language text”

“callLink element” on page 443

“clientCodeSet” on page 245

“serverCodeSet” on page 259

“sysLib.convert” on page 518

“sysVar.callConversionTable” on page 519

Bidirectional language text

Bidirectional (bidi) languages such as Arabic and Hebrew are languages in which the text is presented to the user ordered from right to left, but numbers and Latin alphabetic strings within the text are presented left to right. In addition, the order in which characters appear within program variables can vary. In COBOL environments, the text in program variables is usually in *visual* order, which is the same order in which the text appears on the user interface. In Java environments, the text is usually stored in *logical* order, the order in which the characters are entered in the input field.

These differences in ordering and in other associated presentation characteristics require the program to have the ability to convert bi-directional text strings from one format to another. The bidi conversion attributes are specified in a bidi conversion table (.bct) file created separately from the program. The program references the name of the conversion table to indicate how attribute conversion should be performed.

In all cases, the bidi conversion table reference is specified as the 1 to 8 character file name without the .bct extension. For example, if you have created a bidi conversion table named mybct.bct, you can set the value of formConversionTable in a program by adding the following statement at the beginning of the program:

```
sysVar.formConversionTable = "mybct.bct" ;
```

Your tasks are as follows:

- Create bidi conversion tables that specify the transformations that should occur. Note that different tables are needed for converting data being passed between a Java client and a COBOL host and for converting data to be displayed in a text or print form in a Java environment.
- Reference the appropriate table in the appropriate situation:
 - When generating a COBOL program that uses forms, data tables, or literals with bidi language text, set the build descriptor option bidiConversionTable

- When generating Java programs that call remote COBOL programs, customize the linkage options part so that the property `conversionTable` is in the `callLink` element (or, for asynchronous transfers, in the `asynchLink` element) for the invoked program:
 - You can specify a conversion table as the value of that property; or
 - You can set the property to `programControlled`, which means that the invoking program specifies the conversion table before invoking the other program. The invoker specifies the table by assigning the conversion table name to the system variable `sysVar.callConversionTable`.
- When generating a Java program that uses text or print forms with bidi language text, add a statement to the program that assigns the conversion table name to the system function `sysVar.formConversionTable` before showing the form.

You build the bidi conversion table file using the bidi conversion table wizard plugin, which is in the file `BidiConversionTable.zip`:

1. Download the file from the following web site:
2. Unzip the file into your workbench directory
3. To begin running the wizard, click **File > New > Other > BidiConversionTable**.

The name of a table used with EGL programs must have eight characters or less and must have the `.bct` extension.

4. While running the wizard, press F1 for help in choosing the correct options for creating the table.

When creating a bidi conversion table for generating COBOL programs, specify the client encoding and server encoding as shown in the next table.

Language	Client encoding for bidi conversion table	Server encoding for bidi conversion table
Arabic	Cp1256	Cp864
Hebrew	Cp1255	Cp1255

The bidi conversion table controls the transformation of the text from logical to visual order for the COBOL environment, along with any other formatting transformation requested in the table. At program-generation time, a pair of build descriptor options (`clientCodeSet` and `serverCodeSet`) control the conversion of the code page from ASCII to EBCDIC, as shown in the next table.

Language	clientCodeSet	serverCodeSet
Arabic	IBM-864	IBM-420
Hebrew	IBM-1255	IBM-424

Related reference

“`bidiConversionTable`” on page 243

“`clientCodeSet`” on page 245

“Data conversion” on page 279

“`serverCodeSet`” on page 259

“`sysVar.callConversionTable`” on page 519

DatalItem part

A *datalItem* part defines an area of memory that cannot be subdivided. A *datalItem* part is a standalone part, unlike a structure item in a structure.

A *data item* is a memory area that is based on a *datalItem* part. You may use a data item in these ways:

- As a parameter that receives data into a function or program
- As a variable in an EGL function; for instance, in an assignment statement or as an argument that passes data to another function or program

Each data item has a series of properties, whether by default or as specified in the *datalItem* part. For details, see *Overview of EGL properties and overrides*.

Related concepts

"Parts" on page 11

"Overview of EGL properties and overrides" on page 40

"Record parts" on page 490

"Structure" on page 19

"Typedef" on page 20

Related tasks

"Setting preferences for templates" on page 105

Related reference

"EGL source format" on page 292

"Data initialization" on page 277

"Items" on page 43

"Primitive types" on page 27

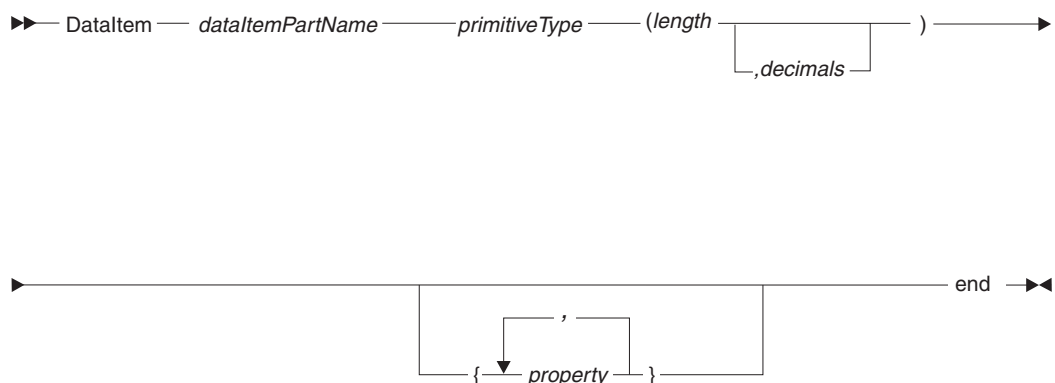
DatalItem part in EGL source format

You declare a *DatalItem* part in an EGL file, which is described in *EGL source format*.

An example of a data item part is as follows:

```
DatalItem myDataItemPart
  BIN(9,2)
end
```

The syntax diagram for a *datalItem* part is as follows:



DataItem dataItemPartName ... end

Identifies the part as a dataItem part and specifies the name. For rules, see *naming conventions*.

primitiveType

The primitive type assigned to the dataItem part.

length

An integer that reflects the length of the dataItem part. The value of any variable that is based on the part includes the specified number of characters or digits.

decimals

For a numeric type (BIN, DECIMAL, NUM, NUMC, or PACF), you may specify *decimals*, which is an integer that represents the number of places after the decimal point. The maximum number of decimal positions is the smaller of two numbers: 18 or the number of digits declared as *length*. The decimal point is not stored with the data.

property

An item property, as described in *Overview of EGL properties and overrides*.

Related concepts

- "DataItem part" on page 284
- "EGL projects, packages, and files" on page 7
- "Overview of EGL properties and overrides" on page 40
- "References to parts" on page 16
- "Parts" on page 11

Related tasks

- "Syntax diagram" on page 506

Related reference

- "EGL source format" on page 292
- "Function part in EGL source format" on page 381
- "Indexed record part in EGL source format" on page 492
- "MQ record part in EGL source format" on page 493
- "Naming conventions" on page 468
- "Primitive types" on page 27
- "Program part in EGL source format" on page 478
- "Relative record part in EGL source format" on page 495
- "Serial record part in EGL source format" on page 497
- "SQL record part in EGL source format" on page 498

DataTable part

An EGL *dataTable part* is primarily composed of these components:

- A structure, with each top-level item defining a column.
- An array of values that are consistent with those columns. Each element of that array defines a row.

A data table of error messages, for example, might include these components:

- The declaration of a numeric field and a character field
- A list of paired values like these—
 - 001 Error 1
 - 002 Error 2
 - 003 Error 3

You do not declare a data table as if you were declaring a record or data item. Instead, any code that can access a table part can treat that part as a variable. For details on part access, see *References to parts*.

Any code that can access a table part has the option of referencing the part name in a Use declaration.

Types of data tables

Some types of data tables are for run-time validation; specifically, to hold data for comparison against form input. (You relate the data table to the input field when you declare the form part.) Three types of validation data tables are available:

matchValidTable

The user's input must match a value in the first data-table column.

matchInvalidTable

The user's input must be different from any value in the first data-table column.

rangeChkTable

The user's input must match a value that is between the values in the first and second column of at least one data-table row. (The range is inclusive; the user's input is valid if it matches a value in the first or second column of any row.)

The other types of data tables are as follows:

msgTable

Contains run-time messages.

basicTable

Contains other information that is used in the program logic; for example, a list of countries and related codes.

Data-table generation

The output generated for a dataTable part varies by the output language:

- If you are generating output in Java, each data table is generated as a pair of files, each named for the data table. One file has the extension .java, the other has the extension .tab. The .tab file is not processed by the Java compiler, but is included in the root of the directory structure that contains the package. If the package is *my.product.package*, for example, the directory structure is *my/product/package*, and the .tab file is in the directory that contains the subdirectory *my*.
- If you are generating output for COBOL, a data table is generated as a separate program, with the file extension .cbl. EGL also produces a binary file that has the file extension .tab. You do not compile that file; it is read as is at runtime.

You do not need to generate the data tables if you are generating into a directory or Java package to which you had previously generated the same data tables.

To save generation time when you do not need to generate data tables, assign NO to the build descriptor option **genTables**.

Properties of the data table

You can set the following properties:

- An **alias** is incorporated into the names of generated output. If you do not specify an alias, the part name (or a truncated version) is used instead.

- The **shared** property indicates whether multiple users can access the data table. The default is *no*.
- The **resident** property indicates whether the data table remains in memory even when no program is using the data table. (The program goes into memory when first accessed.) The default is *no*. You can specify *yes* only if the shared specification is also *yes*.

Related concepts

"References to parts" on page 16

Related reference

"Use declaration" on page 631

DataTable part in EGL source format

You declare a `dataTable` part in an EGL file, which is described in *EGL projects, packages, and files*. This part is a primary part, which means that it must be at the top level of the file and must have the same name as the file.

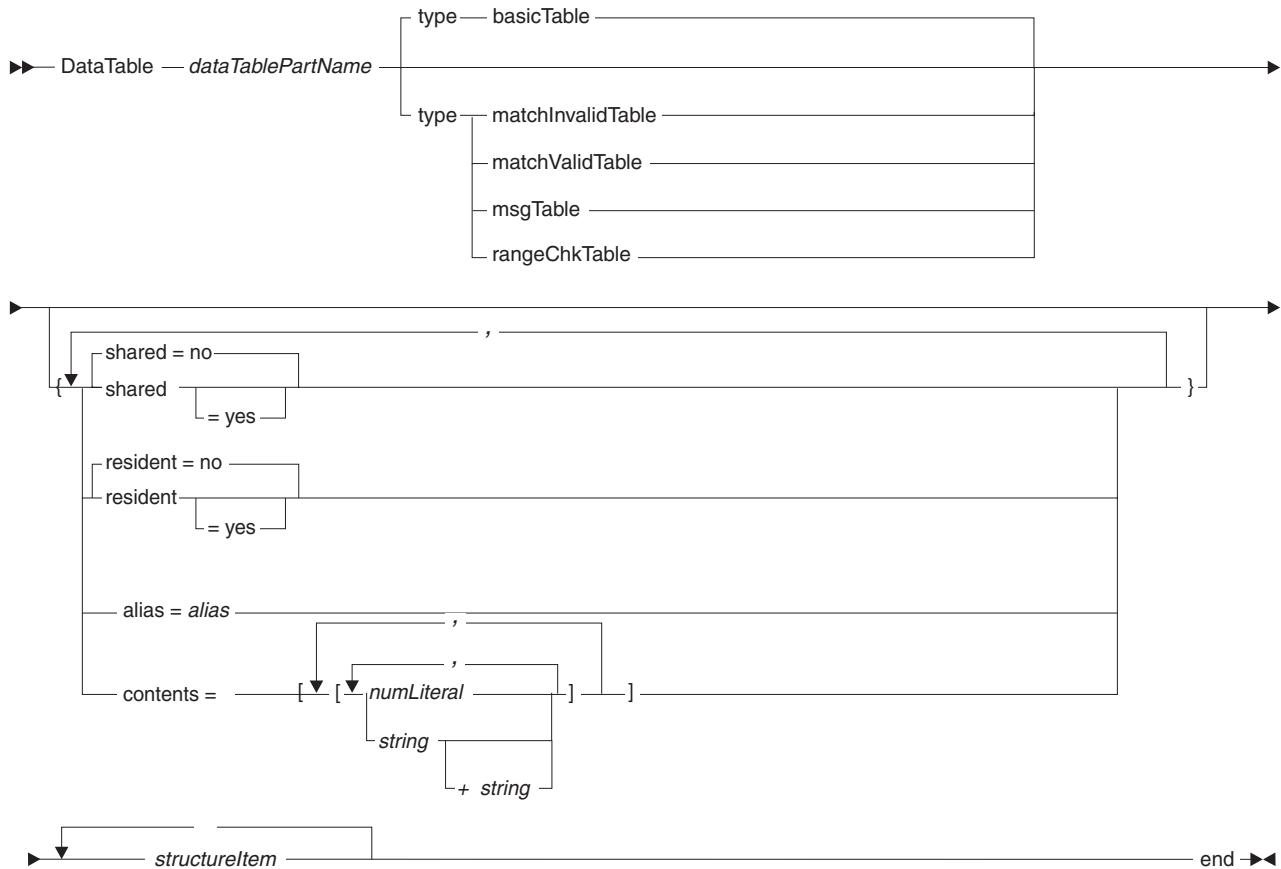
A data table is related to a program by the program's use declaration or (in the case of the program's only message table) by the program's **msgTablePrefix** property. A data table is related to a page handler by the page handler's use declaration.

An example of a `dataTable` part is as follows:

```
DataTable myDataTablePart type basicTable
{
    { shared = yes }
    myColumn1 char(10);
    myColumn2 char(10);
    myColumn3 char(10);

    { contents = [
        [ "row1 col1", "row1 col2", "row1 " + "col3" ] ,
        [ "row2 col1", "row2 col2", "row2 " + "col3" ] ,
        [ "row3 col1", "row3 col2", "row3 col3"      ]
    ]
}
end
```

The syntax diagram for a `dataTable` part is as follows:



DataTable *dataTablePartName* ... end

Identifies the part as a data table and specifies the part name. For the rules of naming, see *Naming conventions*.

basicTable (the default)

Contains information that is used in the program logic; for example, a list of countries and related codes.

matchInvalidTable

Is specified in the **validatorTable** property of a text field to indicate that the user's input must be different from any value in the first column of the data table. The EGL run time acts as follows in response to a validation failure:

- Accesses the table referenced in the **validatorTable** property
- Retrieves the message identified by the item-specific **validatorTableMsgKey** property
- Displays the message in the field identified in the form-specific **msgField** property

matchValidTable

Is specified in the **validatorTable** property of a text field to indicate that the user's input must match a value in the first column of the data table. The EGL run time acts as follows in response to a validation failure:

- Accesses the table referenced in the **validatorTable** property
- Retrieves the message identified by the item-specific **validatorTableMsgKey** property
- Displays the message in the field identified in the form-specific **msgField** property

msgTable

Contains run-time messages. A message is presented in the following circumstance:

- The table is the message table for the program. The association of table to program occurs if the program property **msgTablePrefix** references the *table prefix*, which is the first one to four characters in the name of the data table. The rest of the name is one of the national language codes in the next table.

Language	National language code
Brazilian Portugese	PTB
Chinese, simplified	CHS
Chinese, traditional	CHT
English, uppercase	ENP
English, USA	ENU
French	FRA
German	DEU
Italian	ITA
Japanese, Katakana (single-byte character set)	JPN
Korean	KOR
Spanish	ESP
Swiss German	DES

- The program retrieves and presents a message by one of two mechanisms, as described in *sysLib.displayMsgNum* and *sysLib.validationFailed*.

rangeChkTable

Is specified in the **validatorTable** property of a text field to indicate that the user's input must match a value that is between the values in the first and second column of at least one data-table row. (The range is inclusive; the user's input is valid if it matches a value in the first or second column of any row.) The EGL run time acts as follows in response to a validation failure:

- Accesses the table referenced in the **validatorTable** property
- Retrieves the message identified by the item-specific **validatorTableMsgKey** property
- Displays the message in the field identified in the form-specific **msgField** property

alias

A string that is incorporated into the names of generated output. If you do not specify an alias, the dataTable-part name (or a truncated version) is used instead.

shared

Indicates whether the same instance of a data table is used by multiple programs. Valid values are *yes* and *no* (the default).

The property indicates whether the same instance of a data table is used by every program in the same run unit. If the value of **shared** is *no*, each program in the run unit has a unique copy of the data table.

Changes made at run time are visible to every program that has access to the data table, and the changes remain until the data table is unloaded. In most

cases, the value of the **resident** property (described later) determines when the data table is unloaded; for details, see the description of that property.

resident

Indicates whether the data table is kept in memory even after every program that accessed the data table has ended.

Valid values are *yes* and *no*. The default is *no*.

If you set the **resident** property to *yes*, the data table is shared regardless of the value of **shared**.

The benefits of making a data table resident are as follows:

- The data table retains any values written to it by programs that ran previously
- The table is available for immediate access without additional load processing

A resident data table remains loaded until the run unit ends. A non-resident data table, however, is unloaded when the program that uses it ends.

Note: A data table is loaded into memory (if necessary) at a program's first access, and not when the EGL run time processes a use declaration.

contents

The value of the data table cells, each of which is one of the following kinds:

- A numeric literal
- A string literal or a concatenation of string literals

The kind of content in a given row must be compatible with the top-level structure items, each of which represents a column definition.

structureItem

A structure item, as described in *Structure item in EGL source format*.

Related concepts

"DataTable part" on page 285

"EGL projects, packages, and files" on page 7

"Run unit" on page 504

Related reference

"Naming conventions" on page 468

"Structure item in EGL source format" on page 505

"sysLib.displayMsgNum" on page 527

"sysLib.validationFailed" on page 529

"Use declaration" on page 631

EGL reserved words

EGL includes two categories of reserved words:

- Words that are reserved for specific uses except when you are working on an SQL statement
- Words that are reserved for specific uses when you are working on an SQL statement

Words that are reserved outside of an SQL statement

Outside of SQL statements, the reserved words are as follows in any combination of upper- and lower-case letters:

- absolute, add, after, all, array, at
- before, bigInt, bin, blob, by, byName, byPosition, byte
- call, case, char, clob, close, command, continue, converse, current, currentRow
- database, DataItem, DataTable, date, day, dbChar, dbClob, decimal, decimalFloat, decrement, defined, delete, description, display, dliCall
- else, embed, enable, end, eof, every, execute, exit, externallyDefined
- field, first, float, for, forEach, Form, format, FormGroup, forUpdate, forward, forwarding, freeSql, from, Function
- get, goto, group
- header, help, hex, hide, hold, hour
- if, import, in, inOut, input, insert, int, integerDate, interval, into, is
- keys
- label, languageBundle, last, library, line
- mbChar, menu, minute, money, month, move
- newLine, newWindow, next, noDelete, noInsert, noRefresh, not, nullable, num, number, numc
- of, off, on, onException, onKey, open, option, otherwise, out
- pacf, package, PageHandler, passing, physical, prepare, previous, print, private, Program, prompt, Psb
- Record, relative, replace, report, return, returns, row
- scroll, second, setOption, smallFloat, set, show, singleRow, smallInt, stack
- this, thru, time, timeStamp, to, trailer, transaction, transfer, try, type
- unicode, update, use, using, usingKeys
- varChar, varDbChar, varMbChar, varUnicode
- when, where, while, window, with, withinParent, wrap
- year

Words that are reserved in an SQL statement

In SQL statements, the reserved words are as follows in any combination of upper- and lower-case letters:

- absolute, action, add, alias, all, allocate, alter, and, any, are, as, asc, assertion, at, authorization, avg
- begin, between, bigint, binaryLargeObject, bit, bit_length, blob, boolean, both, by
- call, cascade, cascaded, case, cast, catalog, char, char_length, character, character_length, characterLargeObject, characterVarying, charLargeObject, charVarying, check, clob, close, coalesce, collate, collation, column, comment, commit, connect, connection, constraint, constraints, continue, convert, copy, corresponding, count, create, cross, current, current_date, current_time, current_timestamp, current_user, cursor
- data, database, date, dateTime, day, deallocate, dec, decimal, declare, default, deferrable, deferred, delete, desc, describe, diagnostics, disconnect, distinct, domain, double, doublePrecision, drop
- else, end, endExec, escape, except, exception, exec, execute, exists, explain, external, extract
- false, fetch, first, float, for, foreign, found, from, full
- get, getCurrentConnection, global, go, goto, grant, group
- having, hour
- identity, image, immediate, in, index, indicator, initially, inner, input, insensitive, insert, int, integer, intersect, into, is, isolation
- join
- key
- language, last, leading, left, level, like, local, long, longint, lower, ltrim
- match, max, min, minute, module, month

- national, nationalCharacter, nationalCharacterLargeObject, nationalCharacterVarying, nationalCharLargeObject, nationalCharVarying, natural, nchar, ncharVarying, nclob, next, no, not, null, nullIf, number, numeric
- octet_length, of, on, only, open, option, or, order, outer, output, overlaps
- pad, partial, position, prepare, preserve, primary, prior, privileges, procedure, public
- raw, read, real, references, relative, restrict, revoke, right, rollback, rows, rtrim, runtimeStatistics
- schema, scroll, second, section, select, session, session_user, set, signal, size, smallint, some, space, sql, sqlcode, sqlerror, sqlstate, substr, substring, sum, system_user
- table, tablespace, temporary, terminate, then, time, timestamp, timezone_hour, timezone_minute, tinyint, to, trailing, transaction, translate, translation, trim, true
- uncatalog, union, unique, unknown, update, upper, usage, user, using
- values, varbinary, varchar, varchar2, varying, view
- when, whenever, where, with, work, write
- year
- zone

Related reference

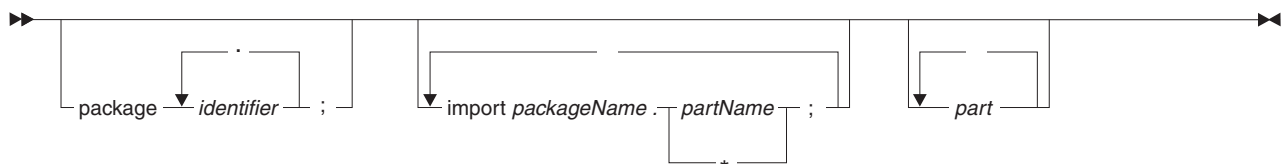
“EGL statements” on page 70

“Naming conventions” on page 468

“reservedWord” on page 259

EGL source format

You declare logic, data, and user-interface parts in EGL files, each of which has the extension *.egl* and is constructed as follows:



package *identifier*

Specifies the name of the package in which the file resides, with each identifier separated from the next by a period.

For an overview, see *EGL projects, packages, and files*.

import *packageName*

Specifies the full name of a package to import. For an overview, see *Import*.

partName

Specifies a single part to import.

- * Indicates that every part in the package is to be imported.

part

One of the EGL logic, data, or user-interface parts.

You may place comments in an EGL file, inside or outside a part.

Related concepts

“EGL projects, packages, and files” on page 7

"Import" on page 26
 "References to parts" on page 16
 "Parts" on page 11

Related tasks

"Syntax diagram" on page 506

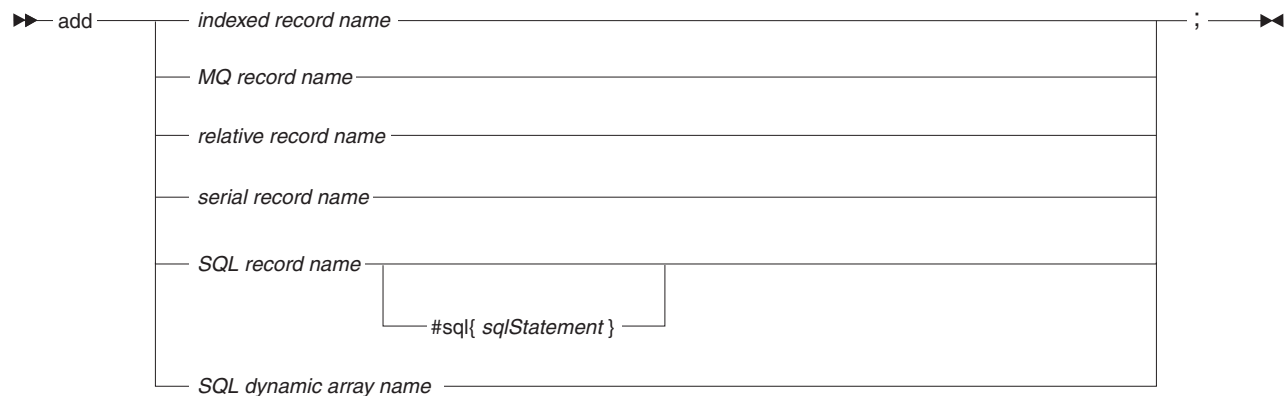
Related reference

"Basic record part in EGL source format" on page 491
 "Comments" on page 305
 "DataItem part in EGL source format" on page 284
 "DataTable part in EGL source format" on page 287
 "FormGroup part in EGL source format" on page 377
 "Form part in EGL source format" on page 370
 "Function part in EGL source format" on page 381
 "Indexed record part in EGL source format" on page 492
 "Library part in EGL source format" on page 434
 "MQ record part in EGL source format" on page 493
 "PageHandler part in EGL source format" on page 472
 "Program part in EGL source format" on page 478
 "Relative record part in EGL source format" on page 495
 "Serial record part in EGL source format" on page 497
 "SQL record part in EGL source format" on page 498

EGL statements

add

The EGL **add** statement places a record in a file, message queue, or database; or places a set of records in a database.



record name

Name of the I/O object to add: an indexed, MQ, relative, serial, or SQL record

with #sql{ sqlStatement }

An explicit SQL INSERT statement. Leave no space after #sql.

SQL dynamic-array name

The name of a dynamic array of SQL records. The elements are inserted into the database, each at the position specified by the element-specific key values. The operation stops at the first error or when all elements are inserted.

An example is as follows:

```

if (userRequest = "A")
  try
    add record1;
  onException
    myErrorHandler(12);
  end
end

```

The behavior of the **add** statement depends on the record type. For details on SQL processing, see *SQL record*.

Indexed record

When you add an indexed record, the key in the record determines the logical position of the record in the file. Adding a record to a file position that is already in use results in the hard I/O error UNIQUE or (if duplicates are allowed) in the soft I/O error DUPLICATE.

MQ record

When you add a MQ record, the record is placed at the end of the queue. This placement occurs because the add invokes one or more MQSeries calls:

- MQCONN connects the generated code to the default queue manager and is invoked when no connection is active
- MQOPEN establishes a connection to the queue and is invoked when a connection is active but the queue is not open
- MQPUT puts the record in the queue and is always invoked unless an error occurred in an earlier MQSeries call

Relative record

When you add a relative record, the key item specifies the position of the record in the file. Adding a record to a file position that is already in use, however, results in the hard I/O error UNIQUE.

The record key item must be available to any function that uses the record and can be any of these:

- An item in the same record
- An item in a record that is global to the program or is local to the function that is running the **add** statement
- A data item that is global to the program or is local to the function that is running the **add** statement

Serial record

When you add a serial record, the record is placed at the end of the file.

If the generated program adds a serial record and then issues a **get next** statement for the same file, the program closes and reopens the file before executing the **get next** statement. A **get next** statement that follows an **add** statement therefore reads the first record in the file. This behavior also occurs when the **get next** and **add** statements are in different programs, and one program calls another.

It is recommended that you avoid having the same file open in more than one program at the same time.

SQL record

Some error conditions are as follows:

- You specify an SQL statement of a type other than INSERT

- You specify some but not all clauses of an SQL INSERT statement
- You specify an SQL INSERT statement (or accept an implicit SQL statement) that has any of these characteristics--
 - Is related to more than one SQL table
 - Includes only host variables that you declared as read only
 - Is associated with a column that either does not exist or is incompatible with the related host variable

The result is as follows when you add an SQL record without specifying an explicit SQL statement:

- The format of the generated SQL INSERT statement is like this--

```
INSERT INTO tableName
(column01, ... columnNN)
values (:recordItem01, ... :recordItemNN)
```

- The key value in the record determines the logical position of the data in the table. A record that does not have a key is handled in accordance with the SQL table definition and the rules of the database.
- As a result of the association of record items and SQL table columns in the record part, the generated code places the data from each record item into the related SQL table column.
- If you declared a record item to be read only, the generated SQL INSERT statement does not include that record item, and the database management system sets the value of the related SQL table column to the default value that was specified when the column was defined.

An example that uses a dynamic array of SQL records is as follows:

```
try
  add employees;
onException
  sysLib.rollback();
end
```

Related concepts

“References to parts” on page 16
 “Record types and properties” on page 13
 “SQL support” on page 171

Related tasks

“Syntax diagram” on page 506

Related reference

“close” on page 303
 “delete” on page 307
 “get” on page 318
 “get next” on page 324
 “get previous” on page 328
 “Exception handling” on page 75
 “execute” on page 309
 “I/O error values” on page 418
 “open” on page 335
 “prepare” on page 339
 “EGL statements” on page 70
 “replace” on page 341
 “SQL item properties” on page 57

Assignments

An EGL assignment can have any of these effects:

- Copies data from one area of memory to another
- Places any of these values into an area of memory:
 - The result of an arithmetic calculation
 - A value returned from a function invocation
 - A literal

►► `target = source ;` ◀◀

target An item, record, or system variable

source An expression or record. An expression may be composed of a complex series of symbols (for example, $a + b + c$) or may be one of these:

- An item or system variable
- A function invocation
- A literal

By default, a record is equivalent to an item of type CHAR, with the total number of bytes equal to the sum of bytes in the structure items. When the record is of type CHAR, the following exceptions apply:

- The content of a record can be assigned to a record or to an item of type CHAR, HEX, or MBCHAR, but not to an item of any other type
- A record can receive data from a record, from a string literal, or from an item of type CHAR, HEX, or MBCHAR, but not from a numeric literal or from an item of a type other than CHAR, HEX, or MBCHAR

Examples of assignments are as follows:

```
z = a + b + c;  
myDate = sysVar.currentShortDate;  
myUser = sysVar.userID;  
myRecord01 = myRecord02;  
myRecord02 = "USER";
```

The next table and the explanations that follow tell the compatibility rules.

Target primitive type	Source primitive (or loose) types that are valid for target
BIN	BIN, INT, BIGINT, SMALLINT, DECIMAL, NUM, NUMBER, NUMC, PACF
CHAR	CHAR, HEX, MBCHAR, NUM, NUMBER
DBCHAR	DBCHAR
HEX	CHAR, HEX
MBCHAR	CHAR, MBCHAR
NUM	BIN, INT, BIGINT, SMALLINT, CHAR, NUM, NUMBER, NUMC, PACF, DECIMAL
NUMC	BIN, INT, BIGINT, SMALLINT, DECIMAL, NUM, NUMBER, NUMC, PACF
PACF	BIN, INT, BIGINT, SMALLINT, DECIMAL, NUM, NUMBER, NUMC, PACF

Target primitive type	Source primitive (or loose) types that are valid for target
DECIMAL	BIN, INT, BIGINT, SMALLINT, DECIMAL, NUM, NUMBER, NUMC, PACF
UNICODE	UNICODE
A special case is as follows: the numeric value returned from a mathematical system word can be assigned to an item of type HEX; for details, see <i>Mathematical (system words)</i> .	

Assignment across numeric types

A value of any of the numeric types (BIN, DECIMAL, NUM, NUMBER, NUMC, PACF) can be assigned to an item of any numeric type and size, and EGL does the conversions necessary to retain the value in the target format. If necessary, non-significant zeros are added or truncated. (Initial zeros in the integer part of a value are non-significant, as are trailing zeros in the fraction part of a value.)

You can use the system variable `sysVar.handleOverflow` to test whether an assignment or arithmetic calculation resulted in an arithmetic overflow, and you can set the system variable `sysVar.overflowIndicator` to specify the consequence of such an overflow.

If an arithmetic overflow occurs, the value in the target item is unchanged. If an arithmetic overflow does not occur, the value assigned to the target item is aligned in accordance with the declaration of the target item.

Let's assume that you are copying an item of type NUM to another and that the run-time value of the source item is 108.314:

- If the target item allows seven digits with one decimal place, the target item receives the value 000108.3, and a numeric overflow is *not* detected. (A loss of precision in a fractional value is not considered an overflow.)
- If the target item allows four digits with two decimal places, a numeric overflow is detected, and the value in the target item is unchanged

Other cross-type assignments

Details on other cross-type assignments are as follows:

- The assignment of a value of type NUM to a target of type CHAR is valid only if the source declaration has no decimal places. This operation is equivalent to a CHAR-to-CHAR assignment.
If the source length is 4 and value is 21, for example, the content is equivalent to "0021", and a length mismatch does not cause an error condition:
 - If the length of the target is 5, the value is stored as "0021 " (a single-byte space was added on the right)
 - If the length of the target is 3, the value is stored as "002" (a digit was truncated on the right)
 If the value of type NUM is negative and assigned to a value of type CHAR, the last byte copied into the item is an unprintable character.
- The assignment of a value of type CHAR to a target of type NUM is valid only in the following case:
 - The source (an item or string expression) has digits with no other characters
 - The target declaration has no decimal place
 This operation is equivalent to a NUM-to-NUM assignment.

If the source length is 4 and value is "0021", for example, the content is equivalent to a numeric 21, and the effect of a length mismatch is shown in these examples:

- If the length of the target is 5, the value is stored as 00021 (a numeric zero was padded on the left)
- If the length of the target is 3, the value is stored as 021 (a non-significant digit was truncated)
- If the length of the target is 1, the value is stored as 1
- The assignment of a value of type NUMC to a target of type CHAR is possible in two steps, which eliminates the sign if the value is positive:
 1. Assign the NUMC value to a target of type NUM
 2. Assign the NUM value to a target of type CHARIf the value of the target of type NUMC is negative, the last byte copied into the target of type CHAR is an unprintable character.
- The assignment of a value of type CHAR to a target of type HEX is valid only if the characters in the source are within the range of hexadecimal digits (0-9, A-F, a-f).
- The assignment of a value of type HEX to a target of type CHAR stores digits and uppercase letters (A-F) in the target.

Padding and truncation with character types

If the target is of a character type (CHAR, DBCHAR, HEX, MBCHAR, UNICODE) and has more space than is required to store a source value, EGL pads data on the right:

- Uses single-byte blanks to pad a target of type CHAR or MBCHAR
- Uses double-byte blanks to pad a target of type DBCHAR
- Uses Unicode double-byte blanks to pad a target of type UNICODE
- Uses binary zeros to pad a target of type HEX, which means (for example) that a source value "0A" is stored in a two-byte target as "0A00" rather than as "000A"

EGL truncates values on the right if the target of a character type has insufficient space to store the source value. No error is signaled. A special case can occur in the following situation:

- The run-time platform supports the EBCDIC character set
- The assignment statement copies a literal of type MBCHAR or an item of type MBCHAR to a shorter item of type MBCHAR
- A byte-by-byte truncation would remove a final shift-in character or split a DBCHAR character

In this situation, EGL truncates characters as needed to ensure that the target item contains a valid string of type MBCHAR, then adds (if necessary) terminating single-byte blanks.

Assignment to or from substructured items

You can assign a substructured item to a non-substructured item or the reverse, and you can assign values between two substructured items. Assume, for example, that variables named *myNum* and *myRecord* are based on the following parts:

```
DataItem myNumPart
  NUM(12)
end

Record myRecordPart type basicRecord
```

```

10 topMost CHAR(4);
20 next01 HEX(4);
20 next02 HEX(4);
end

```

The assignment of a value of type HEX to an item of type NUM is not valid outside of the mathematical system words; but an assignment of the form **myNum = topMost** is valid because **topMost** is of type CHAR. In general terms, the primitive types of the items in the assignment statement guide the assignment, and the primitive types of subordinate items are not taken into account.

The primitive type of a substructured item is CHAR by default. If you assign data to or from a substructured item and do not specify a different primitive type at declaration time, the rules described earlier for items of type CHAR are in effect during the assignment.

Assignment of a record

An assignment of one record to another is equivalent to assigning one substructured item of type CHAR to another. A mismatch in length adds single-byte blanks to the right of the received value or removes single-byte characters from the right of the received value. The assignment does not consider the primitive types of subordinate structure items.

As mentioned earlier, the following exceptions apply:

- The content of a record can be assigned to a record or to an item of type CHAR, HEX, or MBCHAR, but not to an item of any other type
- A record can receive data from a record or from a string literal or from an item of type CHAR, HEX, or MBCHAR, but not from a numeric literal or from an item of a type other than CHAR, HEX, or MBCHAR

Finally, if you assign an SQL record to or from a record of a different type, you must ensure that the non-SQL record has space for the four-byte area that precedes each structure item.

Related concepts

"Syntax diagram" on page 506

Related reference

"sysVar.handleOverflow" on page 532

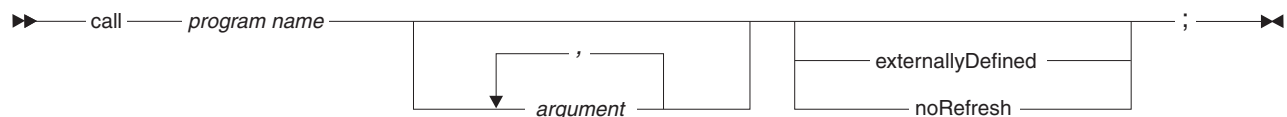
"sysVar.overflowIndicator" on page 534

"Primitive types" on page 27

"EGL statements" on page 70

call

The EGL call statement transfers control to another program and optionally passes a series of values. Control returns to the caller when the called program ends; and if the called program changes any data that was passed by way of a variable, the storage area available to the caller is changed, too.



program name

Name of the called program. The program is either generated by EGL or is considered *externally defined*.

The specified name cannot be a reserved word. If the caller must call a non-EGL program that has the same name as an EGL reserved word, use a different program name in the call statement, then use a linkage options part, **callLink** element to specify an alias, which is the name used at run time.

If the called program is a Java program, the called program name is case-sensitive; *calledProgram* is different from *CALLEDPROGRAM*. Otherwise, the determination of whether the program name is case-sensitive depends on the system on which the called program resides: case-sensitive for UNIX, case-insensitive otherwise.

In the EGL debugger, the called program name is case-insensitive.

argument

One of a series of value references, each separated from the next by a comma. An argument may be an item, a form, a record, a non-numeric literal, or a non-numeric constant.

externallyDefined

An indicator that the program is externally defined. This indicator is available only if you set the project property for VisualAge Generator compatibility.

It is recommended that a non-EGL-generated program be identified as externally defined not in the **call** statement, but in the linkage options part that is used at generation time. (The related property is in the linkage options part, **callLink** element, and is also called **externallyDefined**.)

noRefresh

An indicator that a screen refresh is to be avoided when the called program returns control.

The indicator is supported (at development time) if the program property **VAGCompatibility** is selected or (at generation time) if the build descriptor option **VAGCompatibility** is set to *yes*.

This indicator is appropriate if the caller is in a run unit that presents text forms to a screen and either of these situations is in effect:

- The called program does not present a text form; or
- The caller writes a full-screen text form after the call.

It is recommended that you indicate your preference for screen refresh not in the **call** statement, but in the linkage options part that is used at generation time. (The related property is in the linkage options part, **callLink** element, and is called **refreshScreen**.)

An example is as follows:

```
if (userRequest = "C")
  try
    call programA;
  onException
    myErrorHandler(12);
end
end
```

The number, type, and sequence of arguments in a call statement must correspond to the number, type, and sequence of values expected by the called program.

It is strongly recommended that the number of bytes passed in each argument be the same as the number of bytes expected. In the case of an EGL-generated Java program, a length mismatch causes an error only if the run-time correction of that mismatch causes a type mismatch:

- If the called Java program receives too few bytes, the end of the passed data is padded with blanks.
- If the called Java program receives too many bytes, the end of the passed data is truncated.

In the case of Java, an error occurs if blanks are added to a data item of type NUM, for example, but not if blanks are added to a data item of type CHAR.

The following rules apply to literals and constants:

- The size of a passed literal or constant must equal the size of the receiving parameter
- A numeric literal or constant cannot be passed as an argument
- A literal or constant that includes only single-byte characters may be passed to a parameter of type CHAR or MBCHAR
- A literal or constant that includes only double-byte characters may be passed only to a parameter of type DBCHAR
- A literal or constant that includes a combination of single- and double-byte characters may be passed to a parameter of type MBCHAR

The behavior of call depends partly on the target system, as shown in the next table.

Target system	Platform-specific details
AIX	Recursive calls are supported.
iSeries COBOL	Recursive calls are not supported.
iSeries Java	Recursive calls are supported.
Linux	Recursive calls are supported.
z/OS batch	Recursive calls are not supported.
Windows 2000, Windows NT	Recursive calls are supported.
z/OS UNIX System Services	Recursive calls are supported.

The call is affected by the linkage options part, if any, that is used at generation time. (You include a linkage options part by setting the build descriptor option **linkage**.)

For details on linkage, see *Linkage options part*.

Related concepts

“Linkage options part” on page 439

“Syntax diagram” on page 506

Related reference

"EGL statements" on page 70 "Exception handling" on page 75

"linkage" on page 256

"Primitive types" on page 27

case

The EGL **case** statement marks the start of multiple sets of statements, where at most only one of those sets is run. The **case** statement is equivalent to a C or Java **switch** statement that has a break at the end of each case clause.



criterion

An item, constant, literal, or system variable, including `sysVar.eventKey` or `sysVar.systemType`.

If you specify *criterion*, each of the subsequent when clauses (if any) must contain one or more instances of *matchExpression*. If you do not specify *criterion*, each of the subsequent when clauses (if any) must contain a *logical expression*.

when

The beginning of a clause that is invoked only in these cases:

- You specified a *criterion*, and the when clause is the first to contain a *matchExpression* that is equal to the *criterion*; or
- You did not specify a *criterion*, and the when clause is the first to contain a *logical expression* that evaluates to true.

If you wish the when clause to have no effect when invoked, code the clause without EGL statements.

A **case** statement may have any number of when clauses.

matchExpression

One of the following values:

- A numeric or string expression
- A symbol for comparison to `sysVar.eventKey` or `sysVar.systemType`

The primitive type of *matchExpression* value must be compatible with the primitive type of the criterion value. For details on compatibility, see *Logical expressions*.

logicalExpression

A logical expression.

statement

An EGL statement.

otherwise

The beginning of a clause that is invoked if no when clause runs.

After the statements run in a when or otherwise clause, control passes to the EGL statement that immediately follows the end of the **case** statement. Control does not fall through to the next when clause under any circumstance. If no when clause is invoked and no default clause is in use, control also passes to the next statement immediately following the end of the **case** statement, and no error situation is in effect.

An example of a **case** statement is as follows:

```
case (myRecord.requestID)
  when (1)
    myFirstFunction();
  when (2, 3, 4)
    try
      call myProgram;
    onException
      myCallFunction(12);
    end
  otherwise
    myDefaultFunction();
end
```

If a single clause includes multiple instances of *matchExpression* (2, 3, 4 in the previous example), evaluation of those instances is from left to right, and the evaluation stops as soon as one *matchExpression* is found that corresponds to the criterion value.

Related tasks

“Syntax diagram” on page 506

Related reference

“EGL statements” on page 70

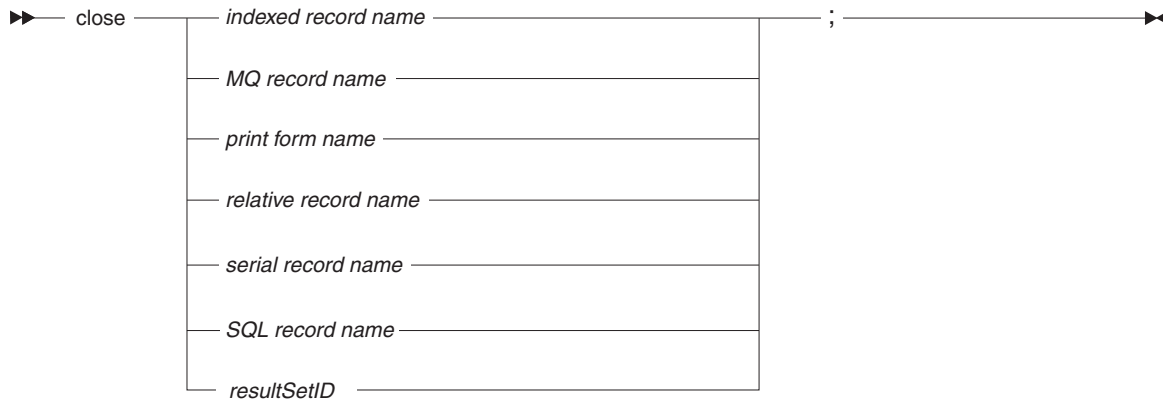
“Logical expressions” on page 358

“`sysVar.eventKey`” on page 622

“`sysVar.systemType`” on page 628

close

The EGL **close** statement disconnects a printer; or closes the file or message queue associated with a given record; or, in the case of an SQL record, closes the cursor that was open by an EGL **open** or **get** statement.



name

Name of the I/O object that is associated with the resource being closed; that object is a print form or an indexed, MQ, relative, serial, or SQL record

resultSetIdentifier

For SQL processing only, an ID that ties the **close** statement to a **get** or **open** statement run earlier in the same program. For details, see *resultSetID*.

Example:

```

if (userRequest = "C")
  try
    close fileA;
  onException
    myErrorHandler(12);
end
end

```

The behavior of a **close** statement depends on the type of I/O object.

Indexed, serial, or relative record

When you use the name of an indexed, serial, or relative record in a **close** statement, EGL closes the file associated with that record.

If a file is open and you use the fileAssociation item to change the resource name associated with that file, EGL closes the file automatically before executing the next statement that affects the file. For details, see *resourceAssociation*.

EGL also closes any file that is open when the program ends.

MQ record

When you use the name of a MQ record in a **close** statement, EGL ensures that the MQSeries command MQCLOSE is executed for the message queue associated with that record.

Print form

If the I/O object is a print form, the close statement issues a form feed and either disconnects from the printer or (if the print form is spooled to a file) closes the file.

Before you use sysVar.printerAssociation to change the print destination, close the printer or file specified by the current value of sysVar.printerAssociation. Issue a close statement option for each print destination, as multiple printer or print files can be open at the same time.

EGL run time ensures that all printers are closed when the program ends.

SQL record

When you use the name of an SQL record in a **close** statement, EGL closes the SQL cursor that is open for that record.

EGL automatically closes a cursor in these cases:

- A cursor-processing loop follows an **open** statement and continues until a No Record Found (NRF) condition indicates that all rows in the set were processed
- EGL runs a **get** statement for an SQL record when a single row is read and neither **forUpdate** nor **singleRow** was specified as an option
- EGL runs a **replace** or **delete** statement that uses the cursor opened by a **get** statement; in this case, **forUpdate** was specified as an option in the **get** statement
- EGL begins to process an **open** or **get** statement for a record that is associated with an open cursor; the close precedes the other processing
- The program runs either **sysLib.commit** or **sysLib.rollback**; but the close does not occur if the option **withHold** is in effect, as explained in relation to **open**

EGL closes all open cursors in this case:

- The program is of type **textUI**, does an automatic commit before conversing a form, and is unaffected by the option **withHold** when the converse occurs; for details on **textUI** programs and the **converse** statement, see *Segmentation*

Related concepts

"Record types and properties" on page 13

"resultSetID" on page 181

"Segmentation in text applications" on page 151

"SQL support" on page 171

Related tasks

"Syntax diagram" on page 506

Related reference

"add" on page 293

"delete" on page 307

"EGL statements" on page 70

"Exception handling" on page 75

"execute" on page 309

"get" on page 318

"get next" on page 324

"get previous" on page 328

"I/O error values" on page 418

"open" on page 335

"prepare" on page 339

"replace" on page 341

"recordName.resourceAssociation" on page 539

"SQL item properties" on page 57

"sysLib.commit" on page 541

"sysLib.rollback" on page 550

"sysVar.printerAssociation" on page 623

"sysVar.terminalID" on page 629

Comments

A *comment* in an EGL file is created in either of the following ways:

- Double right slashes (//) indicate that the subsequent characters are a comment, up to and including the end-of-line character
- A single or multiline comment is delimited by a right slash and asterisk at the start (/*) and by an asterisk and right slash at the end (*/); this form of comment is valid anywhere that a white-space character is valid

You may place a comment inside or outside of an executable statement, as in this example:

```
/* the assignment e = f occurs if a = b or if c = d */
if (a = b           // one comparison
    || /* OR; another comparison */ c = d)
    e = f;
end
```

EGL does not support embedded comments, so the following entries cause an error:

```
/* this line starts a comment /* and
   this line ends the comment, */
   but this line is not inside a comment at all */
```

The comment in the first two lines includes a second comment delimiter (/*). An error results only when EGL tries to interpret the third line as source code.

The following is valid:

```
a = b;  /* this line starts a comment // and
         this line ends the comment */
```

The double right slashes (//) in the last example are themselves part of a larger comment.

Between the symbols #sql{ and }, the EGL comments described earlier are not valid. The following statements apply:

- An SQL comment begins with a double hyphen (--) at the beginning of a line or after white space and continues until the end of the line
- Comments are not available inside a string literal. A series of characters in that literal is interpreted as text even in these contexts:
 - A prepare statement
 - The **defaultSelectCondition** property of a record of type SQLRecord

Related concepts

“EGL projects, packages, and files” on page 7

Related reference

“EGL source format” on page 292

“EGL statements” on page 70

converse

The EGL **converse** statement presents a text form in a text application.

The program waits for a user response, receives the text form from the user, and continues processing with the statement that follows the **converse** statement.

For an overview of text-form processing, see these pages in order:

1. Text forms

2. Segmentation

►► — converse — *textFormName* ————— ; —◄◄

textFormName

Name of a text form that is visible to the program. For details on visibility, see *References to parts*.

An example is as follows:

```
converse myTextForm;
```

These considerations apply:

- In relation to text forms, a **converse** statement is always valid in a called program; but if you are running a main program that is segmented, the **converse** statement is not valid in these kinds of code--
 - A function that has parameters, local storage, or return values
 - A function that is invoked (directly or indirectly) by a function that has parameters, local storage, or return values.

Related concepts

“References to parts” on page 16

“Segmentation in text applications” on page 151

delete

The EGL **delete** statement removes either a record from a file or a row from a database.

►► — delete ————— *indexed record name* ————— ; —◄◄

<i>relative record name</i>
<i>SQL record name</i>

from *resultSetID*

record name

Name of the I/O object: an indexed, relative, or SQL record associated with the file record or SQL row being deleted

from *resultSetID*

ID that ties the **delete** statement to a **get** or **open** statement run earlier in the same program. For details, see *resultSetID*.

An example is as follows:

```
if (userRequest = "D")
  try
    get myRecord forUpdate;
  onException
    myErrorHandler(12);  // exits the program
end
```

```

try
  delete myRecord;
onException
  myErrorHandler(16);
end
end

```

The behavior of the **delete** statement depends on the record type. For details on SQL processing, see *SQL record*.

Indexed or relative record

If you want to delete an indexed or relative record, do as follows:

- Issue a **get** statement for the record and specify the `forUpdate` option
- Issue the **delete** statement, with no intervening I/O operation against the same file

After you issue the **get** statement, the effect of the next I/O operation on the same file is as follows:

- If the next I/O operation is a **replace** statement on the same EGL record, the record is changed in the file
- If the next I/O operation is a **delete** statement on the same EGL record, the record in the file is marked for deletion
- If the next I/O operation is a **get** on the same file (with the `forUpdate` option), a subsequent **replace** or **delete** is valid on the newly read file record
- If the next I/O operation is a **get** on the same EGL record (with no `forUpdate` option) or is a **close** on the same file, the file record is released without change

For details on the `forUpdate` option, see *get*.

SQL record

In the case of SQL processing, you must use the `forUpdate` option on an EGL **get** or **open** statement to retrieve a row for subsequent deletion:

- You can issue a **get** statement to retrieve the row; or
- You can issue an **open** statement to select a set of rows and then invoke a **get next** statement to retrieve the row of interest.

In either case, the EGL **delete** statement is represented in the generated code by an SQL DELETE statement that references the current row in a cursor. You cannot modify that SQL statement, which is formatted as follows:

```

DELETE FROM tableName
WHERE CURRENT OF cursor

```

If you wish to write your own SQL DELETE statement, use the EGL **execute** statement.

You cannot use a single EGL **delete** statement to remove rows from multiple SQL tables.

Related concepts

“Record types and properties” on page 13

“resultSetID” on page 181

“Run unit” on page 504

“SQL support” on page 171

Related tasks

"Syntax diagram" on page 506

Related reference

"add" on page 293

"close" on page 303

"EGL statements" on page 70

"Exception handling" on page 75

"execute"

"get" on page 318

"get next" on page 324

"get previous" on page 328

"I/O error values" on page 418

"prepare" on page 339

"open" on page 335

"replace" on page 341

"SQL item properties" on page 57

display

The EGL **display** statement adds a text form to a run-time buffer but does not present data to the screen. For details on the run-time behavior, see *Text forms*.

Note: If you are working in VisualAge Generator compatibility mode, you can issue a statement of the following form:

```
display printForm;
```

printForm

Name of a print form that is visible to the program.

In that case, **display** is equivalent to **print**.

► — display — *textFormName* ————— ; ————— ◀◀

textFormName

Name of a text form that is visible to the program. For details on visibility, see *References to parts*.

Related concepts

"References to parts" on page 16

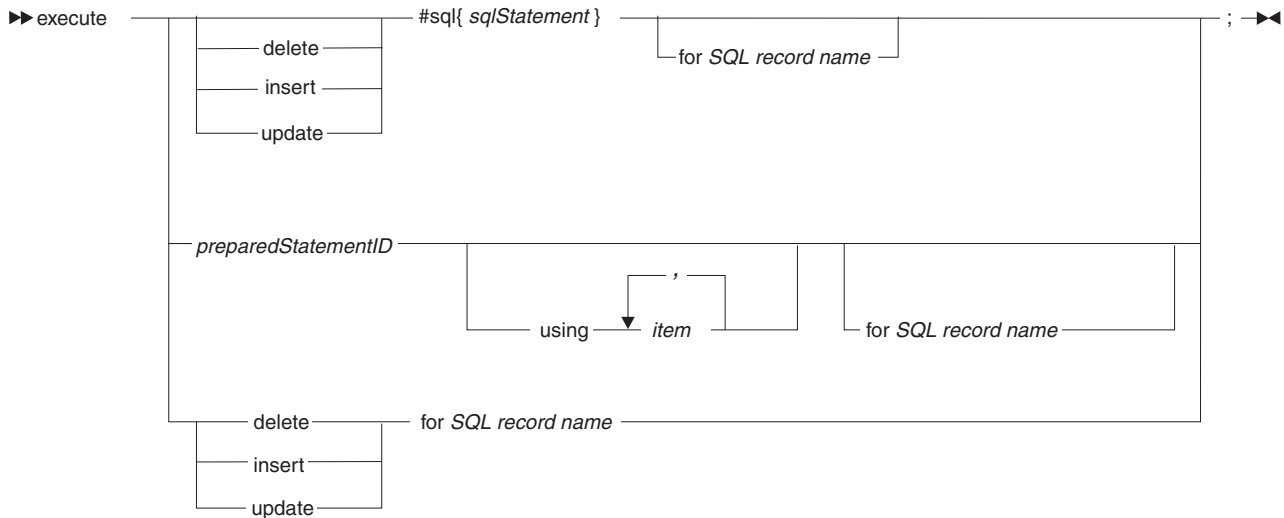
"Text forms" on page 367

Related reference

"print" on page 341

execute

The EGL **execute** statement lets you write one or more SQL statements; in particular, SQL data-definition statements (of type CREATE TABLE, for example) and data-manipulation statements (of type INSERT or UPDATE, for example)



#sql{ sqlStatement }

An explicit SQL statement. If you want the SQL statement to update or delete a row in a result set, code an SQL UPDATE or DELETE statement that includes the following clause:

WHERE CURRENT OF *resultSetID*

resultSetID

The resultSetID specified in the EGL open statement that made the result set available.

Leave no space between #sql and the left brace.

for SQL record name

Name of an SQL record.

If you specify a statement type (delete, insert, or update), EGL uses the SQL record to build an implicit SQL statement, as described later. In any case, you can use the SQL record to test the outcome of the operation.

preparedStatementID

Refers to an EGL prepare statement that has the specified ID. If you do not reference a prepare statement, you must specify either an explicit SQL statement or a combination of an SQL record and a statement type (delete, insert, or update).

delete, insert, update

Indicates that EGL is to provide an implicit SQL statement of the specified type. A declaration-time error occurs if you specify a statement type but not an SQL record name.

If you do not set a statement type, you must specify either an explicit SQL statement or a reference to a prepare statement.

For an overview of implicit SQL statements, see *SQL support*.

Several example statements are as follows (assuming that employeeRecord is an SQL record):

```

execute
  #sql{
    create table employee (
      empnum decimal(6,0) not null,
      empname char(40) not null,

```



```

        empphone char(10) not null)
    };

execute update for employeeRecord;

execute
#sql{
    call aStoredProcedure( :argumentItem)
};

```

You can use an **execute** statement to issue SQL statements of the following types:

- ALTER
- CALL
- CREATE ALIAS
- CREATE INDEX
- CREATE SYNONYM
- CREATE TABLE
- CREATE VIEW
- DECLARE global temporary table
- DELETE
- DROP INDEX
- DROP SYNONYM
- DROP TABLE
- DROP VIEW
- GRANT
- INSERT
- LOCK
- RENAME
- REVOKE
- SAVEPOINT
- SET
- SIGNAL
- UPDATE
- VALUES

You cannot use an **execute** statement to issue SQL statements of the following types:

- CLOSE
- COMMIT
- CONNECT
- CREATE FUNCTION
- CREATE PROCEDURE
- DECLARE CURSOR
- DESCRIBE
- DISCONNECT
- EXECUTE
- EXECUTE IMMEDIATE
- FETCH
- OPEN
- PREPARE
- ROLLBACK WORK
- SELECT
- INCLUDE SQLCA
- INCLUDE SQLDA
- WHENEVER

Implicit SQL DELETE

The effect of requesting an implicit SQL DELETE statement is that an SQL record property (**defaultSelectCondition**) determines what table rows are deleted, so long as the value in each SQL table key column is equal to the value in the corresponding key item of the SQL record. If you specify neither a record key nor a default selection condition, all table rows are deleted.

The implicit SQL DELETE statement for a particular record is similar to the following statement:

```
DELETE FROM tableName
WHERE keyColumn01 = :keyItem01
```

You cannot use a single EGL statement to delete rows from more than one database table.

Implicit SQL INSERT

The effect of requesting an implicit SQL INSERT statement is as follows by default:

- The key value in the record determines the logical position of the data in the table. A record that does not have a key is handled in accordance with the SQL table definition and the rules of the database.
- As a result of the association of record items and SQL table columns in the record part, the generated code places the data from each record item into the related SQL table column.
- If you declared a record item to be read only, the generated SQL INSERT statement does not include that record item, and the database management system sets the value of the related SQL table column to the default value that was specified when the column was defined.

The format of the implicit SQL INSERT statement is like this:

```
INSERT INTO tableName
(column01, ... columnNN)
values (:recordItem01, ... :recordItemNN)
```

Some error conditions are as follows:

- You specify an SQL statement of a type other than INSERT
- You specify some but not all clauses of an SQL INSERT statement
- You specify an SQL INSERT statement (or accept an implicit SQL statement) that has any of these characteristics--
 - Is related to more than one SQL table
 - Includes only host variables that you declared as read only
 - Is associated with a column that either does not exist or is incompatible with the related host variable

Implicit SQL UPDATE

The effect of requesting an implicit SQL UPDATE statement is as follows by default:

- An SQL record property (**defaultSelectCondition**) determines what table rows are selected, so long as the value in each SQL table key column is equal to the value in the corresponding key item of the SQL record. If you specify neither a record key nor a default selection condition, all table rows are updated.
- As a result of the association of record items and SQL table columns in the SQL record declaration, a given SQL table column receives the content of the related

record item. If an SQL table column is associated with a record item that is read only, however, that column is not updated.

The format of the implicit SQL UPDATE statement for a particular record is similar to the following statement:

```
UPDATE tableName
SET    column01 = :recordItem01,
       column02 = :recordItem01, ...
       columnNN = :recordItemNN
WHERE  keyColumn01 = :keyItem01
```

An error occurs in any of the following cases:

- All the items are identified as read only
- The statement attempts to update more than one SQL table
- An item whose value is being written to the database is associated with a column that either does not exist at run time or is incompatible with that item

Related concepts

"Record types and properties" on page 13

"SQL support" on page 171

"References to parts" on page 16

Related tasks

"Syntax diagram" on page 506

Related reference

"add" on page 293

"close" on page 303

"delete" on page 307

"EGL statements" on page 70

"Exception handling" on page 75

"get" on page 318

"get next" on page 324

"get previous" on page 328

"I/O error values" on page 418

"open" on page 335

"prepare" on page 339

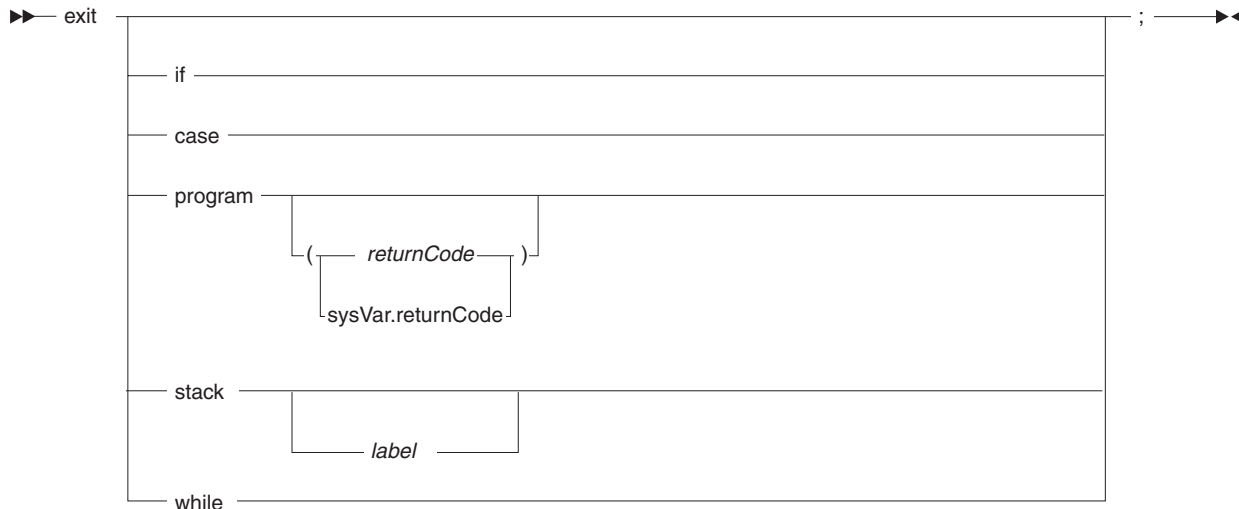
"replace" on page 341

"SQL item properties" on page 57

"sysVar.terminalID" on page 629

exit

The EGL **exit** statement leaves the specified block, which by default is the block that immediately contains the **exit** statement.



case

Leaves the most recently entered **case** statement in which the **exit** statement resides. Continues processing after the **case** statement.

An error occurs if the **exit** statement is not inside a **case** statement that begins in the same function.

if Leaves the most recently entered **if** statement in which the **exit** statement resides. Continues processing after the **if** statement.

An error occurs if the **exit** statement is not inside an **if** statement that begins in the same function.

program

Leaves the program.

The value in the system variable **sysVar.returnCode** is returned to the operating system in any of the following cases:

- The program ends with an **exit** statement that does not include a return code
- The program ends with an **exit** statement that returns **sysVar.returnCode**
- The program ends without a terminating **exit** statement

If the program ends with a terminating **exit** statement that includes a return code other than **sysVar.returnCode**, the specified value is used in place of any value that may be in **sysVar.returnCode**.

returnCode

A literal integer or an item or constant that resolves to an integer. The return code is made available to the operating system.

For details on return codes, see *sysVar.returnCode*.

sysVar.returnCode

The system variable that includes the value returned to the operating system.

For details, see *sysVar.returnCode*.

stack

Returns control to the main function without setting a return value for the current function. A statement of the form *exit stack* removes all references to the intermediate functions in the run-time *stack*, which is a list of functions;

specifically, the current function plus the series of functions whose running made possible the running of the current function.

If you do not specify a *label* (as described later), processing continues at the statement after the most recently run function invocation in the main function. If you specify a label, processing continues at the statement that follows the label in the main function. The label may be before or after the most recently run function invocation in the main function.

If you specify an exit statement of the form *exit stack* in the main function, the next statement is processed, even if you specify a label. For details on how to go to a specified label in the current function, see *goTo*.

label

A series of characters that are displayed in the main function and outside of any blocks, including these:

- if
- else
- inside a **case** statement
- while
- try

When displayed at the location where processing continues, the label is followed by colon. For details on valid characters for the label, see *Naming conventions*.

Related reference

"goTo" on page 332

"Naming conventions" on page 468

"sysVar.returnValue" on page 535

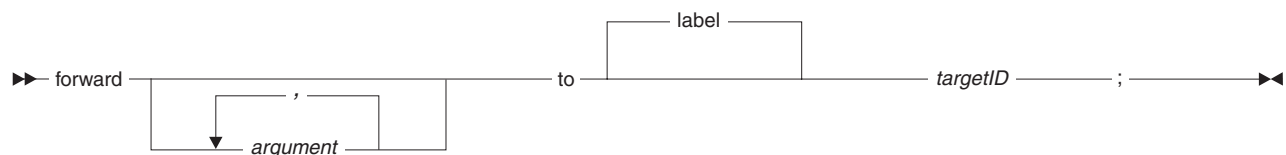
forward

The EGL **forward** statement is invoked from a page handler or from an EGL program. The primary purpose is to display a Web page with variable information; but the statement also can invoke a servlet or Java program that runs in the Web application server.

The statement acts as follows:

1. Commits recoverable resources, closes files, and releases locks
2. Forwards control
3. Ends the code that runs the **forward** statement

The syntax diagram is as follows:



argument

An item or record that is passed to the code being invoked. The names of an argument and its corresponding parameter must be the same in all cases. You may not pass literals.

Various restrictions are in effect:

- If you are invoking a page handler, the arguments must be compatible with the parameters specified for the **onPageLoad** function of the page handler. The function (if any) may have any valid name and is referenced by the page handler property **OnPageLoad**.
- If you are invoking a program, the arguments must be compatible with the program parameters.

The following details may be of interest, depending on how you are using the technology:

- The argument must be named the same as the corresponding parameter because the name is used as a key in storing and retrieving the argument value on the Web application server.
- Instead of passing an argument, the invoker can do as follows before invoking the **forward** statement:
 - Place a value in the request block by invoking the system function `sysLib.setRequestAttr`; or
 - Place a value in the session block by invoking the system function `sysLib.setSessionAttr`.

In this case, the receiver does not receive the value as an argument, but by invoking the appropriate system function:

- `sysLib.getRequestAttr` (to access data from the request block); or
- `sysLib.getSessionAttr` (to access data from the session block).
- A character item is passed as an object of type Java String.
- A record is passed as a Java Bean.

to label *targetID*

Specifies a Java Server Faces (JSF) label, which identifies a mapping in a run-time JSF-based configuration file. The mapping in turn identifies the object to invoke, whether a JSP (usually one associated with an EGL page handler), an EGL program, a non-EGL program, or a servlet. The word **label** is optional, and *targetID* is a quoted string.

Related reference

“Function invocations”

“`sysLib.getRequestAttr`” on page 608

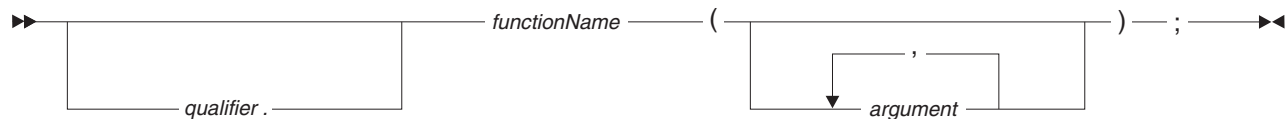
“`sysLib.getSessionAttr`” on page 609

“`sysVar.transferName`” on page 630

Function invocations

A function invocation runs an EGL-generated function or a system function, with these effects:

- Any arguments that are variables are passed *by reference*, which means that changes to the corresponding parameter change the area of memory available to the invoker
- When the invoked function ends, processing continues with the statement that follows the invocation



qualifier

One of the following symbols:

- The name of the library in which the function resides; or
- The name of the package in which the function resides, optionally followed by a period and the name of the library in which the function resides.
- *this* (identifies a function in the current program)

For details on the circumstances in which the qualifier is unnecessary, see *References to parts*.

function name

Name of the invoked function.

argument

Either a literal or the name of one of these data areas:

- A constant
- A data item
- A record
- A dynamic array of records or data items

If the invoked function returns a value, the function invocation can be used only as the source value in an assignment statement. Regardless of whether the function returns a value, the invocation is followed by a semicolon, as in these examples:

```
biggestNumber = maxOf(firstNumber,secondNumber);
readNumbers();
```

The following rules apply:

- If a function parameter is based on a primitive type, loose type, or a dataItem part, you can pass a literal, a constant, or an item. The arguments in the invocation must correspond to the number, order, and types of parameters in the function parameter list. The function invocation fails at run time in the case of a type mismatch, but in the case of a loose type, the test of a mismatch is less strict than in other cases, as described in *Function part*.
- If a function parameter is a record, the function argument must be a record.
- The structure in the argument record must correspond to the structure in the parameter record.

The following rules apply to literals and constants:

- A numeric literal or constant can be passed only if the receiving parameter is a numeric type
- A literal or constant that includes only single-byte characters may be passed to a parameter of type CHAR or MBCHAR
- A literal or constant that includes only double-byte characters may be passed only to a parameter of type DBCHAR
- A literal or constant that includes a combination of single- and double-byte characters may be passed to a parameter of type MBCHAR
- A literal or constant cannot be passed to a parameter of type HEX

Related concepts

"Function part" on page 380

"References to parts" on page 16

"Syntax diagram" on page 506

Related tasks

"Assignments" on page 296

Related reference

"EGL statements" on page 70

"Function part in EGL source format" on page 381

"Primitive types" on page 27

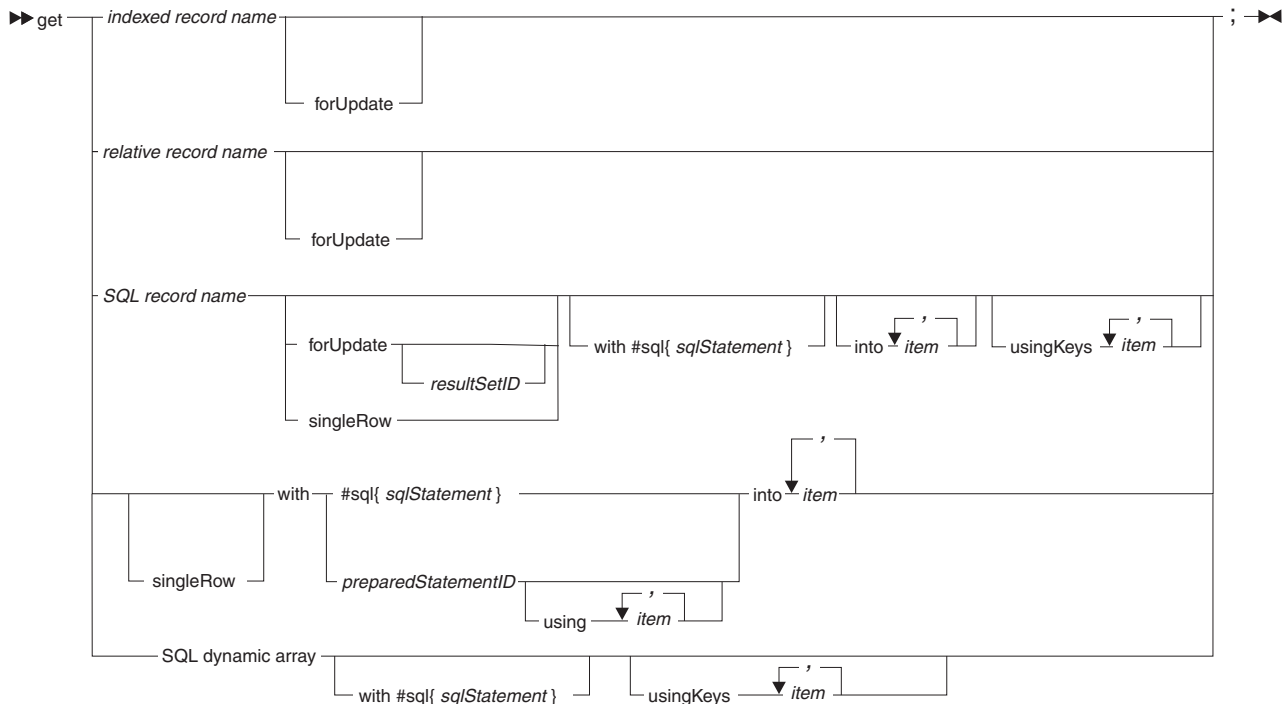
"System words" on page 79

"System words in alphabetical order" on page 508

get

The EGL **get** statement retrieves a single file record or database row and provides an option that lets you replace or delete the stored data later in your code. In addition, this statement allows you to retrieve a set of database rows and place each succeeding row into the next SQL record in a dynamic array.

The **get** statement is sometimes identified as **get by key value** and is distinct from **get next** and **get previous**.



record name

Name of an I/O object: an indexed, relative, or SQL record. For SQL processing, the record name is required if the EGL INTO clause (described later) is not specified.

forUpdate

Option that lets you use a later EGL statement to replace or delete the data that was retrieved from the file or database.

If the resource is recoverable (as in the case of a VSAM file or SQL database), the **forUpdate** option locks the record so that it cannot be changed by other programs until a commit occurs. For details on commit processing, see *Logical unit of work*.

resultSetID

A result-set identifier for use in an EGL **replace**, **delete**, or **execute** statement, as well as in an EGL **close** statement. For details, see *resultSetID*.

singleRow

Option that causes generation of more efficient SQL, as is appropriate when you are sure that the search criterion in the **get** statement applies to only one row and when you do not intend to update or delete the row. A run-time I/O error results if you specify this option when the search criterion applies to multiple rows. For additional details, see *SQL record*.

#sql{ sqlStatement }

An explicit SQL SELECT statement, as described in *SQL support*. Leave no space between **#sql** and the left brace.

into ... item

An EGL INTO clause, which identifies the EGL host variables that receive values from a relational database. This clause is required when you are processing SQL, in either of these cases:

- An SQL record is not specified; or
- Both an SQL record and an explicit SQL SELECT statement are specified, but a column in the SQL SELECT clause is not associated with a record item. (The association is in the SQL record part, as noted in *SQL item properties*.)

In a clause like this one (which is outside of an **#sql{ }** block), do not include a semicolon before the name of a host variable.

preparedStatementID

The identifier of an EGL **prepare** statement that prepares an SQL SELECT statement at run time. The **get** statement runs the SQL SELECT statement dynamically. For details, see *prepare*.

using ... item

A USING clause, which identifies the EGL host variables that are made available to the prepared SQL SELECT statement at run time. In a clause like this one (which is outside of an **sql-and-end** block), do not include a semicolon before the name of a host variable.

usingKeys ... item

Identifies a list of key items that are used to build the key-value component of the WHERE clause in an implicit SQL statement. The implicit SQL statement is used at run time if you do not specify an explicit SQL statement.

If you do not specify a **usingKeys** clause, the key-value component of the implicit statement is based on the SQL record part that is either referenced in the **get** statement or is the basis of the dynamic array referenced in the **get** statement.

In the case of a dynamic array, the items in the **usingKeys** clause (or the host variables in the SQL record) must *not* be in the SQL record that is the basis of the dynamic array.

The **usingKeys** information is ignored if you specify an explicit SQL statement.

SQL dynamic array

Name of a dynamic array that is composed of SQL records.

The following example shows how to read and replace a file record:

```
emp.empnum = 1;           // sets the key in record emp

try
  get emp forUpdate;
onException
  myErrorHandler(8); // exits the program
end

emp.empname = emp.empname + " Smith";

try
  replace emp;
onException
  myErrorHandler(12);
end
```

The next **get** statement uses the SQL record **emp** when retrieving a database row, with no subsequent update or deletion possible:

```
try
  get emp singleRow into empname with
    #sql{
      select empname
      from Employee
      where empnum = :empnum
    };
onException
  myErrorHandler(8);
end
```

The next example uses the same SQL record to replace an SQL row:

```
try
  get emp forUpdate into empname with
    #sql{
      select empname
      from Employee
      where empnum = :empnum
    };

onException
  myErrorHandler(8); // exits the program
end

emp.empname = emp.empname + " Smith";

try
  replace emp;
onException
  myErrorHandler(12);
end
```

Details on the **get** statement depend on the record type. For details on SQL processing, see *SQL record*.

Indexed record

When you issue a **get** statement against an indexed record, the key value in the record determines what record is retrieved from the file.

If you want to replace or delete an indexed (or relative) record, you must issue a **get** statement for the record and then issue the file-changing statement (**replace** or **delete**), with no intervening I/O operation against the same file. After you issue the **get** statement, the effect of the next I/O operation on the same file is as follows:

- If the next I/O operation is a **replace** statement on the same EGL record, the record is changed in the file
- If the next I/O operation is a **delete** statement on the same EGL record, the record in the file is marked for deletion
- If the next I/O operation is a **get** statement on a record in the same file and includes the `forUpdate` option, a subsequent **replace** or **delete** statement is valid on the newly read file record
- If the next I/O operation is a **get** statement on the same EGL record (with no `forUpdate` option) or is a **close** statement on the same file, the file record is released without change

If the file is a VSAM file, the EGL **get** statement (with the `forUpdate` option) prevents the record from being changed by other programs. In iSeries COBOL programs, the lock remains until a commit occurs, which may not happen until the end of the run unit, as described in *Run unit*.

Relative record

When you issue a **get** statement against a relative record, the key item associated with the record determines what record is retrieved from the file. The key item must be available to any function that uses the record and can be any of these:

- An item in the same record
- An item in a record that is global to the program or is local to the function that is running the **get** statement
- A data item that is global to the program or is local to the function that is running the **get** statement

If you want to replace or delete an indexed (or relative) record, you must issue a **get** statement for the record and then issue the file-changing statement (**replace** or **delete**), with no intervening I/O operation against the same file. After you issue the **get** statement, the effect of the next I/O operation on the same file is as follows:

- If the next I/O operation is a **replace** statement on the same EGL record, the record is changed in the file
- If the next I/O operation is a **delete** statement on the same EGL record, the record in the file is marked for deletion
- If the next I/O operation is a **get** on the same file (with the `forUpdate` option), a subsequent replace or delete is valid on the newly read file record
- If the next I/O operation is a **get** on the same EGL record (with no `forUpdate` option) or is a **close** on the same file, the file record is released without change

SQL record

The EGL **get** statement results in an SQL SELECT statement in the generated code. If you specify the `singleRow` option, the SQL SELECT statement is a stand-alone statement. Alternatively, the SQL SELECT statement is a clause in a cursor, as described in *SQL support*.

Error conditions: The following conditions are among those that are not valid when you use a **get** statement to read data from a relational database:

- You specify an SQL statement of a type other than SELECT
- You specify an SQL INTO clause directly in an SQL SELECT statement
- Aside from an SQL INTO clause, you specify some but not all of the clauses of an SQL SELECT statement
- You specify (or accept) an SQL SELECT statement that is associated with a column that either does not exist or is incompatible with the related host variable

The following error conditions are among those that can occur when you use the `forUpdate` option:

- You specify (or accept) an SQL statement that shows an intent to update multiple tables; or
- You use an SQL record as an I/O object, and all the record items are read only.

Also, the following situation causes an error:

- You customize an EGL **get** statement with the `forUpdate` option, but fail to indicate that a particular SQL table column is available for update; and
- The replace statement that is related to that **get** statement tries to revise the column.

You can solve the previous mismatch in any of these ways:

- When you customize the EGL **get** statement, include the column name in the SQL SELECT statement, FOR UPDATE OF clause; or
- When you customize the EGL replace statement, eliminate reference to the column in the SQL UPDATE statement, SET clause; or
- Accept the defaults for both the **get** and **replace** statements.

Implicit SQL SELECT statement: When you specify an SQL record as an I/O object for the **get** statement but do not specify an explicit SQL statement, the implicit SQL SELECT has the following characteristics:

- The record-specific property called **defaultSelectCondition** determines what table row is selected, so long as the value in each SQL table key column is equal to the value in the corresponding key item of the SQL record. If you specify neither a record key nor a default selection condition, all table rows are selected. If multiple table rows are selected for any reason, the first retrieved row is placed in the record.
- As a result of the association of record items and SQL table columns in the record definition, a given item receives the content of the related SQL table column.
- If you specify the `forUpdate` option, the SQL SELECT FOR UPDATE statement does not include record items that are read only.
- The SQL SELECT statement for a particular record is similar to the following statement, except that the FOR UPDATE OF clause is present only if the **get** statement includes the `forUpdate` option :

```
SELECT column01,
       column02, ...
       columnNN
FROM   tableName
WHERE  keyColumn01 = :keyItem01
FOR UPDATE OF
       column01,
       column02, ...
       columnNN
```

The SQL INTO clause on the standalone SQL SELECT or on the cursor-related FETCH statement is similar to this clause:

```
INTO    :recordItem01,  
        :recordItem02, ...  
        :recordItemNN
```

EGL derives the SQL INTO clause if the SQL record is accompanied by an explicit SQL SELECT statement when you have not specified an INTO clause. The items in the derived INTO clause are those that are associated with the columns listed in the SELECT clause of the SQL statement. (The item-and-column association is in the SQL record part, as noted in *SQL item properties*.) An EGL INTO clause is required if a column is not associated with an item.

When you specify a dynamic array of SQL records as an I/O object for the **get** statement but do not specify an explicit SQL statement, the implicit SQL SELECT is similar to that described for a single SQL record, with these differences:

- The key-value component of the query is a set of relationships that is based on a greater-than-or-equal-to condition:

```
keyColumn01 >= :keyItem01 &  
keyColumn02 >= :keyItem02 &  
.  
.  
.  
keyColumnN  >= :keyItemN
```

- The items in the **usingKeys** clause (or the host variables in the SQL record) must *not* be in the SQL record that is the basis of the dynamic array.

Related concepts

“Logical unit of work” on page 159

“Record types and properties” on page 13

“References to parts” on page 16

“resultSetID” on page 181

“SQL support” on page 171

Related tasks

“Syntax diagram” on page 506

Related reference

“add” on page 293

“close” on page 303

“delete” on page 307

“EGL statements” on page 70

“Exception handling” on page 75

“execute” on page 309

“get next” on page 324

“get previous” on page 328

“I/O error values” on page 418

“open” on page 335

“prepare” on page 339

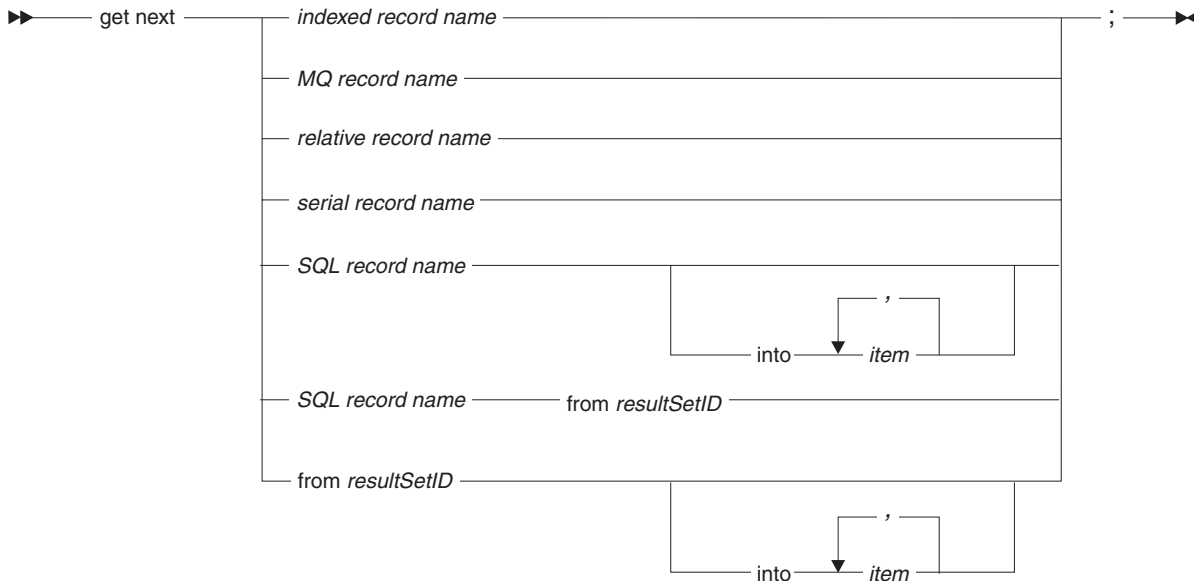
“replace” on page 341

“SQL item properties” on page 57

“sysVar.terminalID” on page 629

get next

The EGL **get next** statement reads the next record from a file or message queue, or the next row from a database.



record name

Name of the I/O object: an indexed, MQ, relative, serial, or SQL record.

from*resultSetID*

For SQL processing only, an ID that ties the **get next** statement to an **open** statement run earlier in the same program. For details, see *resultSetID*.

into

Begins an EGL into clause, which lists the items that receive values from a relational-database table.

item

An item that receives the value of a particular column. Do *not* precede the item name with a colon (:).

An example of file access is as follows:

```
try
  open record1 forUpdate;
  onException
    myErrorHandler(8);
  return;
end
try
  get next record1;
  onException
    myErrorHandler(12);
  return;
end

while (record1 not endOfFile)
  makeChanges(record1); // process the record

  try
    replace record1;
  onException
    myErrorHandler(16);
```

```

        return;
    end

    try
        get next record1;
    onException
        myErrorHandler(12);
        return;
    end
end // end while

sysLib.commit();

```

Details on the **get next** statement depend on the record type. For details on SQL processing, see *SQL record*.

Indexed record

When a **get next** statement operates on an indexed record, the effect is based on the current file position, which is set by either of these operations:

- A successful input or output (I/O) operation such as a **get** statement or another **get next** statement; or
- A **set** statement of the form *set record position*.

Rules are as follows:

- When the file is not open, the **get next** statement reads a record with the lowest key value in the file.
- Each subsequent **get next** statement reads a record that has the next highest key value in relation to the current file position. An exception for duplicate keys is described later.
- After a **get next** statement reads the record with the highest key value in the file, the next **get next** statement results in the I/O error value **endOfFile**.
- The current file position is affected by any of these operations:
 - An EGL **set** statement of the form *set record position* establishes a file position based on the *set value*, which is the key value in the indexed record that is referenced by the **set** statement. The subsequent **get next** statement against the same indexed record reads the file record that has a key value equal to or greater than the set value. If no such record exists, the result of the **get next** is **endOfFile**.
 - A successful I/O statement other than a **get next** statement establishes a new file position, and the subsequent **get next** statement issued against the same EGL record reads the *next* file record. After a **get previous** statement reads a file record, for example, the **get next** statement either reads the file record with the next-highest key or returns **endOfFile**.
 - If a **get previous** statement returns **endOfFile**, the subsequent **get next** statement retrieves the first record in the file.
 - After an unsuccessful **get**, **get next**, or **get previous** statement, the file position is undefined and must be re-established by a **set** statement of the form *set record position* or by an I/O operation other than a **get next** or **get previous** statement.
- When you are using an alternate index and duplicate keys are in the file, the following rules apply:
 - Retrieval of a record with a higher-valued key occurs only after **get next** statements have read all the records that have the same key as the most recently retrieved record. The order in which duplicate-keyed records are retrieved is the order in which VSAM returns the records.

- If a **get next** follows a successful I/O operation other than a **get next**, the **get next** skips over any duplicate-keyed records and retrieves the record with the next higher key.
- The EGL error value **duplicate** is not set when your program retrieves the last record in a group of records containing the same key.

Consider a file in which the keys are as follows:

1, 2, 2, 2, 3, 4

Each of the following tables illustrates the effect of running a sequence of EGL statements on the same indexed record.

The next two tables apply to EGL-generated COBOL code.

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for COBOL
get	2	2 (the first of three)	duplicate
get next	any	3	—

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for COBOL
set (of the form <i>set record position</i>)	2	no retrieval	duplicate
get next	any	2 (the first of three)	duplicate
get next	any	2 (the second)	duplicate
get next	any	2 (the third)	—
get next	any	3	—

The next two tables apply to EGL-generated Java code.

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for Java
get	2	2 (the first of three)	duplicate
get next	any	2 (the second)	duplicate
get next	any	2 (the third)	—
get next	any	3	—

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for Java
set (of the form <i>set record position</i>)	2	no retrieval	duplicate
get next	any	2 (the first of three)	—
get next	any	2 (the second)	duplicate
get next	any	2 (the third)	—
get next	any	3	—

Message queue

When a **get next** operates on a MQ record, the first record in the queue is read into the MQ record. This placement occurs because the **get next** invokes one or more MQSeries calls:

- MQCONN connects the generated code to the default queue manager and is invoked when no connection is active
- MQOPEN establishes a connection to the queue and is invoked when a connection is active but the queue is not open
- MQGET removes the record from the queue and is always invoked unless an error occurred in an earlier MQSeries call

Relative record

When a **get next** statement operates on a relative record, the effect is based on the current file position, which is set by a successful input or output (I/O) operation such as a **get** statement or another **get next** statement. Rules are as follows:

- When the file is not open, the **get next** statement reads the first record in the file.
- Each subsequent **get next** reads a record that has the next highest key value in relation to the current file position.
- A **get next** does not return **noRecordFound** if the next record is deleted. Instead, the **get next** skips deleted records and retrieves the next record in the file.
- After a **get next** statement reads the record with the highest key value in the file, the next **get next** statement results in the EGL error value **endOfFile**.
- The current file position is affected by any of these operations:
 - A successful I/O statement other than a **get next** establishes a new file position, and the subsequent **get next** against the same EGL record reads the *next* file record.
 - After an unsuccessful **get**, **get next**, or **get previous** statement, the file position is undefined and must be re-established by a **set** statement of the form *set record position* or by an I/O operation other than a **get next** statement.
- After a **get next** statement reads the last record in the file, the next **get next** statement results in the EGL error values **endOfFile** and **noRecordFound**.

Serial record

When a **get next** statement operates on a serial record, the effect is based on the current file position, which is set by another **get next** statement. Rules are as follows:

- When the file is not open, the **get next** statement reads the first record in the file.
- Each subsequent **get next** statement reads the next record.
- After a **get next** statement reads the last record, the subsequent **get next** statement results in the EGL error value **endOfFile**.
- If the generated code adds a serial record and then issues the equivalent of a **get next** statement on the same file, EGL closes and reopens the file before executing the **get next** statement. A **get next** statement that follows an **add** statement therefore reads the first record in the file. This behavior also occurs when the **get next** and **add** statements are in different programs, and one program calls another.

It is recommended that you avoid having the same file open in more than one program at the same time.

SQL record

When a **get next** statement operates on an SQL record, your code reads the next row from those selected by an **open** statement. If you issue a **get next** statement to retrieve a row that was selected by an **open** statement that has the **forUpdate** option, you can do any of these:

- Change the row with an EGL **replace** statement
- Remove the row with an EGL **delete** statement
- Change or remove the row with an EGL **execute** statement

An SQL FETCH statement represents the EGL **get next** statement in the generated code. The format of the generated SQL statement is as follows and cannot be changed, except to set the INTO clause:

```
FETCH cursor USING DESCRIPTOR SQLDA INTO ...
```

If you issue a **get next** statement that attempts to access a row that is beyond the last selected row, EGL sets the SQL record to **noRecordFound** and closes the cursor.

Finally, when you specify SQL COMMIT or sysLib.commit, your code retains position in the cursor that was declared in the **open** statement, but only in the following case:

- You are generating a COBOL program; and
- You use the hold option in the **open**.

Related concepts

"Record types and properties" on page 13

"resultSetID" on page 181

"SQL support" on page 171

Related tasks

"Syntax diagram" on page 506

Related reference

"add" on page 293

"close" on page 303

"delete" on page 307

"Exception handling" on page 75

"execute" on page 309

"get" on page 318

"get previous"

"I/O error values" on page 418

"EGL statements" on page 70

"open" on page 335

"prepare" on page 339

"replace" on page 341

"set" on page 345

"SQL item properties" on page 57

get previous

The EGL **get previous** statement reads the previous record in the file that is associated with a specified EGL indexed record.

►► — get previous ————— *indexed record name* ————— ; —►►

indexed record name

Name of the indexed record

An example is as follows:

```
record1.hexKey = "FF";
set record1 position;

try
  get previous record1;
onException
  myErrorHandler(8);
return;
end

while (record1 not endOfFile)
  processRecord(record1); // handle the data

  try
    get previous record1;
  onException
    myErrorHandler(8);
  return;
end
end
```

When a **get previous** statement operates on an indexed record, the effect is based on the current file position, which is set by either of these operations:

- A successful input or output (I/O) operation such as a **get** statement or another **get previous** statement; or
- A **set statement** of the form *set record position*.

Rules are as follows:

- When the file is not open, the **get previous** statement reads a record with the highest key value in the file.
- Each subsequent **get previous** reads a record that has the next lowest key value in relation to the current file position. An exception for duplicate keys is described later.
- After a **get previous** statement reads the record with the lowest key value in the file, the next **get previous** statement results in the EGL error value **endOfFile**.
- The current file position is affected by any of these operations:
 - An EGL **set statement** of the form *set record position* establishes a file position based on the *set value*, which is the key value in the indexed record that is referenced by the **set statement**. The subsequent **get previous** statement against the same indexed record reads the file record that has a key value equal to or less than the set value. If no such record exists, the result of the **get previous** statement is **endOfFile**.
If the set value is filled with hexadecimal FF, the result of a **set** statement of the form *set record position* is as follows:
 - The **set** statement establishes a file position after the last record in the file
 - If a **get previous** statement is the next I/O operation, the generated code retrieves the last record in the file
 - A successful I/O statement other than a **get previous** statement establishes a new file position, and the subsequent **get previous** statement against the

same EGL record reads the *previous* file record. After a **get next** statement reads a file record, for example, the **get previous** statement either reads the file record with the next-lowest key or returns **endOfFile**.

- If a **get next** statement returns **endOfFile**, the subsequent **get previous** statement retrieves the last record in the file.
- After an unsuccessful **get**, **get next**, or **get previous** statement, the file position is undefined and must be re-established by a **set** statement of the form *set record position* or by an I/O operation other than a **get next** or **get previous** statement.
- When you are using an alternate index and duplicate keys are in the file, the following rules apply:
 - Retrieval of a record with a lower-valued key occurs only after **get previous** statements have read all the records that have the same key as the most recently retrieved record. The order in which duplicate-keyed records are retrieved is the order in which VSAM returns the records.
 - If a **get previous** statement follows a successful I/O operation other than a **get previous**, the **get previous** statement skips over any duplicate-keyed records and retrieves the record with the next lower key.
 - The EGL error value **duplicate** is not set when your program retrieves the last record in a group of records containing the same key.

Consider a file in which the keys in an alternate index are as follows:

1, 2, 2, 2, 3, 4

Each of the following tables illustrates the effect of running a sequence of EGL statements on the same indexed record.

The next three tables apply to EGL-generated COBOL code.

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for COBOL
get	3	3	—
get previous	any	2 (the first of three)	duplicate
get previous	any	2 (the second)	duplicate
get previous	any	2 (the third)	—
get previous	any	1	—

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for COBOL
set (of the form <i>set record position</i>)	2	—	—
get next	any	2 (the first)	duplicate
get next	any	2 (the second)	—
get previous	any	1	—
get previous	any	--	endOfFile

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for COBOL
set (of the form <i>set record position</i>)	1	--	--

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for COBOL
get previous	any	1	--

The next three tables apply to EGL-generated Java code.

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for Java
get	3	3	—
get previous	any	2 (the first of three)	duplicate
get previous	any	2 (the second)	duplicate
get previous	any	2 (the third)	—
get previous	any	1	—

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for Java
set (of the form <i>set record position</i>)	2	—	duplicate
get next	any	2 (the first)	—
get next	any	2 (the second)	duplicate
get previous	any	1	—
get previous	any	--	endOfFile

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for Java
set (of the form <i>set record position</i>)	1	--	--
get previous	any	1	--

Related concepts

“Record types and properties” on page 13

Related tasks

“Syntax diagram” on page 506

Related reference

“add” on page 293

“close” on page 303

“delete” on page 307

“Exception handling” on page 75

“execute” on page 309

“get” on page 318

“get next” on page 324

“I/O error values” on page 418

“open” on page 335

“prepare” on page 339

“EGL statements” on page 70

“replace” on page 341

“set” on page 345

goTo

The EGL **goTo** statement causes processing to continue at a specified label, which must be in the same function as the statement and outside of a block.

► goto *label* : ————— ◀◀

label

A series of characters that are displayed elsewhere in the function, outside of any blocks, including these:

- if
- else
- when (in a **case** statement)
- while
- try

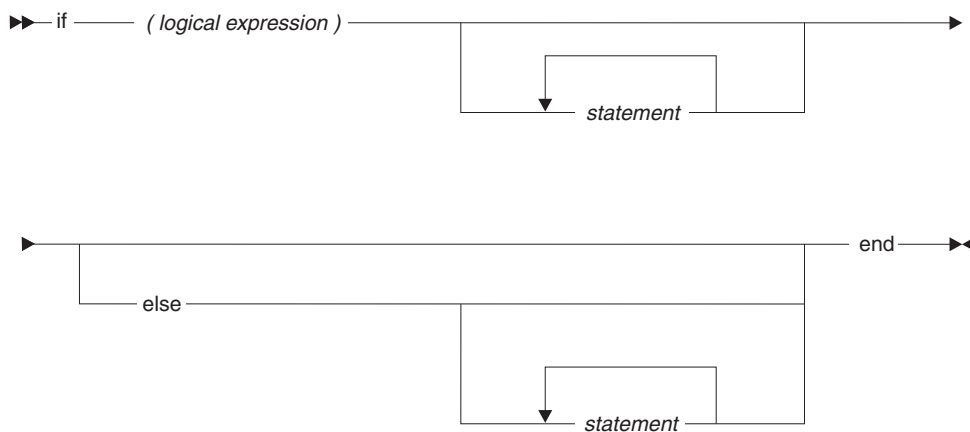
When displayed at the location where processing continues, the label is followed by colon. For details on valid characters for the label, see *Naming conventions*.

Related reference

"Naming conventions" on page 468

if, else

The EGL keyword **if** marks the start of a set of statements (if any) that run only if a logical expression resolves to true. The optional keyword **else** marks the start of an alternative set of statements (if any) that run only if the logical expression resolves to false. The keyword **end** marks the close of the *if* statement.



logical expression

An expression (a series of operands and operators) that evaluates to true or false

statement

One or more EGL statements

You may nest **if** and other end-terminated statements to any level. Each **end** keyword refers to the most recent statement that was not ended and that begins with one of these keywords:

- **if**
- **case**
- **try**
- **while**

None of those statements is followed by a semicolon.

An example is as follows:

```

if (userRequest = "U")
  try
    update myRecord;
    onException
      myErrorHandler(12); // ends program
  end
  try
    myRecord.myItem=25;
    replace record1;
    onException
      myErrorHandler(16);
  end
else
  try
    add record2;
    onException
      myErrorHandler(18); // ends program
  end
  if (sysVar.systemType is WIN)
    myFunction01();
  else
    myFunction02();
  end
end
end

```

Related tasks

“Syntax diagram” on page 506

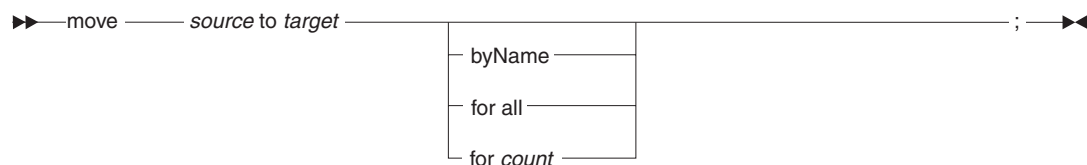
Related reference

“Logical expressions” on page 358

“EGL statements” on page 70

move

The EGL **move** statement copies data, in most cases from the named items in one structure to the same-named items in another.



source

One of these:

- A record (including an element of a record array)
- A form
- An item with or without a substructure
- A literal

The effect of the **move** statement varies by the source type:

- In the case of a record or form, **byName** is the default.
- In the case of a substructured item, the operation copies data by name only if you specify **byName**; otherwise, the operation is equivalent to one of these:
 - An assignment statement; or
 - Multiple assignment statements, if the target is an array of records. In this case, the operation is affected by whether you specify **for all** (the default) or **for count**, as described later.
- In the case of a literal or an item that is not substructured, use of **byName** is not valid; the operation is always equivalent to an assignment statement or (if the target is an array of records) to multiple assignment statements, as described later.

target

One of these:

- A record array
- A record (including an element of a record array)
- A form
- An item with or without a substructure

byName

When you specify **byName** and the target is not an array of records, the value of each subordinate item in the source structure is copied to the corresponding item with the same name in the target structure. The operation occurs in the order in which the items appear in the source structure. The operation is not valid if two or more items in the source or target structure have the same name.

Items whose name is an asterisk (*) are not themselves available, but their substructured, named items are available.

When you specify **byName** and the target is an array of records, the named areas in the source are copied to the same-named areas in successive elements of the target array. The operation is equivalent to a series of **move** statements, each of which assigns values to another array element.

for all

Meaningful only when the target is an array; otherwise, **for all** is ignored.

If the source is an element of an array, the operation moves that element and each subsequent element of the source array to an element in the target array.

Either the source or target array can be longer, and the process ends after data is copied between the last elements for which a match is found.

If the source is not an array, the operation uses the source value to initialize every element in the target array.

Depending on the source type, the operation is equivalent to a series of **move** statements or a series of **assignment** statements, as described earlier for *source*.

for *count*

Meaningful only when the target is an array; otherwise, **for *count*** is ignored.

count can be any of these:

- An integer literal
- A variable that resolves to an integer
- A numeric expression; but not a function invocation

If the source is an element of an array, the operation moves that element and each subsequent element of the source array to the corresponding target-array element, but only for the number of elements specified in *count*. The subscripts on the source and target arrays are the starting points for the move. For example, you can specify the following statement:

```
move source[2] to target[7] for 3;
```

The previous example moves elements 2, 3, and 4 from source into the elements 7, 8, and 9 in target.

Either the source or target array can be longer, and the process ends after the first of two events occurs:

- Data is copied between the last elements for which the copy is requested; or
- Data is copied between the last elements for which a match is found.

If the source is not an element of an array, the operation uses the source value to initialize the elements of the target array, beginning with the subscript specified on the target array and continuing for the number of elements specified in *count*. If the number of elements in the target array exceeds the number in *count*, the operation initializes all the elements in the target array.

Depending on the source type, the operation is equivalent to a series of **move** statements or a series of **assignment** statements, as described earlier for *source*.

An example is as follows:

```
move myRecord01 to myRecord02 byName;
```

Moves are checked for item-to-item compatibility. The rules for truncation, padding, and type conversion are the same as those detailed for the assignment statement.

When you are working with dynamic arrays, the last element in the array is determined by the array's current size. For details, see *Arrays*.

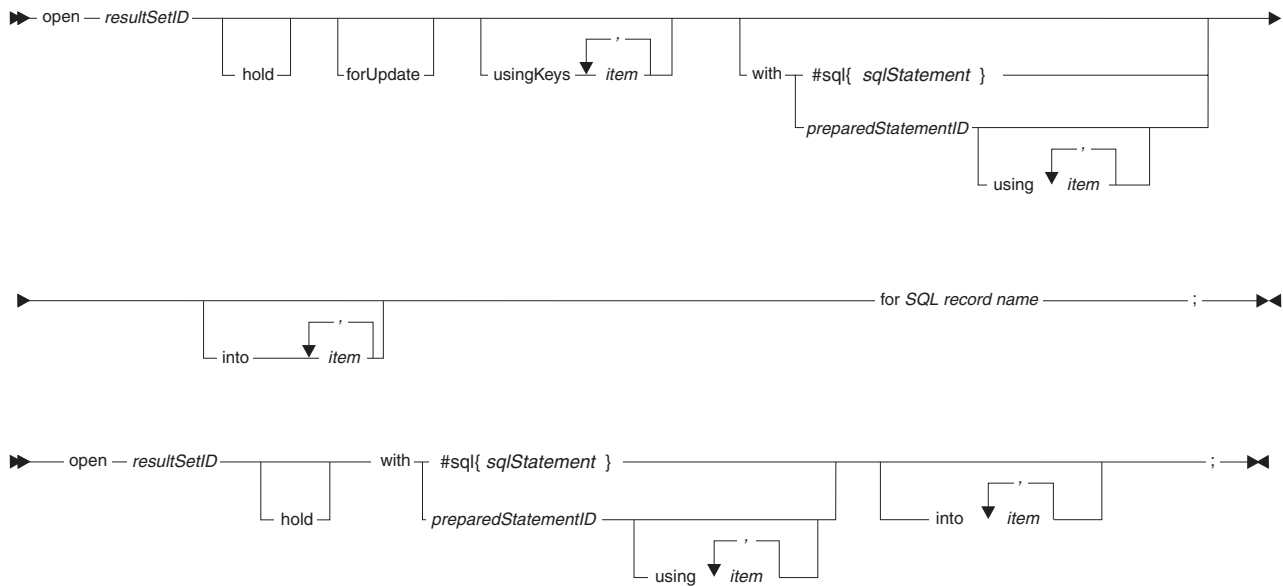
Related reference

"Arrays" on page 64

"Assignments" on page 296

open

The EGL **open** statement selects a set of rows from a relational database for later retrieval with **get next** statements. The **open** statement may operate on a cursor or on a called procedure.



resultSetID

ID that ties the open statement to later **get next**, **replace**, **delete**, and **close** statements. For details, see *resultSetID*.

hold

Maintains position in a result set when a commit occurs.

Note: The **hold** option is available only for COBOL programs.

The **hold** option is appropriate in the following case:

- You are using the EGL **open** statement to open a cursor rather than a stored procedure; and
- You want to commit changes periodically without losing your position in the result set; and
- Your database management system supports use of the WITH HOLD option in the SQL cursor declaration.

You code might do as follows, for example:

1. Declare and open a cursor by running an EGL **open** statement
2. Fetch a row by running an EGL **get next** statement
3. Do the following in a loop--
 - a. Process the data in some way
 - b. Update the row by running an EGL **replace** statement
 - c. Commit changes by running the system function `sysLib.commit`
 - d. Fetch another row by running an EGL **get next** statement

If you do not specify **hold**, the first run of step 3d fails because the cursor is no longer open.

Cursors for which you specify **hold** are not closed on a commit, but a rollback or database connect closes all cursors.

If you have no need to retain cursor position across a commit, do not specify **hold**.

forUpdate

Option that lets you use a later EGL statement to replace or delete the data that was retrieved from the database.

You cannot specify **forUpdate** if you are calling a stored procedure to retrieve a result set.

usingKeys ... item

Identifies a list of key items that are used to build the key-value component of the WHERE clause in an implicit SQL statement. The implicit SQL statement is used at run time if you do not specify an explicit SQL statement.

If you do not specify a **usingKeys** clause, the key-value component of the implicit statement is based on the SQL record part that is referenced in the **open** statement.

The **usingKeys** information is ignored if you specify an explicit SQL statement.

with #sql{ sqlStatement }

An explicit SQL SELECT statement, which is optional if you also specify an SQL record. Leave no space between the **#sql** and the left brace.

into ... item

An INTO clause, which identifies the EGL host variables that receive values from the cursor or stored procedure. In a clause like this one (which is outside of a **#sql{ }** block), do not include a semicolon before the name of a host variable.

with preparedStatementID

The identifier of an EGL **prepare** statement that prepares an SQL SELECT or CALL statement at run time. The **open** statement runs the SQL SELECT or CALL statement dynamically. For details, see *prepare*.

using ... item

A USING clause, which identifies the EGL host variables that are made available to the prepared SQL SELECT or CALL statement at run time. In a clause like this one (which is outside of a **#sql{ }** block), do not include a semicolon before the name of a host variable.

SQL record name

Name of a record of type `SQLRecord`. Either the record name or a value for *sqlStatement* is required; if *sqlStatement* is omitted, the SQL SELECT statement is derived from the SQL record.

Examples are as follows (assuming an SQL record called *emp*):

```
open empSetId forUpdate for emp;

open x1 with
  #sql{
    select empnum, empname, empphone
    from employee
    where empnum >= :empnum
    for update of empname, empphone
  }

open x2 with
  #sql{
    select empname, empphone
    from employee
    where empnum = :empnum
  }
for emp;
```

```

open x3 with
  #sql{
    call aResultSetStoredProc(:argumentItem)
  }

```

Default processing

The effect of an open statement is as follows by default, when you specify an SQL record:

- The open statement makes a set of rows available. Each column in the selected rows is associated with a structure item, and except for the columns that are associated with a read-only structure item, all the columns are available for subsequent update by an EGL replace statement.
- If you declare only one key item for the SQL record, the open statement selects all rows that fulfill the record-specific **default select condition**, so long as the value in the SQL table key column is greater than or equal to the value in the key item of the SQL record.
- If multiple keys are declared for the SQL record, the record-specific **default select condition** is the only search criterion, and the **open** statement retrieves all rows that meet that criterion.
- If you specify neither a record key nor a default selection condition, the **open** statement selects all rows in the table.
- The selected rows are not sorted.

The EGL **open** statement is represented in the generated code by a cursor declaration that includes an SQL SELECT or an SQL SELECT FOR UPDATE statement. The following is true by default:

- The FOR UPDATE clause (if any) does not include structure items that are read only
- The SQL SELECT statement for a particular record is similar to the following statement:

```

SELECT column01,
       column02, ...
       columnNN
INTO   :recordItem01,
       :recordItem02, ...
       :recordItemNN
FROM   tableName
WHERE  keyColumn01 = :keyItem01
FOR UPDATE OF
       column01,
       column02, ...
       columnNN

```

You may override the default by specifying an SQL statement in the EGL **open** statement.

Error conditions

Various conditions are not valid, including these:

- You include an SQL statement that lacks a clause required for SELECT; the required clauses are SELECT, FROM, and (if you specify **forUpdate**) FOR UPDATE OF
- Your SQL record is associated with a column that either does not exist at run time or is incompatible with the related structure item
- You specify the option **forUpdate**, and your code tries to run an **open** statement against either of the following SQL records:
 - An SQL record whose only structure items are read only; or

- An SQL record that is related to more than one SQL table.

A problem also arises in the following case:

1. You customize an EGL **open** statement for update, but fail to indicate that a particular SQL table column is available for update; and
2. The **replace** statement that is related to the **open** statement tries to revise the column.

You can solve this problem in any of these ways:

- When you customize the EGL **open** statement, include the column name in the SQL SELECT statement, FOR UPDATE OF clause; or
- When you customize the EGL **replace** statement, eliminate reference to the column in the SQL UPDATE statement, SET clause; or
- Accept the defaults for both the **open** and **replace** statements.

Related concepts

“Record types and properties” on page 13

"SQL support" on page 171

"resultSetID" on page 181

"References to parts" on page 16

Related tasks

“Syntax diagram” on page 506

Related reference

“add” on page 293

“close” on page 303

“delete” on page 307

“EGL statements” on page 70

“Exception handling” on page 75

"execute" on page 309

"get" on page 318

“get next” on page 324

"get previous" on page 328

"I/O error values" on page 418

“prepare”

“replace” on page 341

“SQL item properties” on page 57

“sysVar.terminalID” on page 629

prepare

The EGL **prepare** statement specifies an SQL PREPARE statement, which optionally includes details that are known only at run time. You run the prepared SQL statement by running an EGL **execute** statement or (if the SQL statement returns a result set) by running an EGL **open** or **get** statement.

►► prepare — *preparedStatementID* — from *stringExpression* — ; ►►
 for *SQL record name*

preparedStatementID

An identifier that relates the **prepare** statement to the **execute**, **open**, or **get** statement.

stringExpression

A *string expression* that contains a valid SQL statement.

SQL record name

Name of an SQL record. Specifying this name has two benefits:

- The EGL editor provides a dialog to derive an SQL statement based on your specifications
- After the **prepare** statement runs, you can test the record name against an I/O error value to determine whether the statement succeeded, as in the following example:

```
try
  prepare prep01 from
    "insert into " + aTableName +
    "(empnum, empname) " +
    "value ?, ?"
  for empRecord;

onException
  if empRecord is unique
    myErrorHandler(8);
  else
    myErrorHandler(12);
  end
end
```

Another example is as follows:

```
myString =
  "insert into myTable " + "(empnum, empname) " +
  "value ?, ?";

try
  prepare myStatement
  from myString;
onException
  myErrorHandler(12);    // exits the program
end

try
  execute myStatement
  using :myRecord.empnum,
        :myRecord.empname;
onException
  myErrorHandler(15);
end
```

As shown in the previous examples, the developer can use a question mark (?) where a host variable should appear. Then, the name of the host variable used at run time is placed in the using clause of the **execute**, **open**, or **get** statement that runs the prepared statement.

A **prepare** statement that acts on a row of a result set may include a phrase of the format *where current of resultSetIdentifier*. This technique is valid only in the following situation:

- The phrase is coded in a literal; and
- The result set is open when the **prepare** statement runs.

An example is as follows:

```

prepare prep02 from
  "update myTable " +
  "set empname = ?, empphone = ? where current of x1" ;

execute prep02 using empname, empphone ;

```

Related concepts

"References to parts" on page 16
 "Record types and properties" on page 13
 "SQL support" on page 171

Related tasks

"Syntax diagram" on page 506

Related reference

"add" on page 293
 "close" on page 303
 "delete" on page 307
 "Exception handling" on page 75
 "execute" on page 309
 "get" on page 318
 "get next" on page 324
 "get previous" on page 328
 "I/O error values" on page 418
 "EGL statements" on page 70
 "open" on page 335
 "replace"
 "SQL item properties" on page 57
 "String expressions" on page 365

print

The EGL **print** statement adds a print form to a run-time buffer, as described in *Print forms*.

► `print printFormName ;` ————— ❧

printFormName

Name of a print form that is visible to the program. For details on visibility, see *References to parts*.

Related concepts

"Print forms" on page 368
 "References to parts" on page 16

replace

The EGL **replace** statement puts a changed record into a file or database.



SQL record

In the case of SQL processing, the EGL **replace** statement results in an SQL UPDATE statement in the generated code.

You must retrieve a row for subsequent replacement, in either of two ways:

- Issue a **get** statement (with the `forUpdate` option) to retrieve the row; or
- Issue an **open** statement to select a set of rows, then invoke a **get next** statement to retrieve the row of interest.

Error conditions: The following conditions are among those that are not valid when you use a **replace** statement:

- You specify an SQL statement of a type other than UPDATE
- You specify some but not all clauses of an SQL UPDATE statement
- You do not specify a *resultSetID* value when one is necessary; for details, see *resultSetID*
- You specify (or accept) an UPDATE statement that has one of these characteristics--
 - Updates multiple tables
 - Is associated with a column that either does not exist or is incompatible with the related host variable
- You use an SQL record as an I/O object, and all the record items are read only

The following situation also causes an error:

- You customize an EGL **get** statement with the `forUpdate` option, but fail to indicate that a particular SQL table column is available for update; and
- The **replace** statement that is related to the **get** statement tries to revise the column.

You can solve the previous mismatch in any of these ways:

- When you customize the EGL **get** statement, include the column name in the SQL SELECT statement, FOR UPDATE OF clause; or
- When you customize the EGL **replace** statement, eliminate reference to the column in the SQL UPDATE statement, SET clause; or
- Accept the defaults for both the **get** and **replace** statements.

Implicit SQL statement: By default, the effect of a **replace** statement that writes an SQL record is as follows:

- As a result of the association of record items and SQL table columns in the record declaration, the generated code copies the data from each record item into the related SQL table column
- If you defined a record item to be read only, the value in the column that corresponds to that record item is unaffected

The SQL statement has these characteristics by default:

- The SQL UPDATE statement does not include record items that are read only
- The SQL UPDATE statement for a particular record is similar to the following statement:

```
UPDATE tableName
SET column01 = :recordItem01,
    column02 = :recordItem02,
    .
```

```

      .
      .
columnNN = :recordItemNN    WHERE CURRENT OF cursor

```

Related concepts

“Record types and properties” on page 13
 “References to parts” on page 16
 “resultSetID” on page 181
 “Run unit” on page 504
 “SQL support” on page 171

Related tasks

“Syntax diagram” on page 506

Related reference

“add” on page 293
 “close” on page 303
 “delete” on page 307
 “EGL statements” on page 70
 “Exception handling” on page 75
 “execute” on page 309
 “get” on page 318
 “get next” on page 324
 “get previous” on page 328
 “I/O error values” on page 418
 “open” on page 335
 “prepare” on page 339
 “SQL item properties” on page 57
 “sysVar.terminalID” on page 629

return

The EGL **return** statement exits from a function and optionally returns a value to the invoking function.



returnValue

An item, literal, or constant that is compatible with the **returns** specification in the EGL function declaration.

Although an item must correspond in all ways to the **returns** specification, the rules for literals and constants are as follows:

- A numeric literal or constant can be returned only if the primitive type in the **returns** specification is a numeric type
- A literal or constant that includes only single-byte characters can be returned only if the primitive type in the **returns** specification is CHAR or MBCHAR
- A literal or constant that includes only double-byte characters can be returned only if the primitive type in the **returns** specification is DBCHAR

- A literal or constant that includes a combination of single- and double-byte characters can be returned only if the primitive type in the **returns** specification is MBCHAR
- A literal or constant cannot be returned if the primitive type in the **returns** specification is HEX

A function that includes a **returns** specification must terminate with a **return** statement that includes a value. A function that lacks a **returns** specification may terminate with a **return** statement, which must not include a value.

The **return** statement gives control to the first statement that follows invocation of the function, even if the statement is in an **OnException** clause in a try block.

set

The following sections describe the effect of an EGL **set** statement:

- “Effect on a record as a whole”
- “Effect on a form as a whole” on page 346
- “Effect on an SQL item” on page 347
- “Effect on a field in a text form” on page 348

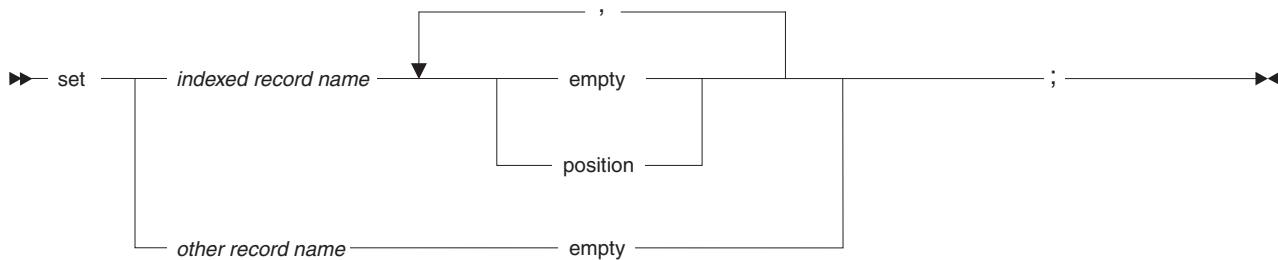
Effect on a record as a whole

The next table describes the **set** statements that affect a record as a whole or an array of records.

Format of set statement	Effect
set record empty	<p>Initializes the each of the elementary structure items, which are those at the lowest level of the record structure. The effect on each item depends on the primitive type, as described in <i>Data initialization</i>.</p> <p>This format is available for an array of records.</p>
set record position	<p>Establishes position in the VSAM file associated with an indexed record, as described later.</p> <p>This format is not available for an array of records.</p>

For an indexed record, you can combine statement formats, inserting a comma to separate the options **empty** and **position**. The options take effect in the order in which they appear in the **set** statement.

The syntax diagram is as follows:



indexed record name

Name of a record of type `indexedRecord`. You can specify a record array in a statement of the format *set record empty*, but not in a statement of the format *set record position*.

other record name

Name of a record of any type other than `indexedRecord`. You can specify a record array in a statement of the format *set record empty*.

empty

As described in the previous table.

position

Establishes a file position based on the *set value*, which is the key value in an indexed record. The overall effect depends on the next input or output operation that your code performs against the same indexed record:

- If the next operation is an EGL **get next** statement, that statement reads the first file record that has a key value equal to or greater than the set value. If no such record exists, the result of the **get next** statement is **endOfFile**.
- If the next operation after *set record position* is an EGL **get previous** statement, that statement reads the first file record that has a key value equal to or less than the set value. If no such record exists, the result of **get previous** is **endOfFile**.
- Any other operation after *set record position* resets the file position, and the *set record position* has no effect.

If the set value is filled with hexadecimal FF values, the following is true:

- The *set record position* establishes a file position after the last record in the file
- If the next operation is a **get previous** statement, the last record in the file is retrieved

Effect on a form as a whole

The next table describes the **set** statements that affect a form as a whole.

Format of set statement	Effect
set form alarm	For text forms only; sounds an alarm the next time that a converse statement presents the form.
set form empty	Initializes the value of each field in the form, clearing any content. The effect on a given field depends on the primitive type, as described in <i>Data initialization</i> .
set form initial	Resets each form field to its originally defined state, as expressed in the form declaration. Changes that were made by the program are canceled.

Format of set statement	Effect
set form initialAttributes	Resets each form field to its originally defined state, as expressed in the form declaration, but with one exception: The statement does not use the value property, which specifies the current content of a given field, to reset the contents of the fields.

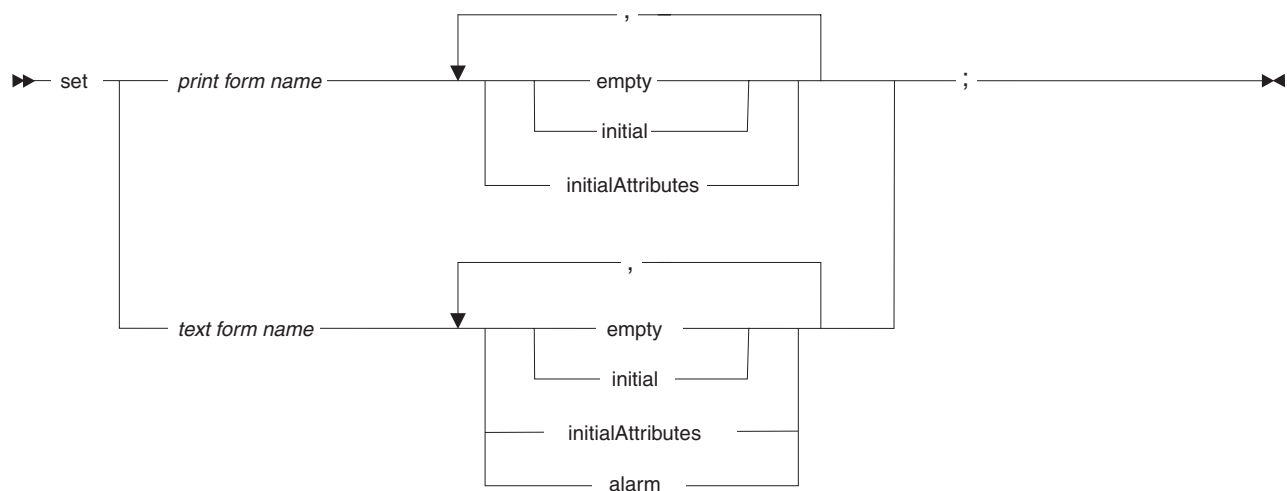
You can combine statement formats, inserting a comma to separate options such as **empty** and **alarm**. Of the following formats, you can choose one or none:

- *set form empty*
- *set form initial*

Of the following formats, you can choose one, both, or none:

- *set form alarm* (available only for text forms)
- *set form initialAttributes*

The syntax diagram is as follows:



print form name

Name of a form of type *print*, as described in *Form part*.

text form name

Name of a form of type *text*, as described in *Form part*.

The options are as described in the previous table.

Effect on an SQL item

The next table describes the format of the **set** statement that affects an item in an SQL record.

Format of set statement	Effect
set item null	Nulls the item, as is valid only for structure items that are in records of type SQLRecord, and only if the IsNullable property of the item is set to <i>yes</i> . (Arrays are not valid in records of type SQLRecord.) For details on null processing, see <i>SQL item properties</i> .

The syntax diagram is as follows:

►► set ——— SQL item name ——— null ——— ; ◄◄

SQL item name

Name of the item.

The options are as described in the previous table.

Effect on a field in a text form

The next table describes the **set** statements that affect a field or an array of fields in a text form. A given **set** statement can combine options only in a particular set of ways, as described later.

Note: Many of the actions described are dependent on the device where the text form is displayed. It is recommended that you test your output on each of the devices that you are supporting.

Format of set statement	Effect
set field blink	Causes the text to blink repeatedly.
set field bold	Cause the text to appear in boldface.
set field cursor	Positions the cursor in the specified field. If the field identifies an array and has no occurs value, the cursor is positioned at the first array element by default. If your program runs multiple statements of the format <i>set field cursor</i> , the last is in effect when the converse statement runs.
set field defaultColor	Sets the field-specific color property to <i>defaultColor</i> , which means that other conditions determine the displayed color. For details, see <i>Field-presentation properties</i> .
set field empty	Initializes the value of the field, clearing any content. The effect on a given field depends on the primitive type, as described in <i>Data initialization</i> .

Format of set statement	Effect
set field full	<p>Sets an empty, blank, or null field to a series of identical characters before the form is presented:</p> <ul style="list-style-type: none"> The character is an asterisk (*) if the field property fillCharacter is the following value (which is also the default value for fillCharacter): <ul style="list-style-type: none"> 0 for fields of type HEX space for fields of a numeric type null for other fields If fillCharacter is not set as described, the character is identical to the value of fillCharacter. <p>The on-form characters are returned to the program only if the modified data tag for the field is set, as described in <i>Modified data tag and property</i>. A user who changes the field must remove all the on-field characters to prevent their return to the program.</p> <p>Use of <i>set field full</i> has an effect only if the form group is generated with the build descriptor option <i>setFormItemFull</i>.</p> <p>A field of type MBCHAR is considered to be empty if it contains all single-byte spaces. In relation to such fields, <i>set field full</i> assigns a series of single-byte characters.</p>
set field initial	Resets the field to its originally defined state, independent of any changes made by the program
set field initialAttributes	Resets the field to its originally defined state, without using the value property (which specifies the current content of the field)
set field invisible	Removes any indication that the field is on the form.
set field modified	Sets the modified data tag, as described in <i>Modified data tag and property</i> .
set field noHighlight	Eliminates the special effects of blink, reverse, and underline.
set field normal	<p>Resets the fields as described in relation to the following formats:</p> <ul style="list-style-type: none"> Set field normalIntensity Set field unmodified Set field unprotected <p>For details, see the next table.</p>
set field normalIntensity	Sets the field to be visible, without boldface.
set field protect	Sets the field so that the user cannot overwrite the value in it. See also <i>set field skip</i> .

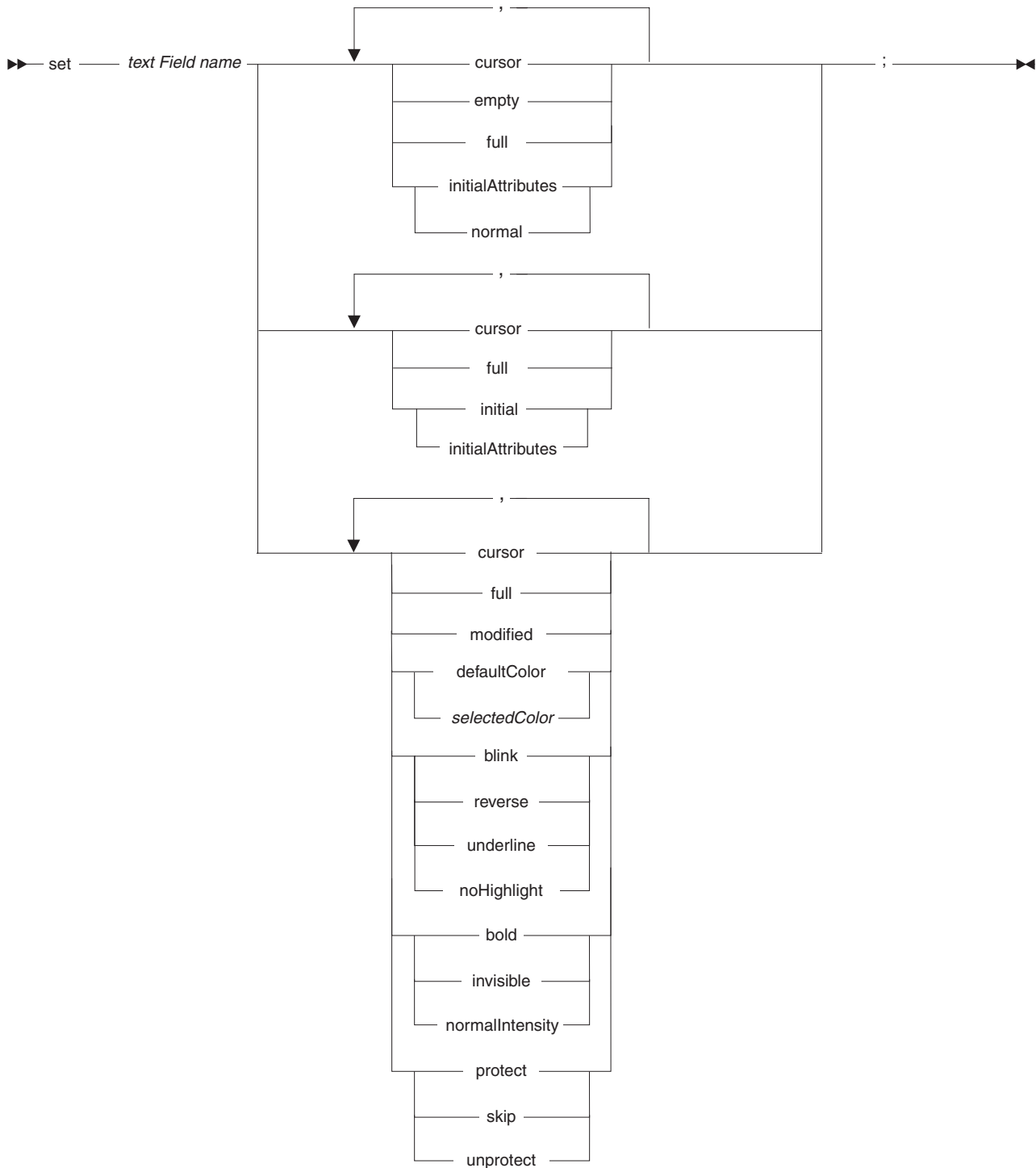
Format of set statement	Effect
set field reverse	Reverses the text and background colors, so that (for example) if the display has a dark background with light letters, the background becomes light and the text becomes dark.
set field <i>selectedColor</i>	Sets the field-specific color property to the value you specify. The valid values for <i>selectedColor</i> are as follows: <ul style="list-style-type: none"> • black • blue • green • pink • red • turquoise • white • yellow
set field skip	Sets the field so that the user cannot overwrite the value in it. In addition, the cursor skips the field in either of these cases: <ul style="list-style-type: none"> • The user is working on the previous field in the tab order and either presses Tab or fills that previous field with content; or • The user is working on the next field in the tab order and presses Shift Tab.
set field underline	Places an underline at the bottom of the field.
set field unprotect	Sets the field so that the user can overwrite the value in it.

You can combine statement formats, inserting a comma to separate options such as **cursor** and **full**, in any of three ways:

1. You can construct a **set** statement as follows--
 - Choose one or none of these field-attribute formats:
 - *set field initialAttributes*
 - *set field normal*
 - Choose any number of the next formats:
 - *set field cursor*
 - *set field empty*
 - *set field full*
2. Second, you can construct a **set** statement from any number of the next formats:
 - *set field cursor*
 - *set field full*
 - *set field initial* or *set field initialAttributes*
3. Last, you can construct a **set** statement as follows--
 - Choose any number of the next formats:
 - *set field cursor*
 - *set field full*

- *set field modified*
- Choose one or none of the color formats:
 - *set field defaultColor*
 - *set field selectedColor*
- Choose one or none of the highlight formats:
 - *set field blink*
 - *set field reverse*
 - *set field underline*
 - *set field noHighlight*
- Choose one or none of the intensity formats:
 - *set field bold*
 - *set field invisible*
 - *set field normalIntensity*
- Choose one or none of the protection formats:
 - *set field protect*
 - *set field skip*
 - *set field unprotect*

The syntax diagram is as follows:



field name

Name of the field in a text form. The name may refer to an array of fields.

The options are as described in the previous table.

Related concepts

"Form part" on page 366

"Modified data tag and modified property" on page 152

"Syntax diagram" on page 506

Related reference

"Data initialization" on page 277
"EGL statements" on page 70
"Field-presentation properties" on page 45
"get next" on page 324
"get previous" on page 328
"Items" on page 43
"setFormItemFull" on page 261
"SQL item properties" on page 57

show

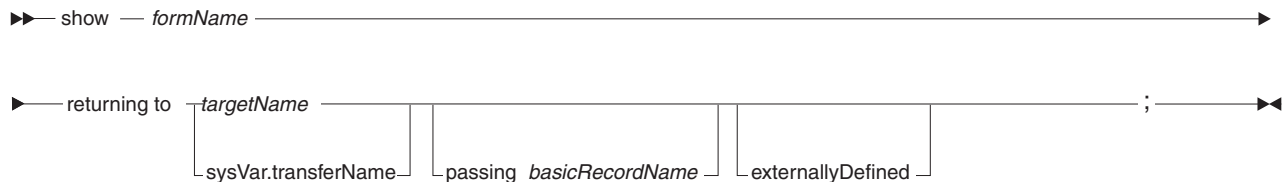
The **show** statement presents a text form from a main program:

1. Commits recoverable resources, closes files, and releases locks
2. Optionally, passes a basic record for use by the program that is specified in the **show** statement's returning clause (if any)
3. Ends the first program
4. Presents the text form

The **show** statement is not available in a called program.

If you include a returning clause in the **show** statement, the EGL run time invokes the specified program when the user presses an event key. The form data is assigned to the receiving program's *input form*. The passed record (unchanged by user input) is assigned to the receiving program's *input record*.

If you do not include a returning clause, the operation ends when the text form is presented.



formPartName

Name of a text form that is visible to the program. For details on visibility, see *References to parts*. If you include a returning clause in the statement, the text form must be equivalent to the text form specified in the **inputForm** property of the program being invoked.

targetName

Name of the program that is invoked after the user submits the text form.

sysVar.transferName

A system variable that contains the identifier of the program to be invoked. Use this variable to set the identifier at run time.

basicRecordName

Name of a record of type *basicRecord*. The content is assigned to the receiving program's *input record*.

externallyDefined

An indicator that the program is externally defined. This indicator is available

only if you set the project property for VisualAge Generator compatibility and is appropriate only if you are generating a COBOL program.

It is recommended that a non-EGL-generated program be identified as externally defined not in the **show** statement, but in the linkage options part that is used at generation time. (The related property is in the linkage options part, `transferLink` element, and is also called **externallyDefined**.) You can make the identification, however, in either way.

Related concepts

"References to parts" on page 16

Related reference

"`sysVar.transferName`" on page 630

transfer

The EGL **transfer** statement gives control from one main program to another, ends the transferring program, and optionally passes a record whose data is accepted into the receiving program's *input record*. You cannot use a **transfer** statement in a called program.

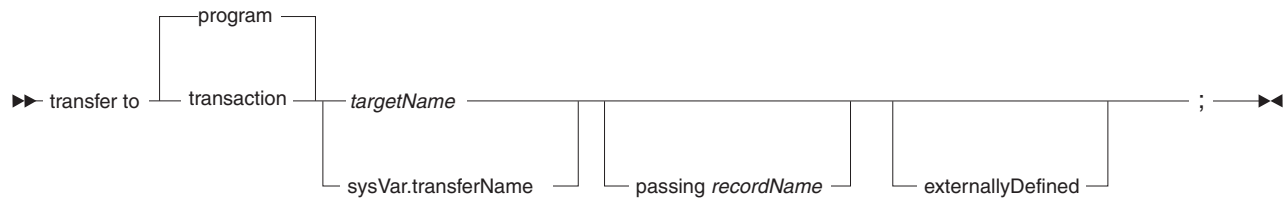
Your program can transfer control by a statement of the form *transfer to a transaction* or by a statement of the form *transfer to a program*:

- A transfer to a transaction acts as follows--
 - In a program that runs as a Java main text or main batch program, the behavior depends on the setting of build descriptor option **synchOnTrxTransfer**--
 - If the value of **synchOnTrxTransfer** is YES, the transfer statement commits recoverable resources, closes files, closes cursors, and starts a program in the same run unit.
 - If the value of **synchOnTrxTransfer** is NO (the default), the transfer statement also starts a program in the same run unit, but does not close or commit resources, which are available to the invoked program.
 - In a page handler, a transfer to a transaction is not valid; use the **forward** statement instead.
- A *transfer to a program* does not commit or rollback recoverable resources, but closes files, closes cursors, and starts a program in the same run unit.

The linkage options part, **transferLink** element has no effect when you are transferring control from Java code to Java code, but is meaningful otherwise.

If you are transferring control code to code that was *not* written with EGL or VisualAge Generator, it is recommended that you set the linkage options part, **transferLink** element. Set the **linkType** property to *externallyDefined*.

If you are running in VisualAge Generator compatibility mode, you can specify the option **externallyDefined** in the transfer statement, as occurs for programs migrated from VisualAge Generator; but it is recommended that you set the equivalent value in the linkage options part instead. For details on VisualAge Generator compatibility mode, see *Compatibility with VisualAge Generator*.



program *targetName* (the default)

The program that receives control. If you are generating for COBOL and specify a program name of more than 8 characters, the program name is truncated to 8 characters with character substitutions (if needed), as described in *Name aliasing*.

transaction *targetName*

The program that receives control, as described earlier.

sysVar.transferName

A system function that contains a target name that can be set at run time. For details, see *sysVar.transferName*.

passing *recordName*

A record that is received as the input record in the target program. The passed record may be of any type, but the length and primitive types must be compatible with the record that receives the data. The input record in the target program must be of type *basicRecord*.

externallyDefined

Not recommended for new development, as described earlier.

Related concepts

"Compatibility with VisualAge Generator" on page 276

"Name aliasing" on page 463

Related reference

"sysVar.transferName" on page 630

try

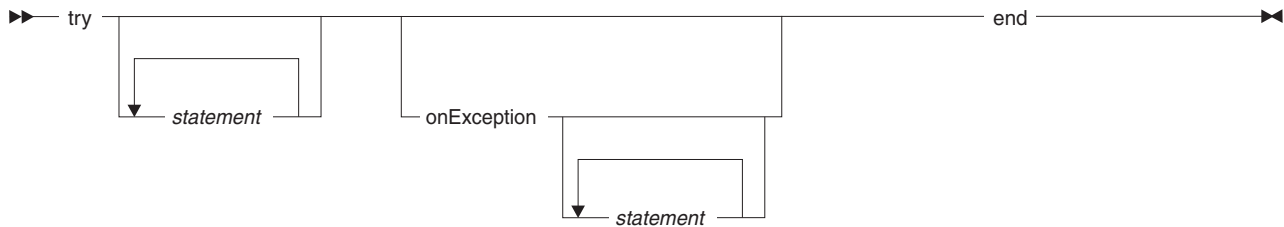
The EGL **try** statement indicates that the program continues running if a statement of any of the following kinds results in an error and is within the **try** statement:

- An input/output (I/O) statement
- A system-function invocation
- A **call** statement

If an exception occurs, processing resumes at the first statement in the **onException** block (if any), or at the first statement following the end of the **try** statement. A hard I/O error, however, is handled only if the system variable **sysVar.handleHardIOErrors** is set to 1; otherwise, the program displays a message (if possible) and ends.

A **try** statement has no effect on run-time behavior when an exception occurs in a function or program that is invoked from within the **try** statement.

For other details, see *Exception handling*.



statement

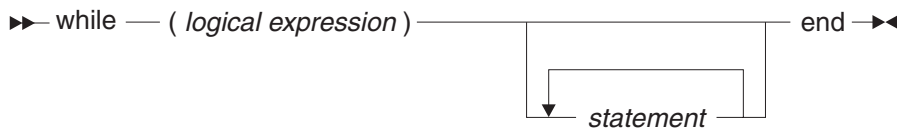
Any EGL statement.

OnException

A block of statements that run if an exception condition occurs.

while

The EGL keyword *while* marks the start of a set of statements that run in a loop. The first run occurs only if a logical expression resolves to true, and each subsequent iteration depends on the same test. The keyword *end* marks the close of the *while* statement.



logical expression

An expression (a series of operands and operators) that evaluates to true or false

statement

A statement in the EGL language

An example is as follows:

```

sum = 0;
i = 1;
while (i < 4)
    sum = inputArray[i] + sum;
    i = i + 1;
end
  
```

Related tasks

"Syntax diagram" on page 506

Related reference

"Logical expressions" on page 358

"EGL statements" on page 70

EGL system limits

No EGL-defined limits are in effect for the number of parts or the number of hierarchical levels in an EGL file. The following limits apply, however:

- A program can use no more than 32767 data items, structure items, and literals.
- A call statement can have no more than 30 arguments; also, these restrictions apply to the size of the arguments in total--

- Can be no more than 32567 if remoteCall or.ejbCall is the value of the **type** property for the call.

Both properties are in the linkage options part, callLink element.

- An item can be no more than 32767 bytes.
- In most cases, a numeric literal or item can have no more than 18 digits plus a sign, decimal point, or both; but an item that receives the result created by invoking the **mathLib.round** function can be more than 17 digits plus a sign, decimal point, or both.
- A static array can have no more than 7 dimensions and can have no more than 32767 elements in total.
- The situation for a dynamic array is as follows:
 - A dynamic array can have no more than 7 dimensions. The number of dimensions in a dynamic record array is one (for the record array declaration) plus the number of dimensions in the record structure.
 - A dynamic array can have a maximum size no greater than 2,147,483,647 elements for Java environments and no greater than 1,044,472 elements for COBOL. Those numbers are in effect if you do not specify a maximum size, but the size that can be allocated is further limited by the memory available at run time.
 - The total size for all arguments that can be passed on a remote call is limited by the maximum buffer size supported for the protocol.

When a program is generated for CICS for z/OS, additional limits are as follows:

- The maximum number of bytes for an indexed, relative, or serial record that accesses a VSAM file is 32688 for a journaled record, 32763 for a non-journaled record.
- The maximum number of bytes for a record that accesses a transient data queue is 32763.
- The maximum number of bytes for a relative or serial record that accesses a temporary storage queue is 32762.
- The maximum number of bytes for a record that accesses a spool file is 32763.
- The total size for all dynamic-array arguments that can be passed on a remote call is 32 kilobytes for a call that uses the CICS external call interface.

Related reference

“callLink element” on page 443

“Items” on page 43

“mathLib.round” on page 594

“Naming conventions” on page 468

“parmForm in callLink element” on page 452

“type in callLink element” on page 458

Expressions

An expression is a series of operands and operators that you specify when you write a program or function script.

Each expression resolves to a particular type of value at run time. A *numeric expression* resolves to a number; a *string expression* resolves to a series of characters; and a *logical expression* resolves to true or false.

Related reference

"Logical expressions"

"Numeric expressions" on page 364

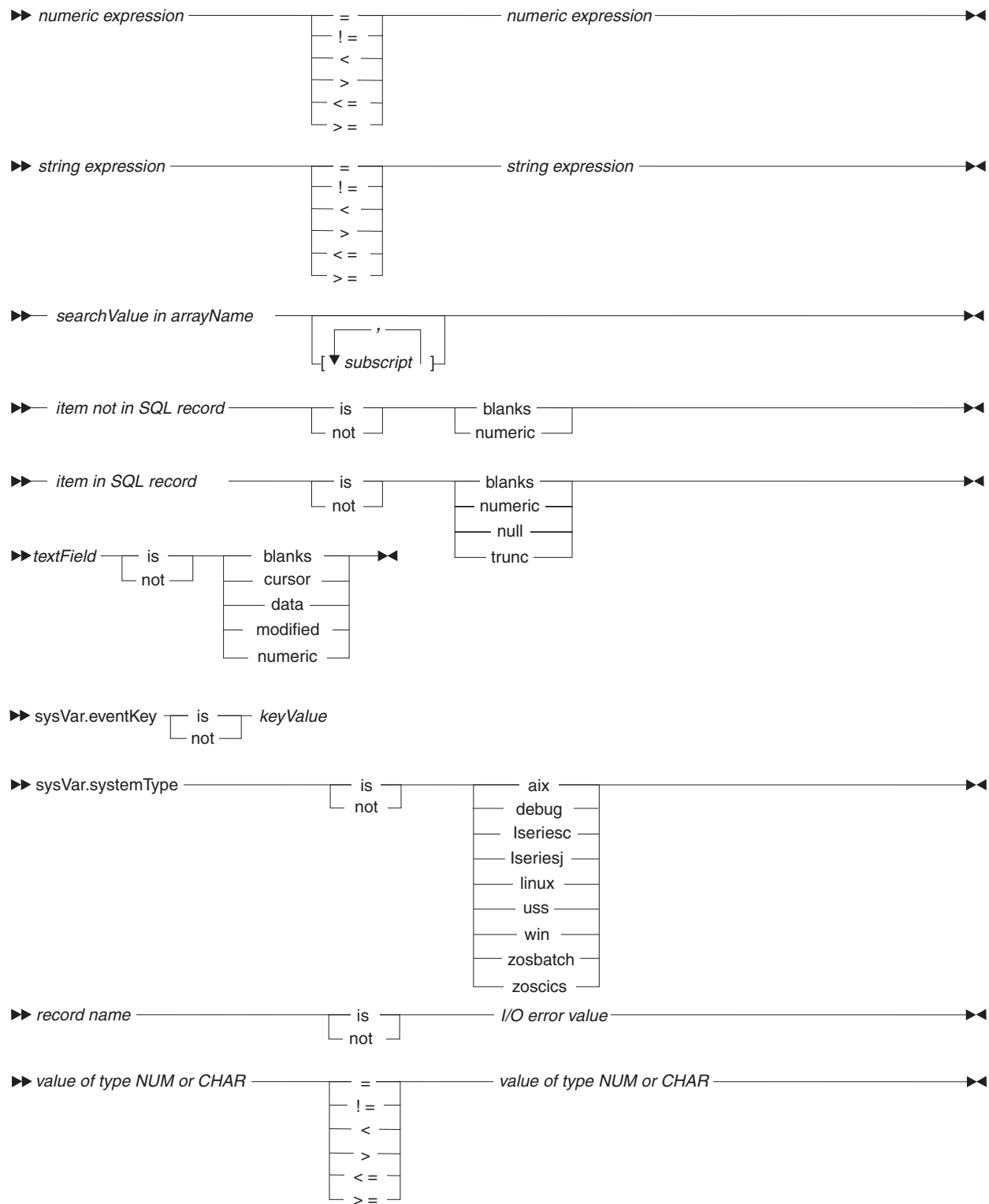
"String expressions" on page 365

Logical expressions

A *logical expression* resolves to true or false and is used as the criteria in an **if** or **while** statement.

Elementary logical expressions

An elementary logical expression is composed of an operand, a comparison operator, and a second operand, as shown in this syntax diagram and the subsequent table:



First operand	Comparison Operator	Second operand
<i>numeric expression</i>	One of these: =, !=, <, >, <=, >=	<i>numeric expression</i>

First operand	Comparison Operator	Second operand
<i>string expression</i>	One of these: =, !=, <, >, <=, >=	<i>string expression</i>
<i>Value of type NUM or CHAR, as described for the second operand</i>	One of these: =, !=, <, >, <=, >=	<i>Value of type NUM or CHAR, which can be any of these:</i> <ul style="list-style-type: none"> • An item that is of type NUM and has no decimal places • An integer literal • An item or literal of type CHAR
<i>searchValue</i>	in	<i>arrayName</i> ; for details, see <i>in</i> .
<i>item not in SQL record</i>	One of these: <ul style="list-style-type: none"> • is • not 	One of these: <ul style="list-style-type: none"> • blanks (for testing whether the value of a character item is or is not blanks only) • numeric (for testing whether the value of an item of type CHAR or MBCHAR is or is not numeric)
<i>item in an SQL record</i>	One of these: <ul style="list-style-type: none"> • is • not 	One of these: <ul style="list-style-type: none"> • blanks (for testing whether the value of a character item is or is not blanks only) • null (for testing whether the item was set to null either by a set statement or by reading from a relational database) • numeric (for testing whether the value of an item of type CHAR or MBCHAR is or is not numeric) • trunc (for testing whether non-blank characters were deleted on the right when a single- or double-byte character value was last read from a relational database into the item) <p>The trunc test can resolve to true only when the database column is longer than the item. The value for the test is false after a value is moved to the item or after the item is set to null.</p>

First operand	Comparison Operator	Second operand
<i>textField</i> (the name of a field in a text form)	One of these: <ul style="list-style-type: none"> • is • not 	One of these: <ul style="list-style-type: none"> • blanks (for testing whether the value of the text field is or is not limited to blanks or nulls). The test for blanks is based on the user's last input to the form, not on the current contents of the form item; and a test that uses <i>is</i> is true in these cases: <ul style="list-style-type: none"> – The user's last input was blanks or null; or – The user entered no data in the field since the start of the program or since a set statement ran that was of type <i>set form initial</i>. • cursor (for testing whether the user left the cursor in the specified text field). • data (for testing whether data other than blanks or nulls is in the specified text field). • modified (for testing whether the field's modified data tag is set, as described in Modified data tag and property). • numeric (for testing whether the value of an item of type CHAR or MBCHAR is or is not numeric).
<i>sysVar.eventKey</i>	One of these: <ul style="list-style-type: none"> • is • not 	For details, see <i>sysVar.eventKey</i> .
<i>sysVar.systemType</i>	One of these: <ul style="list-style-type: none"> • is • not 	For details, see <i>sysVar.systemType</i> . You cannot use <i>is</i> or <i>not</i> to test a value returned by <i>sysLib.getVAGSysType</i> .
<i>record name</i>	One of these: <ul style="list-style-type: none"> • is • not 	An I/O error value appropriate for the record organization. See <i>I/O error values</i> .

The next table lists the comparison operators, each of which resolves to true or false.

Operator	Purpose
=	The <i>equal</i> operator indicates whether two operands have the same value.
!=	The <i>not equal</i> operator indicates whether two operands have different values.
<	The <i>less than</i> operator indicates whether the first of two operands is numerically less than the second.
>	The <i>greater than</i> operator indicates whether the first of two operands is numerically greater than the second.
<=	The <i>less than or equal to</i> operator indicates whether the first of two operands is numerically less than or equal to the second.

Operator	Purpose
>=	The <i>greater than or equal to</i> operator indicates whether the first of two operands is numerically greater than or equal to the second.
in	The <i>in</i> operator indicates whether the first of two operands is a value in the second operand, which references an array. For details, see <i>in</i> .
is	The <i>is</i> operator indicates whether the first of two operands is in the category of the second. For details, see the previous table.
not	The <i>not</i> operator indicates whether the first of two operands is not in the category of the second. For details, see the previous table.

The next table and the explanations that follow tell the compatibility rules when the operands are of the specified types.

Primitive type of first operand	Primitive type of second operand
BIN	BIN, DECIMAL, NUM, NUMC, PACF
CHAR	CHAR, HEX, MBCHAR, NUM
DBCHAR	DBCHAR
DECIMAL	BIN, DECIMAL, NUM, NUMC, PACF
HEX	CHAR, HEX
MBCHAR	CHAR, MBCHAR
NUM	BIN, CHAR, DECIMAL, NUM, NUMC, PACF,
NUMC	BIN, DECIMAL, NUM, NUMC, PACF
PACF	BIN, DECIMAL, NUM, NUMC, PACF
UNICODE	UNICODE

Details are as follows:

- A value of any of the numeric types (BIN, DECIMAL, NUM, NUMC, PACF) can be compared to a value of any numeric type and size, and EGL does temporary conversions as appropriate. An equality comparison of equivalent fractions (like 1.4 and 1.40) evaluates to true, even if the decimal places are different.
- A value of type CHAR can be compared to a value of type HEX only if each character of type CHAR is within the range of hexadecimal digits (0-9, A-F, a-f). EGL temporarily converts any lowercase letters to uppercase in the value of type CHAR.
- If a comparison includes two values of character type (CHAR, DBCHAR, HEX, MBCHAR, UNICODE) and one value has fewer bytes than the other, a temporary conversion pads the shorter value on the right:
 - In a comparison with a value of type MBCHAR, a value of type CHAR is padded on the right with single-byte blanks
 - In a comparison with a value of type HEX, a value of type CHAR is padded on the right with binary zeros
 - A value of type DBCHAR is padded on the right with double-byte blanks
 - A value of type UNICODE is padded on the right with Unicode double-byte blanks
 - A value of type HEX is padded on the right with binary zeros, which means (for example) that if a value "0A" must be expanded to two bytes, the value for comparison purposes is "0A00" rather than "000A"

- A value of type CHAR can be compared to a value of type NUM only if these conditions apply:

- The value of type CHAR has single-byte digits, with no other characters
- The definition of the value of type NUM has no decimal point

A CHAR-to-NUM comparison works as follows:

- A temporary conversion puts the NUM value into a CHAR format. The numeric characters are left-justified, with additional single-byte blanks as needed. If a NUM-type item of length 4 has a value of 7, for example, the value is treated as "7" with three blanks on the right.
- If the length of the items do not match, a temporary conversion pads the shorter value with blanks on the right.
- The comparison checks the values byte-by-byte. Consider two examples:
 - A CHAR-type item of length 2 and value "7 " (including a blank) is equal to a NUM-type item of length 1 and value 7 because the temporary item that is based on the NUM-type field also includes a final blank
 - A CHAR-type item of value "8" is greater than a NUM-type item of value of 534 because the "8" comes after "5" in the ASCII or EBCDIC search order

Complex logical expressions

You can build a more complex expression by using either an *and* (&&) or *or* operator (||) to combine a pair of more elementary expressions. In addition, you can use the *not* operator (!), as described later.

If a logical expression is composed of elementary logical expressions that are combined by *or* operators, EGL evaluates the expression in accordance with the rules of precedence, but stops the evaluation if one of the elementary logical expressions resolves to true. Consider an example:

```
item01 = item02 || 3 in array03 || x = y
```

If item01 does not equal item02, evaluation proceeds. If the value 3 is in array03, however, the overall expression is proven to be true, and the last elementary logical expression (x = y) is not evaluated.

Similarly, if elementary logical expressions are combined by *and* operators, EGL stops the evaluation if one of the elementary logical expressions resolves to false. In the following example, evaluation stops as soon as item01 is found to be unequal to item02:

```
item01 = item02 && 3 in array03 && x = y
```

You may use paired parentheses in a logical expression for any of these purposes:

- To change the order of evaluation.
- To clarify your meaning.
- To make possible the use of the *not* operator (!), which resolves to a Boolean value (true or false) opposite to the value of a logical expression that immediately follows. The subsequent expression must be in parentheses.

Examples

In reviewing the examples that follow, assume that value1 contains "1", value2 contains "2", and so on:

```
/* = true */
value5 < value2 + value4
```

```
/* = false */
!(value1 is numeric)
```

```

/* = true when the generated output runs
   on Windows 2000, Windows NT,
   or z/OS UNIX System Services */
sysVar.systemType is WIN || sysVar.systemType is USS

/* = true */
(value6 < 5 || value2 + 3 >= value5) && value2 = 2

```

Related concepts

“Java access functions” on page 556

“Modified data tag and modified property” on page 152

Related tasks

“Syntax diagram” on page 506

Related reference

“case” on page 302

“Exception handling” on page 75

“Expressions” on page 357

“I/O error values” on page 418

“if, else” on page 332

“in” on page 416

“Items” on page 43

“Numeric expressions”

“Operators and precedence” on page 39

“Primitive types” on page 27

“String expressions” on page 365

“sysVar.eventKey” on page 622

“sysLib.getVAGSysType” on page 613

“sysVar.systemType” on page 628

“while” on page 356

Numeric expressions

A *numeric expression* resolves to a number, and you specify such an expression in various situations; for example, on the right side of an assignment statement. A numeric expression may be composed of any of these:

- A numeric operand, which is one of these:
 - An item (or a system variable) that contains a number. The item may be preceded with a sign.
 - A numeric literal, which may begin with a sign, but always has a series of digits and may include a single decimal point.
 - A function invocation that returns a number. (You cannot reference a function in a logical expression, however.)
- A numeric operand, followed by a numeric operator, followed by a second numeric operand. (The operands cannot reference a function.)
- A more complex expression formed by using a numeric operator to combine a pair of more elementary expressions. (The expression cannot reference a function.)

You may use paired parentheses in a numeric expression to change the order of evaluation or to clarify your meaning.

In reviewing the examples that follow, assume that intValue1 equals 1, intValue2 equals 2, and so on, and that each value has no decimal places:

```

/* = -8, with the parentheses overriding
    the usual precedence of * and + */
intValue2 * (intValue1 - 5)

/* = -2, with a unary minus as the last operator */
intValue2 + -4

/* = 1.4, if the expression is assigned to an
    item with at least one decimal place. */
intValue7 / intValue5

/* = 2, which is a remainder
    expressed as an integer value */
intValue7 % intValue5

```

For COBOL output, a numeric expression may give an unexpected result if an intermediate, calculated value has more than 30 or 31 digits; the exact number of digits depends on the ARITH compiler option.

For Java output, a numeric expression may give an unexpected result if an intermediate, calculated value requires more than 128 bits.

Related reference

“Items” on page 43
 “Expressions” on page 357
 “Logical expressions” on page 358
 “Operators and precedence” on page 39
 “Primitive types” on page 27
 “String expressions”

String expressions

A *string expression* resolves to a series of characters, and you specify such an expression in various situations; for example, on the right side of an assignment statement. The single operand that constitutes a string expression may be any of these:

- An item (or system variable) that contains a series of characters.
- A *character literal*, which is a series of characters delimited by double quote marks. Such a literal may include single-byte characters, double-byte characters, or a combination of the two. A series of character literals that are separated one from the next by a plus sign (+) is itself a character literal; for example, the following statement assigns *WebSphere* to *myString*:

```
myString = "Web" + "Sphere";
```
- A function invocation that returns a series of characters. (You cannot reference a function or system function in a logical expression, however.)

Although an item or return value can contain Unicode characters, a literal cannot.

Any character preceded with the escape character (\) is included in the string expression. In particular, you can use the escape character to include the following characters in a literal, item, or return value:

- A double quote mark (")
- A backslash (\)
- A carriage return, as indicated by \r
- A newline character, as indicated by \n

Examples are as follows:

```
myString = "He said, \"Escape while you can!\"";  
myString2 = "Is a backslash (\\) needed?";
```

An error occurs if a literal has no ending quote mark:

```
myString3 = "Escape is impossible\";
```

Related reference

“Items” on page 43

“Expressions” on page 357

“Logical expressions” on page 358

“Numeric expressions” on page 364

“Operators and precedence” on page 39

“Primitive types” on page 27

Form part

A *form part* is a unit of presentation. It describes the layout and characteristics of a set of fields that are shown to the user at one time.

You do not declare a form as if you were declaring a record or data item. To access a form part, your program must include a use declaration that refers to the related form group.

A form part is of one of two types, *text* or *print*:

- A form of type *text* defines a layout that is displayed in a 3270 screen or in a command window. With one exception, any text form can have both constant fields and variable fields, including variable fields that accept user input. The exception is a *help form*, which is solely for presenting constant information.
- A form of type *print* defines a layout that is sent to a printer. Any print form can have both constant and variable fields.

Form properties determine the size and position of the output on a screen or page and specify formatting characteristics of that output.

A given form can be displayed on one or more *devices*, each of which is an output peripheral or is the operational equivalent of an output peripheral:

- A *screen device* is a terminal, monitor, or terminal emulator. The output surface is a screen.
- A *print device* is a file that can be sent to a printer or is the printer itself. The output surface is a page.

Whether of type *text* or *print*, a form is further categorized as follows:

- A *fixed form* has a specific starting row and column in relation to the output surface of the device. You could assign a fixed print form, for example, to start at line 10, column 1 on a page.
- A *floating form* has no specific starting row or column; instead, the placement of a floating form is at the next unoccupied line in an output surface sub-area that you declare. The declared sub-area is called a *floating area*.

You might declare a floating area to be a rectangle that starts at line 10, extends through line 20, and is the maximum width of the output device. If you have a one-line floating form of the same width, you can construct a loop that acts as follows for each of 20 times:

1. Places data in the floating map
2. Writes the floating map to the next line in the floating area

One or more floating areas are declared in the `FormGroup` part, but only one can accept floating forms for a particular device. If you try to present a floating form in the absence of a floating area, the entire output surface is treated as a floating area.

- A *partial form* is smaller than the standard size of the output surface for a particular device. You can declare and position partial forms so that multiple forms are displayed at different horizontal positions. Although you can specify the starting and ending columns for a partial form, you cannot display forms that are next to one another.

Additional details are specific to the form type:

- Print forms
- Text forms

Related concepts

“Print forms” on page 368

“Text forms”

Related reference

“`FormGroup` part in EGL source format” on page 377

“Form part in EGL source format” on page 370

Text forms

Forms and their types are introduced in *Form part*. The current page outlines how to present text forms.

The **converse** statement is sufficient for giving the user access to a single, fixed text form. The logical flow of your program continues only after the user responds to the displayed form. You can also construct output from multiple forms, as in the following case:

- At the top of the output, a fixed form identifies a purchasing company and an order number
- In a subsequent floating area, a series of identically formatted floating forms identify each item of the company’s order
- At the bottom of the output, a fixed form indicates the number of screens needed to scroll through the list of items

Two steps are necessary:

1. First, you construct the order-and-item output by coding a series of **display** statements, each of which adds a form to a run-time buffer but does not present data to the screen. Each **display** statement operates on one of the following forms:
 - Top form
 - Floating form, as presented by a **display** statement that is invoked repeatedly in a loop
 - Bottom form
2. Next, the EGL run time presents all the buffered text forms to the output device in response to either of these situations:
 - The program runs a **converse** statement; or
 - The program ends.

In most cases, you present the last form of your screen output by coding a **converse** statement rather than a **display** statement.

The fixed forms each have an on-screen position, so the order in which you specify them, in relation to each other and in relation to the repeated display of floating forms, does not matter. The contents of the buffer are erased when output is sent to the screen.

If you overlay one text form with another, no error occurs, but the following statements apply:

- If a partial form overlays any lines in another fixed form, EGL replaces the existing form without clearing the rest of the output from the buffer. If you want to erase the existing output before displaying the new form, invoke the system function `sysLib.clearScreen` before issuing the **display** or **converse** statement for the new form.
- If you use a **display** or **converse** statement to place a floating map beyond the bottom of the floating area, all the floating forms in that floating area are erased, and the added form is placed on the first line of the same floating area.
- If a floating form overlays a fixed form, these statements apply- -
 - Only the fixed-form lines that are in the floating area are overwritten by the floating form
 - The result is unpredictable if a fixed-form line is overwritten by a floating-form line that includes a variable field

Whether you are presenting one form or many, the output destination is the screen device at which the user began the run unit.

Related concepts

“Form part” on page 366

Related reference

“Form part in EGL source format” on page 370

“FormGroup part in EGL source format” on page 377

“`sysLib.clearScreen`” on page 613

Print forms

Forms and their types are introduced in *Form part*. The current page outlines how to present print forms.

Print process

Printing is a two-step process:

- First, you code **print** statements, each of which adds a form to a run-time buffer
- Next, the EGL run time adds the symbols needed to start a new page, sends all the buffered forms to a print device, and erases the contents of the buffer. Those services are provided in response to any of the following circumstances:
 - The program runs a **close** statement on a print form that is destined for the same print device; or
 - The program is in segmented mode (as described in *Segmentation*) and runs a **converse** statement; or
 - The program was called by a non-EGL (and non-VisualAge Generator) program, and the called program ends; or
 - The main program in the run unit ends; or

- The system variable **sysVar.printerAssociation** (which assigns an output destination for print forms) is set in a COBOL program that is running on iSeries.

In the case of multiform output, the **print** statements must be invoked in the order in which you want to present the forms. Consider the following example:

- At the top of the output, a fixed form identifies a purchasing company and an order number
- In a subsequent floating area, a series of identically formatted floating forms identify each item of the company's order
- At the bottom of the output, a fixed form indicates the number of screens or pages needed to scroll through the list of items

You can achieve that output by submitting a series of **print** statements that each operate on a print form. Those statements reference the forms *in the following order*:

1. Top form
2. Floating form, as presented by a **print** statement that is invoked repeatedly in a loop
3. Bottom form

The symbols needed to start a new page are inserted in various circumstances, but you can cause the insertion by invoking the system function `sysLib.pageEject` before issuing a **print** statement.

Considerations for fixed forms

The following statements apply to fixed forms:

- If you issue a print statement for a fixed form that has a starting line greater than the current line, EGL inserts the symbols needed to advance the print device to the specified line. Similarly, if you issue a print statement for a fixed form that has a starting line less than the current line, EGL inserts the symbols needed to start a new page.
- If a fixed form overlays *some but not all* lines in another fixed form, EGL automatically inserts the symbols needed to start a new page and places the second fixed form on the new page.
- If a fixed form overlays *all* lines in another fixed form, EGL replaces the existing form without clearing the rest of the output from the buffer. To keep the existing output and place the new form on the next page, invoke the system function `sysLib.pageEject` before issuing the **print** statement for the new form.

Considerations for floating forms

The following mistakes can occur if you are using floating forms:

- You issue a **print** statement to place a floating form beyond the end of the floating area; or
- You issue a **print** statement that at least partially overlays a floating area with a fixed form, then issue a **print** statement to add a floating form to the floating area.

The result in either case is that EGL inserts the symbols needed to start a new page, and the floating form is placed on the first line of the floating area on the new page. If the page is similar to the order-and-item output described earlier, for example, the new page does not include the topmost fixed form.

Print destination

When EGL processes a **close** statement to present a print file, the output is sent to a printer or data set. You can specify the destination at any of three times:

- At test time (as described in *EGL debugger*)
- At generation time (as described in *Resource associations and file types*)
- At run time (as described in relation to the system variable `sysVar.printerAssociation`)

Related concepts

"EGL debugger" on page 107

"FormGroup part in EGL source format" on page 377

"Form part in EGL source format"

"Form part" on page 366

"Resource associations and file types" on page 157

"Segmentation in text applications" on page 151

Related reference

"sysLib.pageEject" on page 615

"sysVar.printerAssociation" on page 623

Form part in EGL source format

You declare a form part in an EGL file, which is described in *EGL source format*. If a form part is accessed by only one form group, it is recommended that the form part be embedded in the formGroup part. If a form part is accessed by multiple form groups, it is necessary to specify the form part at the top level of an EGL file.

An example of a text form is as follows:

```
Form myTextForm type textForm
{
    formsize= (24, 80),
    position= (1, 1),
    validationBypassKeys=(pf3, pf4),
    helpKey=pf1,
    helpForm=myHelpForm,
    msgField=myMsg,
    alias = form1
}

* { position=(1, 31), value="Sample Menu" } ;
* { position=(3, 18), value="Activity:" } ;
* { position=(3, 61), value="Command Code:" } ;

activity char(42)[5] { position=(4,18), protect=skip } ;

commandCode char(10)[5] { position=(4,61), protect=skip } ;

* { position=(10, 1), value="Response:" } ;
response char(228) { position=(10, 12), protect=skip } ;

* { position=(13, 1), value="Command:" } ;
myCommand char(70) { position=(13,10) } ;

* { position=(14, 1), value="Enter=Run F3=Exit" } ;

myMsg char(70) { position=(20,4) };

end
```

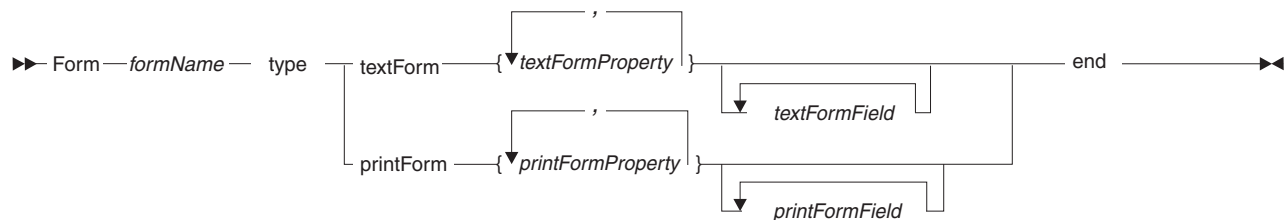
An example of a print form is as follows:

```

Form myPrintForm type printForm
{
    formsize= (48, 80),
    position= (1, 1),
    msgField=myMsg,
    alias = form2
}
* { position=(1, 10), value="Your ID: " } ;
ID char(70) { position=(1, 30) };
myMsg char(70) { position=(20, 4) };
end

```

The diagram of a form part is as follows:



Form *formName* ... end

Identifies the part as a form and specifies the part name. For the rules of naming, see Naming conventions.

textForm

Indicates that the form is a text form.

textFormProperty

A text-form property. For details, see *Text form*.

textFormField

A text-form field. For details, see *Form fields*.

printForm

Indicates that the form is a print form.

printFormProperty

A print-form property. For details, see *Print form*.

printFormField

A print-form field. For details, see *Form fields*.

Text-form properties

The text-form properties are as follows:

formsize = (*rows*, *columns*)

Number of rows and columns in the online presentation area. This property is required.

The column value is equivalent to the number of single-byte characters that can be displayed across the presentation area.

position = (*row*, *column*)

Row and column at which the form is displayed in the presentation area. If you omit this property, the form is a floating form and is displayed in the floating area, at the next free line where the entire form can fit in the floating area.

validationBypassKeys = *bypassKeyValue*

Identifies one or more user keystrokes that causes the EGL run time to skip

input-field validations. This property is useful for reserving a keystroke that ends the program quickly. The *bypassKeyVal* option is as follows:

pf*n*

The name of an F or PF key, including a number between 1 and 24, inclusive

Note: Function keys on a PC keyboard are often *F* keys such as F1, but EGL uses the IBM *PF* terminology so that (for example) F1 is called PF1.

If you wish to specify more than one key value, delimit the set of values with parentheses and separate each value from the next with a comma, as in the following example:

```
validationBypassKeys = (pf3, pf4)
```

helpKey = helpKeyVal

Identifies a user keystroke that causes the EGL run time to present a help form to the user. The *helpKeyVal* option is as follows:

pf*n*

The name of an F or PF key, including a number between 1 and 24, inclusive

Note: Function keys on a PC keyboard are often *F* keys such as F1, but EGL uses the IBM *PF* terminology so that (for example) F1 is called PF1.

helpForm = formName

Name of the help form that is specific to the text form.

msgField = fieldName

Name of the text-form field that displays a message in response to a validation error or in response to the running of sysLib.displayMsgNum.

alias = alias

An alias of 8 characters or less, for use by the EGL run time. An alias is necessary in a COBOL environment if the form name is longer than 8 characters.

Print-form properties

The print-form properties are as follows:

formsize = (rows, columns)

Number of rows and columns in the online presentation area. This property is required.

The column value is equivalent to the number of single-byte characters that can be displayed across the presentation area.

position = (row, column)

Row and column at which the form is displayed in the presentation area. If you omit this property, the form is a floating form and is displayed in the floating area, at the next free line where the entire form can fit in the floating area.

addSpaceForSOSI = yes, addSpaceForSOSI = no

Directs report production in COBOL environments. If you set the property to *no*, the EGL run time prints the line as is, including any shift-out/shift-in

(SO/SI) characters that are in fields of type MBCHAR. If you set the property to *yes* (the default), the EGL run time puts a space in place of each SO/SI character.

To cause the printed form to use the same columns as in the form definition, specify *no* for printers that print a space for each SO or SI character and specify *yes* for printers that strip SO or SI characters from the print line.

msgField = *fieldName*

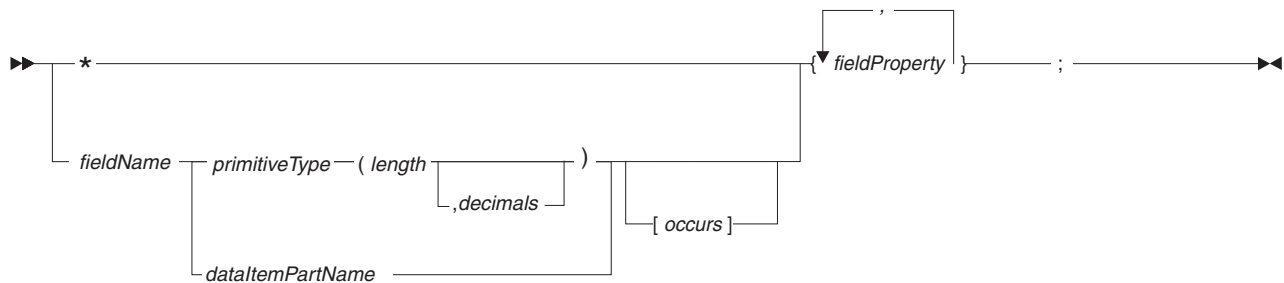
Name of the text-form field that displays a message in response to the running of sysLib.displayMsgNum.

alias = *alias*

An alias of 8 characters or less, for use by the EGL run time. An alias is necessary in a COBOL environment if the form name is longer than 8 characters.

Form fields

The diagram of a form field is as follows:



- * Indicates that the field is a constant field. It has no name but has a constant value, which is specified in the field-specific **value** property. Statements in your code cannot access the value in a constant field.

fieldProperty

A field property. For details, see *Field properties*.

fieldName

Specifies the name of the field. For rules, see *Naming conventions*.

Your code can access the value of a named field, which is also called a *variable field*.

If a text form contains a variable field that starts on one line and ends on another, the text form can be displayed only on screens where the screen width equals the width of the form.

occurs

The number of elements in a field array. Only one-dimensional arrays are supported. For further details, see *For field arrays*.

primitiveType

The primitive type assigned to the field. This specification affects the maximum length; but any numeric field is generated as type NUM.

Forms that contain fields of type DBCHAR can only be used on systems and devices that support double-byte character sets. Similarly, forms that contain fields of type MBCHAR can only be used on systems and devices that support multiple-byte character sets.

The primitive type UNICODE is not supported for text or print forms.

length

The field's length, which is an integer that represents the maximum number of characters or digits that can be placed in the field.

decimals

For a numeric type (BIN, DECIMAL, NUM, NUMC, or PACF), you may specify *decimals*, which is an integer that represents the number of places after the decimal point. The maximum number of decimal positions is the smaller of two numbers: 18 or the number of digits declared as *length*. The decimal point is not stored with the data.

dataItemPartName

The name of a dataItem part that is a model of format for the field, as described in *intypeDef*. The dataItem part must be visible to the form part, as described in *References to parts*.

Field properties

This section describes properties that are useful only in form fields. Additional properties are also of interest as described in *Field-presentation properties*, *Formatting properties*, and *Validation properties*.

For any field: The following properties are useful for any field on a form:

position = (*row*, *column*)

Row and column of the attribute byte that precedes the field. This property is required.

value = *stringLiteral*

A character string that is displayed in the field.

This property can be specified for any item; for example, in a dataItem part declaration.

Note: If VisualAge Generator compatibility is in effect and you set the text-form property **value**, the content of that property is available in the program only after the user has returned the form. For this reason, the value that you set in the program does not need to be valid for the item in the program.

fieldLen = *lengthInBytes*

Field length; the number of single-byte characters that can be displayed in the field. This value does not include the preceding attribute byte.

The value of **fieldLen** for numeric fields must be great enough to display the largest number that can be held in the field, plus (if the number has decimal places) a decimal point. The value of **fieldLen** for a field of type CHAR, DBCHAR, MBCHAR, or UNICODE must be large enough to account for the double-byte characters, as well as any shift-in/shift-out characters.

The default **fieldLen** is the number of bytes needed to display the largest number possible for the primitive type, including all formatting characters.

For variable text fields: The following properties are useful for variable text fields:

cursor = *no*, **cursor** = *yes*

Indicates whether the on-screen cursor is at the beginning of the field when the form is first displayed. Only one field in the form can have the cursor property set to *yes*. The default is *no*.

detectable = no, detectable = yes

Specifies whether the field's modified data tag is set when the field is selected by a light pen or (for emulator sessions) by a cursor click.

The **detectable** property is available only for COBOL programs and only for text-form fields whose **intensity** property is other than *invisible*.

The initial character in the field content (as specified in the **value** property) must be a *designator character*, which indicates what action is taken when the user clicks on the field. The most common designator characters are as follows:

- &** Causes an *immediate detect*, which means that clicking the field at run time is equivalent to modifying the field and pressing the ENTER key.
- ?** Causes a *delayed detect*, which means that clicking the field at run time is equivalent to modifying the field, but that the program receives the form information only when the user presses the ENTER key or clicks a field that is configured for an immediate detect.

To prevent the user from changing the designator character in a variable field, set the **protect** property to *yes* or *skip*.

modified = no, modified = yes

Indicates whether the program will consider the field to have been modified, regardless of whether the user changed the value. For details, see *Modified data tag and modified property*.

The default is *no*.

protect = no, protect = skip, protect = yes

Specifies whether the user can access the field. Valid values are as follows:

no (the default for variable fields)

Sets the field so that the user can overwrite the value in it.

skip (the default for constant fields)

Sets the field so that the user cannot overwrite the value in it. In addition, the cursor skips the field in either of these cases:

- The user is working on the previous field in the tab order and either presses **Tab** or fills that previous field with content; or
- The user is working on the next field in the tab order and presses **Shift Tab**.

yes

Sets the field so that the user cannot overwrite the value in it.

validationOrder = integer

Indicates the field's position in the validation order. The default order in which the fields are validated is the order of the fields on screen, left to right, top to bottom.

For field arrays: One-dimensional arrays are supported on text and print forms. In an array declaration, the value of the **occurs** property is greater than 1, as in this example:

```
myArray char(1)[3];
```

Array elements are positioned in relation to the placement specified for the first element in the array. The default behavior is to position the elements vertically on consecutive rows.

Use the following properties to vary the default behavior:

columns = *numberOfElements*

Number of array elements in each row. The default is 1.

linesBetweenRows = *numberOfLines*

Number of lines between each row that contains array elements. The default is 0.

spacesBetweenColumns = *numberOfSpaces*

Number of spaces between each array element. The default is 1.

indexOrientation = **down**, **indexOrientation** = **across**

Specifies how the program references the elements of an array:

- If you set **indexOrientation** to *down*, elements are numbered from top to bottom, then left to right, so that the elements in a given column are numbered sequentially. The value of **indexOrientation** is *down* by default.
- If you set **indexOrientation** to *across*, elements are numbered from left to right, then top to bottom, so that the elements in a given row are numbered sequentially.

You can override properties for an array element. In the following field declaration, for example, the **cursor** property is overridden in the second element of `myArray`:

```
myArray char(10)[5]
{position=(4,61), protect=skip, myArray[2] { cursor } };
```

Related concepts

"Modified data tag and modified property" on page 152

"Overview of EGL properties and overrides" on page 40

"Print forms" on page 368

"References to parts" on page 16

"Text forms" on page 367

"Typedef" on page 20

Related reference

"Field-presentation properties" on page 45

"Formatting properties" on page 47

"Naming conventions" on page 468

"NUM" on page 33

"Primitive types" on page 27

"sysLib.displayMsgNum" on page 527

"Validation properties" on page 59

FormGroup part

An EGL FormGroup part serves two purposes:

- Defines a collection of text and print forms. (Forms that are unique to the part are defined within the part or are included by way of a Use declaration. Forms that are common to several FormGroup parts are included by way of a Use declaration.)
- Defines zero to many *floating areas*, as described in *Form part*

You do not declare a form group as if you were declaring a record or data item. Instead, your program accesses a FormGroup part (and the related forms) only if the following statements apply:

- The location of the FormGroup part is accessible to the program, as described in *References to parts*
- A Use declaration in the program references the FormGroup part

A program can include no more than two formGroup parts; and if two are specified, one must be a *help group*. A help group contains one or more *help forms*, which are read-only forms that give information in response to a user keystroke.

Forms are available at run time only if you generate the FormGroup. The generated output for Java is a class for the FormGroup part and a class for each Form part. The generated output for a COBOL program is as follows:

- Text forms are generated into an object module
- Print forms are generated into a printing-services program

At preparation time, each of those entities is processed into a separate run-time load module. The EGL run time handles the interaction of your generated program and the form-specific code.

Form parts cannot be generated separately.

Related concepts

"Form part" on page 366

"References to parts" on page 16

Related reference

"Use declaration" on page 631

FormGroup part in EGL source format

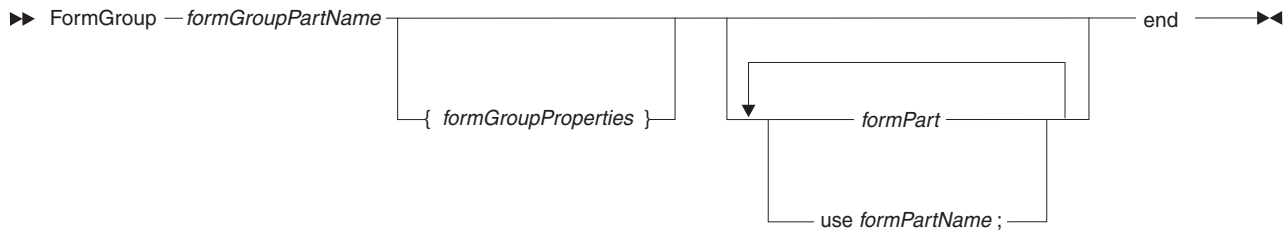
You declare a formGroup part in an EGL file, which is described in *EGL source format*. This part is a primary part, which means that it must be at the top level of the file and must have the same name as the file.

A program can only use forms that are associated with a form group referenced by the program's use declaration.

An example of a formGroup part is as follows:

```
FormGroup myFormGroup
{
  validationBypassKeys = pf3,
  helpKey = pf1,
  pfKeyEquate = yes,
  screenFloatingArea
  {
    screenSize = (24,80),
    topMargin = 0,
    bottomMargin = 0,
    leftMargin = 0,
    rightMargin = 0
  },
  printFloatingArea
  {
    pageSize = (60,80),
    topMargin = 3,
    bottomMargin = 3,
    leftMargin = 5,
    rightMargin = 5
  }
}
use myForm01;
use myForm02;
end
```

The diagram of a `formGroup` part is as follows:



FormGroup *formGroupName* ... **end**

Identifies the part as a form group and specifies the part name. For the rules of naming, see *Naming conventions*.

formGroupProperties

A series of properties, each separated from the next by a comma. Each property is described later.

formPart

A text or print form, as described in *Form part in EGL source format*.

use *formPartName*

A use declaration that provides access to a form that is not embedded in the form group.

The form group properties are as follows:

alias

A string that is incorporated into the names of generated output. If you do not specify an alias, the `formGroup`-part name is used instead.

validationBypassKeys = *bypassKeyValue*

Identifies one or more user keystrokes that causes the EGL run time to skip input-field validations. This property is useful for reserving a keystroke that ends the program quickly. Each *bypassKeyValue* option is as follows:

pf*n*

The name of an F or PF key, including a number between 1 and 24, inclusive.

Note: Function keys on a PC keyboard are often *F* keys such as F1, but EGL uses the IBM *PF* terminology so that (for example) F1 is called PF1.

If you wish to specify more than one key value, delimit the set of values with parentheses and separate each value from the next with a comma, as in the following example:

```
validationBypassKeys = (pf3, pf4)
```

helpKey = *helpKeyValue*

Identifies a user keystroke that causes the EGL run time to present a help form to the user. The *helpKeyValue* option is as follows:

pf*n*

The name of an f or pf key, including a number between 1 and 24, inclusive.

Note: Function keys on a PC keyboard are often *f* keys such as f1, but EGL uses the IBM *pf* terminology so that (for example) f1 is called pf1.

pfKeyEquate = yes, pfKeyEquate = no

Specifies whether the keystroke that is registered when the user presses a high-numbered function key (PF13 through PF24) is the same as the keystroke that is registered when the user presses a function key that is lower by 12. For details, see *pfKeyEquate*.

screenFloatingArea { properties }

Defines the floating area used for output to a screen. For an overview of floating areas, see *Form part*. For property details, see the next section.

printFloatingArea { properties }

Defines the floating area used for printable output. For an overview of floating areas, see *Form part*. For property details, see *Properties of a print floating area*.

Properties of a screen floating area

The set of properties after **screenFloatingArea** is delimited by braces ({ }), and each property is separated from the next by a comma. The properties are as follows:

screenSize = (rows, columns)

Number of rows and columns in the online presentation area, including any lines or columns used as margins. The default is as follows:

screenSize=(24,80)

topMargin= rows

Number of lines left blank at the top of the presentation area. The default is 0.

bottomMargin= rows

Number of lines left blank at the bottom of the presentation area. The default is 0.

leftMargin= columns

Number of columns left blank at the left of the presentation area. The default is 0.

rightMargin= columns

Number of columns left blank at the right of the presentation area. The default is 0.

Properties of a print floating area

The set of properties after **printFloatingArea** is delimited by braces ({ }), and each property is separated from the next by a comma. The properties are as follows:

pageSize = (rows, columns)

Number of rows and columns in the printable presentation area, including any lines or columns used as margins. This property is required if you specify a print floating area.

deviceType = singleByte, deviceType = doubleByte

Specifies whether the floating-area declaration is for a printer that supports single-byte output (as is the default) or double-byte output. Specify **doubleByte** if any of the forms include items of type DBCHAR or MBCHAR.

topMargin = rows

Number of lines left blank at the top of the presentation area. The default is 0.

bottomMargin = rows

Number of lines left blank at the bottom of the presentation area. The default is 0.

leftMargin = columns

Number of columns left blank at the left of the presentation area. The default is 0.

rightMargin = *columns*

Number of columns left blank at the right of the presentation area. The default is 0.

Related concepts

"EGL projects, packages, and files" on page 7

"FormGroup part" on page 376

"Form part" on page 366

Related reference

"EGL source format" on page 292

"Form part in EGL source format" on page 370

"Naming conventions" on page 468

"pfKeyEquate"

"Use declaration" on page 631

pfKeyEquate

When you declare a form group that references a text form, the property *pfKeyEquate* specifies whether the keystroke that is registered when the user presses a high-numbered function key (PF13 through PF24) is the same as the keystroke that is registered when the user presses a function key that is lower by 12.

If you accept the default value of *yes* for *pfKeyEquate*, your logical expressions are able to reference only 12 of the function keys because (for example) PF2 is the same as PF14.

Note: Function keys on a PC keyboard are often *F* keys such as F1, but EGL uses the IBM *PF* terminology so that (for example) F1 is called PF1.

Related concepts

"FormGroup part" on page 376

Related reference

"FormGroup part in EGL source format" on page 377

Function part

A *function part* is a logical unit that either contains the first code in the program or is invoked from another function. The function that contains the first code in the program is called *main*.

The function part can include the following properties:

- A *return value*, which describes the data that the function part returns to the caller
- A set of *parameters*, each of which references memory that is allocated and passed by another logic part
- A set of other *variables*, each of which allocates other memory that is local to the function
- EGL statements

The function *main* is unusual in that it cannot return a value or include parameters, and it must be declared inside a program part.

Related concepts

"Parts" on page 11

"Program part" on page 477

"References to variables and constants" on page 34

"SQL support" on page 171

Related reference

"Data initialization" on page 277

"EGL source format" on page 292

"Function invocations" on page 316

"EGL statements" on page 70

Function part in EGL source format

You can declare functions in an EGL file, as described in *EGL source format*.

The following example shows a program part with two embedded functions, along with a stand-alone function and a stand-alone record part:

```
Program myProgram(employeeNum INT)
{includeReferencedFunctions = yes}

// program-global variable
employees record_ws;
employeeName char(20);

// a required embedded function
Function main()

// initialize employee names
recd_init();

// get the correct employee name
// based on the employeeNum passed
employeeName = getEmployeeName(employeeNum);
end

// another embedded function
Function recd_init()
employees.name[1] = "Employee 1";
employees.name[2] = "Employee 2";
end
end

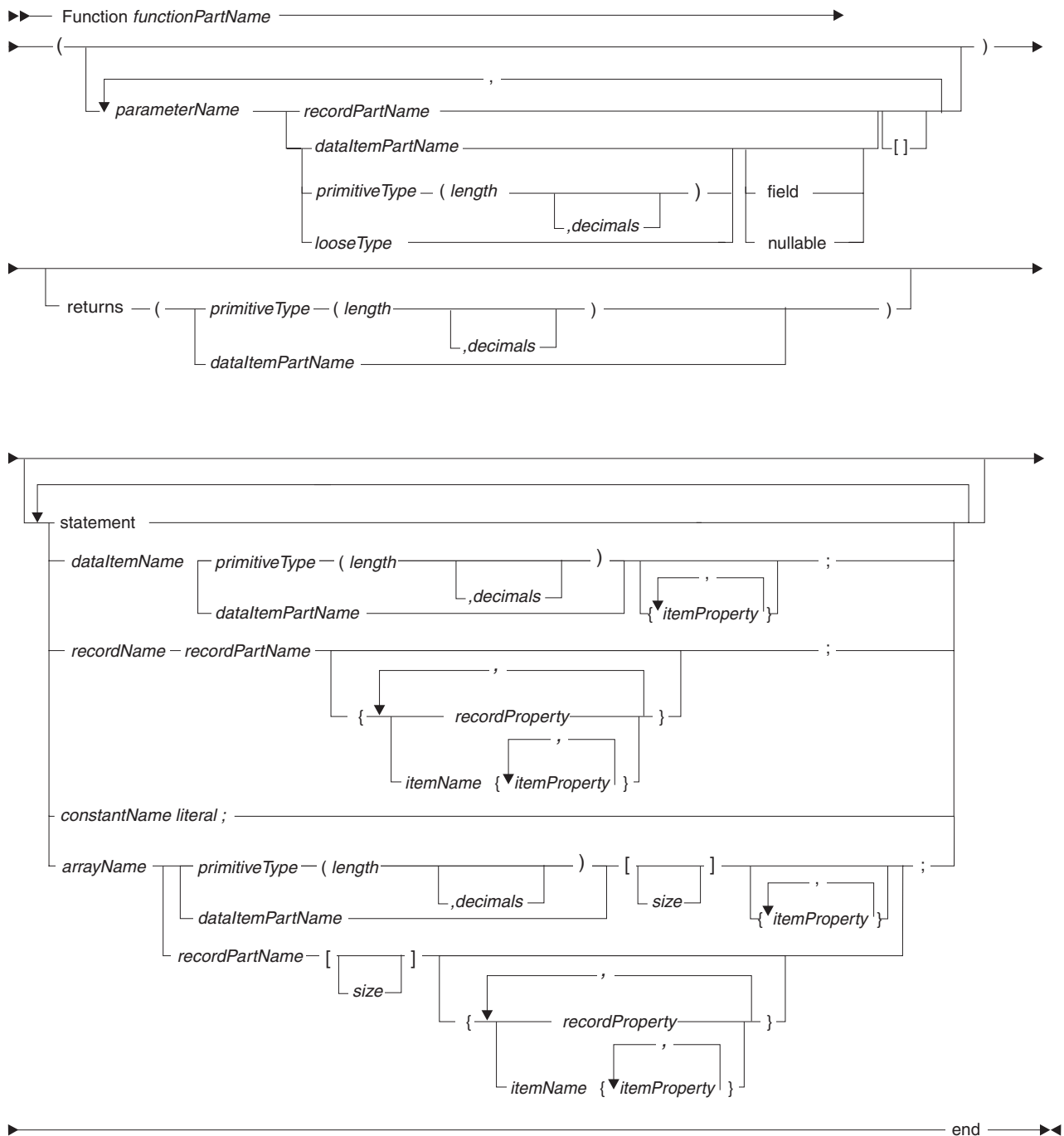
// stand-alone function
Function getEmployeeName(employeeNum INT) returns (CHAR(20))

// local variable
index BIN(4);
index = syslib.size(employees.name);
if (employeeNum > index)
return("Error");
else
return(employees.name[employeeNum]);
end

end

// record part that acts as a typeDef for employees
Record record_ws type basicRecord
10 name CHAR(20)[2];
end
```

The syntax diagram for a function part is as follows:



Function *functionPartName* ... end

Identifies the part as a function and specifies the part name. For the rules of naming, see *Naming conventions*.

parameterName

Specifies the name of a parameter, which may be a record or data item; or an array of records or data items. For rules, see *Naming conventions*.

If a given parameter receives data from a variable (not from a constant or literal), changes to the parameter change the variable in the invoking function.

A parameter that ends with brackets ([]) is a dynamic array, and the other specifications declare aspects of each element of that array.

recordPartName (**after** *parameterName*)

A record part that is visible to the function and that is acting as a typedef (a model of format) for a parameter. For details on what parts are visible, see *References to parts*.

The following statements apply to input or output (I/O) against the specified record:

- A record passed from another function in the same program includes record state such as the I/O error value *endOfFile*, but only if the record is of the same record type as the parameter. Similarly, any change in the record state is returned to the caller, so if you perform I/O against a record parameter, any tests on that record can occur in the current function, in the caller, or in a function that is called by the current function.

Library functions do not receive record state.

- Any I/O operation performed against the record uses the record properties specified for the parameter, not the record properties specified for the argument.
- For records of type *indexedRecord*, *mqRecord*, *relativeRecord*, or *serialRecord*, the file or message queue associated with the record declaration is treated as a run-unit resource rather than a program resource. Local record declarations share the same file (or queue) whenever the record property **fileName** (or **queueName**) has the same value. Only one physical file at a time can be associated with a file or queue name no matter how many records are associated with the file or queue in the run unit, and EGL enforces this rule by closing and reopening files as appropriate.

dataItemPartName (**after** *parameterName*)

A dataItem part that is visible to the function and that is acting as a typedef (a model of format) for a parameter.

primitiveType (**after** *parameterName*)

The parameter's primitive type.

length (**after** *parameterName*)

The parameter's length, which is an integer that represents the number of characters or digits in the memory area referenced by *parameterName*.

decimals (**after** *parameterName*)

For a numeric type (BIN, DECIMAL, NUM, NUMC, or PACF), you may specify *decimals*, which is an integer that represents the number of places after the decimal point. The maximum number of decimal positions is the smaller of two numbers: 18 or the number of digits declared as *length*. The decimal point is not stored with the data.

looseType

A loose type is a special kind of primitive type that is used only for function parameters. You use this type if you wish the parameter to accept a range of argument lengths. The benefit is that you can invoke the function repeatedly and can pass an argument of a different length each time.

Valid values are as follows:

- CHAR
- DBCHAR

- HEX
- MBCHAR
- NUMBER
- UNICODE

If you wish the parameter to accept a number of any primitive type and length, specify NUMBER as the loose type. In this case, the number passed to the parameter must not have any decimal places.

If you wish the parameter to accept a string of a particular primitive type but any length, specify CHAR, DBCHAR, MBCHAR, HEX, or UNICODE as the loose type and make sure that the argument is of the corresponding primitive type.

Loose types are not available in functions that are declared in *libraries*.

For details on primitive types, see Primitive types.

field

Indicates that the parameter has form-field attributes such as *blanks* or *numeric*. Those attributes can be tested in a logical expression.

nullable

Indicates the following characteristics of the parameter:

- The parameter can be set to null
- The parameter has access to the state information necessary to test for truncation or null in a logical expression

The **nullable** attribute is meaningful only if the argument passed to the parameter is a structure item in an SQL record. The following rules apply:

- The parameter can be set to null and tested for null only if the item property **isNullable** is set to *yes*.
- In a Java program, the ability to test for truncation is available regardless of the value of **isNullable**. In a COBOL program, however, the ability to test for truncation is available only if **isNullable** is set to *yes*.

returns

Describes the data that the function returns to the invoker. The characteristics of the return value must match the characteristics of the variable that receives the value.

dataItemPartName **(after returns)**

A dataItem part that is visible to the function and that is acting as a typedef (a model of format) for the return value.

primitiveType **(after returns)**

The primitive type of the data returned to the invoker.

length **(after returns)**

The length of the data returned to the invoker. The length is an integer that represents the number of characters or digits in the returned value.

decimals **(after returns)**

For a numeric type (BIN, DECIMAL, NUM, NUMC, or PACF), you may specify *decimals*, which is an integer that represents the number of places after the decimal point. The maximum number of decimal positions is the smaller of two numbers: 18 or the number of digits declared as *length*. The decimal point is not stored with the data.

statement

An EGL statement, as described in *EGL statements*.

variableName

Specifies the name of a local variable other than a parameter. For details on variable usage, see *References to variables and constants*. For other rules, see *Naming conventions*.

dataItemPartName (**after variableName**)

A dataItem part that is visible to the function and that acts as a typedef for the variable.

recordPartName (**after variableName**)

A record part that is visible to the function and that acts as a typedef for the variable.

primitiveType (**after variableName**)

The primitive type, if no typedef is in use.

length (**after variableName**)

The variable's length, which is an integer that represents the maximum number of characters or digits in the memory area referenced by *variableName*.

decimals (**after variableName**)

For a numeric type (BIN, DECIMAL, NUM, NUMC, or PACF), you may specify *decimals*, which is an integer that represents the number of places after the decimal point. The maximum number of decimal positions is the smaller of two numbers: 18 or the number of digits declared as *length*. The decimal point is not stored with the data.

constantName *literal*

Identifies a constant and the literal that is held by the constant. The constant is local to the function.

arrayName

Name of a dynamic or static array of records or data items. If you use this option, the other symbols to the right (*dataItemPartName*, *primitiveType*, and so on) refer to each element of the array.

size

Number of elements in a static array.

Related concepts

"EGL projects, packages, and files" on page 7

"Function part" on page 380

"Import" on page 26

"Library part" on page 433

"Parts" on page 11

"References to parts" on page 16

"References to variables and constants" on page 34

"Typedef" on page 20

Related tasks

"Syntax diagram" on page 506

Related reference

"Basic record part in EGL source format" on page 491

"EGL source format" on page 292

"EGL statements" on page 70

“Indexed record part in EGL source format” on page 492
 “I/O error values” on page 418
 “Logical expressions” on page 358
 “MQ record part in EGL source format” on page 493
 “Naming conventions” on page 468
 “Primitive types” on page 27
 “Program part in EGL source format” on page 478
 “Relative record part in EGL source format” on page 495
 “Serial record part in EGL source format” on page 497
 “SQL record part in EGL source format” on page 498

Generated output

The next table lists the generated output. For details on the names given to each kind of output file, see *Generated output (reference)*.

Output type	Purpose	Generation type
Build plan	Lists the code-preparation steps that will occur on the target platform	All
COBOL program	Runs as a COBOL program on iSeries	COBOL
Enterprise JavaBean (EJB) session bean	Runs in an EJB container	Java wrapper
Java program and related classes	Runs either outside of J2EE or in the context of a J2EE client application, web application, or EJB container	Java
Java wrapper	Invokes an EGL-generated program from non-EGL-generated Java code	Java wrapper
J2EE environment file	Provides entries for insertion into the Java deployment descriptor	Java
Library (generated output)	Provides functions and values for use by other generated output	Java
Linkage properties file	Guides how calls are made from generated Java code, but only if decisions are final at deployment time rather than generation time	Java or Java wrapper
PageHandler part	Creates output that controls a user’s run-time interaction with a Web page	Java
Program properties file	Contains Java run-time properties in a format that is accessible only when you are debugging a Java program in a non-J2EE Java project	Java
Results file	Gives status information on the code-preparation steps that occurred on the target platform	All

Related concepts

“Introduction to EGL” on page 1
 “Java program, page handler, and library” on page 394
 “Java run-time properties” on page 421
 “Run-time configurations” on page 2

Related tasks

“Building EGL output” on page 122

Related reference

“Generated output (reference)”

Generated output (reference)

The output of EGL generation largely depends on whether you are generating COBOL, Java, or a Java wrapper. The next table shows the file names of generated output that do not come from a specific EGL part.

Output type	File name
“Build plan” on page 392	<i>alias</i> BuildPlan.xml
“Enterprise JavaBean (EJB) session bean” on page 393	<i>alias</i> EJBHome.java for the home interface, <i>alias</i> EJB.java for the remote bean interface, and <i>alias</i> EJBBean.java for the bean implementation
“J2EE environment file” on page 394	<i>alias</i> -env.txt
“Program properties file” on page 404	<i>alias</i> .properties
“Results file” on page 405	<i>alias</i> _Results_timeStamp.xml

alias

The alias, if any, that is specified in the program part. If the alias is not specified, the name of the program part is used but is truncated (if necessary) to the maximum number of characters allowed in the run-time environment.

Other characteristics of *alias* are determined by the kind of output:

- If you are generating a COBOL program and related output, all letters of *alias* are uppercase
- If you are generating a Java program, the case of each letter in *alias* is taken without change from the source code
- If you are generating a Java wrapper, the rules for naming the wrapper and EJB session bean are as follows:
 - The first letter in *alias* is uppercase
 - Every subsequent letter is lower case, with this exception: any underscore or hyphen is eliminated, and the subsequent letter is uppercase

timeStamp

The date and time when the file was created. The format reflects the settings on the development operating system.

For details on file names, see the appropriate reference topic:

- “Output of COBOL generation” on page 388
- “Output of Java program generation” on page 388
- “Output of Java wrapper generation” on page 389

Related concepts

“Build plan” on page 392

“Enterprise JavaBean (EJB) session bean” on page 393

“Generated output” on page 386

“Generation” on page 123

“J2EE environment file” on page 394

“Program properties file” on page 404

“Results file” on page 405

Related reference

“Output of COBOL generation”
“Output of Java program generation”
“Output of Java wrapper generation” on page 389

Output of COBOL generation

The COBOL generation outputs for iSeries include a COBOL program, a results file (which includes status information on EGL generation and preparation), and a build plan (if the build descriptor option **buildPlan** is set to *yes*). For additional details, see *the EGL Server Guide for iSeries*, which is available in the help system.

Related concepts

“Build plan” on page 392
“COBOL program” on page 393
“Development process” on page 1

“Program part” on page 477
“Results file” on page 405
“Source of additional information on EGL” on page 5

Related tasks

“Generating for COBOL” on page 132

Related reference

“buildPlan” on page 243
“callLink element” on page 443
“Generated output (reference)” on page 387
“Output of Java program generation”
“Output of Java wrapper generation” on page 389

Output of Java program generation

The output of Java server program generation is as follows:

- A build plan, if the build descriptor option **genProject** is omitted
- Java source code (see *Java program, page handler, and library*)
- Related objects needed to prepare and run your program (see *Java program, page handler, and library*)
- J2EE environment file
- Program properties file
- A results file, if **genProject** is omitted

You can use the EGL generator to generate entire Java programs. Programs and records are generated as separate Java classes. Functions are generated as methods in the program. Data items and structure items are generated as fields of the record or program class to which they belong.

The following table shows the names of the various types of generated Java parts:

Names of generated Java parts

Part type and name	What is generated
Program named P	A class named P in P.java

Names of generated Java parts

Part type and name	What is generated
Function named F in program P	A method of the P class called \$funcF in P.java
Record named R	A class named EzeR in EzeR.java
Basic record named R, parameter to Function F	A class named Eze\$paramR in Eze\$paramR.java
Linkage options part named L	Linkage properties file named L.properties
Library named Lib	A class named Lib in Lib.java
DataTable named DT	A class named EzeDT in EzeDT.java
Form named F	A class named EzeF in EzeF.java
FormGroup named FG	A class named FG in FG.java

- For the indicated part types, it is possible that two or more parts may exist with the same name. In that event the name of the second one will have an additional suffix, \$v2. The name of the third will have a \$v3 suffix, the fourth will have \$v4, etc.

If the naming format would cause two names to be identical, EGL adds a suffix to each file generated after the first. The suffix is as follows:

\$vn

where

n Is an integer assigned in sequential order, beginning with 2.

Related concepts

"Build plan" on page 392

"J2EE environment file" on page 394

"Java program, page handler, and library" on page 394

"Linkage properties file" on page 404

"Program properties file" on page 404

"Results file" on page 405

Related reference

"callLink element" on page 443

Output of Java wrapper generation

The output of Java wrapper generation is as follows:

- A build plan, if the build descriptor option **genProject** is omitted
- JavaBeans™ for wrapping calls to a Java server program (see *Java wrapper*)
- EJB session beans under certain circumstances; for details, see the explanation of the callLink element in *Linkage options part*
- A results file, if **genProject** is omitted

You can use the generated beans to wrap calls to server programs from non-EGL Java classes such as servlets, EJBs, or Java applications. The following types of classes are generated:

- Beans for servers
- Beans for record parameters
- Beans for record array rows

The following table shows the names of the various types of generated Java wrapper parts:

Names of generated Java wrapper parts

Part type and name	What is generated
Program named P	A class named PWrapper in PWrapper.java
Record named R used as a parameter	A class named R in R.java
Substructured area S in record R used as a parameter	A class named R.S in R.java
Linkage options part named L	Linkage properties file named L.properties

1. For the indicated part types, it is possible that two or more parts may exist with the same name. In that event the name of the second one will have an additional suffix, \$v2. The name of the third will have a \$v3 suffix, the fourth will have \$v4, etc.

When you request that a program part be generated as a Java wrapper, EGL produces Java class for each of the following executables:

- The program part
- Each record that is declared as a program parameter
- A session bean, if you specify a linkage options part and a the **callLink** element for the generated program has a link type of **ejbCall**

In addition, the class generated for each record includes an inner class (or a class within an inner class) for each structure item that has these characteristics:

- Is in the internal structure of one of those records
- Has at least one subordinate structure item; in other words, is substructured
- Is an array; in this case, a substructured array

Each generated class is stored in a file. The EGL generator creates names used in Java wrappers as follows:

- The name is converted to lowercase.
- Each hyphen or minus (-) or underscore (_) is deleted. A character that follows a hyphen or underscore is changed to uppercase.
- When the name is used as a class name or within a method name, the first character is translated back to uppercase.

If one of the parameters to the program is a record, EGL generates a wrapper class for that variable as well. If program Prog has a record parameter with a typeDef named Rec, the wrapper class for the parameter will be called Rec. If the typeDef of a parameter has the same name as the program, the wrapper class for the parameter will have a "Record" suffix.

The generator also produces a wrapper if a record parameter has an array item and the item has other items under it. This substructured array wrapper becomes an inner class of the record wrapper. In most cases, a substructured array item called AItem in Rec will be wrapped by a class called Rec.AItem. The record may contain two substructured array items with the same name, in which case the item wrappers are named by using the qualified names of the items. If the qualified name of the first AItem is Top1.AItem and the qualified name of the second is Top2.Middle2.AItem, the classes will be named Rec.Top1\$AItem and

Rec.Top2\$_middle2\$_aItem. If the name of a substructured array is the same as the name of the program, the wrapper class for substructured array will have a Structure suffix.

Methods to set and get the value of low-level items are generated into each record wrapper and substructured array wrapper. If two low-level items in the record or substructured array have the same name, the generator uses the qualified-name scheme described in the previous paragraph.

Additional methods are generated into wrappers for SQL record variables. For each item in the record variable, the generator creates methods to get and set its null indicator value and methods to get and set its SQL length indicator.

You can use the Javadoc tool to build a *classname.html* file once the the class has been compiled. The HTML file describes the public interfaces for the class. If you use HTML files created by Javadoc, be sure that it is an EGL Java wrapper. HTML files generated from a VisualAge Generator Java wrapper are different from those generated from an EGL Java wrapper.

Example: An example of a record part with a substructured array is as follows:

```
Record myRecord type basicRecord
  10 MyTopStructure[3];
    15 MyStructureItem01 CHAR(3);
    15 MyStructureItem02 CHAR(3);
end
```

In relation to the program part, the output file is named as follows:

aliasWrapper.java

where

alias

Is the alias name, if any, that is specified in the program part. If the external name is not specified, the name of the program part is used.

In relation to each record declared as a program parameter, the output file is named as follows:

recordName.java

where

recordName

Is the name of the record part

In relation to a substructured array, the name and position of the inner class depends on whether the array name is unique in the record:

- If the array name is unique in the record, the inner class is within the record class and is named as follows:

recordName.siName

where

recordName

Is the name of the record part

siName

Is the name of the array

- If the array name is not unique in the record, the name of the inner class is based on the fully qualified name of the array, with one qualifier separated from the next by a combination of dollar sign (\$) and underscore (_). For example, if

the array is at the third level of the record, the generated class is an inner class of the record class and is named as follows:

Topname\$_Secondname\$_Siname

where

Topname

Is the name of the top-level structure item

Secondname

Is the name of the second-level structure item

Siname

Is the name of the substructured-array item

If another, same-named array is immediately subordinate to the highest level of the record, the inner class is also within the record class and is named as follows:

Topname\$_Siname

where

Topname

Is the name of the highest-level structure item

Siname

Is the name of the substructured-array item

Finally, consider the following case: a substructured array has a name that is not unique in the record, and the array is subordinate to another substructured array whose name is not unique in the record. The class for the subordinate array is generated as an inner class of an inner class.

When you generate a Java wrapper, you also generate a Java properties file and a linkage properties file if you request that linkage options be set at run time.

Related concepts

“Build plan”

“Enterprise JavaBean (EJB) session bean” on page 393

“Java wrapper” on page 395

“Linkage options part” on page 439

“Linkage properties file” on page 404

“Results file” on page 405

Related tasks

“Generating Java wrappers” on page 131

Related reference

“callLink element” on page 443

“Java wrapper classes” on page 395

Build plan

The build plan is an XML file that makes the following details available at preparation time:

- What files need to be processed on the build machine
- What build scripts are needed to process them
- Where outputs are to be placed

The build plan resides on the development platform and informs the build client of all the build steps. For each step a request is made of the build server.

EGL produces a build plan whenever you generate a COBOL program, Java program, or Java wrapper, unless you set the build descriptor option **buildPlan** to NO.

For details on the name of the build plan, see *Generated output (reference)*.

Related concepts

“Build script” on page 271

“Generated output” on page 386

Related reference

“buildPlan” on page 243

“Generated output (reference)” on page 387

COBOL program

A COBOL program can be generated to run on the iSeries platform.

When you request that a program part be generated as a COBOL program, EGL produces a source file. For details on the file name, see *Generated output (reference)*.

Related concepts

“Generated output” on page 386

“Java program, page handler, and library” on page 394

“Java wrapper” on page 395

“Run-time configurations” on page 2

Related tasks

“Generating for COBOL” on page 132

Related reference

“Generated output (reference)” on page 387

“Output of COBOL generation” on page 388

Enterprise JavaBean (EJB) session bean

An EJB session bean comprises the following components:

- Home interface, which gives a client access to the EJB session bean at run time
- Remote bean interface, which lists the methods that are directly available to that client
- Bean implementation, which contains the logic that is indirectly available to that client

An EJB session bean is an intermediary between one program and another or between an EGL Java wrapper and a program. Generation of the EJB session bean largely depends on settings in the linkage options part that is used at generation time. For details, see *Linkage options part*; in particular, the overview of the **callLink** element.

For details on the output file names, see *Generated output (reference)*.

Related concepts

“Generated output” on page 386

“Linkage options part” on page 439

Related reference

“Generated output (reference)” on page 387

Java program, page handler, and library

When you request that a program part be generated as a Java program, or when you request that a pageHandler or Java-related library part be generated, EGL produces a class and a file for each of the following:

- The program, pageHandler, or library part
- Each record declared either in that part itself or in any function that is invoked directly or indirectly by that part
- Each data table, form group, and form that is used

For details on the class names, see *Output of Java program generation*.

Related concepts

“Library (generated output)” on page 403

“PageHandler part” on page 469

Related tasks

“Setting up the J2EE run-time environment for EGL-generated code” on page 206

Related reference

“Generated output (reference)” on page 387

“Output of Java program generation” on page 388

J2EE environment file

A *J2EE environment file* is a text file that contains property-and-value pairs that are derived from information that you specify when you generate a Java program. The sources of information are the build descriptor, the resource associations part, and the linkage options part.

When you configure the environment of the Java program, you can use the J2EE environment file as the basis of information that you place in the run-time deployment descriptor.

For details on the name of the J2EE environment file, see *Generated output (reference)*.

For details on the different ways that you can set deployment-descriptor values, see *Setting deployment-descriptor values*.

Related concepts

“Run-time configurations” on page 2

Related tasks

“Setting up the J2EE run-time environment for EGL-generated code” on page 206

“Setting deployment-descriptor values” on page 207

Related reference

“Generated output (reference)” on page 387

“genProperties” on page 254

“sqlDB” on page 262

Java wrapper

A Java wrapper is a set of classes that act as an interface between the following executables:

- A servlet or a hand-written Java program, on the one hand
- A generated program or EJB session bean, on the other

You generate the Java wrapper classes if you use a build descriptor that has these characteristics:

- The build descriptor option **enableJavaWrapperGen** is set to **yes** or **only**; and
- The build descriptor option **linkage** references a linkage options part that includes a **callLink** element to guide the call from wrapper to program; and
- One of two statements apply:
 - The call from wrapper to program is by way of an EJB session bean (in which case the **callLink** element, **linkType** property is set to **ejbCall**); or
 - The call from wrapper to program is remote (in which case the **callLink** element, **type** property is set to **remoteCall**); also, the **callLink** element, **javaWrapper** property is set to **yes**.

If an EJB session bean mediates between the Java wrapper classes and an EGL-generated program, you generate the EJB session if you use a build descriptor that has these characteristics:

- The build descriptor option **enableJavaWrapperGen** is set to **yes** or **only**; and
- The build descriptor option **linkage** references a linkage options part that includes a **callLink** element to guide the call from wrapper to EJB session bean (in which case the **type** property of the **callLink** element is set to **ejbCall**).

For further details on using the classes, see *Java wrapper classes*. For details on the class names, see *Generated output (reference)*.

Related concepts

“COBOL program” on page 393

“Generated output” on page 386

“Java program, page handler, and library” on page 394

“Run-time configurations” on page 2

Related tasks

“Generating Java wrappers” on page 131

Related reference

“Generated output (reference)” on page 387

“Java wrapper classes”

“Output of Java wrapper generation” on page 389

Java wrapper classes

When you request that a program part be generated as a Java wrapper, EGL produces a wrapper class for each of the following:

- The generated program
- Each record or form that is declared as a parameter in that program
- Each dynamic array that is declared as a parameter; and if the array is array of records, the class for the dynamic array class is in addition to the class for the record itself
- Each structure item that has these characteristics:

- Is in one of the records or forms for which a wrapper class is generated
- Has at least one subordinate structure item; in other words, is substructured
- Is an array; in this case, a substructured array

An example of a record part with a substructured array is as follows:

```
Record myPart type basicRecord
  10 MyTopStructure CHAR(20)[5];
  20 MyStructureItem01 CHAR(10);
  20 MyStructureItem02 CHAR(10);
end
```

Later descriptions refer to the wrapper classes for a given program as the *program wrapper class*, the *parameter wrapper classes*, the *dynamic array wrapper classes*, and the *substructured-item-array wrapper classes*.

EGL generates a BeanInfo class for each parameter wrapper class, dynamic array wrapper class, or substructured-item-array wrapper class. The BeanInfo class allows the related wrapper class to be used as a Java-compliant Java bean. You probably will not interact with the BeanInfo class.

Overview of how to use the wrapper classes: To use the wrapper classes to communicate with a program generated with VisualAge Generator or EGL, do as follows in the native Java program:

- Instantiate a class (a subclass of CSOPowerServer) to provide middleware services such as converting data between native Java code and a generated program:

```
import com.ibm.vgj.cso.*;

public class MyNativeClass
{
    /* declare a variable for middleware */
    CSOPowerServer powerServer = null;

    try
    {
        powerServer = new CSOLocalPowerServerProxy();
    }
    catch (CSOException exception)
    {
        System.out.println("Error initializing middleware"
            + exception.getMessage());
        System.exit(8);
    }
}
```

- Instantiate a program wrapper class to do as follows:
 - Allocate data structures, including dynamic arrays, if any
 - Provide access to methods that in turn access the generated program

The call to the constructor includes the middleware object:

```
myProgram = new MyprogramWrapper(powerServer);
```

- Declare variables that are based on the parameter wrapper classes:

```
Mypart myParm = myProgram.getMyParm();
Mypart2 myParm2 = myProgram.getMyParm2();
```

If your program has parameters that are dynamic arrays, declare additional variables that are each based on a dynamic array wrapper class:

```
myRecArrayVar myParm3 = myProgram.getMyParm3();
```

For details on interacting with dynamic arrays, see “Dynamic array wrapper classes” on page 401.

- In most cases (as in the previous step), use the parameter variables to reference and change memory that was allocated in the program wrapper object
- Set a userid and password, but only in these cases:
 - The Java wrapper accesses an iSeries-based program by way of the iSeries Toolbox for Java; or
 - The generated program runs on a CICS for z/OS region that authenticates remote access.

The userid and password are not used for database access.

You set and review the userid and password by using the `callOptions` variable of the program object, as in this example:

```
myProgram.callOptions.setUserID("myID");
myProgram.callOptions.setPassword("myWord");
myUserID = myProgram.callOptions.getUserID();
myPassword = myProgram.callOptions.getPassword();
```

- Access the generated program, in most cases by invoking the `execute` method of the program wrapper object:


```
myProgram.execute();
```
- Use the middleware object to establish database transaction control, but only in the following situation:
 - The program wrapper object either is accessing a generated program on CICS for z/OS or is accessing an iSeries-based COBOL program by way of the IBM Toolbox for Java. In the latter case, the value of **remoteComType** for the call is `JAVA400`.
 - In the linkage options part used to generate the wrapper classes, you specified that the database unit of work is under client (in this case, wrapper) control; for details, see the reference to **luwControl** in *callLink element*.

If the database unit of work is under client control, processing includes use of `commit` and `rollback` methods of the middleware object:

```
powerServer.commit();
powerServer.rollback();
```

- Close the middleware object and allow for garbage collection:

```
if (powerServer != null)
{
    try
    {
        powerServer.close();
        powerServer = null;
    }

    catch(CS0Exception error)
    {
        System.out.println("Error closing middleware"
            + error.getMessage());
        System.exit(8);
    }
}
```

The program wrapper class: The program wrapper class includes a private instance variable for each parameter in the generated program. If the parameter is a record or form, the variable refers to an instance of the related parameter wrapper class. If the parameter is a data item, the variable has a primitive Java type.

A table at the end of this help page describes the conversions between EGL and Java types.

A program wrapper object includes the following public methods:

- **get** and **set** methods for each parameter, where the format of the name is as follows:

purposeParmname()

purpose

The word **get** or **set**

Parmname

Name of the data item, record, or form; the first letter is upper case, and aspects of the other letters are determined by the naming convention described in “Naming conventions for Java wrapper classes” on page 402

- An **execute** method for calling the program; you use this method if the data that will be passed as arguments on the call is in the memory allocated for the program wrapper object

Instead of assigning values to the instance variables, you can do as follows:

- Allocate memory for parameter wrapper objects, dynamic array wrapper objects, and primitive types
- Assign values to the memory you allocated
- Pass those values to the program by invoking the **call** method of the program wrapper object rather than the **execute** method

The program wrapper object also includes the `callOptions` variable, which has the following purposes:

- If you generated the Java wrapper so that linkage options for the call are set at generation time, the `callOptions` variable contains the linkage information. For details on when the linkage options are set, see `remoteBind` in *callLink element*.
- If you generated the Java wrapper so that linkage options for the call are set at run time, the `callOptions` variable contains the name of the linkage properties file. The file name is *LO.properties*, where *LO* is the name of the linkage options part used for generation.
- In either case, the `callOptions` variable provides the following methods for setting or getting a userid and password:

```
setPassword(passWord)
setUserid(userid)
getPassword()
getUserid()
```

The `userid` and `password` are used when you set the **remoteComType** property of the *callLink* element to one of the following values:

- CICSECI
- CICSJ2C
- JAVA400

Finally, consider the following situation: your native Java code requires notification when a change is made to a parameter of primitive type. To make such a notification possible, the native code registers as a listener by invoking the **addPropertyChangeListener** method of the program wrapper object. In this case, either of the following situations triggers the `PropertyChange` event that causes the native code to receive notification at run time:

- Your native code invokes a **set** method on a parameter of primitive type
- The generated program returns changed data to a parameter of primitive type

The `PropertyChange` event is described in the JavaBean specification of Sun Microsystems, Inc.

The set of parameter wrapper classes: A parameter wrapper class is produced for each record that is declared as a parameter in the generated program. In the usual case, you use a parameter wrapper class only to declare a variable that references the parameter, as in the following example:

```
Mypart myRecWrapperObject = myProgram.getMyrecord();
```

In this case, you are using the memory allocated by the program wrapper object.

You also can use the parameter wrapper class to declare memory, as is necessary if you invoke the call method (rather than the execute method) of the program object.

The parameter wrapper class includes a set of private instance variables, as follows:

- One variable of a Java primitive type for each of the parameter's low-level structure items, but only for a structure item that is neither an array nor within a substructured array
- One array of a Java primitive type for each EGL structure item that is an array and is not substructured
- An object of an inner, array class for each substructured array that is not itself within a substructured array; the inner class can have nested inner classes to represent subordinate substructured arrays

The parameter wrapper class includes several public methods:

- A set of **get** and **set** methods allows you to get and set each instance variable. The format of each method name is as follows:

purpose
siName()

purpose

The word **get** or **set**.

siName

Name of the structure item. The first letter is upper case, and aspects of the other letters are determined by the naming convention described in "Naming conventions for Java wrapper classes" on page 402.

Note: Structure items that you declared as fillers are included in the program call; but the array wrapper classes do not include public **get** or **set** methods for those structure items.

- The method **equals** allow you to determine whether the values stored in another object of the same class are identical to the values stored in the parameter wrapper object. The method returns **true** only if the classes and values are identical.
- The method **addPropertyChangeListener** is invoked if your program requires notification of a change in a variable of a Java primitive type.
- A second set of **get** and **set** methods allow you to get and set the null indicators for each structure item in an SQL record parameter. The format of each of these method names is as follows:

purpose
siNameNullIndicator()

purpose

The word **get** or **set**.

siName

Name of the structure item. The first letter is upper case, and aspects of the other letters are determined by the naming convention described in “Naming conventions for Java wrapper classes” on page 402.

The set of substructured-item-array wrapper classes: A substructured-item-array wrapper class is an inner class of a parameter class and represents a substructured array in the related parameter. The substructured-item-array wrapper class includes a set of private instance variables that refer to the structure items at and below the array itself:

- One variable of a Java primitive type for each of the array’s low-level structure items, but only for a structure item that is neither an array nor within a substructured array
- One array of a Java primitive type for each EGL structure item that is an array and is not substructured
- An object of an inner, substructured-item-array wrapper class for each substructured array that is not itself within a substructured array; the inner class can have nested inner classes to represent subordinate substructured arrays

The substructured-item-array wrapper class includes the following methods:

- A set of **get** and **set** methods for each instance variable

Note: Structure items that you declared as nameless fillers are used in the program call; but the substructured-item-array wrapper classes do not include public get or set methods for those structure items.

- The method **equals** allows you to determine whether the values stored in another object of the same class are identical to the values stored in the substructured-item-array wrapper object. The method returns **true** only if the classes and values are identical.
- The method **addChangeListener**, for use if your program requires notification of a change in a variable of a Java primitive type

In most cases, the name of the top-most substructured-item-array wrapper class in a parameter wrapper class is of the following form:

ParameterClassname.ArrayClassName

Consider the following record, for example:

```
Record CompanyPart type basicRecord
10 Departments CHAR(20)[5];
  20 CountryCode CHAR(10);
  20 FunctionCode CHAR(10)[3];
    30 FunctionCategory CHAR(4);
    30 FunctionDetail CHAR(6);
end
```

If the parameter Company is based on CompanyPart, you use the string CompanyPart.Departments as the name of the inner class.

An inner class of an inner class extends use of a dotted syntax. In this example, you use the symbol CompanyPart.Departments.Functioncode as the name of the inner class of Departments.

For additional details on how the substructured-item-array wrapper classes are named, see *Output of Java wrapper generation*.

Dynamic array wrapper classes: A dynamic array wrapper class is produced for each dynamic array that is declared as a parameter in the generated program. Consider the following EGL program signature:

```
Program myProgram(intParms int[], recParms MyRec[])
```

The name of the dynamic array wrapper classes are IntParmsArray and MyRecArray.

You use a dynamic array wrapper class to declare a variable that references the dynamic array, as in the following examples:

```
IntParmsArray myIntArrayVar = myProgram.getIntParms();
MyRecArray myRecArrayVar = myProgram.getRecParms();
```

After declaring the variables for each dynamic array, you might add elements:

```
// adding to an array of Java primitives
// is a one-step process
myIntArrayVar.add(new Integer(5));

// adding to an array of records or forms
// requires multiple steps; in this case,
// begin by allocating a new record object
MyRec myLocalRec = (MyRec)myRecArrayVar.makeNewElement();

// the steps to assign values are not shown
// in this example; but after you assign values,
// add the record to the array
myRecArrayVar.add(myLocalRec);

// next, run the program
myProgram.execute();

// when the program returns, you can determine
// the number of elements in the array
int myIntArrayVarSize = myIntArrayVar.size();

// get the first element of the integer array
// and cast it to an Integer object
Integer firstIntElement = (Integer)myIntArrayVar.get(0);

// get the second element of the record array
// and cast it to a MyRec object
MyRec secondRecElement = (MyRec)myRecArrayVar.get(1);
```

As suggested by that example, EGL provides several methods for manipulating the variables that you declared.

Method of the dynamic array class	Purpose
<code>add(int, Object)</code>	To insert an object at the position specified by <i>int</i> and to shift the current and subsequent elements to the right.
<code>add(Object)</code>	To append an object to the end of the dynamic array.
<code>addAll(ArrayList)</code>	To append an ArrayList to the end of the dynamic array.
<code>get()</code>	To retrieve the ArrayList object that contains all elements in the array
<code>get(int)</code>	To retrieve the element that is in the position specified by <i>int</i>

Method of the dynamic array class	Purpose
<code>makeNewElement()</code>	To allocate a new element of the array-specific type and to retrieve that element, without adding that element to the dynamic array.
<code>maxSize()</code>	To retrieve an integer that indicates the maximum (but not actual) number of elements in the dynamic array
<code>remove(<i>int</i>)</code>	To remove the element that is in the position specified by <i>int</i>
<code>set(<i>ArrayList</i>)</code>	To use the specified <i>ArrayList</i> as a replacement for the dynamic array
<code>set(<i>int</i>, <i>Object</i>)</code>	To use the specified object as a replacement for the element that is in the position specified by <i>int</i>
<code>size()</code>	To retrieve the number of elements that are in the dynamic array

Exceptions occur in the following cases, among others:

- If you specify an invalid index in the **get** or **set** method
- If you try to add (or set) an element that is of a class incompatible with the class of each element in the array
- If you try to add elements to a dynamic array when the maximum size of the array cannot support the increase; and if the method **addAll** fails for this reason, the method adds no elements

Naming conventions for Java wrapper classes: EGL creates a name in accordance with these rules:

- If the name is all upper case, lower case all letters.
- If the name is a keyword, precede it with an underline
- If a hyphen or underline is in the name, remove that character and upper case the next letter
- If a dollar sign (\$), at sign (@), or pound sign (#) is in the name, replace each of those characters with a double underscore (__) and precede the name with an underscore (_).
- If the name is used as a class name, upper case the first letter.

The following rules apply to dynamic array wrapper classes:

- In most cases, the name of a class is based on the name of the part declaration (data item, form, or record) that is the basis of each element in the array. For example, if a record part is called `MyRec` and the array declaration is `recParms myRec[]`, the related dynamic array wrapper class is called `MyRecArray`.
- If the array is based on an item declaration that has no related part declaration, the name of the dynamic array class is based on the array name. For example, if the array declaration is `intParms int[]`, the related dynamic array wrapper class is called `IntParmsArray`.

Data type cross-reference: The next table indicates the relationship of EGL primitive types in the generated program and the Java data types in the generated wrapper.

EGL primitive type	Length in chars or digits	Length in bytes	Decimals	Java data type	Maximum precision in Java
CHAR	1-32767	2-32766	NA	String	NA
DBCHAR	1-16383	1-32767	NA	String	NA
MBCHAR	1-32767	1-32767	NA	String	NA
UNICODE	1-16383	2-32766	NA	String	NA
HEX	2-75534	1-32767	NA	byte[]	NA
BIN, SMALLINT	4	2	0	short	4
BIN, INT	9	4	0	int	9
BIN, BIGINT	18	8	0	long	18
BIN	4	2	>0	float	4
BIN	9	4	>0	double	15
BIN	18	8	>0	double	15
DECIMAL, PACF	1-3	1-2	0	short	4
DECIMAL, PACF	4-9	3-5	0	int	9
DECIMAL, PACF	10-18	6-10	0	long	18
DECIMAL, PACF	1-5	1-3	>0	float	6
DECIMAL, PACF	7-18	4-10	>0	double	15
NUM, NUMC	1-4	1-4	0	short	4
NUM, NUMC	5-9	5-9	0	int	9
NUM, NUMC	10-18	10-18	0	long	18
NUM, NUMC	1-6	1-6	>0	float	6
NUM, NUMC	7-18	7-18	>0	double	15

Related concepts

“Java wrapper” on page 395

“Run-time configurations” on page 2

Related tasks

“Generating Java wrappers” on page 131

Related reference

“callLink element” on page 443

“How Java wrapper names are aliased” on page 466

“Linkage properties file (details)” on page 431

“Output of Java wrapper generation” on page 389

“remoteBind in callLink element” on page 454

Library (generated output)

A library part for Java output is generated as a Java class. The name of the class is the same as the part name, but EGL makes substitutions as needed; for details, see *How Java names are aliased*.

In a given run unit, the first program that uses that class loads it, and the class is removed from memory when the run unit ends.

Related concepts

"Library part" on page 433

"Run unit" on page 504

Related tasks

"How Java names are aliased" on page 465

Related reference

"Library part in EGL source format" on page 434

Linkage properties file

A *linkage properties file* is a text file that is used at Java run time to give details on how a generated Java program or wrapper calls a generated Java program in a different process.

The file is applicable only if you specified that linkage options for a Java program or wrapper are set at run time instead of at generation time. You may generate the file or create one from scratch.

For details on when the file is generated and on the file format, see *Linkage properties file (details)*. For details on the name of the generated file, see *Generated output (reference)*. For details on deployment, see *Deploying a linkage properties file*.

Related concepts

"Generated output" on page 386

Related tasks

"Deploying a linkage properties file" on page 223

"Setting up the J2EE run-time environment for EGL-generated code" on page 206

Related reference

"Generated output (reference)" on page 387

"genProperties" on page 254

"Linkage properties file (details)" on page 431

Program properties file

The *program properties file* contains Java run-time properties in a format that is accessible only to a Java program that runs outside of a J2EE environment. For overview information, see *Java run-time properties*.

The program properties file is a text file. Each entry other than comments has the following format:

propertyName = *propertyValue*

propertyName

One of the properties described in *Java run-time properties (details)*

propertyValue

The property value that is available to your program at run time

A comment is any line where the first non-text character is a pound sign (#).

A portion of an example file is as follows:

```
# This file contains properties for generated
# Java programs that are being debugged in a
# non-J2EE Java project
```

```
vgj.nls.code = ENU  
vgj.datemask.gregorian.long.ENU = MM/DD/YYYY
```

For details on the name given to the generated file, see *Generated output (reference)*.

Related concepts

“EGL debugger” on page 107
“Java run-time properties” on page 421

Related tasks

“Generated output (reference)” on page 387

Related reference

“genProperties” on page 254
“J2EE” on page 255
“Java run-time properties (details)” on page 423

Results file

The results file contains status information on the code-preparation steps that were done on the target environment. You receive the file only if EGL attempts to prepare generated output. If you are preparing a COBOL program, you also receive a file for each step in preparation.

Preparation occurs automatically when you generate into a directory and use the following build descriptor options:

- **prep** is set to YES
- **buildPlan** is set to YES

For details on the name of the results file and on the additional files provided to you after a COBOL program is prepared, see *Generated output (reference)*.

Related concepts

“Build descriptor part” on page 234
“Generated output” on page 386

Related tasks

“Generating into a directory” on page 220

Related reference

“Generated output (reference)” on page 387
“buildPlan” on page 243
“prep” on page 258

Generation reference

EGL command file

An EGL command file indicates what EGL files you wish to process when you generate output outside of the workbench, whether you are using the workbench batch interface (command EGLCMD) or the EGL SDK (command EGLSDK). You can create the file in either of two ways:

- By hand, according to the rules described later; or
- By using the EGL Generation wizard, as described in *Generating in the workbench*.

The command file is an XML file, and the file name must have the extension .xml, in any combination of uppercase and lowercase letters. The file content must conform to the following document type definition (DTD), which is in the WebSphere Studio installation directory (like c:\myStudio) :

```
wstools\eclipse\plugins\
com.ibm.etools.egl.utilities_version\
dtd\eglcommands_5_1.dtd
```

version

The installed version of the plugin; for example, 5.1.2

The following table shows the elements and attributes supported by the DTD. The element and attribute names are case sensitive.

Element	Attribute	Attribute value
EGLCOMMANDS (required)	eglp	<p>As described in <i>eglp</i>, the <i>eglp</i> attribute identifies directories to search when EGL uses an import statement to resolve the name of a part. The attribute is optional and if present, references a quoted string that has one or more directory names, each separated from the next by a semicolon.</p> <p>The attribute is used only if the command EGLSDK is referencing the command file. If the command EGLCMD is in use, the value of <i>eglp</i> is ignored; instead, import statements are resolved in accordance with the EGL project path, as described in <i>Import</i>.</p>
buildDescriptor (optional; you can avoid specifying this value if you are using a master build descriptor, as described in <i>Build descriptor part</i>)	name	<p>The name of a build descriptor part that guides generation. The build descriptor must be at the top level of an EGL build (.eglbld) file.</p> <p>Build descriptor options that you specify when invoking EGLCMD or EGLSDK take precedence over options in the build descriptor (if any) that is listed in the EGL command file.</p>
	file	<p>The absolute or relative path of the EGL file that contains the build descriptor. Relative paths specified for EGLCMD are relative to the path name of the Enterprise Developer workspace. Relative paths specified for EGLSDK are relative to the directory in which you run the command.</p> <p>The path must be in double quotes if the path includes a space.</p>
generate (optional)	file	<p>The absolute or relative path of the EGL file that contains the part you want to process. Relative paths specified for EGLCMD are relative to the path name of the Enterprise Developer workspace. Relative paths specified for EGLSDK are relative to the directory in which you run the command.</p> <p>The path must be in double quotes if the path includes a space.</p> <p>If you omit the file attribute, no generation occurs.</p>

Examples of command files

This section shows two command files. The results produced by either file are the same whether you use the EGLCMD command or the EGLSDK command, if you run the EGLSDK command in the directory where the EGL program files reside.

The following command file contains a generate command that uses the build descriptor myBDescPart to generate the program myProgram.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EGLCOMMANDS PUBLIC "-//IBM//DTD EGLCOMMANDS 5.0//EN" "">
<EGLCOMMANDS eglpath="C:\mydata\entdev\workspace\projectinteract">
  <generate file="projectinteract\myProgram.eglpgm">
    <buildDescriptor name="myBDescPart" file="projectinteract\mybdesc.eglbld"/>
  </generate>
</EGLCOMMANDS>
```

The next example contains two generate commands, both of which implicitly use a master build descriptor.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EGLCOMMANDS PUBLIC "-//IBM//DTD EGLCOMMANDS 5.0//EN" "">
<EGLCOMMANDS eglpath="C:\mydata\entdev\workspace\projecttrade">
  <generate file="projecttrade\program2.eglpgm"/>
  <generate file="projecttrade\program3.eglpgm"/>
</EGLCOMMANDS>
```

Related concepts

“Build descriptor part” on page 234

“Generation from the EGL SDK” on page 127

“Generation from the workbench batch interface” on page 126

“Import” on page 26

Related tasks

“Generating from the EGL SDK” on page 128

“Generating from the workbench batch interface” on page 127

“Generating in the workbench” on page 125

Related reference

“EGLCMD”

“EGL command file” on page 405

“EGL build path and eglpath” on page 409

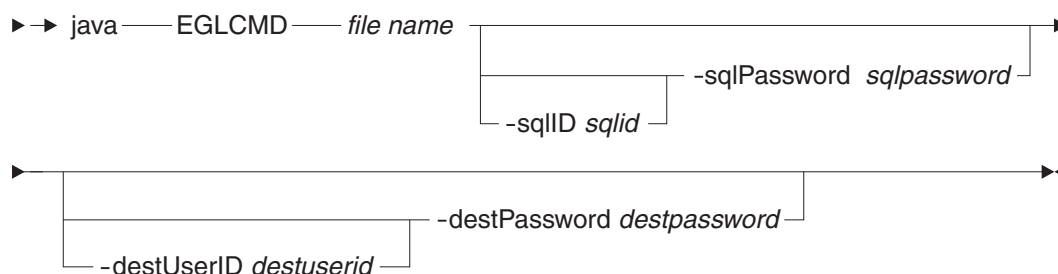
“EGLSDK” on page 410

EGLCMD

The command EGLCMD gives you access to the Workbench batch interface, as described in *Generation from the workbench batch interface*.

Syntax

The syntax for invoking EGLCMD is as follows:



generate

Indicates that the command itself references the EGL file and build descriptor part that are used to generate output. In this case, the command EGLCMD does not reference a command file.

cmdFile

Specifies the absolute or relative path of the file described in *EGL command file*. Relative paths are relative to the directory in which you run the command.

Embed the path in double quotes.

-generateFile *genFile*

Specifies the absolute or relative path of the EGL file that contains the part you want to process. Relative paths are relative to the directory in which you run the command.

Embed the path in double quotes.

-buildDescriptorFile *bdFile*

Specifies the absolute or relative path of the build file that contains the build descriptor. Relative paths are relative to the directory in which you run the command.

Embed the path in double quotes.

-buildDescriptorName *bdName*

Specifies the name of a build descriptor part that guides generation. The build descriptor must be at the top level of an EGL build (.eglbld) file.

-sqlID *sqlID*

Sets the value of build descriptor option sqlID.

-sqlPassword *sqlPW*

Sets the value of build descriptor option `sqlPassword`.

-destUserid *destID*

Sets the value of build descriptor option `destUserID`.

-destPassword *destPW*

Sets the value of build descriptor option `destPassword`.

-data *workSpace*

Specifies the absolute or relative path of the workspace directory. Relative paths are relative to the directory in which you run the command.

If you do not specify a value, the command accesses a subdirectory named *Workspace* in the current directory.

Embed the path in double quotes.

Build descriptor options that you specify when invoking the command EGLCMD take precedence over options in the build descriptor (if any) that is listed in the EGL command file.

Examples

In the commands that follow, each multiline example belongs on a single line:

```
java EGLCMD "commandfile.xml"

java EGLCMD "commandfile.xml" -data "c:\myWorkspace"

java EGLCMD generate
  -generateFile "c:\myProg.eglpgm"
  -buildDescriptorFile "c:\myBuild.eglbld"
  -buildDescriptorName myBuildDescriptor
  -data "myWorkspace"

java EGLCMD "myCommand.xml"
  -sqlID myID -sqlPassword myPW
  -destUserID myUserID -destPassword myPass
  -data "my Workspace"
```

Related concepts

“Generation from the workbench batch interface” on page 126

Related tasks

“Generating from the workbench batch interface” on page 127

“Syntax diagram” on page 506

Related reference

“destPassword” on page 248

“destUserID” on page 249

“sqlID” on page 263

“sqlPassword” on page 264

EGL build path and egldpath

Each EGL project and EGL Web project is associated with an EGL build path so that the project can reference parts in other projects. For details on when the EGL build path is used and on why the order of build-path entries is important, see *References to parts*.

When you specify the EGL build path, you can choose to *export* one or more of the projects that are listed in the build path. Then, when a project refers to the project being declared, each of the exported projects is made available to the referencing project, as in the following example:

- The EGL build path for project A comprises the following projects, in order:

A, B, C, D

Projects B and D are exported.

- The EGL build path for project L comprises the following projects, in order:

L, J, A, Z

- The effective build path for project L also includes the projects that were exported from project A. In this case, the EGL build path for project L is effectively as follows:

L, J, A, B, D, Z

The exported projects are placed after the project that exports them, in the order in which the projects are listed in the build path of the exporting project.

The build path of a project always includes the project itself, which is usually first in build-path order, as is recommended. If you have multiple EGL source folders in your project, all must be listed in the EGL build path for that project, and the order of those folders is used by any project that refers to your project.

It is strongly recommended that you avoid having identically named packages in different projects or in different folders of the same project.

If you generate in the EGL SDK, the situation is as follows:

- Project information is not available.
- The command-line argument *eglp* replaces the functionality of the EGL build path. *eglp* is a list of operating-system directories that are searched when the EGL SDK attempts to resolve a part reference.
- The rules for when *eglp* is used are equivalent to the rules for when the EGL build path is used; however, you cannot export directories as you can export projects.

When you use the EGL SDK, it is strongly recommended that you avoid having identically named packages in different directories.

Related concepts

“Generation from the EGL SDK” on page 127

“References to parts” on page 16

Related tasks

“Generating from the EGL SDK” on page 128

Related reference

“EGLSDK”

EGLSDK

The command EGLSDK gives you access to the EGL Software Development Kit (EGL SDK), as described in *Generation from the EGL SDK*.

Syntax

The syntax for invoking EGLSDK is as follows:



generate

Indicates that the command itself references the EGL file and build descriptor part that are used to generate output. In this case, the command EGLSDK does not reference a command file.

cmdFile

Specifies the absolute or relative path of the file described in *EGL command file*. Relative paths are relative to the directory in which you run the command.

Embed the path in double quotes.

-eglpath eglpath

As described in *eglpath*, the *eglpath* option identifies directories to search when EGL uses an import statement to resolve the name of a part. You specify a quoted string that has one or more directory names, each separated from the next by a semicolon.

-generateFile genFile

The absolute or relative path of the EGL file that contains the part you want to process. Relative paths are relative to the directory in which you run the command.

Embed the path in double quotes.

-buildDescriptorFile bdFile

The absolute or relative path of the build file that contains the build descriptor. Relative paths are relative to the directory in which you run the command.

Embed the path in double quotes.

-buildDescriptorName bdName

The name of a build descriptor part that guides generation. The build descriptor must be at the top level of an EGL build (.eglbld) file.

-sqlID *sqlID*

Sets the value of build descriptor option sqlID.

-sqlPassword *sqlPW*

Sets the value of build descriptor option sqlPassword.

-destUserid *destID*

Sets the value of build descriptor option destUserID.

-destPassword *destPW*

Sets the value of build descriptor option destPassword.

The eglpath value that you specify when invoking the command EGLSDK takes precedence over any eglpath value in an EGL command file. Similarly, build descriptor options that you specify when invoking the command take precedence over options in any build descriptor that is listed in an EGL command file.

Examples

In the commands that follow, each multiline example belongs on a single line:

```
java EGLSDK "commandfile.xml"
```

```
java EGLSDK "commandfile.xml"  
-eglpath "c:\myGroup;h:\myCorp"
```

```
java EGLSDK generate  
-eglpath "c:\myGroup;h:\myCorp"  
-generateFile "c:\myProg.eglpqm"  
-buildDescriptorFile "c:\myBuild.eglbld"  
-buildDescriptorName myBuildDescriptor
```

```
java EGLSDK "myCommand.xml"  
-sqlID myID -sqlPassword myPW  
-destUserID myUserID -destPassword myPass
```

Related concepts

"Build descriptor part" on page 234

"Generation from the EGL SDK" on page 127

"Import" on page 26

"Master build descriptor" on page 413

Related tasks

"Generating from the EGL SDK" on page 128

Related reference

"destPassword" on page 248

"destUserID" on page 249

"EGL build path and eglpath" on page 409

"sqlID" on page 263

"sqlPassword" on page 264

"Syntax diagram" on page 506

Format of eglmaster.properties file

The eglmaster.properties file is a Java properties file that the EGL SDK uses to specify the name and file path name of the master build descriptor. This properties file must be contained in a directory that is specified in the CLASSPATH variable of the process that invokes the EGLSDK command. The format of the eglmaster.properties file is as follows:

```
masterBuildDescriptorName=desc  
masterBuildDescriptorFile=path
```

where:

desc

The name of the master build descriptor

path

The fully qualified path name of the EGL file in which the master build descriptor used by the EGL SDK is declared

The content of this file must follow the rules of a Java properties file. You can use either a slash (/) or two backslashes (\\) to separate file names within a path name.

You must specify both the **masterBuildDescriptorName** and **masterBuildDescriptorFile** keywords in the properties file. Otherwise the `eglmaster.properties` file is ignored.

Following is an example of the contents of an `eglmaster.properties` file:

```
# Specify the name of the master build descriptor:
masterBuildDescriptorName=MYBUILDDESRIPTOR
# Specify the file that contains the master build descriptor:
masterBuildDescriptorFile=d:/egl/build descriptors/master.egl
```

Related concepts

"Master build descriptor"

Related tasks

"Choosing options for COBOL generation" on page 132

"Choosing options for Java generation" on page 129

Related reference

"Build descriptor options" on page 237

"EGLSDK" on page 410

"Format of master build descriptor plugin.xml file" on page 414

Master build descriptor

An installation can provide its own set of default values for build options and control whether those default values can be overridden.

To set up the master build descriptor, create two build descriptor parts in the same build file, the first referencing the second by use of the build descriptor option **nextBuildDescriptor**. The options in the first part specify default values for options that may not be overridden. The options in the second part specify default values for options that can be overridden.

To install the master build descriptor in the workbench, add a plugin xml file like following to the workbench plugins directory:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
  id="egl.master.build.descriptor.plugin"
  name="EGL Master Build Descriptor Plug-in"
  version="5.0"
  vendor-name="IBM">
  <requires />
  <runtime />
  <extension point =
    "com.ibm.etools.egl.generation.base.framework.masterBuildDescriptor">
    <masterBuildDescriptor
```

```

file = "filePath.buildFileName"
name = "masterBuildPartName" />
</extension>
</plugin>

```

The file path (*filePath*) is in relation to the workspace directory.

If you are using the EGL SDK, you declare the name and file path name of the master build descriptor in a file named `eglmaster.properties`. This file must be in a directory that is listed in the CLASSPATH environment variable. The format of the properties file is as follows:

```

masterBuildDescriptorName=masterBuildPartName
masterBuildDescriptorFile=fullyQualifiedPathforEGLBuildFile

```

Related concepts

"Build" on page 121

"Build descriptor part" on page 234

"Build plan" on page 392

"EGL projects, packages, and files" on page 7

Related tasks

"Adding a build descriptor part" on page 92

Related reference

"Build descriptor options" on page 237

"Format of `eglmaster.properties` file" on page 412

"Format of master build descriptor `plugin.xml` file"

Format of master build descriptor `plugin.xml` file

The master build descriptor `plugin.xml` file is an XML file that the workbench uses to specify the name and file path name of the master build descriptor. You need this only if you need a master build descriptor to enforce certain options to be used for generation and you are generating from the workbench or are using the `EGLCMD` command. You must put this `plugin.xml` file in a directory in the plugins directory. The format of the file is as follows:

```

<?xml version="1.0" encoding="UTF-8"?>
<plugin
  id="id"
  name="plg"
  version="5.0"
  vendor-name="com">
  <requires />
  <runtime />
  <extension point =
    "com.ibm.etools.egl.generation.base.framework.masterBuildDescriptor">
    <masterBuildDescriptor file = "bfil" name = "mas" />
  </extension>
</plugin>

```

where:

id The identifier for the plug-in

plg

The name of the plug-in

com

The name of your company

bfil

The path name of a file containing a master build descriptor, of the form *project/folder/file*, relative to Enterprise Developer's workspace directory, where:

project

The name of the project directory

folder

The name of a directory within the project directory

file The name of a file that contains a master build descriptor

mas

The name of a master build descriptor

The content of this file must follow the rules of an XML file. To separate file names within a path name you must use the slash (/) character.

You must specify both the name attribute and the file attribute. Otherwise the plugin.xml file is ignored.

Following is an example of the contents of plugin.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Example master BuildDescriptor Plugin -->

<plugin
  id="example.master.BuildDescriptor.plugin"
  name="Example master BuildDescriptor plug-in"
  version="5.0"
  vendor-name="IBM">
  <requires />
  <runtime />
  <!-- ===== -->
  <!-- Register the master BuildDescriptor -->
  <!-- ===== -->
  <extension point =
    "com.ibm.etools.egl.generation.base.framework.masterBuildDescriptor" >
    <masterBuildDescriptor file
      = "myProject/myFolder/myFile.eglbld" name = "masterBD" />
    </extension>
</plugin>
```

Related concepts

"Build descriptor part" on page 234

"Master build descriptor" on page 413

Related tasks

"Generating from the workbench batch interface" on page 127

"Generating in the workbench" on page 125

Related reference

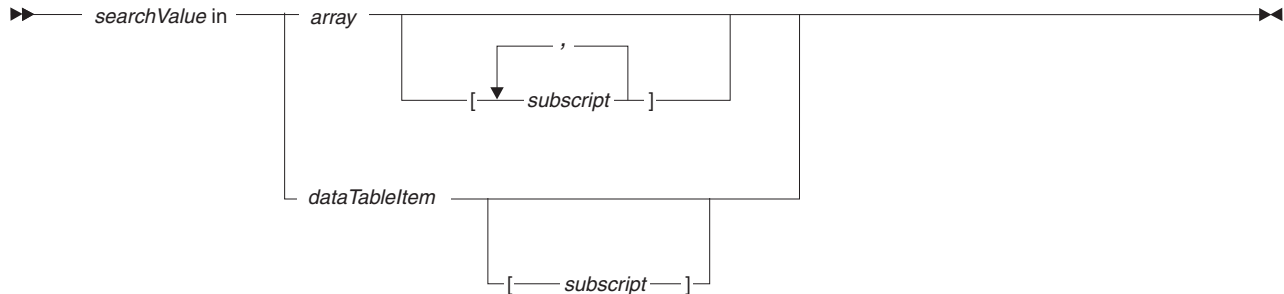
"Build descriptor options" on page 237

"EGLCMD" on page 407

"Format of eglmaster.properties file" on page 412

in

The operator **in** is a binary operator used in an elementary logical expression that has the following format:



searchValue

A literal or item, but not a system variable.

array A one-dimensional or multidimensional array. The operator **in** operates on a one-dimensional array, which may be part of a multidimensional array.

subscript

An integer, or an item (or system variable) that resolves to an integer. The value of a subscript is an index that refers to a specific element in an array.

An item used as a subscript of an array can not itself be an array element. In each of the following examples, `myItemB[1]` is both a subscript and an array element; as a result, the following syntax is *not* valid:

```
/* the next syntax is not valid */
myItemA[myItemB[1]]

// this next syntax is not valid; but only
// because myItemB is myItemB[1], the
// first element of a one-dimensional array
myItemA[myItemB]
```

dataTableItem

The name of a `dataTable` item. The item represents a column in the data table. The **in** operator interacts with that column as if the column were a one-dimensional array.

The logical expression resolves to true if the generated program finds the search value. The search begins at the element identified by the last array subscript. If *array* is a one-dimensional array, the last subscript is optional and defaults to 1. If *array* is a multidimensional array, the following statements are true:

- A subscript must be present for each dimension
- The generated program searches the one-dimensional array that is identified by the sequence of subscripts other than the last subscript
- The search begins at the element identified by the last subscript

In relation to both one-dimensional and multidimensional arrays, the search ends at the last element of the one-dimensional array under review.

The logical expression that includes **in** resolves to false in either of these cases:

- The search value is not found

- The value of the last subscript is greater than the number of entries in the one-dimensional array being searched

If the elementary logical expression resolves to true, the operation **in** sets the system variable **sysVar.arrayIndex** to the subscript value of the element that contains the search value. If the expression resolves to false, the operation sets **sysVar.arrayIndex** to zero.

Examples with a one-dimensional array

Let's assume that the structure item myString is substructured to an array of three characters:

```
structureItem name="myString" length=3
structureItem name="myArray" occurs=3 length=1
```

The next table shows the effect of the operator **in** if myString is "ABC".

Logical expression	Value of expression	Value of sysVar. ArrayIndex	Comment
"A" in myArray	true	1	The subscript of a single-dimension array defaults to 1
"C" in myArray[2]	true	3	Search begins at second element
"A" in myArray[2]	false	0	The search ends at the last element

Examples with a multidimension array

Let's assume that the array myArray01D is substructured to an array of three characters:

```
structureItem name="myArray01D" occurs=3 length=3
structureItem name="myArray02D" occurs=3 length=1
```

In this example, myArray01D is a one-dimensional array, with each element containing a string that is substructured to an array of three characters. myArray02D is a two-dimensional array, with each element (such as myArray02D[1,1]) containing a single character.

If the content of myArray01D is "ABC", "DEF", and "GHI", the content of myArray02D is as follows:

```
"A"  "B"  "C"
"D"  "E"  "F"
"G"  "H"  "I"
```

The next table shows the effect of the operator **in**.

Logical expression	Value of expression	Value of sysVar. ArrayIndex	Comment
"DEF" in myArray01D	true	2	A reference to a one-dimensional array does not require a subscript; by default, the search begins at the first element

Logical expression	Value of expression	Value of sysVar. ArrayIndex	Comment
"C" in myArray02D[1]	—	—	The expression is invalid because a reference to a multidimensional array must include a subscript for each dimension
"I" in myArray02D[3,2]	true	3	Search begins at the third row, second element
"G" in myArray02D[3,2]	false	0	Search ends at the last element of the row being reviewed
"G" in myArray02D[2,4]	false	0	The second subscript is greater than the number of columns available to search

Related tasks

"Syntax diagram" on page 506

Related reference

"Arrays" on page 64

"Logical expressions" on page 358

"Operators and precedence" on page 39

"sysVar.arrayIndex" on page 621

I/O error values

The next table describes the EGL error values for input/output (I/O) operations that affect databases, files, and MQSeries message queues. The values associated with hard errors are available to your code only if the system variable `sysVar.handleHardIOErrors` is set to 1, as described in *Exception handling*.

Error value	Type of error	Type of Record	Meaning of error value
deadLock	Hard	SQL	Two program instances are trying to change a record, but neither can do so without system intervention. If you are accessing an SQL table in DB2, deadlock indicates that the value of <code>sqlcode</code> is -911.
duplicate	Soft	Indexed or Relative	Your code tried to access a record having a key that already exists, and the attempt succeeded. For details, see <i>duplicate</i> .
endOfFile	Soft	Indexed, Relative, Serial	For details, see <i>endOfFile</i> .
ioError	Hard or Soft	Any	EGL received a non-zero return code from the I/O operation.
format	Hard	Any	The accessed file is incompatible with the record definition. For details, see <i>format</i> .

Error value	Type of error	Type of Record	Meaning of error value
fileNotAvailable	Hard	Any	fileNotAvailable is possible for any I/O operation and could indicate, for example, that another program is using the file or that resources needed to access the file are scarce.
fileNotFound	Hard	Indexed, Message queue, Relative, Serial	A file was not found.
full	Hard	Indexed, Relative, Serial	full is set in these cases: <ul style="list-style-type: none"> An indexed or serial file is full
hardIOError	Hard	Any	A hard error occurred, which is any error except endOfFile, noRecordFound, or duplicate.
noRecordFound	Soft	Any	For details, see <i>noRecordFound</i> .
unique	Hard	Indexed, Relative, or SQL	UNQ indicates <i>unique</i> : your code tried to add or replace a record having a key that already exists, and the attempt failed. For details, see <i>unique</i> .

duplicate

For an indexed or relative record, **duplicate** is set in these cases:

- An **add** statement tries to insert a record whose key or record ID already exists in the file or in an alternate index, and the insertion succeeds.
- A **replace** statement overwrites a record successfully, and the replacement values include a key that is the same as the alternate-index key of another record.
- A **get**, **get next**, or **get previous** statement reads a record successfully (or a **set** statement of the form *set record position* runs successfully), and a second record has the same key.

The **duplicate** setting is returned only if the access method returns the information, as is true on some operating systems but not on others. The option is not available during SQL database access.

If you are accessing an emulated VSAM file from an EGL-generated COBOL program on iSeries, see *Association elements* for a description of the **duplicates** property in the resource associations part that is used at generation time.

endOfFile

endOfFile is set in these conditions:

- Your code issues a **get next** statement for a serial or relative record when the related file pointer is at the end of the file. The pointer is at the end when the last record in the file was accessed by a previous **get** or **get next** statement.
- Your code issues a **get next** statement for an indexed record when the related file pointer is at the end of file, as occurs in these situations:
 - The last record in the file was accessed by a previous **get** or **get next** statement; or
 - The last record in the file was accessed by a previous **set** statement of type *set record position* when either of these conditions applies:

- The key value matched the key of the last record in the file; or
- Every byte in the key value was set to hexadecimal FF. (If a **set** statement of type *set record position* runs with a key value set to all hexadecimal FF, the statement sets the position pointer to the end of the file.)
- Your code issues a **get previous** statement for an indexed record when the related file pointer is at the beginning of file, as occurs in these situations:
 - The first record in the file was accessed by a previous **get** or **get previous** statement;
 - Your code did not previously access the same file; or
 - A **set** statement of type *set record position* ran with a key when no keys in the file were previous to that key.
- A **get next** statement attempts to retrieve data from an empty or uninitialized file into an indexed record.
(An empty file is one from which all records have been deleted. An uninitialized file is one that has never had any records added to it.)
- A **get previous** statement attempts to retrieve data from an empty file into an indexed record.
- In relation to COBOL generation, a **get previous** statement attempts to retrieve data from an uninitialized file into an indexed record.

format

format can result from any kind of I/O operation and could be set for these reasons, among others:

- **Record format**
The file format (fixed or variable length) is different from the EGL record format.
- **Record length**
In relation to fixed-length records, the length of a record in the file is different from the length of the EGL record. In relation to variable-length records, the length of a record in the file is larger than the length of the EGL record.
- **File type**
The file type specified for the record does not match the file type at run time.
- **Key length**
The key length in the file is different from the key length in the EGL indexed record.
- **Key offset**
The key position in the file is different from the key position in the EGL indexed record.

noRecordFound

noRecordFound is set in these conditions:

- For an indexed record, no record is found that matches the key specified in a **get** statement.
- For EGL-generated Java, your code issues a **get next** or **get previous** statement for an indexed record when the VSAM file is empty or uninitialized.
- For a relative record, no record is found that matches the record ID specified in a **get** statement. Alternatively, a **get next** statement tries to access a record that is beyond the end of the file.
- For a SQL record, no row is found that matches the specified SELECT statement; or a **get next** statement occurs when no selected rows are left to review.

unique

For an indexed or relative record, **unique** is set in these cases:

- An **add** statement tries to insert a record whose key or record ID already exists in the file or in an alternate index, and the insertion fails because of the duplication.
- A **replace** statement fails to overwrite a record because the replacement values include a key that is the same as the alternate-index key of another record.

The **unique** setting is returned only if the access method returns the information, as is true on some operating systems but not on others.

During SQL database access, **unique** is set when a SQL row being added or replaced has a key that already exists in a unique index. The corresponding sqlcode is -803.

Related reference

"add" on page 293

"Association elements" on page 231

"close" on page 303

"delete" on page 307

"Exception handling" on page 75

"execute" on page 309

"get" on page 318

"get next" on page 324

"get previous" on page 328

"Logical expressions" on page 358

"open" on page 335

"prepare" on page 339

"replace" on page 341

Java run-time properties

An EGL-generated Java program uses a set of run-time properties that provide information such as how to access the databases and files that are used by the program.

In a J2EE environment

In relation to a generated Java program that will run in a J2EE environment, these situations are possible:

- EGL can generate the run-time properties directly into a J2EE deployment descriptor. In this case, EGL overwrites properties that already exist and appends properties that do not exist. The program accesses the J2EE deployment descriptor at run time.
- Alternatively, EGL can generate the run-time properties into a J2EE environment file. You can customize the properties in that file, then copy them into the J2EE deployment descriptor.
- You can avoid generating the run-time properties at all, in which case you must write any needed properties by hand.

In a J2EE module, every program has the same run-time properties because all code in the module shares the same deployment descriptor.

In a non-J2EE Java environment

In relation to a generated Java program that runs outside of a J2EE environment, you can generate the run-time properties into a program properties file or code that file by hand. (The program properties file provides the kind of information that is available in the deployment descriptor, but the format of the properties is different.)

In a non-J2EE Java environment, properties can be specified in these two program properties files:

- **vgj.properties**; and
- A file named as follows--
programName.properties
programName
The first program in the run unit

If a same-named property is set in both files, the property in the file named for the program is used.

Use of **vgj.properties** is especially appropriate when the first program of a run unit does not access a file or database but calls programs that do:

- When generating the caller, you can generate a properties file named for the program, and the content might include no database- or file-related properties
- When you generate the called program, you can generate **vgj.properties**, and the content would be available for both programs

Neither file is mandatory, and simple programs do not need either one.

Build descriptors and program properties

Your choices are submitted to EGL as build-descriptor-option values:

- To generate properties into a J2EE deployment descriptor, set **J2EE** to YES; set **genProperties** to PROGRAM or GLOBAL; and generate into a J2EE project.
- To generate properties into a J2EE environment file, set **J2EE** to YES; set **genProperties** to PROGRAM or GLOBAL; and do either of these:
 - Generate into a directory (in which case you use the build descriptor option **genDirectory** rather than **genProject**); or
 - Generate into a non-J2EE project.
- To generate a program properties file with the same name as the program being generated, set **J2EE** to NO; set **genProperties** to PROGRAM; and generate into a project other than a J2EE project.
- To generate a program properties file **vgj.properties**, set **J2EE** to NO; set **genProperties** to GLOBAL; and generate into a project other than a J2EE project.
- To avoid generating properties, set **genProperties** to NO.

For additional information

For details on generating properties into a deployment descriptor or into a J2EE environment file, see *Setting deployment-descriptor values*.

For details on the meaning of the run-time properties, see Java run-time properties (reference).

Related concepts

"EGL debugger" on page 107
"J2EE environment file" on page 394
"Program properties file" on page 404
"Run unit" on page 504

Related tasks

"Generating into a directory" on page 220
"Generating into a project" on page 214
"Setting up the J2EE run-time environment for EGL-generated code" on page 206
"Setting deployment-descriptor values" on page 207
"Updating the deployment descriptor manually" on page 216
"Updating the J2EE environment file" on page 224

Related reference

"genProperties" on page 254
"J2EE" on page 255
"Java run-time properties (details)"

Java run-time properties (details)

The next table describes the properties that can be included in the deployment descriptor or program properties file, as well as the source of the value generated into the J2EE environment file, if any. The Java type for each property is `java.lang.String` unless the description column says otherwise.

Run-time property	Description	Source of the generated value
<code>cso.cicsj2c.timeout</code>	<p>Specifies the number of milliseconds before a timeout occurs during a call that uses protocol CICSJ2C. The default value is 30000, which represents 30 seconds. If the value is set to 0, no timeout occurs. The value must be greater than or equal to 0.</p> <p>The Java type in this case is <code>Java.lang.Integer</code>.</p> <p>The property has no effect on calls when the code is running in WebSphere 390; for details, see <i>Setting up the J2EE server for CICSJ2C calls</i>.</p>	Build descriptor option cicsj2cTimeout
<code>cso.linkageOptions.LO</code>	<p>Specifies the name of a linkage properties file that guides how the generated program or wrapper calls other programs. <i>LO</i> is the name of the linkage options part used at generation. For details, see <i>Deploying a linkage properties file</i>.</p>	<i>LO</i> is from the build descriptor option linkage

Run-time property	Description	Source of the generated value
tcpiplistener.port	Specifies the number of the port on which an EGL TCP/IP listener (of class CSOTcpipListener or CSOTcpipListenerJ2EE) listens. No default exists. For details, see <i>Setting up the TCP/IP listener</i> . The Java type in this case is Java.lang.Integer.	Not generated
tcpiplistener.trace.file	Specifies the name of the file in which to record the activity of one or more EGL TCP/IP listeners (each is of class CSOTcpipListener or CSOTcpipListenerJ2EE). The default file is tcpiplistener.out.	Not generated; tracing is only for use by IBM
tcpiplistener.trace.flag	Specifies whether to trace the activity of one or more EGL TCP/IP listeners (each of class CSOTcpipListener or CSOTcpipListenerJ2EE). Select one of these: <ul style="list-style-type: none"> • 1 for recording the activity into the file identified in property tcpiplistener.trace.flag • 0 (the default value) for not recording the activity The Java type in this case is Java.lang.Integer. For details, see <i>Setting up the TCP/IP listener</i> .	Not generated; tracing is only for use by IBM
vgj.datemask. gregorian.long.locale	Contains the date mask used in either of two cases: <ul style="list-style-type: none"> • The Java code generated for the system variable <i>sysVar.currentFormattedDate</i> is invoked; or • EGL validates a page item or text-form field that has a length of 10 or more, if the item property dateFormat is set to <i>systemGregorian</i>. <i>locale</i> is the code specified in property vgj.nls.code . In Web applications, you may change the date-mask property in use by assigning a different value to <i>sysLib.setLocale</i> .	Build descriptor value for the long Gregorian date mask; the default value is specific to the locale

Run-time property	Description	Source of the generated value
vgj.datemask. gregorian.short. <i>locale</i>	<p>Contains the date mask used when EGL validates a page item or text-form field that has a length of less than 10, if the item property dateFormat is set to <i>systemGregorian</i>.</p> <p><i>locale</i> is the code specified in property vgj.nls.code. In Web applications, you may change the date-mask property in use by assigning a different value to <code>sysLib.setLocale</code>.</p>	Build descriptor value for the short Gregorian date mask; the default value is specific to the locale
vgj.datemask. julian.long. <i>locale</i>	<p>Contains the date mask used in either of two cases:</p> <ul style="list-style-type: none"> • The Java code generated for the system variable <code>sysVar.currentFormattedJulianDate</code> is invoked; or • EGL validates a page item or text-form field that has a length of 10 or more, if the item property dateFormat is set to <i>systemJulian</i>. <p><i>locale</i> is the code specified in property vgj.nls.code. In Web applications, you may change the date-mask property in use by assigning a different value to <code>sysLib.setLocale</code>.</p>	Build descriptor value for the long Julian date mask; the default value is specific to the locale
vgj.datemask. julian.short. <i>locale</i>	<p>Contains the date mask used when EGL validates a page item or text-form field that has a length of less than 10, if the item property dateFormat is set to <i>systemJulian</i>.</p> <p><i>locale</i> is the code specified in property vgj.nls.code. In Web applications, you may change the date-mask property in use by assigning a different value to <code>sysLib.setLocale</code>.</p>	Build descriptor value for the short Julian date mask; the default value is specific to the locale

Run-time property	Description	Source of the generated value
vgj.jdbc.database. <i>SN</i>	<p>Specifies the JDBC database name that is used when a database connection is made by way of the system function sysLib.connect or sysLib.connectionService.</p> <p>The meaning of the value is different for J2EE connections as compared with standard (non-J2EE) connections:</p> <ul style="list-style-type: none"> • In relation to J2EE connections (as is needed in a production environment), the value is the name to which the datasource is bound in the JNDI registry; for example, jdbc/MyDB • In relation to a standard JDBC connection (as may be used for debugging), the value is the connection URL; for example, jdbc:db2:MyDB <p>You must customize the name of the property itself when you specify a substitution value for <i>SN</i>, at deployment time. The substitution value in turn must match either the server name that is included in the invocation of sysLib.connectionService or the database name that is included in the invocation of sysLib.connect.</p>	Build descriptor value for the database name that you want to associate with the specified "server name"

Run-time property	Description	Source of the generated value
vgj.jdbc.default.database. <i>programName</i>	<p>Specifies the default database name that is used for an SQL I/O operation if no prior database connection exists. EGL includes the program name (or program alias, if any) as a substitution value for <i>programName</i> so that each program has its own default database. The program name is optional, however, and a property named <code>vgj.jdbc.default.database</code> is used as a default for any program that is not referenced in a program-specific property of this kind.</p> <p>The meaning of the value in the property itself is different for J2EE connections as compared with non-J2EE connections:</p> <ul style="list-style-type: none"> • In relation to J2EE connections, the value is the name to which the datasource is bound in the JNDI registry; for example, <code>jdbc/MyDB</code> • In relation to a standard JDBC connection, the value is the connection URL; for example, <code>jdbc:db2:MyDB</code> 	<p>Depends on the connection type:</p> <ul style="list-style-type: none"> • For J2EE connections, build descriptor option sqlJNDIName • For non-J2EE connections, build descriptor option sqlDB
vgj.jdbc.default.password	<p>Specifies the password for accessing the database connection identified in vgj.jdbc.default.database.</p> <p>To avoid exposing passwords in the J2EE environment file, do one of the following tasks:</p> <ul style="list-style-type: none"> • Specify a password in program and function scripts by using the system function <code>sysLib.connect</code> or <code>sysLib.connectionService</code>; or • Include a user ID and password in the datasource specification in the web application server, as described in <i>Setting up a J2EE JDBC connection</i>. 	Build descriptor option sqlPassword
vgj.jdbc.default.userid	Specifies the userid for accessing the database connection identified in vgj.jdbc.default.database .	Build descriptor option sqlID
vgj.jdbc.drivers	Specifies the driver class for accessing the database connection identified in vgj.jdbc.default.database . This property is not present in the deployment descriptor or J2EE environment file and is used only for a standard (non-J2EE) JDBC connection.	Build descriptor option sqlJDBCClass

Run-time property	Description	Source of the generated value
vgj.nls.code	<p>Specifies the three-letter NLS code of the program. For a list of valid values, see <code>targetNLS</code>.</p> <p>If the property is not set, these rules apply:</p> <ul style="list-style-type: none"> • The value defaults to the NLS code that corresponds to the default Java locale • The value is <code>ENU</code> if the default Java locale does not correspond to any of the NLS codes supported by EGL 	Build descriptor option targetNLS
vgj.nls.currency	Specifies the character used as a currency symbol. The default is determined by the locale associated with vgj.nls.code .	Build descriptor option currencySymbol
vgj.nls.number.decimal	Specifies the character used as a decimal symbol. The default is determined by the locale associated with vgj.nls.code .	Build descriptor option decimalSymbol
vgj.ra.QN.conversionTable	Specifies the name of the conversion table used by a generated Java program during access of the MQSeries message queue identified by <i>QN</i> . Valid values are <code>programControlled</code> , <code>NONE</code> , or a conversion table name. The default is <code>NONE</code> .	Resource associations property conversionTable
vgj.ra.FN.fileType	<p>Specifies the type of file associated with <i>FN</i>, which is a file or queue name identified in the record part. The property value is <code>seqws</code> or <code>mq</code>, as described in <i>Record and file type cross-reference</i>.</p> <p>You must specify this deployment descriptor property for each logical file that your program uses.</p>	Resource associations property fileType
vgj.ra.FN.replace	<p>Specifies the effect of an add statement on a record associated with <i>FN</i>, which is a file name identified in a record. Select one of two values:</p> <ul style="list-style-type: none"> • 1 if the statement replaces the file record • 0 (the default) if the statement appends a record to the file <p>The Java type in this case is <code>java.lang.Integer</code>.</p>	Resource associations property replace

Run-time property	Description	Source of the generated value
vgj.ra.FN.systemName	<p>Specifies the name of the physical file or message queue associated with <i>FN</i>, which is a file or queue name identified in the record part.</p> <p>You must specify this deployment descriptor property for each logical file that your program uses.</p>	Resource associations property systemName
vgj.ra.FN.text	<p>Specifies whether to cause a generated Java program to do the following when accessing a file by way of a serial record:</p> <ul style="list-style-type: none"> • Append end-of-line characters during the add operation. On non-UNIX platforms, those characters are a carriage return and linefeed; on UNIX platforms, the only character is a linefeed. • Remove end-of-line characters during the get next operation. <p><i>FN</i> is the file name associated with the serial record.</p> <p>Select one of these values:</p> <ul style="list-style-type: none"> • 1 for make the changes • 0 (the default) for refrain from making the changes <p>The Java type in this case is <code>java.lang.Integer</code>.</p>	Resource associations property text
vgj.trace.device.option	<p>Destination of trace data, if any. Select one of these values:</p> <ul style="list-style-type: none"> • 0 for write to <code>System.out</code> • 1 for write to <code>System.err</code> • 2 (the default) for write to the file specified in vgj.trace.device.spec with this exception: for VSAM I/O traces, write to <code>vsam.out</code> <p>The Java type in this case is <code>java.lang.Integer</code>.</p>	The generated value, if any, is 2
vgj.trace.device.spec	<p>Specifies the name of the output file if vgj.trace.device.option is set to 2. An exception is that VSAM I/O traces are written to <code>vsam.out</code>.</p>	The generated value, if any, is <code>vgjtrace.out</code>

Run-time property	Description	Source of the generated value
vgj.trace.type	<p>Specifies the run-time trace setting. Sum the values of interest to get the tracing you want:</p> <ul style="list-style-type: none"> • -1 for trace all • 0 for no trace (the default) • 1 for general trace, including function invocations and call statements • 2 for system functions that handle math • 4 for system functions that handle strings • 16 for data passed on a call statement • 32 for the linkage options used on a call • 128 for jdbc I/O • 256 for file I/O • 512 for all the properties except vgj.jdbc.default.password <p>The Java type in this case is java.lang.Integer.</p>	The generated value, if any, is 0

Related concepts

“Java run-time properties” on page 421

“Linkage properties file” on page 404

Related tasks

“Deploying a linkage properties file” on page 223

“Setting up a J2EE JDBC connection” on page 209

“Setting up the J2EE run-time environment for EGL-generated code” on page 206

“Setting up the TCP/IP listener” on page 210

“Understanding how a standard JDBC connection is made” on page 208

Related reference

“callLink element” on page 443

“cicsj2cTimeout” on page 244

“decimalSymbol” on page 246

“linkage” on page 256

“Linkage properties file (details)” on page 431

“Record and file type cross-reference” on page 159

“sqlDB” on page 262

“sqlID” on page 263

“sqlJDBCClass” on page 263

“sqlJNDIName” on page 264

“sqlPassword” on page 264

“sysLib.connect” on page 543

“sysLib.connectionService” on page 545

“sysLib.setLocale” on page 615

“sysVar.currentFormattedDate” on page 522

"sysVar.currentFormattedJulianDate" on page 523
"sysVar.currentShortDate" on page 525
"sysVar.currentShortJulianDate" on page 525
"targetNLS" on page 267

Linkage properties file (details)

When you generate a calling Java program or wrapper, you can specify that linkage information is required at run time. You make that specification by setting the linkage-option values for the called program as follows:

- The value of the callLink element property **type** is remoteCall or ejbCall; and
- The value of the callLink element property **remoteBind** is RUNTIME.

A linkage properties file may be handwritten, but EGL generates a file if (in addition to the settings described earlier) you generate a Java program or wrapper with the build descriptor option **genProperties** set to GLOBAL or PROGRAM.

How the linkage properties file is identified at run time

If the callLink element property **remoteBind** for a called program was set to RUNTIME in the linkage options part, the linkage properties file is sought at run time; but the source of the file name is different for Java programs and Java wrappers:

- A Java program checks the Java run-time property **cso.linkageOptions.LO**, where *LO* is the name of the linkage options part used for generation. If the property is not present, the EGL run-time code seeks a linkage properties file named **LO.properties**. Again, *LO* is the name of the linkage options part used for generation.

In this case, if the EGL run-time code seeks a linkage properties file but is unable to find that file, an error occurs on the first call statement that requires use of that file. For details on the result, see Exception handling.

- The Java wrapper stores the name of the linkage properties file in the program object variable *callOptions*, which is of type CSOCallOptions. The generated name of the file is **LO.properties**, where *LO* is the name of the linkage options part used for generation.

In this case, if the Java Virtual Machine seeks a linkage properties file but is unable to find that file, the program object throws an exception of type CSOException.

Format of the linkage properties file

As used during run time, the linkage properties file includes a series of entries to handle each call from the generated Java program or wrapper that you are deploying.

The primary entry is of type **cso.serverLinkage** and can include any property-and-value pair that you can set in a callLink element of the linkage options part, with the following exceptions:

- Property **remoteBind** is necessarily RUNTIME and should not appear
- Property **type** cannot be localCall, because linkage for local calls must be established at generation time

cso.serverLinkage entries: In the most elementary case, each entry in the linkage properties file is of type **cso.serverLinkage**. The format of the entry is as follows:

`cso.serverLinkage.programName.property=value`

programName

The name of the called program. If the called program is generated by EGL, the name you specify is that of a program part.

property

Any of the properties appropriate for a Java program, except for properties **remoteBind** and **pgmName**. For details, see *callLink element*.

value

A value that is valid for the specified property.

An example for called program *XYZ* is as follows, where *xxx* refers to a case-sensitive string:

```
cso.serverLinkage.Xyz.type=ejbCall
cso.serverLinkage.Xyz.remoteComType=TCPIP
cso.serverLinkage.Xyz.remotePgmType=EGL
cso.serverLinkage.Xyz.externalName=xxx
cso.serverLinkage.Xyz.package=xxx
cso.serverLinkage.Xyz.conversionTable=xxx
cso.serverLinkage.Xyz.location=xxx
cso.serverLinkage.Xyz.serverID=xxx
cso.serverLinkage.Xyz.parmForm=COMMDATA
cso.serverLinkage.Xyz.providerURL=xxx
cso.serverLinkage.Xyz.luwControl=CLIENT
```

The literal values TCPIP, EGL, and so on are not case sensitive and are examples of valid data.

cso.application entries: If you wish to create a series of *cso.serverLinkage* entries that refer to any of several called programs, precede those entries with one or more entries of type *cso.application*. Your purpose in this case is to equate a single application name to multiple program names. In the subsequent *cso.serverLinkage* entries, you use the application name instead of *programName*; then, at Java run time, those *cso.serverLinkage* entries handle calls to any of several programs.

The format of a *cso.application* entry is as follows:

```
cso.application.wildProgramName.appName
```

wildProgramName

A valid program name, an asterisk, or the beginning of a valid program name followed by an asterisk. The asterisk is the wild-card equivalent of one or more characters and provides a way to identify a set of names.

If *wildProgramName* refers to a program that is generated by EGL, any program name included in *wildProgramName* is the name of a program part.

appName

A series of characters that conforms to the EGL naming conventions. The value of *appName* is used in subsequent *cso.serverLinkage* entries.

The following example show use of an asterisk as a wild-card character. The *cso.serverLinkage* entries in this example handle any call to a program whose name begins with *XYZ*:

```
cso.application.Xyz*=myApp
cso.serverLinkage.myApp.type=remoteCall
cso.serverLinkage.myApp.remoteComType=TCPIP
cso.serverLinkage.myApp.remotePgmType=EGL
cso.serverLinkage.myApp.externalName=xxx
cso.serverLinkage.myApp.package=xxx
cso.serverLinkage.myApp.conversionTable=xxx
cso.serverLinkage.myApp.location=xxx
```

```
cso.serverLinkage.myApp.serverID=xxx  
cso.serverLinkage.myApp.parmForm=COMMDATA  
cso.serverLinkage.myApp.luwControl=CLIENT
```

The following example shows use of the same `cso.serverLinkage` entries to handle calls to any of several programs, even though the names of those programs do not begin with the same characters:

```
cso.application.Abc=myApp  
cso.application.Def=myApp  
cso.application.Xyz=myApp  
cso.serverLinkage.myApp.type=remoteCall  
cso.serverLinkage.myApp.remoteComType=TCPIP  
cso.serverLinkage.myApp.remotePgmType=EGL  
cso.serverLinkage.myApp.externalName=xxx  
cso.serverLinkage.myApp.package=xxx  
cso.serverLinkage.myApp.conversionTable=xxx  
cso.serverLinkage.myApp.location=xxx  
cso.serverLinkage.myApp.serverID=xxx  
cso.serverLinkage.myApp.parmForm=COMMDATA  
cso.serverLinkage.myApp.luwControl=CLIENT
```

If multiple `cso.application` entries are valid for a program, EGL uses the first entry that applies.

Related concepts

“Linkage options part” on page 439

“Linkage properties file” on page 404

Related tasks

“Editing the `callLink` element of a linkage options part” on page 96

“Setting up the J2EE run-time environment for EGL-generated code” on page 206

Related reference

“`callLink` element” on page 443

“Exception handling” on page 75

“Java run-time properties (details)” on page 423

“Naming conventions” on page 468

Library part

A *library part* contains a set of functions, variables, and constants that can be used by programs, page handlers, or other libraries. It is recommended that you use libraries to maximize your reuse of common code and values.

Rules are as follows:

- Libraries are available only if you generating Java output.
- You can reference a library’s functions, variables, and constants without specifying the library name, but only if you include the library in a program-specific Use declaration.
- Library functions can access any system variables that are associated with the invoking program or page handler.
- Library functions can use any statements except these:
 - converse
 - display
 - forward

- show
- transfer
- A library that accesses a text or print form must include a use declaration for the related form group. The form can be presented only in non-segmented mode because the library cannot go out of memory except as described later.
- You can use the modifier **private** on a function, variable, or constant declaration to keep the element from being used outside the library.
- Library functions that are declared as public (as is the default) are available outside the library and cannot have parameters that are of a loose type, which is a special kind of primitive type that is available only if you wish the parameter to accept a range of argument lengths. For details on the loose type, see *Function part in EGL source format*.

The library is generated separately from the parts that use it. At run time, the library is loaded when first used and is unloaded when the program or page handler that accessed the library leaves memory, as occurs when the run unit ends or when a program (if segmented) issues a converse statement. .

A page handler gets a new copy of the library whenever the page handler is loaded. Also, a library that is invoked by another library remains in memory as long as the invoking library does.

In EGL-generated Java code, a library that is used only for its constants is not loaded at run time because constants are generated as literals in the programs and page handlers that reference them.

Related reference

“converse” on page 306

“forward” on page 315

“Function part in EGL source format” on page 381

“Run unit” on page 504

“Segmentation in text applications” on page 151

“show” on page 353

“transfer” on page 354

“Use declaration” on page 631

Library part in EGL source format

You declare a library part in an EGL file, which is described in *EGL source format*.

An example of a library part is as follows:

```
Library CustomerLib3
```

```
// Use declarations
Use StatusLib;
```

```
// Data Declarations
exceptionId ExceptionId ;
```

```
// Retrieve one customer for an email
// In: customer, with emailAddress set
// Out: customer, status
```

```
Function getCustomerByEmail ( customer CustomerForEmail, status int )
    status = StatusLib.success;
    try
        get customer ;
    onException
```

```

        exceptionId = "getCustomerByEmail" ;
        status = sqlCode ;
    end
    commit();
end

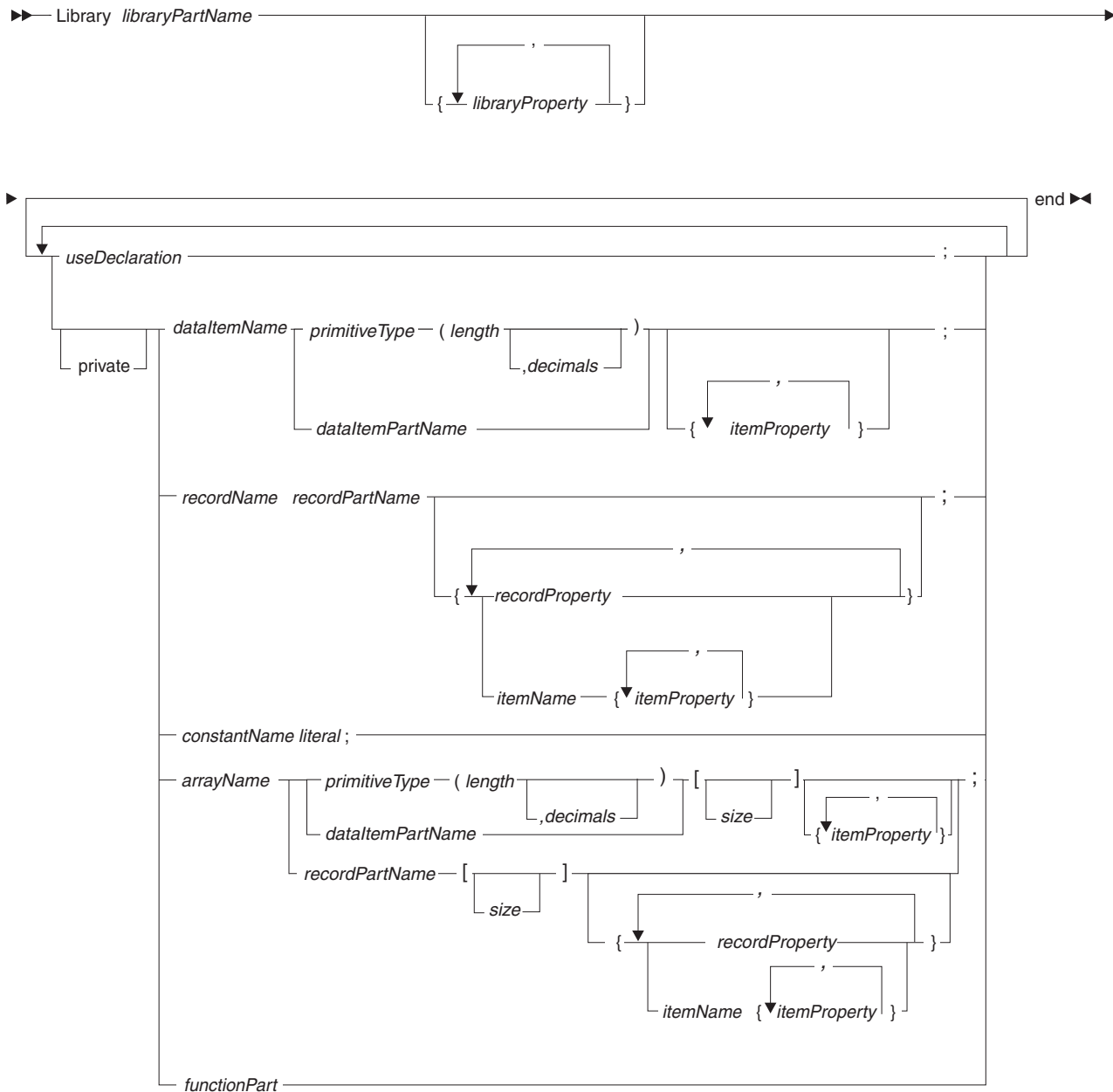
// Retrieve one customer for a customer ID
//   In: customer, with customer ID set
//   Out: customer, status
Function getCustomerByCustomerId ( customer Customer, status int )
    status = StatusLib.success;
    try
        get customer ;
    onException
        exceptionId = "getCustomerByCusomerId" ;
        status = sqlCode ;
    end
    commit();
end

// Retrieve multiple customers for an email
//   In: startId
//   Out: customers, status
Function getCustomersByCustomerId
( startId CustomerId, customers Customer[], status int )
    status = StatusLib.success;
    try
        get customers usingKeys startId ;
    onException
        exceptionId = "getCustomerForEmail" ;
        status = sqlCode ;
    end
    commit();
end

end

```

The diagram of a library part is as follows:



Library libraryPartName ... end

Identifies the part as a library part and specifies the name. If you do not set the **alias** property (as described later), the name of the generated library is either *libraryPartName* or, if you are generating COBOL, the first eight characters of *libraryPartName*.

For other rules, see *Naming conventions*.

libraryProperties

The library properties are as follows:

- **alias**
- **allowUnqualifiedItemReferences**
- **includeReferencedFunctions**
- **messageTablePrefix**

All are optional:

- **alias** = *alias* identifies a string that is incorporated into the names of generated output. If you do not set the **alias** property, the program-part name (or a truncated version) is used instead.
- **allowUnqualifiedItemReferences** = **no**, **allowUnqualifiedItemReferences** = **yes** specifies whether to allow your code to reference structure items but to exclude the name of the *container*, which is the data table, record, or form that holds the structure item. Consider the following record part, for example:

```
Record aRecordPart type basicRecord
  10 myItem01 CHAR(5);
  10 myItem02 CHAR(5);
end
```

The following variable is based on that part:

```
myRecord aRecordPart;
```

If you accept the default value of **allowUnqualifiedItemReferences** (*no*), you must specify the record name when referring to myItem01, as in this assignment:

```
myValue = myRecord.myItem01;
```

If you set the property **allowUnqualifiedItemReferences** to *yes*, however, you can avoid specifying the record name:

```
myValue = myItem01;
```

It is recommended that you accept the default value, which promotes a best practice. By specifying the container name, you reduces ambiguity for people who read your code and for EGL.

EGL uses a set of rules to determine the area of memory to which a variable name or item name refers. For details, see *References to variables and constants*.

- **includeReferencedFunctions** = **no**, **includeReferencedFunctions** = **yes** indicates whether the library contains a copy of each function that is neither inside the library nor in a library accessed by the current library. The default value is *no*, which means that you can ignore this property if all functions that are to be part of this library are inside the library.
If the library is using shared functions that are not in the library, generation is possible only if you set the property **includeReferencedFunctions** to *yes*.
- **msgTablePrefix** = *prefix* specifies the first one to the four characters in the name of a data table that is used as a message table. (The message table is available to forms that are output by library functions.) The other characters in the name correspond to one of the national language codes listed in *DataTable part in EGL source format*.

useDeclaration

Provides easier access to a data table or library, and is needed to access forms in a form group. For details, see *Use declaration*.

private

Indicates that the variable, constant, or function is unavailable outside the library. If you omit the term **private**, the variable, constant, or function is available.

dataItemName

Name of a data item. For the rules of naming, see *Naming conventions*.

primitiveType

The primitive type of a data item or (in relation to an array) the primitive type of an array element.

length

The parameter's length or (in relation to an array) the length of an array element. The length is an integer that represents the number of characters or digits in the memory area referenced either by *dataItemName* or (in the case of an array) *dynamicArrayName*.

decimals

For a numeric type (BIN, DECIMAL, NUM, NUMC, or PACF), you may specify *decimals*, which is an integer that represents the number of places after the decimal point. The maximum number of decimal positions is the smaller of two numbers: 18 or the number of digits declared as *length*. The decimal point is not stored with the data.

dataItemPartName

The name of a dataItem part that is visible to the program. For details on visibility, see *References to parts*.

The part acts as a model of format, as described in *Typedef*.

recordName

Name of a record. For the rules of naming, see *Naming conventions*.

recordPartName

Name of a record part that is visible to the program. For details on visibility, see *References to parts*.

The part acts as a model of format, as described in *Typedef*.

constantName literal

Name and value of a constant. The value is either a quoted string or a number. For the rules of naming, see *Naming conventions*.

itemProperty

An item-specific property-and-value pair, as described in *Overview of EGL properties and overrides*.

recordProperty

A record-specific property-and-value pair. For details on the available properties, see the reference topic for the record type of interest.

A basic record has no properties.

itemName

Name of a record item whose properties you wish to override. See *Overview of EGL properties and overrides*.

arrayName

Name of a dynamic or static array of records or data items. If you use this option, the other symbols to the right (*dataItemPartName*, *primitiveType*, and so on) refer to each element of the array.

size

Number of elements in the array. If you specify the number of elements, the array is static; otherwise, the array is dynamic.

functionPart

A function. No parameter in the function can be of a loose type. For details, see *Function part in EGL source format*.

Related concepts

"EGL projects, packages, and files" on page 7

"Library part" on page 433

"Overview of EGL properties and overrides" on page 40

"References to parts" on page 16
"References to variables and constants" on page 34
"Typedef" on page 20

Related reference

"Basic record part in EGL source format" on page 491
"DataTable part in EGL source format" on page 287
"EGL source format" on page 292
"Function part in EGL source format" on page 381
"Indexed record part in EGL source format" on page 492
"Input form" on page 486
"Input record" on page 487
"I/O error values" on page 418
"MQ record part in EGL source format" on page 493
"Primitive types" on page 27
"Relative record part in EGL source format" on page 495
"Serial record part in EGL source format" on page 497
"SQL record part in EGL source format" on page 498

"Use declaration" on page 631

Linkage options part

A *linkage options* part specifies details on the following issues:

- How a generated Java program or wrapper calls other generated code
- How a generated COBOL program calls and is called by other generated code
- How a generated Java or COBOL program transfers asynchronously to another generated program
- How a generated COBOL program transfers control and ends processing.

Specifying when linkage options are final

For a generated COBOL program, the linkage options specified at generation time are in effect at run time. For generated Java code, you can choose between two alternatives:

- The linkage options specified at generation time are in effect at run time; or
- The linkage options specified in a linkage properties file at deployment time are in effect at run time. Although you can write that file by hand, EGL generates it in this situation:
 - You set the linkage options property **remoteBind** to **RUNTIME**; and
 - You generate a Java program or wrapper with the build descriptor option **genProperties** set to **GLOBAL** or **PROGRAM**.

For details on using the file, see *Deploying a linkage properties file*. For details on customizing the file, see *Linkage properties file (reference)*.

Elements of a linkage options part

The linkage options part is composed of a set of elements, each of which has a set of properties and values. The following types of elements are available:

- A **callLink** element specifies the linkage conventions that EGL uses for a given call.

If you are generating a COBOL program, the following relationships are in effect:

- If the `callLink` element refers to the generated program, that element determines aspects of the program's own parameters; for example, whether the program expects pointers to data or expects the data itself. Also, that element helps determine whether to generate a Java wrapper that allows access to the COBOL code from native Java code; for an overview, see *Java wrapper*.
- If the `callLink` element refers to a program being called by the generated program, that element specifies how the call is implemented; for example, whether the call is local or remote.

If you are generating Java code, the `callLink` element always applies to a called program. The following relationships are in effect:

- If the `callLink` element refers to the program that you are generating, that element helps determine whether to generate a Java wrapper that allows access to the program from native Java code; for an overview, see *Java wrapper*. If you indicate that the Java wrapper accesses the program by way of an EJB session bean, the `callLink` element also causes generation of an EJB session bean.
- If you are generating a Java program and if the `callLink` element refers to a program that is called by that program, the `callLink` element specifies how the call is implemented; for example, whether the call is local or remote. If you indicate that the calling Java program makes the call through an EJB session bean, the `callLink` element causes generation of an EJB session bean.
- An *asynchLink* element specifies how a generated Java or COBOL program transfers asynchronously to another program, as occurs when the transferring program invokes the system function `sysLib.startTransaction`.
- A *transferToProgram* element specifies how a generated COBOL program transfers control to a program and ends processing. This element is not used for Java output and is meaningful only for a main program that issues a **transfer** statement of the type *transfer to program*.
- A *transferToTransaction* element specifies how a generated program transfers control to a transaction and ends processing. This element is meaningful only for a main program that issues a **transfer** statement of the type *transfer to transaction*. The element is unnecessary, however, when the target program is generated with VisualAge Generator or (in the absence of an alias) with EGL.

Identifying the programs or records to which an element refers

In each element, a property (for example, **pgmName**) identifies the programs or records to which the element refers; and unless otherwise stated, the value of that property can be a valid name, an asterisk, or the beginning of a valid name followed by an asterisk. The asterisk is the wild-card equivalent of one or more characters and provides a way to identify a set of names.

Consider a `callLink` element that includes the following value for the **pgmName** property:

```
myProg*
```

That element pertains to any EGL program part that begins with the letters *myProg*.

If multiple elements are valid, EGL uses the first element that applies. A series of `callLink` elements, for example, might be characterized by these **pgmName** values, in order:

```
YourProgram
YourProg*
*
```

Consider the element associated with the last value, where the value of **pgmName** is only an asterisk. Such an element could apply to any program; but in relation to a particular program, the last element applies only if the previous elements do not. If your program calls `YourProgram01`, for instance, the linkage specified in the second element (`YourProg*`) supersedes the third element (`*`) to define how EGL handles the call.

In most cases, elements with more specific names should precede those with more general names. In the previous example, the element with the asterisk is appropriately positioned to provide the default linkage specifications.

Related concepts

“Java wrapper” on page 395
“Parts” on page 11

Related tasks

“Adding a linkage options part” on page 95
“Deploying a linkage properties file” on page 223
“Editing the `asynchLink` element of a linkage options part” on page 97
“Editing the `callLink` element of a linkage options part” on page 96

“Editing the transfer-related elements of a linkage options part” on page 98
“Setting up the J2EE run-time environment for EGL-generated code” on page 206

Related reference

“`asynchLink` element”
“`call`” on page 299
“`callLink` element” on page 443

“`linkage`” on page 256
“Linkage properties file (details)” on page 431

“`sysLib.startTransaction`” on page 617
“`transfer`” on page 354
“`transferToProgram` element” on page 459
“`transferToProgram` element” on page 459

asynchLink element

An *asynchLink* element of a linkage options part specifies how a generated Java or COBOL program invokes another program asynchronously, as occurs when the originating program invokes the system function `sysLib.startTransaction`.

You can avoid specifying an *asynchLink* element if you accept the default behavior, which assumes that the created transaction is to be started from the same Java package.

Each element includes the property `recordName`, which references a record that is also referenced in the specific **`sysLib.startTransaction`** function whose action is being modified.

For Java programs, the other property is **package**, which is needed only if the source for the invoked program is in a package that is different from the invoker's package.

Related concepts

"Linkage options part" on page 439

Related reference

"package in asynchLink element"

"recordName in asynchLink element"

package in asynchLink element

The linkage options part, asynchLink element, property **package** is valid only for Java output and specifies the name of the package that contains the program being invoked. The default is the package of the invoking program.

The package name that is used in generated Java programs is the package name of the EGL program, but in lower case; and when EGL generates output from the asynchLink element, the value of **package** is changed (if necessary) to lower case.

Related concepts

"Linkage options part" on page 439

Related reference

"asynchLink element" on page 441

"recordName in asynchLink element"

recordName in asynchLink element

The linkage options part, asynchLink element, property **recordName** specifies the name of the record that is used in the system function sysLib.startTransaction. In this case, the record name is used to identify which program or transaction is associated with the asynchLink element.

You can use an asterisk (*) as a global substitution character in the record name; however, that character is valid only as the last character. For details, see *Linkage options part*.

Related concepts

"Linkage options part" on page 439

Related reference

"asynchLink element" on page 441

"package in asynchLink element"

"sysLib.startTransaction" on page 617

callLink element

The callLink element of a linkage options part specifies the type of linkage used in a call. Each element includes these properties:

- pgmName
- type

The value of the **type** property determines what additional properties are available, as shown in the next sections:

- “If callLink type is localCall (the default)”
- “If callLink type is remoteCall”
- “If callLink type is ejbCall” on page 444

If callLink type is localCall (the default)

Set property **type** to localCall when you are generating a Java program that calls a generated Java program that resides in the same thread. In this case, EGL middleware is not in use, and the following properties are meaningful for a callLink element in which **pgmName** identifies the called program--

- “alias in callLink element” on page 445
- “package in callLink element” on page 451
- “pgmName in callLink element” on page 453
- “type in callLink element” on page 458

You do not need to specify a callLink element for the call if the called program is in the same package as the caller and if either of these conditions is in effect:

- You do not specify an external name for the called program; or
- The external name for the called program is identical to the part name for that program.

The value of **type** cannot be localCall when you are generating a Java wrapper.

If callLink type is remoteCall

Set property **type** to remoteCall when you are generating a Java program or wrapper, and the Java code calls a program that runs in a different thread. The call is not by way of a generated EJB session bean. In this case, EGL middleware is in use, and the following properties are meaningful for a callLink element in which **pgmName** identifies the called program--

- “alias in callLink element” on page 445
- “conversionTable in callLink element” on page 445
- “location in callLink element” on page 449
- “package in callLink element” on page 451 (used only if the generated code is calling a Java program that is stored in another package)
- “pgmName in callLink element” on page 453
- “remoteBind in callLink element” on page 454
- “remoteComType in callLink element” on page 455
- “remotePgmType in callLink element” on page 457
- “serverID in callLink element” on page 458
- “type in callLink element” on page 458

If callLink type is ejbCall

Set property **type** to `ejbCall` when a `callLink` element is required to handle either of the following situations:

- You are generating a Java wrapper and intend to call the related, generated program by way of a generated EJB session bean
- You are generating a Java program and intend to call another generated program by way of a generated EJB session bean

In this case, EGL middleware is in use, and the following properties are meaningful for a `callLink` element in which **pgmName** identifies the called program:

- “alias in callLink element” on page 445
- “conversionTable in callLink element” on page 445
- “location in callLink element” on page 449
- “package in callLink element” on page 451 (used only if the generated Java code is calling a Java program that is stored in a package other than the package in which the EJB session bean resides)
- “parmForm in callLink element” on page 452 (used only if the generated Java code is calling a program that runs on CICS)
- “pgmName in callLink element” on page 453
- “providerURL in callLink element” on page 453
- “remoteBind in callLink element” on page 454
- “remoteComType in callLink element” on page 455
- “remotePgmType in callLink element” on page 457
- “serverID in callLink element” on page 458
- “type in callLink element” on page 458

Related concepts

“Linkage options part” on page 439

“Run-time configurations” on page 2

Related tasks

“Editing the callLink element of a linkage options part” on page 96

Related reference

“alias in callLink element” on page 445

“conversionTable in callLink element” on page 445

“linkType in callLink element” on page 448

“location in callLink element” on page 449

“package in callLink element” on page 451

“pgmName in callLink element” on page 453

“providerURL in callLink element” on page 453

“remoteBind in callLink element” on page 454

“remoteComType in callLink element” on page 455

“remotePgmType in callLink element” on page 457

“serverID in callLink element” on page 458

“type in callLink element” on page 458

alias in callLink element

The linkage options part, callLink element, property **alias** specifies the run-time name of the program identified in property **pgmName**. The property is meaningful only when **pgmName** refers to a program that is called by the program being generated.

The value of this property must match the alias (if any) you specified when declaring the program. If you did not specify an alias when declaring the program, either set the callLink element property **alias** to the name of the program part or do not set the property at all.

Related concepts

"Linkage options part" on page 439

Related tasks

"Editing the callLink element of a linkage options part" on page 96

Related reference

"callLink element" on page 443

"pgmName in callLink element" on page 453

conversionTable in callLink element

The linkage options part, callLink element, property **conversionTable** specifies the name of the conversion table that is used to convert data on a call. The property is meaningful only when **pgmName** identifies a program that is called by the generated program or wrapper.

When you generate a COBOL program, these details are in effect:

- The property **conversionTable** is useful only in this case--
 - The call is to a non-EGL-generated program
 - The called program runs on a platform that supports the ASCII character set
- The property is available only if the value of property **type** is `remoteCall`
- The default is that no conversion occurs

When you generate a Java program or wrapper, the following details are in effect:

- When the call is to a non-Java program, a default conversion occurs in accordance with the character set (ASCII or EBCDIC) used on the calling platform. You must specify a value for **conversionTable** in the following case--
 - The caller is Java code and is on a machine that supports one character set (EBCDIC or ASCII); and
 - The called program is non-Java and is on a machine that supports the other character set.
- An attempt to specify a conversion table has no effect when EGL-generated Java code calls a Java program, except in the case of bidirectional text.
- The property **conversionTable** is available only if the value of property **type** is `ejbCall` or `remoteCall`.

Select one of the following values:

conversion table name

The caller uses the conversion table specified. For a list of tables, see *Data conversion*.

- * Uses the default conversion table. For a COBOL client, the name of that table is

ELAx_{xxx}, where the value of *xxx* is the value of build descriptor option `targetNLS`. For a Java client, the selected table is based either on the locale of the client machine or (if the client is running on a Web application server) on the locale of that server. If an unrecognized locale is found, English is assumed.

For a list of tables, see *Data conversion*.

programControlled

The caller uses the conversion table name that is in the system item `sysVar.callConversionTable` at run time. If `sysVar.callConversionTable` contains blanks, no conversion occurs.

Related concepts

"Linkage options part" on page 439

Related tasks

"Editing the `callLink` element of a linkage options part" on page 96

Related reference

"Bidirectional language text" on page 282

"`callLink` element" on page 443

"Data conversion" on page 279

"`pgmName` in `callLink` element" on page 453

"`sysLib.convert`" on page 518

"`targetNLS`" on page 267

"`type` in `callLink` element" on page 458

ctgKeyStore in callLink element

The linkage options part, `callLink` element, property **ctgKeyStore** is the name of the key store generated with the Java tool `keytool.exe` or with the CICS Transaction Gateway tool `IKEYMAN`. This property is required when the value of property **remoteComType** is set to `CICSSSL`.

Related concepts

"Linkage options part" on page 439

Related reference

"`callLink` element" on page 443

"`ctgKeyStorePassword` in `callLink` element"

"`remoteComType` in `callLink` element" on page 455

ctgKeyStorePassword in callLink element

The linkage options part, `callLink` element, property **ctgKeyStorePassword** is the password used when generating the key store.

Related concepts

"Linkage options part" on page 439

Related reference

"`callLink` element" on page 443

"`ctgKeyStore` in `callLink` element"

"`remoteComType` in `callLink` element" on page 455

ctgLocation in callLink element

The linkage options part, callLink element, property **ctgLocation** is the URL for accessing a CICS Transaction Gateway (CTG) server, as is used if the value of property **remoteComType** is CICSECI or CICSSSL. Specify the related port by setting the property **ctgPort**.

Related concepts

"Linkage options part" on page 439

Related reference

"callLink element" on page 443

"remoteComType in callLink element" on page 455

ctgPort in callLink element

The linkage options part, callLink element, property **ctgPort** is the port for accessing a CICS Transaction Gateway (CTG) server, as is used if the value of property **remoteComType** is CICSECI or CICSSSL. Specify the related URL by setting the property **ctgLocation**.

If the case of CICSSSL, the value of **ctgPort** is the TCP/IP port on which a CTG JSSE listener is listening for requests; and if **ctgPort** is not specified, the CTG default port of 8050 is used.

Related concepts

"Linkage options part" on page 439

Related reference

"callLink element" on page 443

"ctgLocation in callLink element"

"remoteComType in callLink element" on page 455

JavaWrapper in callLink element

The linkage options part, callLink element, property **javaWrapper** indicates whether to allow generation of Java wrapper classes that can invoke the program being generated.

Valid values are as follows:

No (the default)

Do not allow generation of Java wrapper classes.

Yes

Allow that generation to occur. The generation occurs only if the build descriptor option **enableJavaWrapperGen** is set to **yes** or **only**.

Your choice for **javaWrapper** property has an effect only when you are setting up a remote call, as occurs when the value of the callLink property **type** is **remoteCall**. In contrast, if you are setting up a call to the program by way of an EJB, the value of **javaWrapper** is always **yes**; and if you are setting up a local call, the value of **javaWrapper** is always **no**.

If you are generating in the workbench or from the workbench batch interface, the build descriptor option **genProject** identifies the project that receives the classes. If

genProject is not specified (or if you are generating in the EGL SDK), the wrapper classes are placed in the directory specified by the build descriptor option **genDirectory**.

Related concepts

“Linkage options part” on page 439

Related reference

“callLink element” on page 443

“genDirectory” on page 251

“genProject” on page 252

linkType in callLink element

The linkage options part, callLink element, property **linkType** specifies the type of linkage when the value of property **type** is localCall.

If you are generating a COBOL or Java program, **linkType** is meaningful when property **pgmName** refers to a program that is called by the program being generated. If you are generating a Java wrapper, property **type** must be remoteCall or ejbCall, and **linkType** is not available.

Select a value from this list:

DYNAMIC

If you are generating a COBOL program, specifies that the call is a dynamic COBOL call.

If you are generating a Java program, specifies that the call is to a Java program in the same thread. DYNAMIC is the default value when you are generating a Java program.

STATIC

If you are generating a COBOL program, specifies that a static COBOL call occurs, which means that you must link-edit the called program with the calling program.

If you are generating a Java program, STATIC is equivalent to DYNAMIC.

Related concepts

“Linkage options part” on page 439

Related tasks

“Editing the callLink element of a linkage options part” on page 96

Related reference

“callLink element” on page 443

“pgmName in callLink element” on page 453

“type in callLink element” on page 458

library in callLink element

The linkage options part, callLink element, property **library** specifies the DLL or library that contains the called program when the value of the **type** property is ejbCall or remoteCall:

- If your program is calling a remote COBOL program on iSeries, the **library** property refers to the iSeries library that contains the program to be called.
- If your EGL-generated Java program is calling a remote, non-EGL generated program on iSeries (for example, a C or C++ service program), the called

program belongs to an iSeries library, and the **library** property refers to the name of the program that contains the entry point to be called. Set the other callLink properties as follows:

- Set the **pgmName** property to the name of the entry point
- Set the **remoteComType** property to direct or distinct
- Set the **remotePgmType** property to externallyDefined
- Set the **location** property to the name of the iSeries library
- Otherwise, if the calling program is an EGL-generated Java program not on iSeries, the **library** property refers to the name of a DLL that contains an entry point to be called locally as a native program. The entry point is identified by the **pgmName** property; but you need to specify the **library** property only if the names of the entry point and DLL are different.

To call a native DLL, set the other callLink properties as follows:

- Set the **remoteComType** property to direct
- Set the **remotePgmType** property to externallyDefined
- Set the **type** property to remoteCall because EGL middleware is used even though the DLL is called on the machine where the Java program is running.

Related concepts

“Linkage options part” on page 439

Related reference

“callLink element” on page 443

location in callLink element

The linkage options part, callLink element, property **location** specifies how the location of a called program is determined at run time. The property **location** is applicable in the following situation:

- The value of property **type** is ejbCall or remoteCall;
- The value of property **remoteComType** is JAVA400, CICSECI, CICSSSL, CICSJ2C, or TCPIP; and
- One of these statements applies:
 - If you are generating a COBOL or Java program, property **pgmName** refers to a program that is called by the program being generated
 - If you are generating a Java wrapper, **pgmName** refers to a program that is called by way of the Java wrapper

Select a value from this list:

programControlled

Specifies that the location of the called program is obtained from the system function sysVar.remoteSystemID when the call occurs.

system name

Specifies the location where the called program resides.

If you are generating a Java program or wrapper, the meaning of this property depends on property **remoteComType**:

- If the value of **remoteComType** is JAVA400, **location** refers to the iSeries system identifier
- If the value of **remoteComType** is CICSECI or CICSSSL, **location** refers to the CICS system identifier

- If the value of **remoteComType** is CICSJ2C, **location** refers to the JNDI name of the ConnectionFactory object that you establish for the CICS transaction invoked by the call. You establish that ConnectionFactory object when setting up the J2EE server, as described in *Setting up the J2EE server for CICSJ2C calls*. By convention, the name of the ConnectionFactory object begins with eis/, as in the following example:

eis/CICS1

- If the value of **remoteComType** is TCPIP, **location** refers to the TCP/IP hostname, and no default value exists
- If all the next conditions apply, **location** refers to the library of the called program--
 - The called program is an EGL-generated Java program that runs locally on iSeries
 - The value of **remoteComType** is DIRECT or DISTINCT
 - The value of **remotePgmType** is EXTERNALLYDEFINED

Related concepts

“Linkage options part” on page 439

Related tasks

“Editing the callLink element of a linkage options part” on page 96

“Setting up the J2EE server for CICSJ2C calls” on page 213

Related reference

“callLink element” on page 443

“sysVar.remoteSystemID” on page 625

“pgmName in callLink element” on page 453

“remoteComType in callLink element” on page 455

“type in callLink element” on page 458

luwControl in callLink element

The linkage options part, callLink element, property **luwControl** specifies whether the caller or called program controls the unit of work. This property is applicable only in the following situation:

- The value of property **type** is remoteCall; and
- You are generating a Java program or wrapper--
 - If you are generating a Java program, property **pgmName** refers to a CICS-based program that is called by the program being generated
 - If you are generating a Java wrapper, **pgmName** refers to a CICS-based program that is called by way of the Java wrapper

Select one of the following values:

CLIENT

Specifies that the unit of work is under the caller’s control. Updates by the called program are not committed or rolled back until the caller requests commit or rollback. If the called program issues a commit or rollback, a run-time error occurs.

CLIENT is the default value, unless a caller-controlled unit of work is not supported on the platform where the called program resides.

SERVER

Specifies that a unit of work started by the called program is independent of any unit of work controlled by the calling program. In the called program, these rules apply:

- The first change to a recoverable resource begins a unit of work
- Use of the system functions `sysLib.commit` and `sysLib.rollback` are valid

On a call from EGL-generated Java code to a VisualAge Generator COBOL program, a commit (or rollback on abnormal termination) is issued automatically when the called program returns. That command affects only the changes that were made by the called program.

When the property **type** is `ejbCall`, the run-time behavior is as described for `SERVER`.

Related concepts

"Linkage options part" on page 439

"Logical unit of work" on page 159

Related tasks

"Editing the `callLink` element of a linkage options part" on page 96

Related reference

"`callLink` element" on page 443

"`sysLib.commit`" on page 541

"`sysLib.rollback`" on page 550

"`pgmName` in `callLink` element" on page 453

"`type` in `callLink` element" on page 458

package in callLink element

The linkage options part, `callLink` element, property **package** identifies the Java package in which a called Java program resides. The property is useful whether property **type** is `ejbcall`, `localCall`, or `remoteCall`.

If you are generating a Java program, **package** is meaningful when property **pgmName** refers to a program that is called by the program being generated. If you are generating a Java wrapper, **package** is meaningful when property **pgmName** refers to the program that is called by way of the Java wrapper.

If the **package** property is not specified, the called program is assumed to be in the same package as the caller.

The package name that is used in generated Java programs is the package name of the EGL program, but in lower case; and when EGL generates output from the `callLink` element, the value of **package** is changed (if necessary) to lower case.

Related concepts

"Linkage options part" on page 439

Related tasks

"Editing the `callLink` element of a linkage options part" on page 96

"Setting up the J2EE run-time environment for EGL-generated code" on page 206

Related reference

"`callLink` element" on page 443

"`pgmName` in `callLink` element" on page 453

"`type` in `callLink` element" on page 458

parmForm in callLink element

The linkage options part, callLink element, property **parmForm** specifies the format of call parameters.

If you are generating a COBOL program, **parmForm** is applicable in this situation:

- Property **pgmName** refers to the program being generated or to a CICS-based program that is called by the program being generated; and
- Property **type** is localCall or remoteCall--
 - If the **type** is localCall, the valid **parmForm** values (as described later) are COMMDATA, COMMPTR (the default), and OSLINK
 - If the **type** is remoteCall, the valid **parmForm** values are COMMDATA (the default) and (if you are referring to a COBOL program called from Java code) COMMPTR.

If you are generating a Java program, **parmForm** is applicable in this situation:

- Property **pgmName** refers to a CICS-based program that is called by the program being generated; and
- Property **type** is ejbCall or remoteCall; in either case, the valid **parmForm** values (as described later) are COMMDATA (the default) and COMMPTR.

If you are generating a Java wrapper, **parmForm** is applicable in this case:

- Property **pgmName** refers to a generated COBOL program that is called by way of the Java wrapper; and
- Property **type** is ejbCall or remoteCall; in either case, the valid **parmForm** values (as described later) are COMMDATA (the default) or COMMPTR.

Select a value from this list:

COMMDATA

Specifies that the caller places business data (rather than pointers to data) in the COMMAREA.

Each argument value is moved to the buffer adjoining the previous value without regard for boundary alignment.

COMMDATA is the default value if the property **type** is ejbCall or remoteCall.

COMMPTR

Specifies that the caller acts as follows:

- Places a series of 4-byte pointers in the COMMAREA, one pointer per argument passed
- Sets the high-order bit of the last pointer to 1

COMMPTR is the default value if the value of property **type** is localCall.

OSLINK

Specifies that the standard COBOL parameter-passing conventions are in effect, with the called program expecting pointers to data, but without the CICS EIB or COMMAREA.

OSLINK is available only when **type** is localCall, **linkType** is DYNAMIC or STATIC, and you are generating a COBOL program.

Related concepts

"Linkage options part" on page 439

Related tasks

"Editing the callLink element of a linkage options part" on page 96

Related reference

"callLink element" on page 443

"linkType in callLink element" on page 448

"parmForm in callLink element" on page 452

"pgmName in callLink element"

"type in callLink element" on page 458

pgmName in callLink element

The linkage options part, callLink element, property **pgmName** specifies the name of the program part to which the callLink element refers.

You can use an asterisk (*) as a global substitution character in the program name; however, that character is valid only as the last character. For details, see *Linkage options part*.

Related concepts

"Linkage options part" on page 439

Related tasks

"Editing the callLink element of a linkage options part" on page 96

Related reference

"callLink element" on page 443

providerURL in callLink element

The linkage options part, callLink element, property **providerURL** specifies the host name and port number of the name server used by an EGL-generated Java program or wrapper to locate an EJB session bean that in turn calls an EGL-generated Java program. The property must have the following format:

`iiop://hostName:portNumber`

hostName

The IP address or host name of the machine on which the name server runs

portNumber

The port number on which the name server listens

The property **providerURL** is applicable only in the following situation:

- The value of property **type** is `ejbCall`; and
- Property **pgmName** refers to the program being called from the Java program or wrapper being generated.

Enclose the URL in double quote marks to avoid a problem either with periods or with the colon that precedes the port number.

A default is used if you do not specify a value for **providerURL**. The default directs an EJB client to look for the name server that is on the local host and that listens on port 900. The default is equivalent to the following URL:

`"iiop://"`

The following **providerURL** value directs an EJB client to look for a remote name server that is called *bankserver.mybank.com* and that listens on port 9019:

`"iiop://bankserver.mybank.com:9019"`

The following property value directs an EJB client to look for a remote name server that is called *bankserver.mybank.com* and that listens on port 900:

```
"iiop://bankserver.mybank.com"
```

Related concepts

"Linkage options part" on page 439

Related tasks

"Editing the callLink element of a linkage options part" on page 96

"Setting up the J2EE run-time environment for EGL-generated code" on page 206

Related reference

"callLink element" on page 443

"pgmName in callLink element" on page 453

"type in callLink element" on page 458

refreshScreen in callLink element

The linkage options part, callLink element, property **refreshScreen** indicates whether an automatic screen refresh is to occur when the called program returns control. Valid values are *yes* (the default) and *no*.

Set **refreshScreen** to *no* if the caller is in a run unit that presents text forms to a screen and either of these situations applies:

- The called program does not present a text form; or
- The caller writes a full-screen text form after the call.

The property **refreshScreen** applies only in these cases:

- The callLink **type** property is *localCall*; or
- The callLink **type** property is *remoteCall* when the *remoteComType* property is *direct* or *distinct*.

The property is ignored if you include the **noRefresh** indicator on the call statement.

Related reference

"call" on page 299

remoteBind in callLink element

The linkage options part, callLink element, property **remoteBind** specifies whether linkage options are determined at generation time or at run time. This property is applicable only in the following situation:

- The value of property **type** is *ejbCall* or *remoteCall*; and
- You are generating a Java program or wrapper. The property **pgmName** may refer to a program that is called by the program being generated, in which case the entry refers to the call from program to program. Alternatively, the property may refer to the program being generated, in which case the entry refers to the call from wrapper to program.

Select one of these values:

GENERATION

The linkage options specified at generation time are necessarily in use at run time. **GENERATION** is the default value.

RUNTIME

The linkage options specified at generation time can be revised at deployment time. In this case, you must include a linkage properties file in the run-time environment.

EGL generates a linkage properties file in the following situation:

- You are generating a Java program or wrapper;
- You set the property **remoteBind** to **RUNTIME**; and
- You generate with the build descriptor option **genProperties** set to **GLOBAL** or **PROGRAM**.

Related concepts

"Linkage options part" on page 439

"Linkage properties file" on page 404

Related tasks

"Deploying a linkage properties file" on page 223

"Editing the callLink element of a linkage options part" on page 96

Related reference

"callLink element" on page 443

"genProperties" on page 254

"Linkage properties file (details)" on page 431

"pgmName in callLink element" on page 453

"type in callLink element" on page 458

remoteComType in callLink element

The linkage options part, callLink element, property **remoteComType** specifies the communication protocol used in the following case:

- The value of property **type** is **ejbCall** or **remoteCall**; and
- You are generating a Java program or wrapper--
 - If you are generating a Java program, property **pgmName** refers to a program that is called by the program being generated
 - If you are generating a Java wrapper, **pgmName** refers to a program that is called by way of the Java wrapper

Select one of the following values.

DEBUG

Causes the called program to run in the EGL debugger, even when the calling program is running in a Java run-time or Java debug environment. You might use this setting in the following cases:

- You are running a Java program that uses an EGL Java wrapper to call a program written with EGL; or
- You are running an EGL-generated calling program that calls a program written with EGL.

The preceding situations can occur outside the WebSphere Test Environment, but can also occur within that environment, as when a JSP invokes a program written with EGL. In any case, the effect is to invoke the EGL source, not an EGL-generated program.

If you are using the WebSphere Test Environment, the caller and called programs must both be running there; the call cannot be from a remote machine.

When you use DEBUG, you set the following properties in the same **callLink** element--

- **library**, which names the project that contains the called program
- **package**, which identifies the package that contains the called program; but you do not need to set this property if the caller and called programs are in the same package

If the caller is not running in the EGL debugger and is not running in the WebSphere Test Environment, you must set these properties of the callLink element:

- **serverid**, which should specify the listener's port number if it's not 8346; and
- **location**, which must contain the hostname of the machine where the Eclipse workbench is running.

DIRECT

Specifies that the calling program or wrapper uses a direct local call, which means that the calling and called code run in the same thread. No TCP/IP listener is involved, and the value of property **location** is ignored. DIRECT is the default.

A calling Java program does not use the EGL middleware, but a calling wrapper uses that middleware to handle data conversion between EGL and Java primitive types.

DISTINCT

Specifies that a new run unit is started when calling a program locally. The call is still considered to be remote because EGL middleware is involved.

You can use this value for an EGL-generated Java program that calls a C or C++ program.

CICSECI

Specifies use of the CICS Transaction Gateway (CTG) ECI interface, as is needed when you are debugging or running non-J2EE code that accesses CICS.

CTG Java classes are used to implement this protocol. To specify the URL and port for a CTG server, assign values to the callLink element, properties **ctgLocation** and **ctgPort**. To identify the CICS region where the called program resides, specify the **location** property.

CICSJ2C

Specifies use of a J2C connector for the CICS Transaction Gateway.

CICSSSL

Specifies use of the Secure Socket Layer (SSL) features of CICS Transaction Gateway (CTG). The JSSE implementation of SSL is supported.

CTG Java classes are used to implement this protocol. To specify additional information for a CTG server, assign values to the following callLink element properties:

- **ctgKeyStore**
- **ctgKeyStorePassword**
- **ctgLocation**
- **ctgPort**, which in this case is the TCP/IP port on which a CTG JSSE listener is listening for requests. If **ctgPort** is not specified, the CTG default port of 8050 is used.

To identify the CICS region where the called program resides, specify the location property.

JAVA400

Specifies use of the IBM Toolbox for Java to communicate between a Java wrapper or program and a COBOL program that was generated (by EGL or VisualAge Generator) for iSeries.

TCPIP

Specifies that the EGL middleware uses TCP/IP.

Related concepts

"Linkage options part" on page 439

Related tasks

"Editing the callLink element of a linkage options part" on page 96

Related reference

"ctgKeyStore in callLink element" on page 446

"ctgKeyStorePassword in callLink element" on page 446

"ctgLocation in callLink element" on page 447

"ctgPort in callLink element" on page 447

"Editing the callLink element of a linkage options part" on page 96

"Setting up the J2EE server for CICSJ2C calls" on page 213

"Setting up the TCP/IP listener" on page 210

remotePgmType in callLink element

The linkage options part, callLink element, property **remotePgmType** specifies the kind of program being called. The property is applicable in the following situation:

- The value of property **type** is **ejbCall** or **remoteCall**; and
- One of these statements applies:
 - If you are generating a program (rather than a wrapper), property **pgmName** refers to a program that is called by the program being generated.
The called program is one of the following kinds:
 - An EGL-generated Java program
 - A non-EGL-generated C or C++ program
 - A program that runs on CICS and has CICS commands
 - If you are generating a Java wrapper, **pgmName** refers to the program that is called by way of the Java wrapper.

EGL

The called program is a COBOL or Java program that was generated by EGL or by VisualAge Generator; in this case, the caller is a COBOL program, Java program, or Java wrapper. This value is the default.

Related concepts

"Linkage options part" on page 439

"Run-time configurations" on page 2

Related tasks

"Editing the callLink element of a linkage options part" on page 96

Related reference

"callLink element" on page 443

"pgmName in callLink element" on page 453

"type in callLink element" on page 458

serverID in callLink element

The linkage options part, callLink element, property **serverID** specifies one of the following values:

- The TCP/IP port number of a called program's listener; but only if the TCP/IP protocol is in use. In this case, no default exists.
- The ID of a CICS transaction being invoked, but only when access to CICS is by the ECI interface or Secure Socket Layer features of the CICS Transaction Gateway. In this case, the default is the CICS server system mirror transaction.

The property is used only in the following situation:

- The value of property **type** is **ejbCall** or **remoteCall**;
- The value of **remoteComType** is **TCPIP**, **CICSECI**, or **CICSSSL**; and
- You are generating a Java program or wrapper--
 - If you are generating a Java program, property **pgmName** refers to a program that is called by the program being generated
 - If you are generating a Java wrapper, **pgmName** refers to a program that is called by way of the Java wrapper

Related concepts

"Linkage options part" on page 439

Related tasks

"Editing the callLink element of a linkage options part" on page 96

"Setting up the TCP/IP listener" on page 210

Related reference

"callLink element" on page 443

"pgmName in callLink element" on page 453

"remoteComType in callLink element" on page 455

"type in callLink element"

type in callLink element

The linkage options part, callLink element, property **type** specifies the kind of call. Select one of the following values:

ejbCall

Indicates that the generated Java program or wrapper will implement the program call by using an EJB session bean and that the EJB session bean will access the COBOL or Java program identified in the property **pgmName**. The value **ejbCall** is applicable in either of two cases:

- You are generating a Java wrapper for a COBOL or Java program, and the wrapper calls that program by way of an EJB session bean. In this case, the property **pgmName** refers to the program called from the wrapper, and your use of **ejbCall** causes generation of the EJB session bean.
- You are generating a Java program that calls a generated COBOL or Java program by way of an EJB session bean. In this case, the property **pgmName** refers to the called program, and an EJB session bean is not generated.

In either case, if you are using an EJB session bean, you must generate a Java wrapper, if only to generate the EJB session bean.

The generated session bean must be deployed on an enterprise Java server, and one of the following statements must be true:

- The name server used to locate the EJB session bean resides on the same machine as the code calling that session bean; or

- The property **providerURL** identifies where the name server resides.

If you wish to use an EJB session bean, you must generate the calling program or wrapper with a linkage options part in which the value of property **type** for the called program is **ejbCall**. You cannot make the decision to use a session bean at deployment time. If you set the property **remoteBind** to **RUNTIME**, however, you can decide at deployment time *how* the EJB session bean accesses the generated program, although making this decision at generation time is more efficient.

localCall

Specifies that the call does *not* use EGL middleware. The called program in this case is in the same process as the caller.

If the caller is a COBOL program, the situation is further defined by other properties. Most important are **linkType** and (for CICS COBOL programs) **parmForm**. Those properties have default values that you can accept or override.

localCall is the default value

remoteCall

Specifies that the call uses EGL middleware, which adds 12 bytes to the end of the data passed. Those bytes allow the caller to receive a return value from the called program.

If the caller is Java code, communication is handled by the protocol specified in property **remoteComType**; the protocol choice indicates whether the called program is in the same or a different thread.

If variable length records are passed on a call, these statements apply:

- Space is reserved for the maximum length specified for the record
- If the value of **callLink** property **type** is **remoteCall** or **ejbCall**, the variable-length item (if any) must be inside the record

Related concepts

"Linkage options part" on page 439

Related tasks

"Editing the **callLink** element of a linkage options part" on page 96

Related reference

"**callLink** element" on page 443

"**linkType** in **callLink** element" on page 448

"**location** in **callLink** element" on page 449

"**parmForm** in **callLink** element" on page 452

"**pgmName** in **callLink** element" on page 453

"**providerURL** in **callLink** element" on page 453

"**remoteComType** in **callLink** element" on page 455

transferToProgram element

A *transferToProgram* element of a linkage options part specifies how a generated COBOL program transfers control and ends processing.

The element includes these properties:

- **fromPgm**
- **toPgm**

- **linkType**
- **alias** (as is necessary if your code is transferring to a program whose run-time name is different from the name of the related program part)

You can avoid specifying a **transferToProgram** element when the target program is generated with VisualAge Generator or (in the absence of an alias) with EGL.

Related concepts

“Linkage options part” on page 439

“Run unit” on page 504

Related tasks

“Adding a linkage options part” on page 95

“Editing the transfer-related elements of a linkage options part” on page 98

Related reference

“alias in transfer-related linkage elements” on page 462

“fromPgm in transferToProgram element”

“linkType in transferToProgram element”

“toPgm in transfer-related linkage elements” on page 461

fromPgm in transferToProgram element

The linkage options part, transferToProgram element, property **fromPgm** specifies the name of a program part:

- If the target system is CICS for z/OS, the program part is the one that issues the transfer statement.
- If the target system is z/OS but not CICS and if any of the programs in the run unit issue a transfer statement, the property **fromPgm** is assigned the name of the first program in the run unit. For an example, see *transferToProgram element*.

The value of the **fromPgm** property is required and cannot include an asterisk (*).

Related concepts

“Linkage options part” on page 439

“Run unit” on page 504

Related tasks

“Adding a linkage options part” on page 95

“Editing the transfer-related elements of a linkage options part” on page 98

Related reference

“toPgm in transfer-related linkage elements” on page 461

“transfer” on page 354

“transferToProgram element” on page 459

“linkType in transferToProgram element”

linkType in transferToProgram element

The linkage options part, transferToProgram element, property **linkType** specifies the type of linkage to generate in relation to a transfer statement of type *transfer to program*. Valid values are as follows:

Dynamic (the default)

In programs that run on CICS for z/OS, an XCTL implements the transfer statement. In programs that run on z/OS outside of CICS, a dynamic COBOL

call is generated in the first program in the run unit, and the EGL run-time handles processing so that the transfer simulates the behavior of a CICS-based program.

The target program is assumed to be produced by EGL or by VisualAge Generator.

Static

In programs that run on CICS for z/OS, an XCTL implements the transfer statement. In programs that run on z/OS outside of CICS, the following statements apply:

- A static COBOL call is generated
- The EGL run-time handles processing so that the transfer simulates the behavior of a CICS-based program
- The value *Static* is required for target programs that call PL/I programs or that call programs that call PL/I programs.

The target program is assumed to be produced by EGL or by VisualAge Generator.

ExternallyDefined

Specify the value *ExternallyDefined* if you are transferring to a program that was not produced by EGL or VisualAge Generator. In all COBOL target systems, an XCTL implements the transfer statement.

If the program property **VAGCompatibility** is set to *yes*, you can specify *ExternallyDefined* in the transfer statement, as noted in *Compatibility with VisualAge Generator*. It is recommended that the value be specified in the *transferToProgram* element instead, but the value is in effect if specified in either place.

Related concepts

“Compatibility with VisualAge Generator” on page 276

“Linkage options part” on page 439

Related tasks

“Adding a linkage options part” on page 95

“Editing the transfer-related elements of a linkage options part” on page 98

Related reference

“fromPgm in transferToProgram element” on page 460

“toPgm in transfer-related linkage elements”

“transfer” on page 354

“transferToProgram element” on page 459

toPgm in transfer-related linkage elements

In the transfer-related elements of the linkage options part, the required property **toPgm** specifies the name of the program part (or of the non-EGL program) that receives control.

If the function word *sysVar.transferName* is specified as the target in a transfer statement, do not specify that system variable in the related **toPgm** property. Instead, specify the program name that will be in *sysVar.transferName* when the program runs.

Related concepts

“Linkage options part” on page 439

Related tasks

"Adding a linkage options part" on page 95

"Editing the transfer-related elements of a linkage options part" on page 98

Related reference

"sysVar.transferName" on page 630

"transfer" on page 354

transferToTransaction element

A *transferToTransaction* element of a linkage options part specifies how a generated program transfers control to a transaction and ends processing. The element includes the property `toPgm` and may include these properties:

- `alias`, as is necessary if your code is transferring to a program whose run-time name is different from the name of the related program part
- `externallyDefined`, as is necessary if your code is transferring to a program that was not generated with EGL or VisualAge Generator

You can avoid specifying a **transferToTransaction** element when the target program is generated with VisualAge Generator or (in the absence of an alias) with EGL.

Related concepts

"Linkage options part" on page 439

Related tasks

"Adding a linkage options part" on page 95

"Editing the transfer-related elements of a linkage options part" on page 98

Related reference

"alias in transfer-related linkage elements"

"externallyDefined in transferToTransaction element" on page 463

"fromPgm in transferToProgram element" on page 460

"linkType in transferToProgram element" on page 460

"toPgm in transfer-related linkage elements" on page 461

alias in transfer-related linkage elements

In the transfer-related elements of the linkage options part, the property **alias** specifies the run-time name of the program that is identified in property **toPgm**.

The value of this property must match the alias (if any) you specified when declaring the program to which you are transferring. If you did not specify an alias when declaring that program, either set the property **alias** to the name of the program part or do not set the property at all.

Related concepts

"Linkage options part" on page 439

Related tasks

"Adding a linkage options part" on page 95

"Editing the transfer-related elements of a linkage options part" on page 98

Related reference

"transferToProgram element" on page 459

"transferToProgram element" on page 459

"toPgm in transfer-related linkage elements" on page 461

externallyDefined in transferToTransaction element

The linkage options part, transferToTransaction element, property **externallyDefined** indicates whether you are transferring to a program that was produced by software other than EGL or VisualAge Generator. Valid values are **no** (the default) and **yes**.

If you specify **yes**, an XCTL implements the transfer statement in all COBOL target systems.

If the program property **VAGCompatibility** is set to *yes*, you can specify **externallyDefined** in the transfer statement, as noted in *Compatibility with VisualAge Generator*. It is recommended that the value be specified in the transferToTransaction element instead, but the value is in effect if specified in either place.

Related concepts

"Compatibility with VisualAge Generator" on page 276

Related tasks

"Adding a linkage options part" on page 95

"Editing the transfer-related elements of a linkage options part" on page 98

Related reference

"transfer" on page 354

Name aliasing

If you use a name that is not valid in the target generation language, the generator creates and uses an alias for the name in the generated code.

A name may be aliased for the following reasons:

- Differences in identifier characters allowed
- Differences in length limitations
- Differences in support for uppercase and lowercase characters
- Using a word that is a reserved word in the generated language
- Using a word that clashes with the name alias syntax (for example, **class\$** is aliased because **class\$** is the alias for **class** in Java generation)

An alias may be generated by substituting a valid set of characters for an invalid character, by truncating names that are too long, by adding a prefix or suffix to a name, or by producing a completely different name such as **EZE00123**.

Related concepts

"COBOL reserved-word file" on page 275

Related tasks

"Creating an EGL program part" on page 85

Related reference

"How names are aliased" on page 464

"Naming conventions" on page 468

How names are aliased

When you are naming EGL parts, the EGL language let you use names that are not permitted in the programming language being generated. During generation, such names are replaced with names that are permitted in the language being generated.

For example, COBOL does not permit names that contain underscore characters, names that contain lowercase letters, or names longer than 30 characters. If you use such a name in EGL code, the name is replaced with a valid name in the output.

Related concepts

"Name aliasing" on page 463

Related tasks

"Creating an EGL source file" on page 85

Related reference

"How COBOL names are aliased"

"How Java names are aliased" on page 465

How COBOL names are aliased

A COBOL name begins with a letter and comprises from one to 30 characters from the following set: letters A-Z, digits 0-9, and the hyphen or minus sign (-).

An EGL part name may be aliased for any of the following reasons:

- The part name contains invalid COBOL characters
- The part name contains lowercase letters
- The part name is longer than a maximum length
- The part name is not unique in the program
- The part name is a COBOL reserved word

In all cases, all characters are made upper case.

For a subset of parts (specifically, a program, data table, form, form group, or library), you can specify an alias by assigning a value to the **alias** property; and if that value is too long or has characters that are not valid in COBOL, an error occurs. If you did not specify a value for the property and if the value of the part name is too long, the part name is truncated to the maximum, which varies by part type:

- For data tables, 7
- For forms, 8
- For form groups, 6
- For libraries, 8
- For programs, 8

For the other parts (data items, functions, and records), EGL aliases names as follows:

1. Each character that is not valid in COBOL is replaced with an X, except that each underscore is replaced with a hyphen (-); for example, TEMP_ITEM becomes TEMP-ITEM
2. Part names that are longer than a maximum length are changed as follows:

- The name is prefixed with the letters EZE, a hyphen, and a one-to-five-digit number that is unique to the program
- The new name is truncated to the maximum length

The maximum length varies by part type:

- For data items, 27
 - For functions, 18
 - For records, 18
3. If after the previous steps the part name is a duplicate name in the program, the prefix described earlier is added to the beginning of the second and any subsequent occurrences of the part name. The resulting alias is truncated to the maximum length as stated above.
 4. If after steps 1-3 the part name matches a COBOL reserved word, the prefix described earlier is added to the beginning of the part name and the resulting alias is truncated to the maximum length as stated above.
 5. If after steps 1-4 the part name begins or ends with a hyphen, the beginning or ending hyphen is changed to X.

Related concepts

“COBOL reserved-word file” on page 275

Related reference

“Format of COBOL reserved-word file” on page 276

How Java names are aliased

When you give a part a name, that name must be a valid Java identifier, except that you can use a hyphen or minus sign (-) in a part name. However, a hyphen cannot be the first character in a part name.

If you choose a name that is a Java keyword or a name that contains a dollar sign (\$) or a hyphen or minus, the part name will not match the name in the generated output. An aliasing mechanism automatically appends a dollar sign to each part name that is a Java keyword. If you specify a name that contains one or more dollar signs or hyphens, the aliasing mechanism replaces each symbol with a Unicode value as follows:

```
$ $0024
- $002d
```

For example, an item named **class** is aliased to **class\$**, and an item named **class\$** is aliased to **class\$0024**.

The case you use to declare a part name is preserved. Programs XYZ and xyz are generated in XYZ.java and xyz.java respectively. On Windows 2000/NT/XP, if you generate into the same directory parts with names that differ only in case, the older files are overwritten.

EGL package names are always converted to lower case Java package names.

Finally, if the name of a program, page handler, or library matches the name of a class from the Java system package java.lang, a dollar sign is appended to the class name: Object becomes Object\$, Error becomes Error\$, and so on.

Related concepts

“Name aliasing” on page 463

Related reference

"How COBOL names are aliased" on page 464

"How names are aliased" on page 464

How Java wrapper names are aliased

The EGL generator applies the following rules to alias Java wrapper names:

1. If the EGL name is all uppercase, convert it to lowercase.
2. If the name is a class name or a method name, make the first character uppercase. (For example, the getter method for x is **getX()** not **getx()**.)
3. Delete every underscore (_) and hyphen (-). (Hyphens are valid in EGL names if you use VisualAge Generator compatibility mode.) If a letter follows the underscore or hyphen, change that character to uppercase.
4. If the name is a qualified name that uses a period (.) as a separator, replace every period with a low line, and add a low line at the beginning of the name.
5. If the name contains a dollar sign (\$), replace the dollar sign with two low lines and add a low line at the beginning of the name.
6. If a name is a Java keyword, add a low line at the beginning of the name.
7. If the name is * (an asterisk, which represents a filler item), rename the first asterisk **Filler1**, the second asterisk **Filler2**, and so forth.

In addition, special rules apply to Java wrapper class names for program wrappers, record wrappers, and substructured array items. The remaining sections discuss these rules and give an example. In general, if naming conflicts exist between fields within a generated wrapper class, the qualified name is used to determine the class and variable names. If the conflict is still not resolved, an exception is thrown at generation time.

Program wrapper class

Record parameter wrappers are named by using the above rules applied to the type definition name. If the record wrapper class name conflicts with the program class name or the program wrapper class name, **Record** is added at the end of the record wrapper class name.

The rules for variable names are as follows:

1. The record parameter variable is named using above rules applied to the parameter name. Therefore, the **get()** and **set()** methods contain these names rather than the class name.
2. The **get** and **set** methods are named **get** or **set** followed by the parameter name with the above rules applied.

Record wrapper class

The rules for substructured array items class names are as follows:

1. The substructured array item becomes an inner class of the record wrapper class, and the class name is derived by applying the above rules to the item name. If this class name conflicts with the containing record class name, **Structure** is appended to the item class name.
2. If any item class names conflict with each other, the qualified item names are used.

The rules for **get** and **set** method names are as follows:

1. The methods are named **get** or **set** followed by the item name with the above rules applied.

2. If any item names conflict with each other, the qualified item names are used.

Substructured array items class

The rules for substructured array items class names are as follows:

1. The substructured array item becomes an inner class of the wrapper class generated for the containing substructured array item , and the class name is derived by applying the above rules to the item name.
2. If this class name conflicts with the containing substructured array item class name, **Structure** is appended to the item class name.

The rules for **get** and **set** method names are as follows:

1. The methods are named **get** or **set** followed by the item name with the above rules applied.
2. If any item names conflict with each other, the qualified item names are used.

Example

The following sample program and generated output show what should be expected during wrapper generation:

Sample program:

```
Program WrapperAlias(param1 RecordA)
```

```
end
```

```
Record RecordA type basicRecord
```

```
  10 itemA CHAR(10)[1];
  10 item_b CHAR(10)[1];
  10 item$C CHAR(10)[1];
  10 static CHAR(10)[1];
  10 itemC CHAR(20)[1];
    15 item CHAR(10)[1];
    15 itemD CHAR(10)[1];
  10 arrayItem CHAR(20)[5];
    15 innerItem1 CHAR(10)[1];
    15 innerItem2 CHAR(10)[1];
end
```

Generated output:

Names of generated output

Output	Name
Program wrapper class	WrapperaliasWrapper , containing a field param1 , which is an instance of the record wrapper class RecordA
Parameter wrapper classes	RecordA , accessible through the following methods: <ul style="list-style-type: none"> • getItemA (from itemA) • getItemB (from the first item-b) • get_Item__C (from item\$C) • get_Static (from static) • get_ItemC_itemB (from itemB in itemC) • getItemD (from itemD) • getArrayItem (from arrayItem) ArrayItem is an inner class of RecordA that contains fields that can be accessed through getInnerItem1 and getInnerItem2 .

Related concepts

"Compatibility with VisualAge Generator" on page 276

"Java wrapper" on page 395

"Name aliasing" on page 463

Related tasks

"Generating Java wrappers" on page 131

Related reference

"How names are aliased" on page 464

"Java wrapper classes" on page 395

"Naming conventions"

"Output of Java wrapper generation" on page 389

Naming conventions

This page describes the rules for naming parts and variables and for assigning values to properties such as **file name**. For details on how logic parts can reference areas of memory, see *References to variables and constants* and *Arrays*.

Three categories of identifier are in EGL:

- EGL part and variables names, as described later.
- External resource names that are specified as property values in part or variable declarations. These names represent special cases, and the naming conventions depend on the conventions of the run-time system.
- EGL package names such as com.mycom.mypack. In this case, each character sequence is separated from the next by a period, and each sequence follows the naming convention for an EGL part name. For details on the relationship of package names and file structure, see *EGL projects, packages, and files*.

An EGL part or variable name is a series of 1 to 128 characters, with these exceptions:

- In a record part, the name of a logical file or queue can be no more than 8 characters
- In a program part, the *external name* is incorporated into the names of generated output files and Java classes. If the external name is not specified, the name of the program part is used but is truncated (if necessary) to the maximum number of characters allowed in the run-time environment.

Except as noted, a name must begin with a Unicode letter or underscore and can include additional Unicode letters as well as digits and currency symbols. If your code is compatible with VisualAge Generator, the following rules also apply to part and variable names but have no effect on package names:

- Initial character of a name can be an "at" sign (@)
- Subsequent characters can include "at" signs (@), hyphens (-), and pound signs (#)

A name cannot contain embedded blanks or be an EGL reserved word.

The name of either an EGL Web service or an EGL Web service operation must begin with a Unicode letter or an underscore and can include additional Unicode letters as well as digits and underscores, but cannot include hyphens, "at" signs, or currency symbols and cannot be a Java reserved word. The name of an EGL Web service cannot be identical to the name of an operation that is in that service.

Related concepts

“Compatibility with VisualAge Generator” on page 276
“EGL projects, packages, and files” on page 7
“EGL Web service” on page 139
“References to variables and constants” on page 34

Related reference

“Arrays” on page 64
“EGL reserved words” on page 290
“EGL system limits” on page 356
“How names are aliased” on page 464

PageHandler part

An EGL *pageHandler part* (or, more simply, a *page handler*) controls a user’s run-time interaction with a Web page. Specifically, the page handler provides data and services to a JavaServer Page (JSP) that displays the page. You can work most easily by customizing the JSP file and creating the page handler in the WebSphere Studio Page Designer; for details, see *Page Designer support for EGL*.

The page handler itself includes variables and the following kinds of logic--

- An OnPageLoad function, which is invoked the first time that the JSP renders the Web page
- A set of event handler functions, each of which is invoked in response to a specific user action (specifically, by the user clicking a button or hypertext link)
- Optionally, validation functions that are used to validate Web-page input fields
- Private functions that can be invoked only by other page-handler functions

The variables in the page handler are accessed in two ways:

- The run-time environment accesses the data automatically. If a field in the JSP is *bound* to an item in the page handler, the result is as follows--
 - After the OnPageLoad function runs and before the Web page is displayed, each page-handler item value is written to the JSP field to which the data is bound.
 - When the user submits a form in which bound JSP fields reside, the value in each field of the submitted form is copied to the associated page-handler item. Only then is control passed to an event handler. (However, this description does not include the validation steps, which are covered later in this topic.)
- The event handlers and the OnPageLoad function also can interact with the data, as well as with data stores (such as SQL databases) and with called programs.

The pageHandler part should be simple. Although the part might include light-weight data validations such as range checks, you are advised to invoke other programs to perform complex business logic. Database access, for example, should be reserved to a called program.

Output associated with a page handler

When you save a page handler, EGL adds a JSP file to your project for subsequent customization, unless a JSP file of the same name (the name specified in the **view** property of the page handler) is already in the appropriate folder (the folder WebContent\WEB-INF). EGL never overwrites a JSP.

If a Workbench preference is set to automatic build on save, page-handler generation occurs whenever you save the page handler. In any case, when you generate a page handler, the output is composed of the following objects:

- The *page bean* is a Java class that contains data and that provides initialization, data validation, and event-handling services for the Web page.
- A `<managed-bean>` element is placed in the JSF configuration file in your project, to identify the page bean at run time.
- A `<navigation-rule>` element is created in the JSF application configuration file to associate a JSF outcome (the name of the page handler) with the JSP file to be invoked.
- A default JSP file is also generated if one does not exist. Again, the file name is specified in the **view** property of the page handler, and the location is the folder `WebContent\WEB-INF`.

All data tables and records that are used by the part handler are also generated.

Validation

If the JSP-based JSF tags perform data conversion, validation, or event handling, the JSF run time does the necessary processing as soon as the user submits the Web page. If errors are found, the JSF run time may re-display the page without passing control to the run-time page handler. If the page handler receives control, however, the page handler may conduct a set of EGL-based validations.

The EGL-based validations occur if you specify the following details when you declare the page handler:

- The element edits (such as minimum input length) for individual input fields.
- The type-based edits (character, numeric) for individual fields.
- The DataTable edits (range, match valid, and match invalid) for individual input fields, as explained in *DataTable part*.
- The edit functions for individual input fields.
- The edit function for the page handler as a whole.

The page handler oversees the edits in the following order, but only for items whose values were changed by the user:

1. All the elementary and type-based edits, even if some fail
2. (If the prior edits were successful) all the table edits, even if some fail
3. (If the prior edits were successful) all the field-edit functions, even if some fail
4. (If all prior edits were successful) the pageHandler edit function

The page-item property **validationOrder** defines the order in which both the individual input fields are edited and the field validator functions are invoked.

If no `validationOrder` properties are specified, the default is the order of items defined in the page handler, from top to bottom. If `validationOrder` is defined for some but not all of the items in a page handler, validation of all items with the `validationOrder` property occurs first, in the specified order. Then, validation of items without the `validationOrder` property occurs in the order of items in the page handler, from top to bottom.

Run-time scenario

This section gives a technical overview of the run-time interaction of user and Web application server.

When the user invokes a JSP that is supported by a page handler, the following steps occur:

1. The Web application server initializes the environment--
 - a. Constructs a session object to retain data that the user-accessed applications need across multiple interactions
 - b. Constructs a request object to retain data on the user's current interaction
 - c. Invokes the JSP
2. The JSP processes a series of JSF tags to construct a Web page--
 - a. Creates an instance of the page handler, causes the onPageLoad function (if any) to be invoked with user-specified arguments, and places the page handler into the request object
 - b. Accesses data stored in the request and session objects, for inclusion in the Web page

Note: The pageHandler part has a property called OnPageLoad, which identifies the page-handler function that is invoked at JSP startup. The function automatically retrieves any user-supplied arguments that were passed to it; can call other code; and can place additional data in the request or session object of the Web application server; but the function can neither forward control to another page nor cause an error message to be displayed when the page is first presented to the user.

3. The JSP submits the Web page to the user, and the Web application server destroys the response object, leaving the session object and the JSP.

If the user supplies data in the on-screen fields associated with an HTML <FORM> tag and submits the form, the following steps occur:

1. The Web application server re-initializes the environment--
 - a. Constructs a request object
 - b. Places the received data for the submitted form into the page bean, for validation
 - c. Re-invokes the JSP
2. The JSP processes a series of JSF tags to store the received data in the page bean.
3. The run-time page handler validates data:
 - a. Does relatively elementary edits (such as minimum input length), as specified in the pageHandler data declarations
 - b. Invokes any item-specific validation functions, as specified in the pageHandler data declarations
 - c. Invokes a pageHandler validator function, as is needed if you wish to validate one field at least partially on the basis of the content of another field

(For details on validation, see the previous section.)

4. If an error occurs, the EGL run time places errors on a JSF queue, and the JSP re-displays the Web page with embedded messages. If no error occurs, however, the result is as follows:
 - a. Data stored in the page bean is written to the record bean
 - b. Subsequent processing is determined by an event handler, which is identified in the JSF tag that is associated with the user-clicked button or hyperlink.

The event handler may forward processing to a JSF label, which identifies a mapping in a run-time JSF-based configuration file. The mapping in turn identifies the object to invoke, whether a JSP (usually one associated with an EGL page handler) or a servlet.

Related concepts

"References to parts" on page 16

"Web support" on page 135

Related reference

"Page Designer support for EGL" on page 135

"PageHandler part in EGL source format"

"Page item properties" on page 53

PageHandler part in EGL source format

You declare a pageHandler part in an EGL file, which is described in *EGL projects, packages, and files*. This part is a primary part, which means that it must be at the top level of the file and must have the same name as the file.

An example of a pageHandler part is as follows:

```
package pagehandlers ;
/* Page designer requires that all page handler parts
   be in a package named "pagehandlers". */
PageHandler ListCustomers {onPageLoad=onPageLoad}

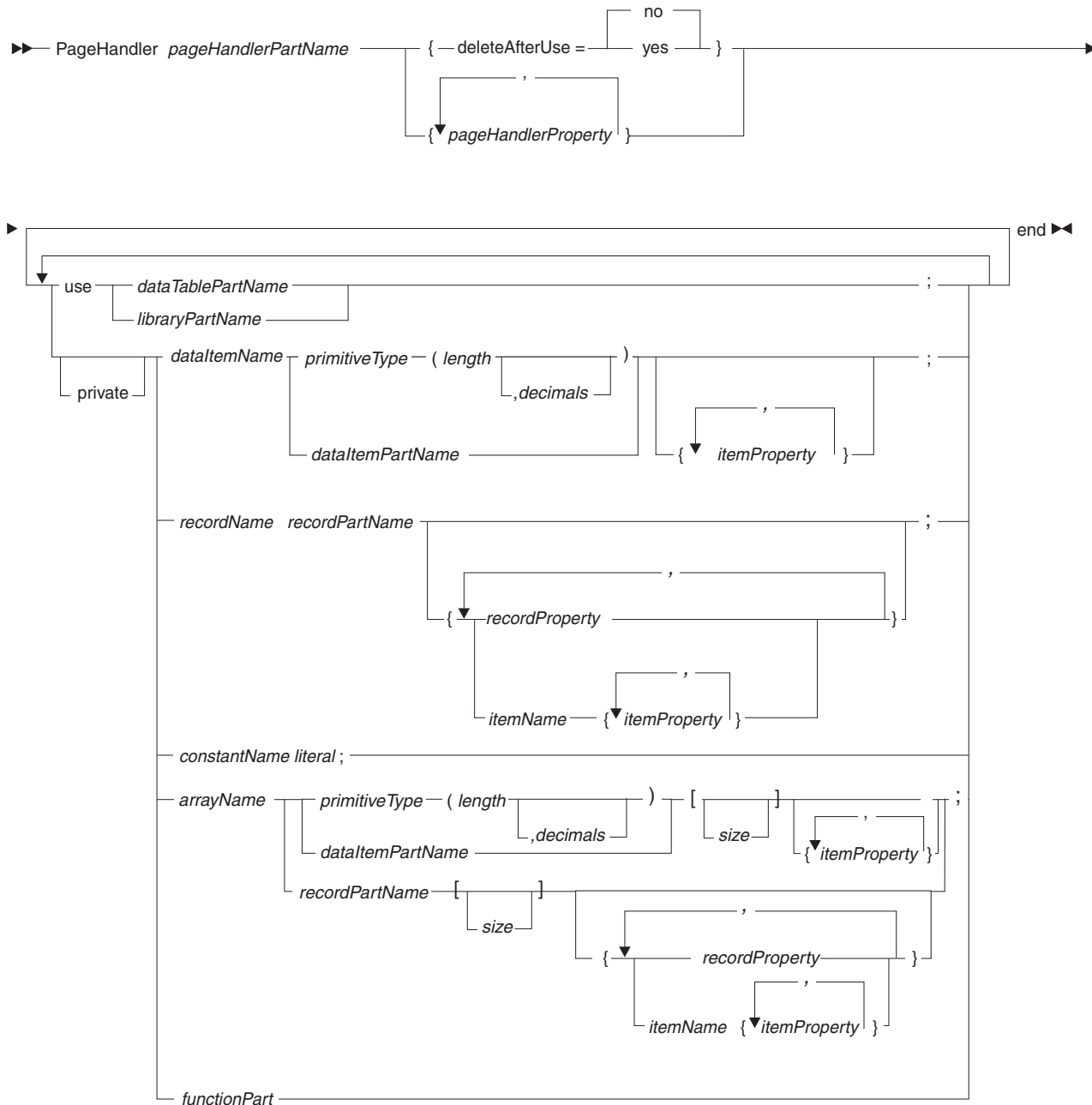
    use CustomerLib3;           // Library for customer table access

    customerList Customer[] {maxSize=100}; // List of customers

    Function onPageLoad()
        startkey CustomerId;    // Starting key to retrieve customers
        status int;             // Result from library call

        startKey = 0;
        // Retrieve up to 100 customer records
        CustomerLib3.getCustomersByCustomerId(startKey, customerList, status);
        if ( status != 0 && status != 100 )
            setError("Retrieval of Customers Failed.");
        end
    End
    function returnToIntroductionClicked()
        forward to "Introduction";
    end
End
```

The diagram of a pageHandler part is as follows:



PageHandler *pageHandlerPartName* ... **end**

Identifies the part as a page handler and specifies the part name. For the rules of naming, see *Naming conventions*.

pageHandlerProperty

A page-handler property. For details, see *Properties of the page handler*.

use *dataTablePartName*, **use** *libraryPartName*

A use declaration that simplifies access of a data table or library. For details, see *Use declaration*.

private

Indicates that the variable, constant, or function is unavailable to the JSP that renders the Web page. If you omit the term **private**, you can bind the variable, constant, or function to a control on the Web page.

dataItemName

Name of a data item (a variable). For rules, see *Naming conventions*.

primitiveType

The primitive type assigned to the data item.

length

The structure item's length, which is an integer. The value of a memory area that is based on the structure item includes the specified number of characters or digits.

decimals

For a numeric type (BIN, DECIMAL, NUM, NUMC, or PACF), you may specify *decimals*, which is an integer that represents the number of places after the decimal point. The maximum number of decimal positions is the smaller of two numbers: 18 or the number of digits declared as *length*. The decimal point is not stored with the data.

dataItemPartName

The name of a dataItem part that is a model of format for the data item, as described in *typeDef*. The dataItem part must be visible to the pageHandler part, as described in *References to parts*.

itemProperty

An item property. For details, see *Page item properties*.

recordName

Name of a record (a variable). For rules, see *Naming conventions*.

recordPartName

The name of a record part that is a model of format for the record, as described in *typeDef*. The record part must be visible to the pageHandler part, as described in *References to parts*.

recordProperty

An override of a record property. For details on the record properties, see one of the following descriptions, depending on the type of record in *recordPartName*:

- Basic record part in EGL source format
- Indexed record part in EGL source format
- MQ record part in EGL source format
- Relative record part in EGL source format
- Serial record part in EGL source format
- SQL record part in EGL source format

itemName

Name of the record item whose properties you intend to override.

itemProperty

An override of an item property. For details, see *Overview of EGL properties and overrides*.

constantName literal

Name and value of a constant. For rules, see *Naming conventions*.

arrayName

Name of a dynamic or static array of records or data items. If you use this option, the other symbols to the right (*dataItemPartName*, *primitiveType*, and so on) refer to each element of the array.

functionPart

An embedded function. For details on the syntax, see *Function part in EGL source format*.

Properties of the page handler

The properties of the page handler are as follows and are optional:

alias = *alias*

A string that is incorporated into the names of generated output. If you do not specify an alias, the `pageHandler` part name is used instead.

allowUnqualifiedItemReferences = **no**, **allowUnqualifiedItemReferences** = **yes**

Specifies whether to allow your code to reference structure items but to exclude the name of the *container*, which is the data table, record, or form that holds the structure item. Consider the following record part, for example:

```
Record aRecordPart type basicRecord
  10 myItem01 CHAR(5);
  10 myItem02 CHAR(5);
end
```

The following variable is based on that part:

```
myRecord aRecordPart;
```

If you accept the default value of **allowUnqualifiedItemReferences** (*no*), you must specify the record name when referring to `myItem01`, as in this assignment:

```
myValue = myRecord.myItem01;
```

If you set the property **allowUnqualifiedItemReferences** to *yes*, however, you can avoid specifying the record name:

```
myValue = myItem01;
```

It is recommended that you accept the default value, which promotes a best practice. By specifying the container name, you reduces ambiguity for people who read your code and for EGL.

EGL uses a set of rules to determine the area of memory to which a variable name or item name refers. For details, see *References to variables and constants*.

includeReferencedFunctions = **no**, **includeReferencedFunctions** = **yes**

Indicates whether the page-handler bean contains a copy of each function that is neither inside the page handler nor in a library accessed by the page handler. The default value is *no*, which means that you can ignore this property if you are fulfilling the following practices at development time, as is recommended:

- Place shared functions in a library
- Place non-shared functions in the page handler

If you are using shared functions that are not in a library, generation is possible only if you set the property **includeReferencedFunctions** to *yes*.

msgResource = *logicalName*

Identifies a Java resource bundle or properties file that is used in error-message presentation. The content of the resource bundle or properties file is composed of a set of keys and related values.

A particular value is displayed in response to the program's invoking the EGL system function `sysLib.setError`, when the invocation includes use of the key for that value.

onPageLoad = *functionName*

The name of a page handler function that receives control when the related JSP initially displays a Web page. This function can be used to set up initial values of the data displayed in the page.

title = *literal*

The **title** property is a bind property, which means that the assigned value is used as a default when you are working in Page Designer. The property specifies the title of the page.

literal is a quoted string.

validationBypassFunctions = (*functionNames*)

Identifies one or more *event handlers*, which are page-handler functions that are associated with a button control in the JSP. Each function name is separated from the next by a comma.

If you specify an event handler in this context, the EGL run time skips input-field and page validations when the user clicks the button or hypertext link that is or related to the event handler. This property is useful for reserving a user action that ends the current page-handler processing and that immediately transfers control to another Web resource.

validator = *functionName*

Identifies the page-handler validator function, which is invoked after all the item validators are invoked, as described in *Validation in Web applications built with EGL*.

view = *JSPFileName*

Identifies the name and subdirectory path to the Java Server Page (JSP) that is bound to the page handler. *JSPFileName* is a quoted string.

The default value is the name of the page handler, with the file extension **.jsp**. If you specify this property, include the file extension, if any.

When you save or generate a page handler, EGL adds a JSP file to your project for subsequent customization, unless a JSP file of the same name (the name specified in the **view** property) is already in the appropriate folder (the folder `WebContent\WEB-INF`). EGL never overwrites a JSP.

Related concepts

"EGL projects, packages, and files" on page 7
"Overview of EGL properties and overrides" on page 40
"PageHandler part" on page 469
"References to parts" on page 16
"References to variables and constants" on page 34
"Typedef" on page 20

Related reference

"Function part in EGL source format" on page 381
"Naming conventions" on page 468
"Page item properties" on page 53
"Primitive types" on page 27
"`sysLib.setError`" on page 528
"Use declaration" on page 631

Program part

A *program part* defines the central logical unit in a run-time COBOL or Java program. For an overview of main and called programs and of the program types (basic and textUI), see *Parts*.

Any kind of program part includes a function called *main*, which represents the logic that runs at program start up. A program can include other functions and can access functions that are outside of the program. The function *main* can invoke those other functions, and any function can give control to other programs.

The most important program properties are as follows:

- Each *parameter* references an area of memory that contains data received from a caller. Parameters are global to the program and are valid only in called programs.
- Each *variable* references an area of memory that is allocated in and global to the program.
- A *form group* is a collection of forms that present data to the user:
 - A basic program can present data to a printer by way of *print forms*
 - A text program can present data interactively (by way of *text forms*) or to a printer

For details, see *FormGroup part*.

- An *input record* is an area of global memory that receives data when control is transferred asynchronously from another program. An input record is available only in a main program.
- In main text programs, the *segmented* property determines what actions are taken automatically before the program issues a **converse** statement to present a text form. For details, see *Segmentation*.
- Also in text programs, an *input form* has one of two purposes at program start up:
 - The form is presented to a user who invokes the program from a monitor or terminal
 - Alternatively, data that was entered by a user is received into the input form, which is a memory area in the program itself. This situation applies only in the case of a *deferred program switch*, which is a two-step transfer of control that is caused by a variant of the **show** statement--
 1. A program submits a text form to the user, then terminates
 2. The user submits the form, and by virtue of information in the form, the submission automatically invokes a second program, which contains the input form

For a complete list of program properties, see *Program part properties*.

Related concepts

"FormGroup part" on page 376

"Function part" on page 380

"Parts" on page 11

"References to variables and constants" on page 34

"Segmentation in text applications" on page 151

Related tasks

"Creating an EGL program part" on page 85

Related reference

“Content assist” on page 103
“Data initialization” on page 277
“EGL source format” on page 292
“EGL statements” on page 70
“Program part in EGL source format”
“Program part properties” on page 485

Program part in EGL source format

You declare a program part in an EGL file, which is described in *EGL source format*. When you write that file, do as follows:

- Include only those parts that are used exclusively by the program
- Do not include other primary parts (dataTable, library, program, or pageHandler)

The next example shows a called program part with two embedded functions, along with a stand-alone function and a stand-alone record part:

```
Program myProgram type basicProgram (employeeNum INT)
{
    includeReferencedFunctions
}

// program-global variables
employees record_ws;
employeeName char(20);

// a required embedded function
Function main()
    // initialize employee names
    recd_init();

    // get the correct employee name
    // based on the employeeNum passed
    employeeName = getEmployeeName(employeeNum);
end

// another embedded function
Function recd_init()
    employees.name[1] = "Employee 1";
    employees.name[2] = "Employee 2";
end

// stand-alone function
Function getEmployeeName(employeeNum INT) returns (CHAR(20))

    // local variable
    index BIN(4);
    index = 2;
    if (employeeNum > index)
        return("Error");
    else
        return(employees.name[employeeNum]);
    end

end

// record part that acts as a typeDef for employees
Record record_ws type basicRecord
    10 name CHAR(20)[2];
end
```


For other details, see the topic for a particular type of program.

Related concepts

"Parts" on page 11

"Program part" on page 477

Related reference

"Basic program in EGL source format"

"EGL source format" on page 292

"Function part in EGL source format" on page 381

"Text UI program in EGL source format" on page 481

Basic program in EGL source format

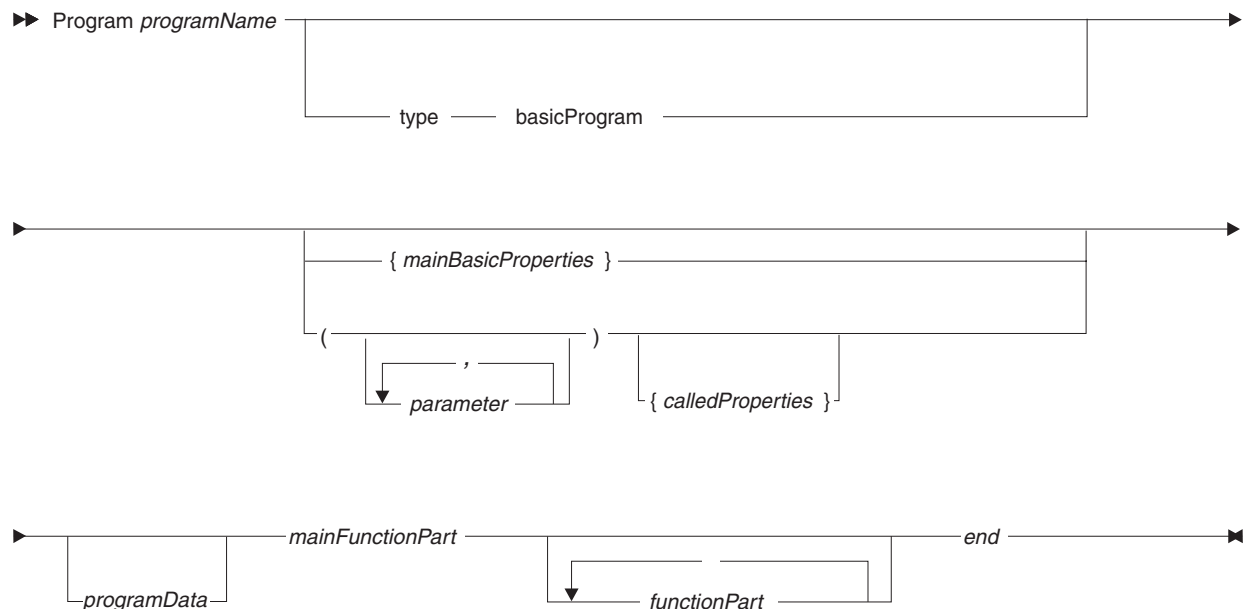
An example of a basic program is as follows:

```
program myCalledProgram type basicProgram
  (buttonPressed int, returnMessage char(25))

  function main()
    returnMessage = "";
    if (buttonPressed = 1)
      returnMessage = "Message1";
    end

    if (buttonPressed = 2)
      returnMessage = "Message2";
    end
  end
end
```

The syntax diagram for a program part of type basicProgram is as follows:



Program *programPartName* ... end

Identifies the part as a program part and specifies the name and type. If the program name is followed by a left parenthesis, the program is a called basic program.

If you do not set the **alias** property (as described later), the name of the generated program is either *programPartName* or, if you are generating COBOL, the first eight characters of *programPartName*.

For other rules, see *Naming conventions*.

mainBasicProperties

The properties for a main basic program are optional:

- **alias**
- **allowUnqualifiedItemReferences**
- **includeReferencedFunctions**
- **inputRecord**
- **msgTablePrefix**

For details, see *Program properties*.

parameter

Specifies the name of a parameter, which may be a data item, record, or form; or a dynamic array of records or data items. For rules, see *naming conventions*.

If the caller's argument is a variable (not a constant or literal), any changes to the parameter change the area of memory available to the caller.

Each parameter is separated from the next by a comma. For other details, see *Program parameters*.

calledProperties

The called properties are optional:

- **alias**
- **allowUnqualifiedItemReferences**
- **includeReferencedFunctions**
- **msgTablePrefix**

For details, see *Program properties*.

programData

Variable and use declarations, as described in *Program data other than parameters*.

mainFunctionPart

A required function named *main*, which takes no parameters. (The only program code that can take parameters is the program itself and functions other than *main*.)

For details on writing a function, see *Function part in EGL source format*.

functionPart

An embedded function, which is private to this program. For details on writing a function, see *Function part in EGL source format*.

Related concepts

- "EGL projects, packages, and files" on page 7
- "Overview of EGL properties and overrides" on page 40
- "Parts" on page 11
- "Program part" on page 477
- "Syntax diagram" on page 506

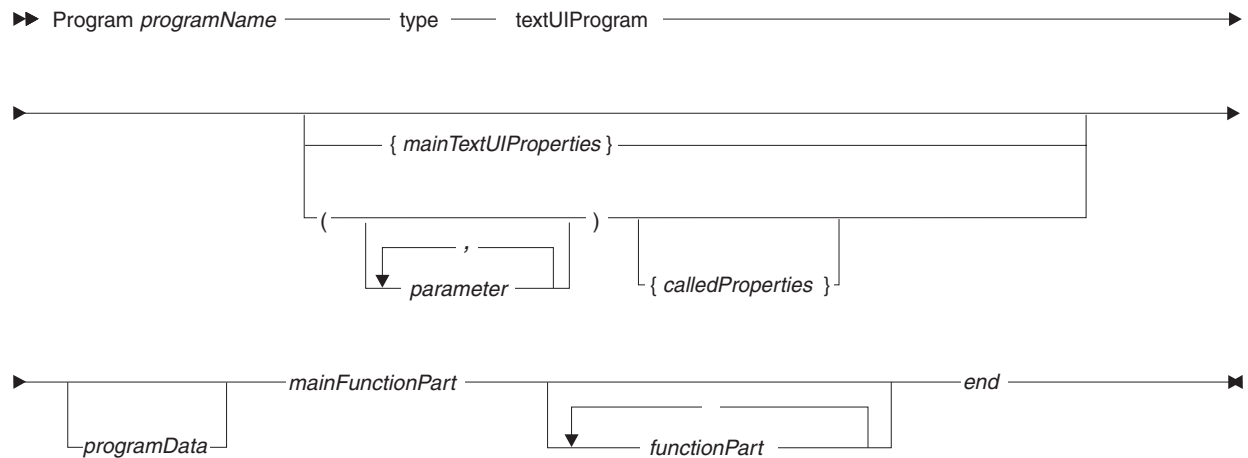
Related reference

- "EGL source format" on page 292
- "Function part in EGL source format" on page 381

“Naming conventions” on page 468
 “Program data other than parameters” on page 487
 “Program parameters” on page 483
 “Program part in EGL source format” on page 478
 “Program part properties” on page 485
 “Use declaration” on page 631

Text UI program in EGL source format

The syntax diagram for a program part of type textUIProgram is as follows:



Program *programPartName* ... end

Identifies the part as a program part and specifies the name and type. If the program name is followed by a left parenthesis, the program is a called basic program.

If you do not set the **alias** property (as described later), the name of the generated program is either *programPartName* or, if you are generating COBOL, the first eight characters of *programPartName*.

For other rules, see *Naming conventions*.

mainTextUIProperties

The properties for a main text UI program are optional:

- **alias**
- **allowUnqualifiedItemReferences**
- **includeReferencedFunctions**
- **inputForm**
- **inputRecord**
- **msgTablePrefix**
- **segmented**

For details, see *Program properties*.

parameter

Specifies the name of a parameter, which may be a data item, record, or form; or a dynamic array of records or data items. For rules, see *naming conventions*.

If the caller’s argument is a variable (not a constant or literal), any changes to the parameter change the area of memory available to the caller.

Each parameter is separated from the next by a comma. For other details, see *Program parameters*.

calledProperties

The called properties are optional:

- **alias**
- **allowUnqualifiedItemReferences**
- **includeReferencedFunctions**
- **msgTablePrefix**

For details, see *Program properties*.

programData

Variable and use declarations, as described in *Program data other than parameters*.

mainFunctionPart

A required function named *main*, which takes no parameters. (The only program code that can take parameters is the program itself and functions other than *main*.)

For details on writing a function, see *Function part in EGL source format*.

functionPart

An embedded function, which is not available to any logic part other than the program. For details on writing a function, see *Function part in EGL source format*.

An example of a Text UI program is as follows:

```
Program HelloWorld type textUIprogram
```

```
{}
```

```
use myFormgroup;
```

```
myMessage char(25);
```

```
function main()
```

```
while (eventKey not pf3)
```

```
    myTextForm.msgField = "                ";
```

```
    myTextForm.msgField=myMessage;
```

```
    converse myTextForm;
```

```
    if (eventKey is pf3)
```

```
        exit program;
```

```
    end
```

```
    if (eventKey is pf1)
```

```
        myMessage = "Hello Word";
```

```
    end
```

```
end
```

```
end
```

```
end
```

Related concepts

"EGL projects, packages, and files" on page 7

"Overview of EGL properties and overrides" on page 40

"Parts" on page 11

"Program part" on page 477

"Segmentation in text applications" on page 151

"Syntax diagram" on page 506

Related reference

"EGL source format" on page 292

"Function part in EGL source format" on page 381

"Naming conventions" on page 468

"Program data other than parameters" on page 487

"Program parameters" on page 483

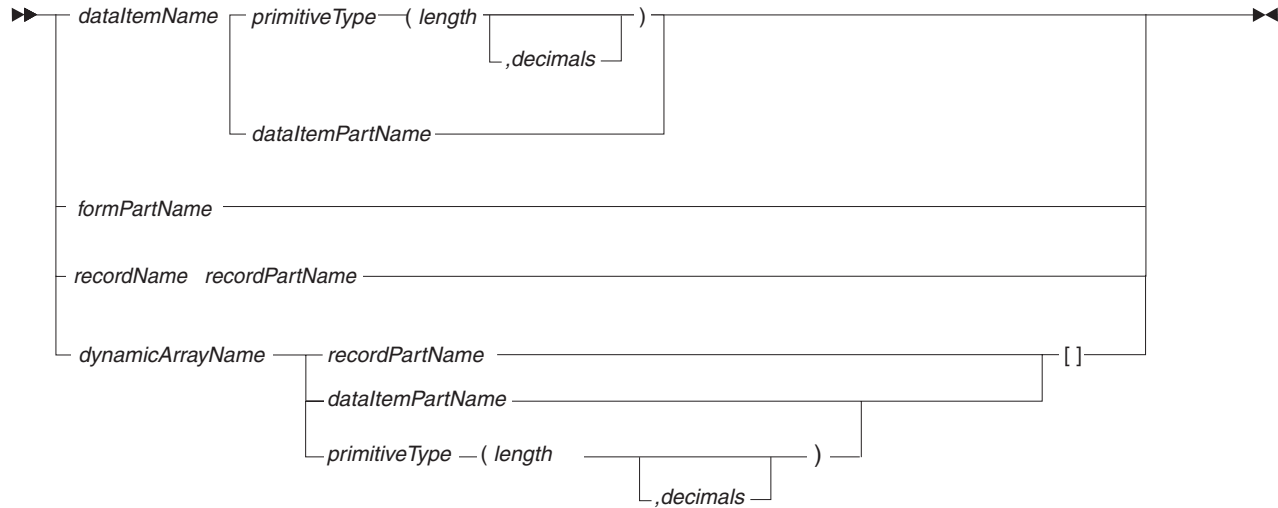
“Program part in EGL source format” on page 478

“Program part properties” on page 485

“Use declaration” on page 631

Program parameters

The syntax diagram for a program parameter is as follows:



dataItemName

Name of a data item. For the rules of naming, see *Naming conventions*.

primitiveType

The primitive type of a data item or (in relation to a dynamic array) the primitive type of an array element.

length

The parameter's length or (in relation to a dynamic array), the length of an array element. The length is an integer that represents the number of characters or digits in the memory area referenced either by *dataItemName* or (in the case of a dynamic-array element), *dynamicArrayName*.

decimals

For a numeric type (BIN, DECIMAL, NUM, NUMC, or PACF), you may specify *decimals*, which is an integer that represents the number of places after the decimal point. The maximum number of decimal positions is the smaller of two numbers: 18 or the number of digits declared as *length*. The decimal point is not stored with the data.

dataItemPartName

The name of a dataItem part that is visible to the program. For details on visibility, see *References to parts*.

The part acts as a model of format, as described in *Typedef*.

formPartName

Name of a form.

The form part must be accessible through a form group in the program's use declaration; and a form accessed as a parameter cannot be displayed to the user.

Even if a program cannot display a form (as is the case for basic program that receives a text form), the program can access the data in a form that was passed to the program.

For the rules of naming, see *Naming conventions*.

recordName

Name of a record. For the rules of naming, see *Naming conventions*.

recordPartName

Name of a record part that is visible to the program. For details on visibility, see *References to parts*.

The part acts as a model of format, as described in *Typedef*.

dynamicArrayName

Name of a dynamic array of records or data items. If you use this option, the other symbols to the right (*dataItemPartName*, *primitiveType*, and so on) refer to each element of the array. For the rules of naming, see *Naming conventions*.

The following statements apply to input or output (I/O) against record parameters:

- A record passed from another program does not include record state such as the I/O error value *endOfFile*. Similarly, any change in the record state is not returned to the caller, so if you perform I/O against a record parameter, any tests on that record must occur before the program ends.
- Any I/O operation performed against the record uses the record properties specified for the parameter, not the record properties specified for the argument.
- For records of type *indexedRecord*, *mqRecord*, *relativeRecord*, or *serialRecord*, the file or message queue associated with the record declaration is treated as a run-unit resource rather than a program resource. Local record declarations share the same file (or queue) whenever the record property **fileName** (or **queueName**) has the same value. Only one physical file at a time can be associated with a file or queue name no matter how many records are associated with the file or queue in the run unit, and EGL enforces this rule by closing and reopening files as appropriate.

Related concepts

"Program part" on page 477

"References to parts" on page 16

"References to variables and constants" on page 34

"Syntax diagram" on page 506

"Typedef" on page 20

Related reference

"Arrays" on page 64

"Basic record part in EGL source format" on page 491

"DataItem part in EGL source format" on page 284

"EGL source format" on page 292

"Indexed record part in EGL source format" on page 492

"Naming conventions" on page 468

"Primitive types" on page 27

"Relative record part in EGL source format" on page 495

"Serial record part in EGL source format" on page 497

"SQL record part in EGL source format" on page 498

Program part properties

Program part properties vary by whether the program is called or main and, if main, by whether the program is of type basic or text UI. The properties are as follows:

alias = *alias*

A string that is incorporated into the names of generated output. If you do not set the **alias** property, the program-part name (or a truncated version) is used instead.

The **alias** property is available in any program.

allowUnqualifiedItemReferences = **no**, **allowUnqualifiedItemReferences** = **yes**

Specifies whether to allow your code to omit container and substructure qualifiers when referencing items in structures.

The **allowUnqualifiedItemReferences** property is available in any program.

Consider the following record part, for example:

```
Record aRecordPart type basicRecord
  10 myItem01 CHAR(5);
  10 myItem02 CHAR(5);
end
```

The following variable is based on that part:

```
myRecord aRecordPart;
```

If you accept the default value of **allowUnqualifiedItemReferences** (*no*), you must specify the record name when referring to `myItem01`, as in this assignment:

```
myValue = myRecord.myItem01;
```

If you set the property **allowUnqualifiedItemReferences** to *yes*, however, you can avoid specifying the record name:

```
myValue = myItem01;
```

It is recommended that you accept the default value, which promotes a best practice. By specifying the container name, you reduces ambiguity for people who read your code and for EGL.

EGL uses a set of rules to determine the area of memory to which a variable name or item name refers. For details, see *References to variables and constants*.

includeReferencedFunctions = **no**, **includeReferencedFunctions** = **yes**

Indicates whether the program contains a copy of each function that is neither inside the program nor in a library accessed by the program.

The **includeReferencedFunctions** property is available in any program.

The default value is *no*, which means that you can ignore this property if you are fulfilling the following practices at development time, as is recommended:

- Place shared functions in a library
- Place non-shared functions in the program

If you are using shared functions that are not in a library, generation is possible only if you set the property **includeReferencedFunctions** to *yes*.

inputForm = *formName*

Identifies a form that is presented to the user before the program logic runs, as described in *Input form*.

The **inputForm** property is available only in main text UI programs.

inputRecord = *inputRecord*

Identifies a global, basic record that a program automatically initializes and that may receive data from a program that uses a **transfer** statement to transfer control. For additional details, see *Input record*.

The **inputRecord** property is available in any main program.

msgTablePrefix = *prefix*

Specifies the first one to the four characters in the name of the data table that is used as the message table for the program. The other characters in the name correspond to one of the national language codes listed in *DataTable part in EGL source format*.

The **msgTablePrefix** property is available in any basic or text UI program.

Programs that run in Web applications do not use a message table, but use a JavaServer Faces message resource. For details on that resource, see the description of the **msgResource** property in:

- *PageHandler part in EGL source format*

segmented = *no*, **segmented** = *yes*

Indicates whether the program is segmented, as explained in *Segmentation*. The default is *no* in main text UI programs. The property is not valid in other types of programs.

Related concepts

"Program part" on page 477

"References to variables and constants" on page 34

"Segmentation in text applications" on page 151

Related reference

"DataTable part in EGL source format" on page 287

"forward" on page 315

"Input form"

"Input record" on page 487

"Naming conventions" on page 468

"PageHandler part in EGL source format" on page 472

"Syntax diagram" on page 506

Input form

When you declare a main program that runs in a text application, you have the option to specify an *input form*, which is a form that is presented to the user before the program logic runs.

Two scenarios are possible:

- If the program is the target of a show-form-returning-to statement from an EGL-generated program, the sending program presents a form to the user, and that form must be identical to the input form of the receiving program. The receiving program is invoked only after the user submits the form. After the user submits the form, the receiving program does not present the input form a second time; instead, the initial logic (the execute function) runs.
- If the program is the target of a transfer statement from a program (EGL or non-EGL) or if the program is invoked by the user or by an operating-system

command, the receiving program converses the input form. (In this case, input fields on that form are initialized before display.) After the user submits the form, the initial logic (the execute function) runs.

The input form must be in the form group that you specified in the program-part declaration.

Related reference

"Data initialization" on page 277

Input record

Any main program part can have an input record, which is a global record that the EGL-generated program automatically initializes. The record must be of type `basicRecord`.

If the program starts as a result of a transfer with a record, the program initializes the input record (which is internal to that program), then assigns the transferred data to the record.

If the input record is longer than the received data, the extra area in the input record retains the values assigned during record initialization. If the input record is shorter than the received data, the extra data is truncated.

If primitive types in the transferred data are incompatible with the primitive types in the equivalent positions in the input record, the receiving program may end abnormally.

Related concepts

"Overview of EGL properties and overrides" on page 40

"Parts" on page 11

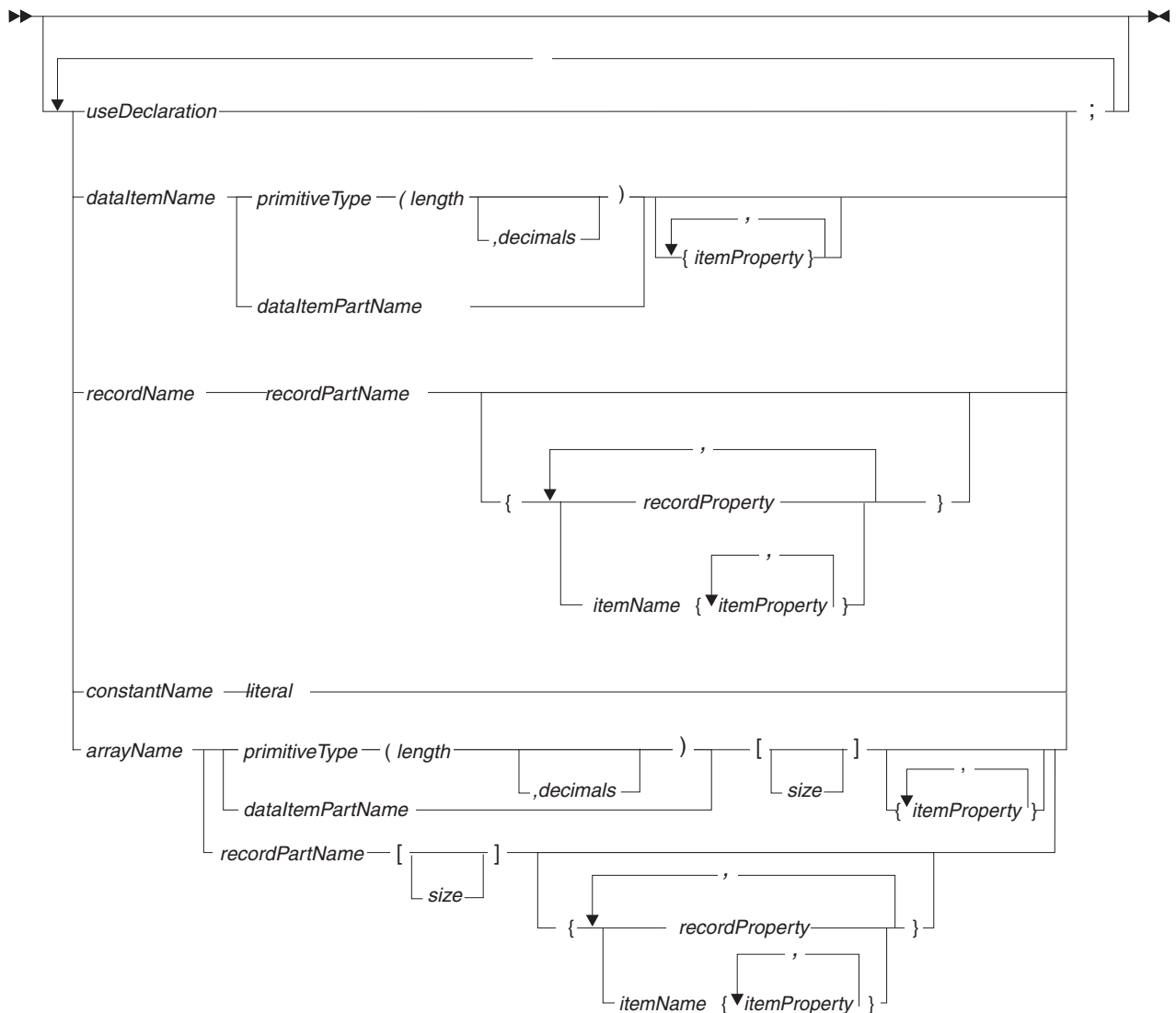
"Compatibility with VisualAge Generator" on page 276

Related reference

"Data initialization" on page 277

Program data other than parameters

The syntax diagram for program data is as follows:



useDeclaration

Provides easier access to a data table or library, and is needed to access to forms in a form group. For details, see *Use declaration*.

dataItemName

Name of a data item. For the rules of naming, see *Naming conventions*.

primitiveType

The primitive type of a data item or (in relation to an array) the primitive type of an array element.

length

The parameter's length or (in relation to an array), the length of an array element. The length is an integer that represents the number of characters or digits in the memory area referenced either by *dataItemName* or (in the case of an array), *dynamicArrayName*.

decimals

For a numeric type (BIN, DECIMAL, NUM, NUMC, or PACF), you may specify *decimals*, which is an integer that represents the number of places after

the decimal point. The maximum number of decimal positions is the smaller of two numbers: 18 or the number of digits declared as *length*. The decimal point is not stored with the data.

dataItemPartName

The name of a dataItem part that is visible to the program. For details on visibility, see *References to parts*.

The part acts as a model of format, as described in *Typedef*.

recordName

Name of a record. For the rules of naming, see *Naming conventions*.

recordPartName

Name of a record part that is visible to the program. For details on visibility, see *References to parts*.

The part acts as a model of format, as described in *Typedef*.

constantName literal

Name and value of a constant. The value is either a quoted string or a number. For the rules of naming, see *Naming conventions*.

itemProperty

An item-specific property-and-value pair, as described in *Overview of EGL properties and overrides*.

recordProperty

A record-specific property-and-value pair. For details on the available properties, see the reference topic for the record type of interest.

A basic record has no properties.

itemName

Name of a record item whose properties you wish to override. See *Overview of EGL properties and overrides*.

arrayName

Name of a dynamic or static array of records or data items. If you use this option, the other symbols to the right (*dataItemPartName*, *primitiveType*, and so on) refer to each element of the array.

size

Number of elements in the array. If you specify the number of elements, the array is static; otherwise, the array is dynamic.

Related concepts

"EGL projects, packages, and files" on page 7

"Overview of EGL properties and overrides" on page 40

"Parts" on page 11

"Program part" on page 477

"References to variables and constants" on page 34

"Segmentation in text applications" on page 151

"Syntax diagram" on page 506

"Typedef" on page 20

Related reference

"Arrays" on page 64

"Data initialization" on page 277

"DataItem part in EGL source format" on page 284

"DataTable part in EGL source format" on page 287

"EGL source format" on page 292

“EGL statements” on page 70
“forward” on page 315
“Function part in EGL source format” on page 381
“Indexed record part in EGL source format” on page 492
“Input form” on page 486
“Input record” on page 487
“I/O error values” on page 418
“MQ record part in EGL source format” on page 493
“Naming conventions” on page 468
“Primitive types” on page 27
“Relative record part in EGL source format” on page 495
“Serial record part in EGL source format” on page 497
“SQL record part in EGL source format” on page 498

“Use declaration” on page 631

Record parts

A *record part* defines a structure (a hierarchical layout of fixed-size data elements in storage) and an optional binding, which is a relationship, of the record to an external data source (file, database, or message queue). The binding is specified by the record type and associated properties. The record is used as a type when declaring variables (including parameters), and the structure defines the layout of the storage allocated for the variables. The data source binding determines the type of I/O operations generated for the variable when the variable is used in an I/O statement.

You may use a record variable in an EGL function in the following contexts:

- In most cases, in a statement that copies data to or from data storage
- In an assignment statement
- As an argument that passes data to another program or function

You also may use a record variable in these ways:

- As a parameter that receives data into a program
- As a parameter that receives data into a function

Related concepts

“DataItem part” on page 284
“Parts” on page 11
“Record types and properties” on page 13
“Resource associations and file types” on page 157
“Structure” on page 19
“Typedef” on page 20

Related tasks

“Setting the default build descriptors” on page 237
“Setting preferences for the EGL editor” on page 104

Related reference

“EGL source format” on page 292
“Data initialization” on page 277
“Items” on page 43
“Primitive types” on page 27

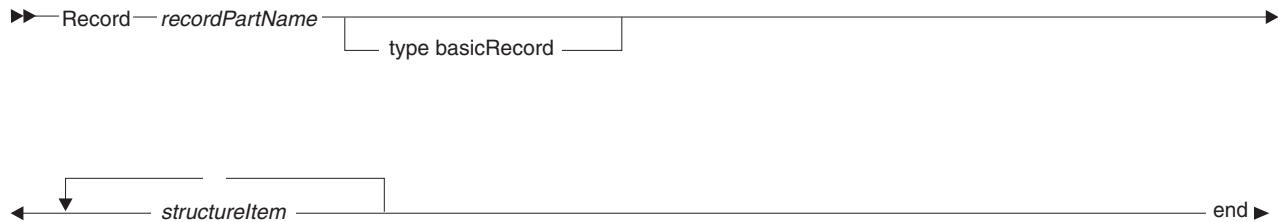
Basic record part in EGL source format

You declare a record part of type `basicRecord` in an EGL file, which is described in *EGL source format*.

An example of a basic record part is as follows:

```
Record myBasicRecordPart type basicRecord
  10 myField01 CHAR(2);
  10 myField02 CHAR(78);
end
```

The syntax diagram for a basic record is as follows:



Record *recordPartName* **basicRecord**

Identifies the part as being of type `basicRecord` and specifies the name. For rules, see *Naming conventions*.

structureItem

A structure item, as described in *Structure item in EGL source format*.

Related concepts

- "EGL projects, packages, and files" on page 7
- "References to parts" on page 16
- "Parts" on page 11
- "Record parts" on page 490
- "References to variables and constants" on page 34
- "Typedef" on page 20

Related tasks

- "Syntax diagram" on page 506

Related reference

- "DataItem part in EGL source format" on page 284
- "EGL source format" on page 292
- "Function part in EGL source format" on page 381
- "Indexed record part in EGL source format" on page 492
- "MQ record part in EGL source format" on page 493
- "Naming conventions" on page 468
- "Primitive types" on page 27
- "Program part in EGL source format" on page 478
- "Properties that support variable-length records" on page 502
- "Relative record part in EGL source format" on page 495
- "Serial record part in EGL source format" on page 497
- "SQL record part in EGL source format" on page 498
- "Structure item in EGL source format" on page 505

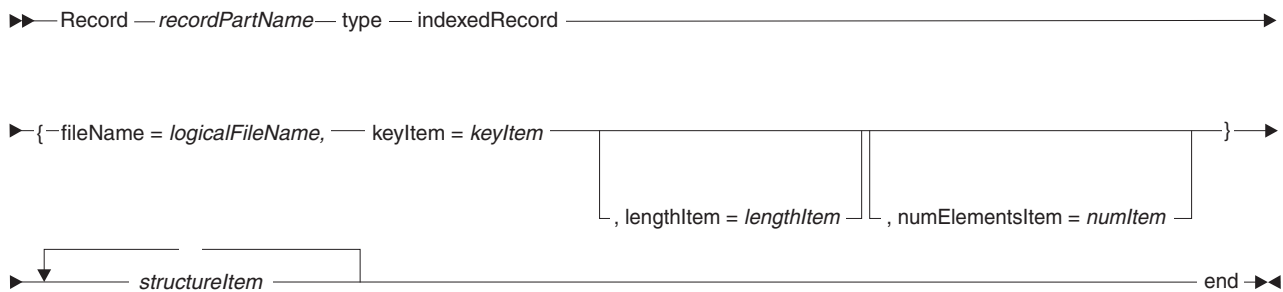
Indexed record part in EGL source format

You declare a record part of type `indexedRecord` in an EGL file, which is described in *EGL source format*.

An example of an indexed record part is as follows:

```
Record myIndexedRecordPart type indexedRecord
{
    fileName = myFile,
    keyItem = myKeyItem
}
10 myKeyItem CHAR(2);
10 myContent CHAR(78);
end
```

The syntax diagram for an indexed record part is as follows:



Record *recordPartName* **indexedRecord**

Identifies the part as being of type `indexedRecord` and specifies the name. For rules, see *Naming conventions*.

fileName = *logicalFileName*

The file name. For details on the meaning of your input, see *Resource associations (overview)*. For rules, see *Naming conventions*.

keyItem = *keyItem*

The key item, which can only be a structure item that is unique in the same record. You must use an unqualified reference for *keyItem*; for example, use *myItem* rather than *myRecord.myItem*. (In a function, however, you can reference that structure item as you would reference any structure item.)

lengthItem = *lengthItem*

The length item, as described in *Properties that support variable-length records*.

numElementsItem = *numElementsItem*

The number of elements item, as described in *Properties that support variable-length records*.

structureItem

A structure item, as described in *Structure item in EGL source format*.

Related concepts

"EGL projects, packages, and files" on page 7

"References to parts" on page 16

"Parts" on page 11

"Record parts" on page 490

"References to variables and constants" on page 34

"Resource associations and file types" on page 157

"Typedef" on page 20

Related tasks

"Syntax diagram" on page 506

Related reference

"Arrays" on page 64

"DataItem part in EGL source format" on page 284

"EGL source format" on page 292

"Function part in EGL source format" on page 381

"MQ record part in EGL source format"

"Naming conventions" on page 468

"Primitive types" on page 27

"Program part in EGL source format" on page 478

"Properties that support variable-length records" on page 502

"Relative record part in EGL source format" on page 495

"Serial record part in EGL source format" on page 497

"SQL record part in EGL source format" on page 498

"Structure item in EGL source format" on page 505

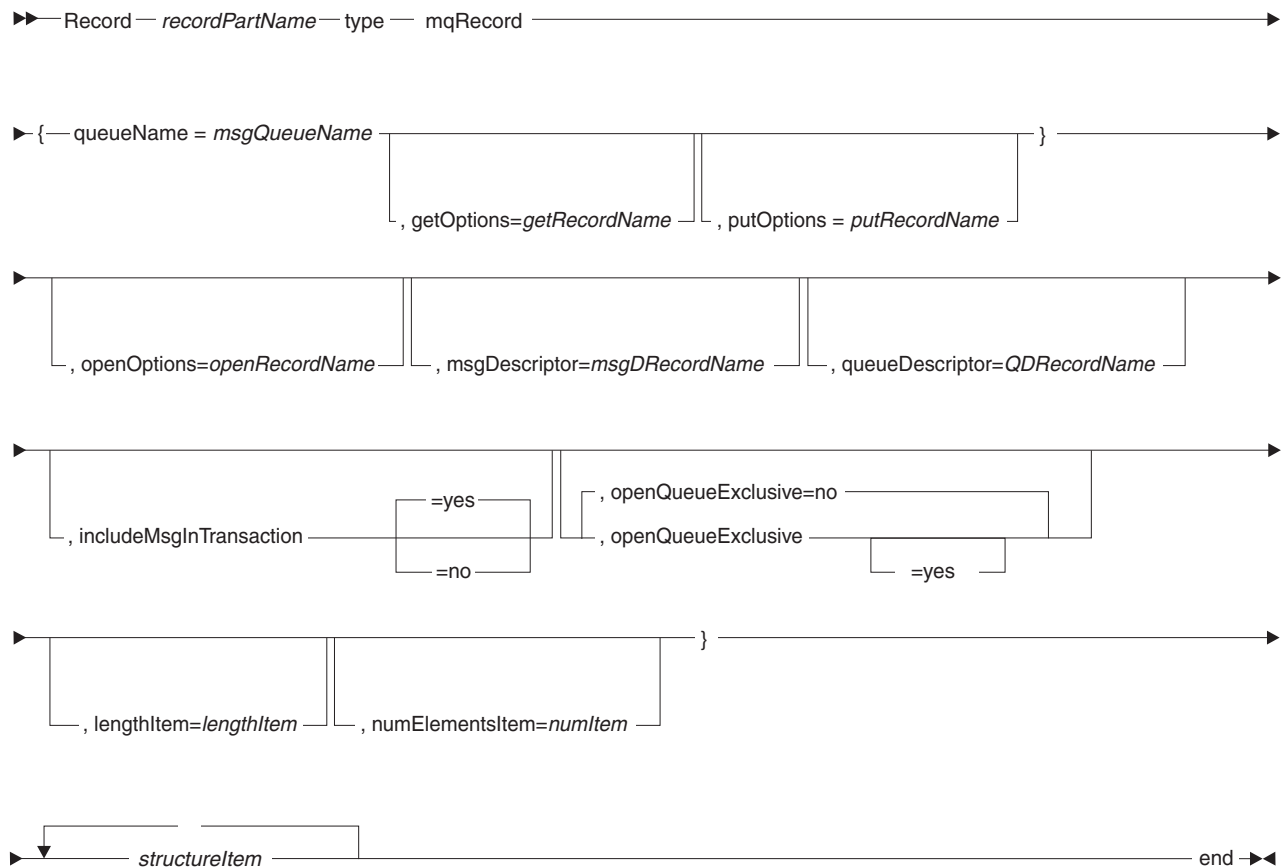
MQ record part in EGL source format

You can declare MQ record parts in a program file or definitions file. For an overview of those files, see *EGL source format*. For an overview of how EGL interacts with MQSeries, see *MQSeries support*.

An example of an MQ record part is as follows:

```
Record myMQRecordPart type mqRecord
{
    queueName = "myQueue"
}
10 myField01 CHAR(2);
10 myField02 CHAR(78);
end
```

The syntax diagram for an MQ record part is as follows:



Record *recordPartName* **mqRecord**

Identifies the part as being of type `mqRecord` and specifies the name. For rules, see *Naming conventions*.

queueName = *queueName*

The queue name, which is the logical queue name and usually not the name of the physical queue. For details on the format of your input, see *MQ record properties*.

getOptions = *getRecordName*

Identifies a program variable (a basic record) that is used as a get options record. For details, see *Options records for MQ records*.

putOptions = *putRecordName*

Identifies a program variable (a basic record) that is used as a put options record. For details, see *Options records for MQ records*.

openOptions = *openRecordName*

Identifies a program variable (a basic record) that is used as an open options record. For details, see *Options records for MQ records*.

msgDescriptor = *msgDRecordName*

Identifies a program variable (a basic record) that is used as a message descriptor. For details, see *Options records for MQ records*.

queueDescriptor = *QDRecordName*

Identifies a program variable (a basic record) that is used as a queue descriptor. For details, see *Options records for MQ records*.

includeMsgInTransaction = yes, includeMsgInTransaction = no

If this property is set to *yes* (the default), each of the record-specific messages is embedded in a transaction, and your code can commit or roll back that transaction. For details on the implications of your choice, see *MQSeries support*.

openQueueExclusive = no, openQueueExclusive = yes

If this property is set to *yes*, your code has the exclusive ability to read from the message queue; otherwise, other programs can read from the queue. The default is *no*. This property is equivalent to the MQSeries option `MQOO_INPUT_EXCLUSIVE`.

lengthItem = lengthItem

The length item, as described in *MQ record properties*.

numElementsItem = numElementsItem

The number of elements item, as described in *MQ record properties*.

structureItem

A structure item, as described in *Structure item in EGL source format*.

Related concepts

"EGL projects, packages, and files" on page 7

"References to parts" on page 16

"MQSeries support" on page 161

"Parts" on page 11

"Record parts" on page 490

"References to variables and constants" on page 34

"Typedef" on page 20

Related tasks

"Syntax diagram" on page 506

Related reference

"Arrays" on page 64

"DataItem part in EGL source format" on page 284

"EGL source format" on page 292

"Function part in EGL source format" on page 381

"Indexed record part in EGL source format" on page 492

"MQ record properties" on page 169

"Naming conventions" on page 468

"Options records for MQ records" on page 169

"Primitive types" on page 27

"Program part in EGL source format" on page 478

"Relative record part in EGL source format"

"Serial record part in EGL source format" on page 497

"SQL record part in EGL source format" on page 498

"Structure item in EGL source format" on page 505

Relative record part in EGL source format

You declare a record part of type `relativeRecord` in an EGL file, which is described in *EGL source format*.

An example of a relative record part is as follows:

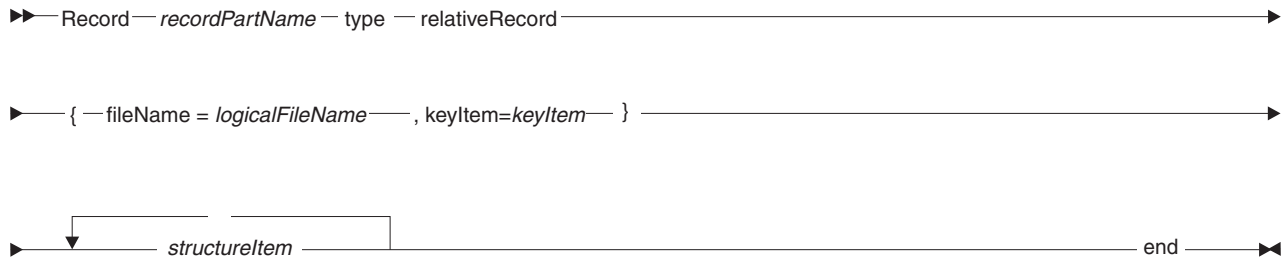
```
Record myRelativeRecordPart type relativeRecord
{
    fileName = myFile,
    keyItem  = myKeyItem
```

```

    }
    10 myKeyItem NUM(4);
    10 myContent CHAR(76);
end

```

The syntax diagram of a relative record part is as follows:



Record *recordPartName* **relativeRecord**

Identifies the part being of type *relativeRecord* and specifies the name. For the rules of naming, see *Naming conventions*.

fileName = *fileName*

The file name. For details on the meaning of your input, see *Resource associations (overview)*. For rules, see *Naming conventions*.

keyItem = *keyItem*

The key item, which can be any of these areas of memory:

- A record item in the same record
- A record item in a record that is global to the program or is local to the function that accesses the record
- A data item that is global to the program or is local to the function that accesses the record

You must use an unqualified reference to name the key item. For example, use *myItem* rather than *myRecord.myItem*. (In a function, however, you can reference the key item as you would reference any item.) The key item must be unique in the local scope of the function that accesses the record or must be absent from local scope and unique in global scope.

The key item has these characteristics:

- Has a primitive type of NUM, BIN, DECIMAL, INT, or SMALLINT
- Contains no decimal place
- Allows for 9 digits at most

Only the **get** and **add** statements use the relative record key item, but the key item must be available to any function that uses the record for file access.

structureItem

A structure item, as described in *Structure item in EGL source format*.

Related concepts

“EGL projects, packages, and files” on page 7

“References to parts” on page 16

“Parts” on page 11

“Record parts” on page 490

“References to variables and constants” on page 34

“Typedef” on page 20

Related tasks

"Syntax diagram" on page 506

Related reference

"Arrays" on page 64

"DataItem part in EGL source format" on page 284

"EGL source format" on page 292

"Function part in EGL source format" on page 381

"Indexed record part in EGL source format" on page 492

"MQ record part in EGL source format" on page 493

"Naming conventions" on page 468

"Primitive types" on page 27

"Program part in EGL source format" on page 478

"Resource associations and file types" on page 157

"Serial record part in EGL source format"

"SQL record part in EGL source format" on page 498

"Structure item in EGL source format" on page 505

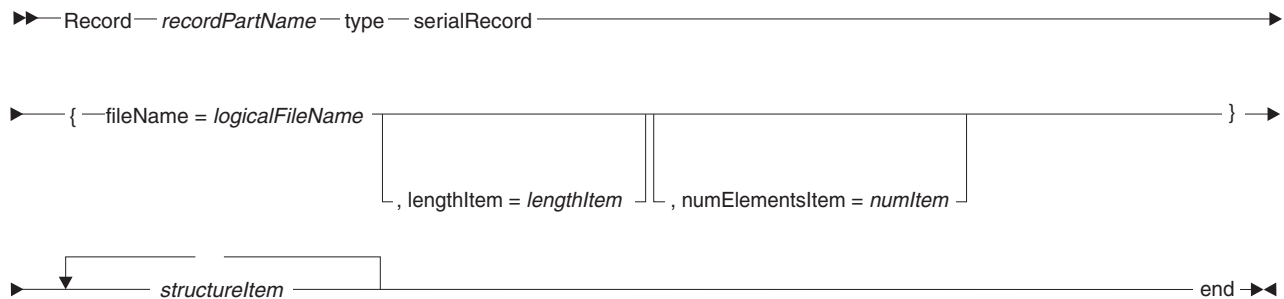
Serial record part in EGL source format

You declare a record part of type `serialRecord` in an EGL file, which is described in *EGL source format*.

An example of a serial record part is as follows:

```
Record mySerialRecordPart type serialRecord
{
    fileName = myFile
}
10 myField01 CHAR(2);
10 myField02 CHAR(78);
end
```

The syntax diagram for a serial record part is as follows:



Record *recordPartName* **serialRecord**

Identifies the part as being of type `serialRecord` and specifies the part name.

For the rules of naming, see *Naming conventions*.

fileName = *fileName*

The file name. For details on the meaning of your input, see *Resource associations (overview)*. For rules, see *Naming conventions*.

lengthItem = *lengthItem*

The length item, as described in *Properties that support variable-length records*.

numElementsItem = *numItem*

The number of elements item, as described in *Properties that support variable-length records*.

structureItem

A structure item, as described in *Structure item in EGL source format*.

Related concepts

"EGL projects, packages, and files" on page 7

"References to parts" on page 16

"Parts" on page 11

"Record parts" on page 490

"References to variables and constants" on page 34

"Resource associations and file types" on page 157

"Typedef" on page 20

Related tasks

"Syntax diagram" on page 506

Related reference

"Arrays" on page 64

"DataItem part in EGL source format" on page 284

"EGL source format" on page 292

"Function part in EGL source format" on page 381

"Indexed record part in EGL source format" on page 492

"MQ record part in EGL source format" on page 493

"Naming conventions" on page 468

"Primitive types" on page 27

"Program part in EGL source format" on page 478

"Properties that support variable-length records" on page 502

"Relative record part in EGL source format" on page 495

"SQL record part in EGL source format"

"Structure item in EGL source format" on page 505

SQL record part in EGL source format

You declare a record part of type `sqlRecord` in an EGL file, which is described in *EGL source format*. For an overview of how EGL interacts with relational databases, see *SQL support*.

An example of a SQL record part is as follows:

```
Record mySQLRecordPart type sqlRecord
{
    tableNames = (mySQLTable T1),
    keyItems = myHostVar01,
    defaultSelectCondition =
        #sql{ // no space between #sql and the brace
            hostVar02 = 4 -- start each SQL comment
                        -- with a double hyphen
        }
}

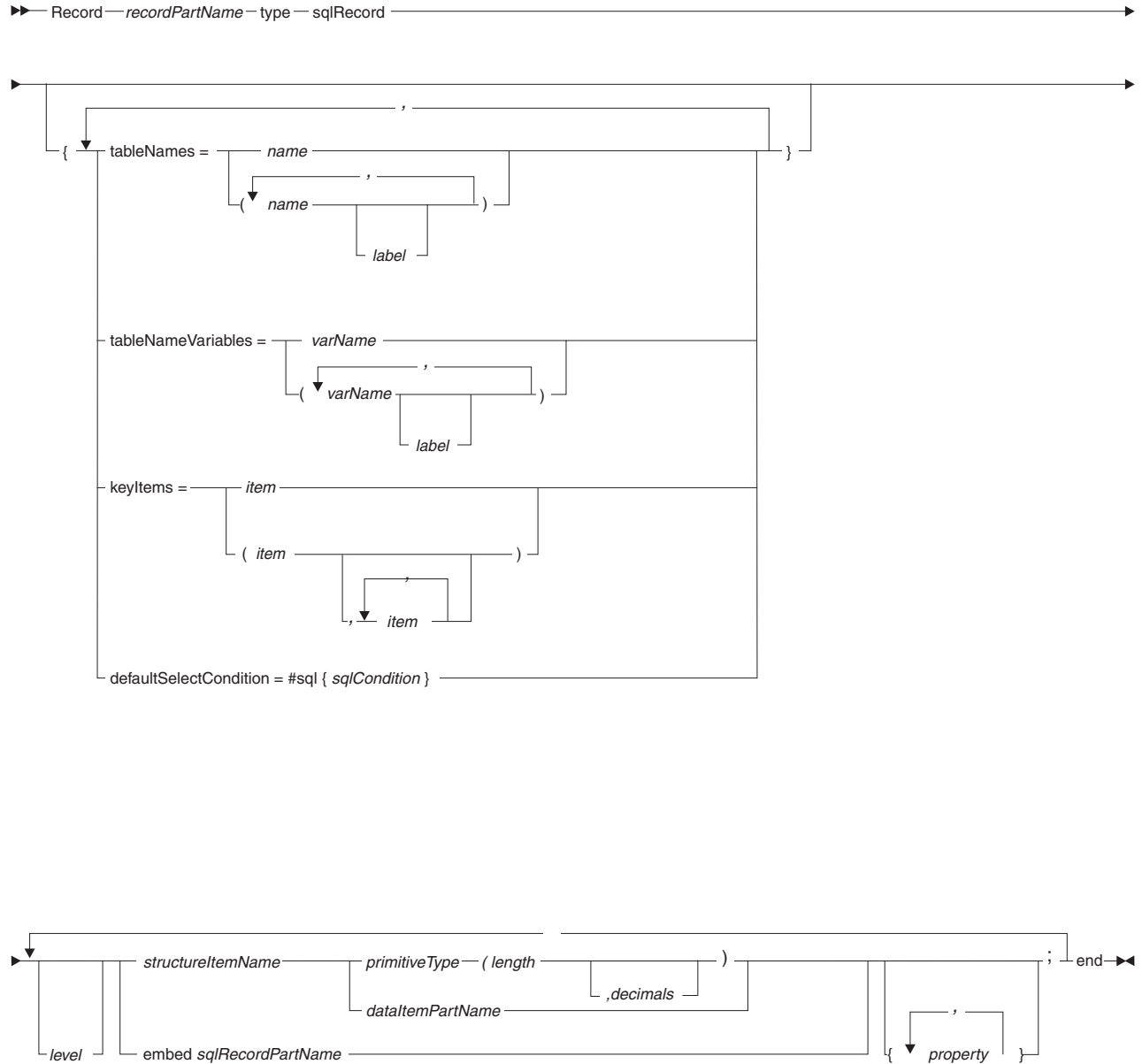
// The structure of an SQL record has no hierarchy
10 myHostVar01 myDataItemPart01
{
    column = column01,
    isNullable = no,
    isReadOnly = no
};
10 myHostVar02 myDataItemPart02
{
    column = column02,
```

```

        isNullable = yes,
        isReadOnly = no
    };
end

```

The syntax diagram for an SQL record part is as follows:



Record recordPartName sqlRecord

Identifies the part as a record part of type sqlRecord and specifies the name.
For rules, see *naming conventions*.

tableNames = (name label, ..., name label)

Lists the table or tables that are accessed by the SQL record. If you specify a label for a given table name, the label is included in the default SQL statements that are associated with the record.

You may include a double quote mark (") in a table name by preceding the quote mark with the escape character (\). That convention is necessary, for example, when a table name is one of these SQL reserved words:

- CALL
- COLUMNS
- FROM
- GROUP
- HAVING
- INSERT
- ORDER
- SELECT
- SET
- UPDATE
- VALUES
- WHERE

Each of those names must be embedded in a doubled pair of quote marks. If the only table name is *SELECT*, for example, the *tableNames* clause is as follows:

```
tableNames=("\"SELECT\"")
```

(A similar situation applies when one of those SQL reserved words is used as a column name.)

tableNameVariables = (varName label, ..., varName label)

Lists one or more table-name variables, each of which contains the name of a table that is accessed by the SQL record. The name of a table is determined only at run time.

The variable may be qualified by a library name and may be subscripted.

If you specify a label for a given table-name variable, the label is included in the default SQL statements that are associated with the record.

You may use table-name variables alone or with table names; but the use of any table-name variable ensures that the characteristics of your SQL statement will be determined only at run time.

You may include a double quote mark (") in a table-name variable by preceding the quote mark with the escape character (\).

keyItems = (item, ..., item)

Indicates whether the column associated with a given record item is part of the key in the database table. If the database table has a composite key, the order of the record items that are defined as keys must match the order of the columns that are keys in the database table.

defaultSelectCondition = #sql { sqlCondition }

Defines part of the search criterion in the WHERE clause of an implicit SQL statement. The value of *defaultSelectCondition* does not include the SQL keyword WHERE.

EGL provides an implicit SQL statement with a WHERE clause when you code one of these EGL statements:

- **get**
- **open**

- **execute** (only when you request an implicit SQL DELETE or UPDATE statement)

The implicit SQL statements are not stored in the EGL source code. For an overview of those statements, see *SQL support*.

level

Integer that indicates the hierarchical position of a structure item.

structureItemName

Name of a structure item. For rules, see *naming conventions*.

primitiveType

The primitive type assigned to the structure item.

length

The structure item's length, which is an integer. The value of a memory area that is based on the structure item includes the specified number of characters or digits.

decimals

For a numeric type (BIN, DECIMAL, NUM, NUMC, or PACF), you may specify *decimals*, which is an integer that represents the number of places after the decimal point. The maximum number of decimal positions is the smaller of two numbers: 18 or the number of digits declared as *length*. The decimal point is not stored with the data.

dataItemPartName

Specifies the name of a dataItem part that acts as a model of format for the structure item being declared. For details, see *typeDef*.

embed *sqlRecordPartName*

Specifies the name of a record part of type sqlRecord and embeds the structure of that record part into the current record. The embedded structure does not add a level of hierarchy to the current record. For details, see *typeDef*.

property

An item property, as described in *Overview of EGL properties and overrides*. In an SQL record, the SQL item properties are particularly important.

Related concepts

"EGL projects, packages, and files" on page 7

"Overview of EGL properties and overrides" on page 40

"Parts" on page 11

"References to parts" on page 16

"Record parts" on page 490

"SQL support" on page 171

"Typedef" on page 20

Related tasks

"Syntax diagram" on page 506

Related reference

"Arrays" on page 64

"DataItem part in EGL source format" on page 284

"EGL source format" on page 292

"Function part in EGL source format" on page 381

"Indexed record part in EGL source format" on page 492

"MQ record part in EGL source format" on page 493

“Naming conventions” on page 468
 “Primitive types” on page 27
 “Program part in EGL source format” on page 478
 “References to variables and constants” on page 34
 “Relative record part in EGL source format” on page 495
 “Serial record part in EGL source format” on page 497
 “SQL item properties” on page 57

Properties that support variable-length records

When you declare a record part, you can include properties that support the use of variable-length records, but only as follows:

- In relation to EGL-generated COBOL programs, you can use variable-length serial or indexed records for accessing VSAM files, and you can use variable-length MQ records for accessing MQSeries message queues
- In relation to EGL-generated Java programs, you can use variable-length serial records for accessing sequential files, variable-length serial or indexed records for accessing VSAM files, and variable-length MQ records for accessing MQSeries message queues

Variable-length records with the `lengthItem` property

The *lengthItem* property, if present, identifies an item that is used when:

- Your code reads a record from a file or queue. The length item receives the number of bytes read into the variable-length record.
- Your code writes a record. The length item specifies the number of bytes to add to the file or queue.

The length item can be any of the following:

- A structure item in the same record
- A structure item in a record that is global to the program or is local to the function that accesses the record (the length item may be qualified with a record variable declared in the program or function)
- A data item that is global to the program or is local to the function that accesses the record

The length item has these characteristics:

- Has a primitive type of BIN, DECIMAL, INT, NUM, or SMALLINT
- Contains no decimal place
- Allows for 9 digits at most

An example of a variable-length record part with the `lengthItem` property is as follows:

```

Record mySerialRecordPart1 type serialRecord
{
  fileName = "myFile",
  lengthItem = myOtherField
}
10 myField01 BIN(4);    // 2 bytes long
10 myField02 NUM(3);    // 3 bytes long
10 myField03 CHAR(20); // 20 bytes long
end
  
```

When writing a record, the value of the length item must fall between item boundaries, unless the item is a character item. For example, a record of type `mySerialRecordPart1` can have the length item, `myOtherField`, set to 2, 5, 6, 7, ... ,

24 , 25. A record with myOtherField set to 2 only contains a value for myField01; a record with myOtherField set to 5 contains values for myField01 and myField02; a record with myOtherField set to 6 through 24 also contains part of myField03.

Variable-length records with the numElementsItem property

The *NumElementsItem* property, if present, identifies an item that is used when your code adds to or updates the file or queue. The variable-length record must have an array as the last, top-level structure item. The value in the number of elements item represents the actual number of array elements that are written. The value can range from 0 to the maximum, which is the *occurs* value specified in the declaration of the last, top-level structure item in the record.

The number of bytes written is equal to the sum of the following:

- The number of bytes in the fixed-length part of the record.
- The value of the number of elements item multiplied by the number of bytes in each element of the ending array.

The number of elements item has these characteristics:

- Has a primitive type of BIN, DECIMAL, INT, NUM, SMALLINT
- Contains no decimal place
- Allows for 9 digits at most

An example of a variable-length record part with the numElementsItem property is as follows:

```
Record mySerialRecordPart2 type serialRecord
{
    fileName = "myFile",
    numElementsItem = myField02
}
10 myField01 BIN(4);    // 2 bytes long
10 myField02 NUM(3);    // 3 bytes long
10 myField03 CHAR(20)[3]; // 60 bytes long
    20 mySubField01 CHAR(10);
    20 mySubField02 CHAR(10);
end
```

Writing a record of type mySerialRecordPart2 with the number of elements item myField02 set to 2 results in a variable-length record with myField01, myField02, and two occurrences of myField03 being written to the file or queue.

The number of elements item must be an item in the fixed-length part of the variable-length record. Use an unqualified reference to name the number of elements item. For example, use myField02 rather than myRecord.myField02.

The number of elements item has no effect when you are reading a record from the file.

Variable-length records with both lengthItem and numElementsItem properties

If both the lengthItem and the numElementsItem properties are specified for a variable-length record, the length of the record is calculated using the number of elements item. The calculated length is moved to the record length item before the record is written to the file.

Variable-length records passed on a call or transfer

If variable-length records are passed on a call, these statements apply:

- Space is reserved for the maximum length specified for the record
- If the value of the callLink element, property **type**, is remoteCall or ejbCall, the length item (if any) must be inside the record; for details, see *callLink element*

Similarly, if variable-length records are passed on a transfer, space is reserved for the maximum length specified for the record.

Related concepts

“MQSeries support” on page 161

“Record types and properties” on page 13

Related reference

“callLink element” on page 443

“MQ record properties” on page 169

Run unit

A *run unit* is a set of programs that are related by local calls or (in some cases) by transfers. Each run unit has these characteristics:

- The programs operate together as a group. When a hard error occurs but is not handled, all the programs in the run unit are removed from memory.
- The programs share the same run-time properties. The same databases and files are available throughout the run unit, for example, and when you invoke sysLib.connect or sysLib.connectionService to connect to a database dynamically, the connection is present in any program that receives control in the same run unit.

Run units are of the following types:

- The *iSeries COBOL run unit* is composed of the main program and the programs called (directly or indirectly) from that program. The run unit ends when a main program ends, as in these cases:
 - The program returns to the non-EGL program from which it was started; or
 - The program issues a **transfer** statement of the form *transfer to a transaction*.
- The *Java run unit* is composed of programs that run in a single thread.

A new run unit can start with a main program, as when the user invokes the program. A **transfer** statement also invokes a main program but continues the same run unit.

In the following cases, a called program is the initial program of a run unit:

- The call is a call from an EJB session bean; or
- The call is a remote call, except that the same run unit continues in the following case--
 - The called program is generated by EGL or VisualAge Generator; and
 - No TCP/IP listener is involved in the call.

All programs in a Java run unit are affected by the same Java run-time properties.

Related concepts

“Java run-time properties” on page 421

“Linkage options part” on page 439

Related reference

“Default database” on page 198

“sysLib.connect” on page 543

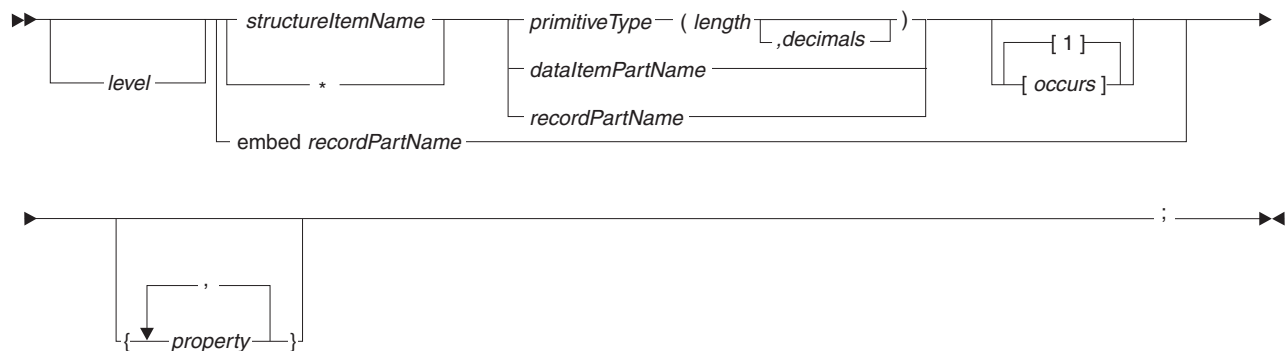
“sysLib.connectionService” on page 545

Structure item in EGL source format

An example of a structure item is as follows:

```
10 address;  
20 street01 CHAR(20);  
20 street02 CHAR(20);
```

The syntax diagram for a structure item is as follows:



level

Integer that indicates the hierarchical position of a structure item.

structureItemName

Name of a structure item. For rules, see *Naming conventions*.

- * Indicates that the structure item describes a *filler*, which is a memory area whose name is of no importance. An asterisk is not valid in a reference to an area of memory, as noted in *References to variables and constants*.

primitiveType

The primitive type assigned to the structure item.

length

The structure item's length, which is an integer. The value of a memory area that is based on the structure item includes the specified number of characters or digits.

decimals

For a numeric type (BIN, DECIMAL, NUM, NUMC, or PACF), you may specify *decimals*, which is an integer that represents the number of places after the decimal point. The maximum number of decimal positions is the smaller of two numbers: 18 or the number of digits declared as *length*. The decimal point is not stored with the data.

dataItemPartName

Specifies the name of a dataItem part that acts as a model of format for the structure item being declared. For details, see *typeDef*.

embed *recordPartName*

Specifies the name of a record part and embeds the structure of that record

part into the current record. The embedded structure does not add a level of hierarchy to the current record. For details, see *typeDef*.

recordPartName

Specifies the name of a record part and includes the structure of that record part in the current record. In the absence of the word *embed*, the record structure is included as a substructure of the structure item being declared. For details, see *typeDef*.

occurs

The number of elements in an array of structure items. The default is 1, which means that the structure item is not an array unless you specify otherwise. For details, see Arrays.

property

An item property, as described in *Overview of EGL properties and overrides*.

Related concepts

"Overview of EGL properties and overrides" on page 40

Related reference

"Arrays" on page 64

"Naming conventions" on page 468

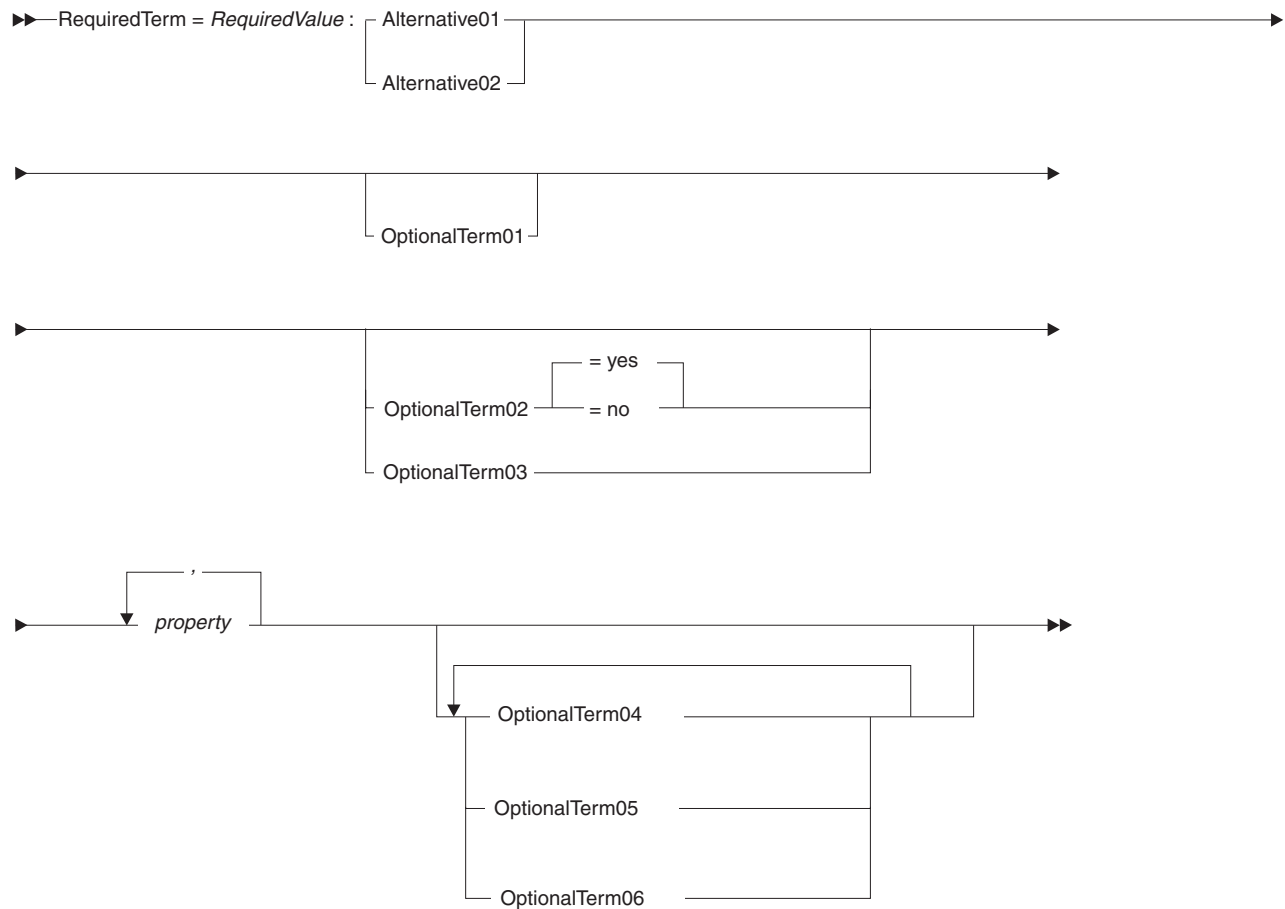
"Primitive types" on page 27

"References to variables and constants" on page 34

"Typedef" on page 20

Syntax diagram

The IBM syntax diagram lets you see quickly how to construct a statement or command. An example of such a diagram is as follows:



Read the diagram from left-to-right, top-to-bottom, following the *main path*, which is the line that begins on the left with double arrowheads (>>). As you follow the main path you may select an entry on a subordinate path, in which case you continue reading from left-to-right along the subordinate path.

In the example, the main path is composed of four line segments. It is important to see this. The second and third line segments of the main path each begins with a single arrowhead (>) and includes subordinate information. The fourth line segment of the main path line also begins with a single arrowhead (>), includes returning arrows and subordinate information, and ends with two arrowheads facing each other (><).

A term (or symbol) that is not in italics must be specified exactly as shown. In the example, you specify the term **RequiredTerm** as is. In contrast, a term in italics is a placeholder for a value that you specify. In the example, you might include any of the following symbols in place of *RequiredValue*:

```
myVariable
50
"0h!"
```

The specific requirements for an italicized term (for example, whether a string or number is appropriate) are explained in the text that follows the syntax diagram, not in the syntax diagram itself.

If a diagram shows a non-alphanumeric character, you type that character as part of the syntax. After you specify a value for *RequiredValue*, for instance, you type a colon (:) and a blank.

If you are allowed to select from any of several terms, the terms are shown in a stack. In the example, you can specify the term **Alternative01** or **Alternative02**.

If (as in this case) you *must select* a term from those listed in a stack, one of the choices (arbitrarily specified) is on the top line of the stack. If you are not required to select a term, the terms are all below the top line of the stack, as is true of **OptionalTerm01**.

A value that is on a path but is shown in an elevated way (as is true of = **yes**) is the default value for the stack in which the value appears. The example indicates that you can specify any of the following strings, and the first two are equivalent:

optionalTerm01 = yes

optionalTerm01

optionalTerm01 = no

OptionalTerm02

An arrow returning to the left above a term indicates that you can use the term repeatedly. In the example, you specify values for *property*, each separated from the next with a comma.

An arrow returning to the left above a vertical stack means that you can choose from the list of entries in any order. In the example, each of the following strings is valid (as are other variations), but none is required:

OptionalTerm04 OptionalTerm05

OptionalTerm06

OptionalTerm04 OptionalTerm06 OptionalTerm05

System words in alphabetical order

This page lists the system functions, then the system variables.

System function/Invocation	Description
<i>arrayName.appendAll</i> <i>arrayName.appendAll</i> (<i>appendArray</i>)	Acts as follows, but only for dynamic arrays: <ul style="list-style-type: none">• Appends to the array that is referenced by <i>arrayName</i>, adding a copy of the array that is referenced by <i>appendArray</i>• Increments the array size by the number of added elements• Assigns an appropriate index value to each of the appended elements
<i>arrayName.appendElement</i> <i>arrayName.appendElement</i> (<i>content</i>)	Places an element to the end of a dynamic array and increments the array size by one; <i>arrayName</i> is the name of the array, and <i>content</i> is the new content (a constant or variable of the appropriate type for the array)

System function/Invocation	Description
<code>arrayName.insertElement</code> <code>arrayName.insertElement</code> <code>(content, index)</code>	Places an element in front of the element that is now at the specified location in a dynamic array, increments the array size by one, and increments the index of each element that resides after the inserted element; <i>arrayName</i> is the name of the array, <i>content</i> is the new content (a constant or variable of the appropriate type for the array), and <i>index</i> is an integer literal or a numeric variable that indicates the location of the new element
<code>arrayName.removeAll</code> <code>arrayName.removeAll()</code>	Removes <i>arrayName</i> (a dynamic array) from memory
<code>arrayName.removeElement</code> <code>arrayName.removeElement</code> <code>(index)</code>	Removes the element at the specified location in <i>arrayName</i> (a dynamic array), decrements the array size by one, and decrements the index of each element that resides after the removed element; <i>index</i> is an integer literal or a numeric variable that indicates the location of the element to be removed
<code>recordName.resourceAssociation</code>	Contains the system resource name associated with the record <i>recordName</i> and allows for a dynamic change to that name
<code>mathLib.abs</code> <code>result = mathLib.abs</code> <code>(numericItem)</code>	Returns absolute value of <i>numericItem</i>
<code>mathLib.acos</code> <code>result = mathLib.acos</code> <code>(numericItem)</code>	Returns arccosine of <i>numericItem</i>
<code>mathLib.asin</code> <code>result = mathLib.asin (numericItem)</code>	Returns arcsine of <i>numericItem</i>
<code>mathLib.atan</code> <code>result = mathLib.atan</code> <code>(numericItem)</code>	Returns arctangent of <i>numericItem</i>
<code>mathLib.atan2</code> <code>result = mathLib.atan2 (y, x)</code>	Computes the principal value of the arc tangent of <i>y/x</i> , using the signs of both arguments to determine the quadrant of the return value
<code>mathLib.ceiling</code> <code>result = mathLib.ceiling</code> <code>(numericItem)</code>	Returns smallest integer not less than <i>numericItem</i>
<code>mathLib.compareNum</code> <code>result = mathLib.compareNum</code> <code>(numericItem1, numericItem2)</code>	Returns a result (-1, 0, or 1) that indicates whether <i>numericItem1</i> is less than, equal to, or greater than <i>numericItem2</i>
<code>mathLib.cos</code> <code>result = mathLib.cos</code> <code>(numericItem)</code>	Returns cosine of <i>numericItem</i>
<code>mathLib.cosh</code> <code>result = mathLib.cosh (numericItem)</code>	Returns hyperbolic cosine of <i>numericItem</i>

System function/Invocation	Description
mathLib.exp <i>result</i> = mathLib.exp (<i>numericItem</i>)	Returns exponential value of <i>numericItem</i>
mathLib.floatingAssign <i>result</i> = mathLib.floatingAssign (<i>numericItem</i>)	Returns <i>numericItem</i> as a double-precision floating-point number
mathLib.floatingDifference <i>result</i> = mathLib.floatingDifference (<i>numericItem1</i> , <i>numericItem2</i>)	Returns the difference between <i>numericItem1</i> and <i>numericItem2</i>
mathLib.floatingMod <i>result</i> = mathLib.floatingMod (<i>numericItem1</i> , <i>numericItem2</i>)	Calculates the floating point remainder of <i>numericItem1</i> divided by <i>numericItem2</i> , with the result having the same sign as <i>numericItem1</i>
mathLib.floatingProduct <i>result</i> = mathLib.floatingProduct (<i>numericItem1</i> , <i>numericItem2</i>)	Returns product of <i>numericItem1</i> and <i>numericItem2</i>
mathLib.floatingQuotient <i>result</i> = mathLib.floatingQuotient (<i>numericItem1</i> , <i>numericItem2</i>)	Returns quotient of <i>numericItem1</i> divided by <i>numericItem2</i>
mathLib.floatingSum <i>result</i> = mathLib.floatingSum (<i>numericItem1</i> , <i>numericItem2</i>)	Returns sum of <i>numericItem1</i> and <i>numericItem2</i>
mathLib.floor <i>result</i> = mathLib.floor (<i>numericItem</i>)	Returns the largest integer not greater than <i>numericItem</i>
mathLib.frexp <i>result</i> = mathLib.frexp (<i>numericItem</i> , <i>integer</i>)	Splits a number into a normalized fraction in the range of .5 to 1 (which is the returned value) and a power of 2 (which is returned in <i>integer</i>)
mathLib.ldexp <i>result</i> = mathLib.ldexp (<i>numericItem</i> , <i>integer</i>)	Returns <i>numericItem</i> multiplied by 2 to the power of <i>integer</i>
mathLib.log <i>result</i> = mathLib.log (<i>numericItem</i>)	Returns the natural logarithm of <i>numericItem</i>
mathLib.log10 <i>result</i> = mathLib.log10 (<i>numericItem</i>)	Returns the base 10 logarithm of <i>numericItem</i>
mathLib.maximum <i>result</i> = mathLib.maximum (<i>numericItem1</i> , <i>numericItem2</i>)	Returns the greater of <i>numericItem1</i> and <i>numericItem2</i>
mathLib.minimum <i>result</i> = mathLib.minimum (<i>numericItem1</i> , <i>numericItem2</i>)	Returns the lesser of <i>numericItem1</i> and <i>numericItem2</i>
mathLib.modf <i>result</i> = mathLib.modf (<i>numericItem1</i> , <i>numericItem2</i>)	Splits <i>numericItem1</i> into integral and fractional parts, both with the same sign as the number; places the integral part in <i>numericItem2</i> ; and returns the fractional part

System function/Invocation	Description
mathLib.pow <i>result</i> = mathLib.pow (<i>numericItem1</i> , <i>numericItem2</i>)	Returns <i>numericItem1</i> raised to the power of <i>numericItem2</i>
mathLib.precision <i>result</i> = mathLib.precision (<i>numericItem</i>)	Returns the maximum precision (in decimal digits) for <i>numericItem</i>
mathLib.round <i>result</i> = mathLib.round (<i>numericItem</i> , <i>integer</i>) <i>result</i> = mathLib.round (<i>numericExpression</i>)	Rounds a number or expression to a nearest value (for example, to the nearest thousands) and returns the result
mathLib.sin <i>result</i> = mathLib.sin (<i>numericItem</i>)	Returns sine of <i>numericItem</i>
mathLib.sinh <i>result</i> = mathLib.sinh (<i>numericItem</i>)	Returns hyperbolic sine of <i>numericItem</i>
mathLib.sqrt <i>result</i> = mathLib.sqrt (<i>numericItem</i>)	Returns the square root of <i>numericItem</i> if <i>numericItem</i> is greater than or equal to zero
mathLib.tan <i>result</i> = mathLib.tan (<i>numericItem</i>)	Returns the tangent of <i>numericItem</i>
mathLib.tanh <i>result</i> = mathLib.tanh (<i>numericItem</i>)	Returns the hyperbolic tangent of <i>numericItem</i>
strLib.compareStr <i>result</i> = strLib.compareStr (<i>target</i> , <i>targetSubstringIndex</i> , <i>targetSubstringLength</i> , <i>source</i> , <i>sourceSubstringIndex</i> , <i>sourceSubstringLength</i>)	Compares two substrings in accordance with their ASCII or EBCDIC order at run time and returns a value (-1, 0, or 1) to indicate which is greater.
strLib.concatenate <i>result</i> = strLib.concatenate (<i>target</i> , <i>source</i>)	Concatenates <i>target</i> and <i>source</i> ; places the new string in <i>target</i> ; and returns an integer that indicates whether <i>target</i> was long enough to contain the new string
strLib.concatenateWithSeparator <i>result</i> = strLib.concatenateWithSeparator (<i>target</i> , <i>source</i> , <i>separator</i>)	Concatenates <i>target</i> and <i>source</i> , inserting <i>separator</i> between them; places the new string in <i>target</i> ; and returns an integer that indicates whether <i>target</i> was long enough to contain the new string
strLib.copyStr <i>result</i> = strLib.copyStr (<i>target</i> , <i>targetSubstringIndex</i> , <i>targetSubstringLength</i> , <i>source</i> , <i>sourceSubstringIndex</i> , <i>sourceSubstringLength</i>)	Copies one substring to another

System function/Invocation	Description
<code>strLib.findStr</code> <code>result = strLib.findStr</code> <code>(source, sourceSubstringIndex</code> <code>, sourceSubstringLength, searchString)</code>	Searches for the first occurrence of a substring within a string
<code>strLib.getNextToken</code> <code>result = strLib.getNextToken</code> <code>(target, source, sourceSubstringIndex,</code> <code>sourceStringLength, characterDelimiter)</code>	Searches a string for the next token and copies the token to <i>target</i>
<code>strLib.setBlankTerminator</code> <code>result = strLib.setBlankTerminator</code> <code>(target)</code>	Replaces a null terminator and any subsequent characters in a string with spaces, so that a string value returned from a C or C++ program can operate correctly in an EGL-generated program
<code>strLib.setNullTerminator</code> <code>result = strLib.setNullTerminator</code> <code>(target)</code>	Changes all trailing spaces in a string to nulls
<code>strLib.setSubStr</code> <code>result = strLib.setSubStr</code> <code>(target, targetSubstringIndex,</code> <code>targetSubstringLength, source)</code>	Replaces each character in a substring with a specified character
<code>strLib.strLen</code> <code>result = strLib.strLen</code> <code>(source)</code>	Returns the number of bytes in an item, excluding any trailing spaces or nulls
<code>sysLib.bytes</code> <code>result = sysLib.bytes</code> <code>(itemOrRecord)</code>	Returns the number of bytes in a named area of memory
<code>sysLib.calculateChkDigitMod10</code> <code>sysLib.calculateChkDigitMod10</code> <code>(input, checkLength, result)</code>	Places a modulus-10 check digit in a character item that begins with a series of integers
<code>sysLib.calculateChkDigitMod11</code> <code>sysLib.calculateChkDigitMod11</code> <code>(input, checkLength, result)</code>	Places a modulus-11 check digit in a character item that begins with a series of integers
<code>sysLib.clearRequestAttr</code> <code>sysLib.clearRequestAttr</code> <code>(key)</code>	Removes the argument that is associated with the specified key in the request object; the function is useful in page handlers and in programs that run in Web applications
<code>sysLib.clearScreen</code> <code>sysLib.clearScreen()</code>	Clears the screen, as is useful before the program issues a converse statement in a text application
<code>sysLib.clearSessionAttr</code> <code>sysLib.clearSessionAttr</code> <code>(key)</code>	Removes the argument that is associated with the specified key in the session object; the function is useful in page handlers and in programs that run in Web applications
<code>sysLib.commit</code> <code>sysLib.commit()</code>	Calls services to save recoverable file, database, and message queue updates since the last commit
<code>sysLib.connect</code> <code>sysLib.connect</code> <code>(database, userID, password,</code> <code>disconnectOption, isolationLevel)</code>	Allows a program to connect to a database at run time

System function/Invocation	Description
sysLib.connectionService sysLib.connectionService (<i>userID, password, serverName, product, release, connectionOption</i>)	Allows a program to connect or disconnect to a database at run time and receives (optionally) the database product name and release level; but this function is supported only when you have requested compatibility with VisualAge Generator
sysLib.convert sysLib.convert (<i>target, direction, conversionTable</i>)	Converts data between EBCDIC (host) and ASCII (workstation) formats or performs code-page conversion within a single format
sysLib.disconnect sysLib.disconnect (<i>database</i>)	Disconnects from the specified database or (if no database is specified) from the current database
sysLib.disconnectAll sysLib.disconnectAll()	Disconnects from all the currently connected databases
sysLib.displayMsgNum sysLib.displayMsgNum (<i>msgNumber</i>)	Retrieves a value from the program's message table
sysLib.fieldInputLength sysLib.displayMsgNum (<i>msgNumber</i>)	Returns the number of characters that the user typed in the input field when the text form was last presented
sysLib.getRequestAttr sysLib.getRequestAttr (<i>key, variable</i>)	Uses a specified key to retrieve an argument from the request object into a specified variable; the function is useful in page handlers and in programs that run in Web applications
sysLib.getSessionAttr sysLib.getSessionAttr (<i>key, variable</i>)	Uses a specified key to retrieve an argument from the session object into a specified variable; the function is useful in page handlers and in programs that run in Web applications
sysLib.getVAGSysType <i>result</i> = sysLib.getVAGSysType()	Identifies the target system in which the program is running; but this function is supported only when you have requested compatibility with VisualAge Generator
sysLib.java <i>result</i> = sysLib.java (<i>identifierOrClass, method, argumentList</i>)	Invokes a method on a native Java object or class and may return a value
sysLib.javaGetField <i>result</i> = sysLib.javaGetField (<i>identifierOrClass, field</i>)	Returns the value of a specified field of a specified object or class
sysLib.javaIsNull <i>result</i> = sysLib.javaIsNull(<i>identifier</i>)	Returns a value (1 for true, 0 for false) to indicate whether a specified identifier refers to a null object
sysLib.javaIsObjID <i>result</i> = sysLib.javaIsObjID (<i>identifier</i>)	Returns a value (1 for true, 0 for false) to indicate whether a specified identifier is in the EGL Java namespace
sysLib.javaRemove sysLib.javaRemove (<i>identifier</i>)	Removes the specified identifier from the EGL Java namespace and, if no other identifiers refer to the object, removes the object

System function/Invocation	Description
sysLib.javaRemoveAll sysLib.javaRemoveAll()	Removes all identifiers and objects from the EGL Java namespace
sysLib.javaSetField sysLib.javaSetField (<i>identifierOrClass, field, value</i>)	Sets the value of a field in a Java object or class
sysLib.javaStore sysLib.javaStore (<i>storeId, method, argumentList</i>)	Invokes a method and places the returned object (or null) into the EGL Java namespace, along with a specified identifier
sysLib.javaStoreCopy sysLib.javaStoreCopy (<i>storeId, method, argumentList</i>)	Creates a new identifier based on another in the EGL Java namespace, so that both refer to the same object
sysLib.javaStoreField sysLib.javaStoreField (<i>storeId, identifierOrClass, field</i>)	Places the value of a class field or object field into the EGL Java namespace
sysLib.javaStoreNew sysLib.javaStoreNew (<i>storeId, class, argumentList</i>)	Invokes the constructor of a class and places the new object into the EGL Java namespace
sysLib.javaType <i>result</i> = sysLib.javaType (<i>identifier</i>)	Returns the fully qualified name of the class of an object in the EGL Java namespace
sysLib.maximumSize <i>result</i> = sysLib.maximumSize (<i>arrayName</i>)	Returns the maximum number of rows that can be in the specified dynamic array of data items or records
sysLib.pageEject sysLib.pageEject()	Advances print-form output to the top of the next page, as is useful before the program issues a print statement
sysLib.queryCurrentDatabase sysLib.queryCurrentDatabase (<i>product, release</i>)	Returns the product and release number of the currently connected database
sysLib.rollback sysLib.rollback()	Calls system services to back out recoverable file, database, and message queue updates since the last commit point
sysLib.setCurrentDatabase sysLib.setCurrentDatabase (<i>database</i>)	Makes the specified database the currently active one
sysLib.setError sysLib.setError (<i>elementInError, msgKey, msgInsertList</i>)	Associates a message with an item in a page handler or with the page handler itself
sysLib.setLocale sysLib.setLocale (<i>languageCode, countryCode, variant</i>)	Sets the Java locale in page handlers and in programs that run in a Web application
sysLib.setRemoteUser sysLib.setRemoteUser (<i>userID, password</i>)	Sets the userid and password that are used on calls to remote programs from Java programs
sysLib.setRequestAttr sysLib.setRequestAttr (<i>key, argument</i>)	Uses a specified key to place a specified argument in the request object; this function is used in page handlers and in programs that run in a Web application

System function/Invocation	Description
sysLib.setSessionAttr sysLib.setSessionAttr (key, argument)	Uses a specified key to place a specified argument in the session object; this function is used in page handlers and in programs that run in a Web application
sysLib.size result = sysLib.size (arrayName)	Returns the number of elements in the array <i>arrayName</i> , which may be a structure-item array, a static array of data items or records, or a dynamic array of data items or records; alternatively, <i>arrayName</i> may be a data table, in which case the function returns the number of rows
sysLib.startTransaction sysLib.startTransaction (request, prID, termID)	Invokes a main program asynchronously, associates that program with a printer or terminal device, and passes a record
sysLib.validationFailed sysLib.validationFailed (msgNumber)	Involved in message presentation on a text or print form
sysLib.verifyChkDigitMod10 sysLib.verifyChkDigitMod10 (input, checkLength, result)	Verifies a modulus-10 check digit in a character item that begins with a series of integers
sysLib.verifyChkDigitMod11 sysLib.verifyChkDigitMod11 (input, checkLength, result)	Verifies a modulus-11 check digit in a character item that begins with a series of integers
sysLib.wait sysLib.wait (timeInSeconds)	Suspends execution for the specified number of seconds.

The next table lists the system variables.

System variable	Description
arrayName.maximumSize	The maximum number of elements in the array <i>arrayName</i> , which must be a dynamic array
sysVar.arrayIndex	One of these: <ul style="list-style-type: none"> Number of the first element in an array that matches the search condition of a simple logical expression with an in operator Zero, if no array element matches the search condition
sysVar.callConversionTable	Name of the conversion table used to dynamically convert data for calling remote programs, for starting remote asynchronous transactions, or for accessing remote files
sysVar.commitOnConverse	Determines whether to cause a commit and a release of resources in a text application, before a non-segmented program issues a converse statement

System variable	Description
<code>sysVar.currentDate</code>	Contains the current system date in eight-digit Gregorian format (YYYYMMDD)
<code>sysVar.currentFormattedDate</code>	Contains the current system date in the system default long Gregorian format
<code>sysVar.currentFormattedJulianDate</code>	Contains the current system date in the system default long Julian format
<code>sysVar.currentFormattedTime</code>	Retrieves the current system time in HH:MM:SS format
<code>sysVar.currentJulianDate</code>	Contains the current system date in seven-digit Julian format (YYYYDDD)
<code>sysVar.currentShortDate</code>	Contains the current system date in six-digit Gregorian format (YYMMDD)
<code>sysVar.currentShortJulianDate</code>	Contains the current system date in five-digit Julian format (YYDDD)
<code>sysVar.errorCode</code>	<p>Receives a status code after any of the following events:</p> <ul style="list-style-type: none"> • The invocation of a call statement, if that statement is in a try block • An I/O operation on an indexed, MQ, relative, or serial file • The invocation of almost any system function in these cases-- <ul style="list-style-type: none"> – The invocation is within a try block; or – The program is running in VisualAge Generator Compatibility mode and <code>sysVar.handleSysLibErrors</code> is set to 1 <p>(For an overview that includes details on <code>sysVar.errorCode</code>, see <i>Exception handling</i>.)</p>
<code>sysVar.eventKey</code>	Identifies the key that the user pressed to return a form to an EGL text program
<code>sysVar.formConversionTable</code>	Name of the conversion table used to dynamically convert data in text and print forms
<code>sysVar.handleHardIOErrors</code>	Controls whether a program continues to run after a hard I/O error occurs on an I/O operation
<code>sysVar.handleOverflow</code>	Controls error processing after an arithmetic overflow
<code>sysVar.handleSysLibErrors</code>	Specifies whether the value of the system variable <code>sysVar.errorCode</code> is affected by the invocation of a system function
<code>sysVar.mqConditionCode</code>	Completion code from an MQSeries API call following an add or scan I/O operation for an MQ record
<code>sysVar.overflowIndicator</code>	Set to 1 when arithmetic overflow occurs

System variable	Description
sysVar.printerAssociation	Allows you to specify, at run time, the destination of the output that is associated with the file name <i>printer</i>
sysVar.remoteSystemID	System identifier for location of remote program or file
sysVar.returnValue	Contains the external return code to be checked when the program ends; the check is by the JCL, by the command processor, or by the calling high-level language program
sysVar.segmentedMode	Changes the effect of the converse statement in a text application
sysVar.sessionID	System-dependent user identifier or terminal identifier for your program
sysVar.sqlca	Contains the entire SQL communication area (SQLCA) returned for the last SQL I/O operation
sysVar.sqlcode	Contains the return code for the most recently completed SQL I/O operation
sysVar.sqlerrd sysVar.sqlerrd[index]	Is a static 6-element array, where each element contains the corresponding SQL communication area (SQLCA) value that was returned from the last SQL I/O option
sysVar.sqlerrmc	Contains the substitution variables for the error message associated with the return code in sysVar.sqlcode
sysVar.sqlIsolationLevel	Indicates the level of independence of one database transaction from another, as is meaningful only if you are generating Java output
sysVar.sqlstate	Contains the SQL state value for the most recently completed SQL I/O operation
sysVar.sqlwarn sysVar.sqlwarn[index]	Is a static 11-element array, where each element contains a warning byte returned in the SQL communications area (SQLCA) for the last SQL I/O operation and where the index is one greater than the warning number in the SQL SQLCA description; sqlwarn[1], for example, refers to SQLWARN0
sysVar.systemType	Identifies environment in which program is running
sysVar.terminalID	Contains the terminal identifier in environments where one is available
sysVar.transactionID	Contains the name of the current transaction ID or the name of the next transaction to be used after a converse statement
sysVar.transferName	Allows you to specify, at run time, the name of the program or transaction to which you want to transfer

System variable	Description
sysVar.userID	Contains the user identifier in environments where one is available
sysVar.validationMsgNum	Contains the value assigned by sysLib.validationFailed

Related reference

“Arrays” on page 64

“EGL statements” on page 70

“System words” on page 79

Data conversion (system words)

The data conversion system words are described in the next table.

System word/Usage	Description
sysLib.convert sysLib.convert (<i>target</i> , <i>direction</i> , <i>conversionTable</i>)	Converts data between EBCDIC (host) and ASCII (workstation) formats or performs code-page conversion within a single format
sysVar.callConversionTable sysVar.callConversionTable	Name of the conversion table used to dynamically convert data for calling remote programs, for starting remote asynchronous transactions, or for accessing remote files
sysVar.formConversionTable sysVar.formConversionTable	Name of the conversion table used to dynamically convert data in text and print forms

Related reference

“Data conversion” on page 279

“EGL statements” on page 70

“Function invocations” on page 316

“System words” on page 79

“System words in alphabetical order” on page 508

sysLib.convert

The system function **sysLib.convert** converts data between EBCDIC (host) and ASCII (workstation) formats or performs code-page conversion within a single format. You can use **sysLib.convert** as the function name in a function invocation statement.

►► sysLib.Convert (*target* , *direction* , *conversionTable*) ; ◀◀

target

Name of the record, data item, or form that has the format you want to convert. The data is converted in place based on the item definition of the lowest-level items (items with no substructure) in the target object.

Variable-length records are converted only for the length of the current record. The length of the current record is calculated using the record's numElementsItem or is set from the record's lengthItem. A conversion error occurs and the program ends if the variable-length record ends in the middle of a numeric field or a DBCHAR character.

direction

Direction of conversion. "R" and "L" (including the quotation marks) are the only valid values. Required if *conversionTable* is specified; optional otherwise.

"R" Default value. The data is assumed to be in remote format and is converted to local format.

"L" Data is assumed to be in local format and is converted to remote format (as defined in the conversion table).

conversionTable

Data item or literal (eight characters, optional) that specifies the name of the conversion table to be used for data conversion. The default value is the conversion table associated with the national language code specified when the program was generated.

Definition considerations: You can use the linkage options part to request that automatic data conversion be generated for remote calls, to start remote asynchronous transactions, or for remote file access. Automatic conversion is always performed using the data structure defined for the argument being converted. If an argument has multiple formats, do not request automatic conversion. Instead, code the program to explicitly call **sysLib.convert** with redefined record declarations that correctly map the current values of the argument.

Example:

```
Record RecordA
  record_type char(3);
  item1 char(20);
end

Record RecordB
  record_type char(3);
  item2 bigint;
  item3 decimal(7);
  item4 char(8);
end

Program ProgramX type basicProgram
  myRecordA RecordA;
  myRecordB RecordB {redefines = myRecordA};
  myConvTable char(8);

  function main();
    myConvTable = "ELACNENU"; // conversion table for US English
    if (myRecordA.record_type = "00A")
      sysLib.convert(myRecordA, "L", myConvTable);
    else;
      sysLib.convert(myRecordB, "L", myConvTable);
    end
    call ProgramY myRecordA;
  end
end
```

Related reference

"Data conversion" on page 279

"Data conversion (system words)" on page 518

"sysVar.callConversionTable"

sysVar.callConversionTable

The system variable **sysVar.callConversionTable** contains the name of the conversion table that is used to convert data when your program does the following at run time:

- Passes arguments in a call to a program on a remote system
- Passes arguments when invoking a remote program by way of the system function `sysLib.startTransaction`
- Accesses a file at a remote location

The conversion occurs when the data is being moved between EBCDIC-based and ASCII-based systems or between systems that use different code pages. Conversion is possible only if the linkage options part used at generation time specifies **PROGRAMCONTROLLED** as the value of property **conversionTable** in the **callLink** or **asynchLink** element. Conversion does not occur, however, if **PROGRAMCONTROLLED** is specified but **sysVar.callConversionTable** is blank.

Characteristics: The characteristics of **sysVar.callConversionTable** are as follows:

Primitive type
CHAR

Data length
8

Value saved across segments?
Yes

Definition considerations: You should use **sysVar.callConversionTable** to switch conversion tables in a program or to turn data conversion on or off in a program.

sysVar.callConversionTable is initialized to blanks. To cause conversion to occur, make sure that the linkage options part includes the value **PROGRAMCONTROLLED**, as described earlier, and move the name of a conversion table to the system variable. You can set **sysVar.callConversionTable** to an asterisk (*) to use the default conversion table for the default national language code. For Java, this setting references the default locale on the target system provided the locale is mapped to one of the languages that can be specified for the **targetNLS** build descriptor option. For COBOL, this setting references the default national language code you specified when you installed EGL Server for iSeries.

Conversion is performed on the system that originates the call, invocation, or file access. When you define multiple levels of a record structure, conversion is performed on the lowest level items (the items with no substructure).

You can use **sysVar.callConversionTable** in these ways:

- As the source or target operand in an assignment or **move** statement
- As a variable in a logical expression
- As an argument in a **return** or **exit** statement

A comparison of **sysVar.callConversionTable** with another value tests true only if the match is exact. If you initialize **sysVar.callConversionTable** with a lowercase value, for example, the lowercase value matches only a lowercase value.

The value that you place in **sysVar.callConversionTable** remains unchanged for purposes of comparison.

Example:

```
sysVar.callConversionTable = "ELACNENU";
// conversion table for US English COBOL generation
```

Related reference

"Data conversion" on page 279

"Data conversion (system words)" on page 518

"sysLib.startTransaction" on page 617

"targetNLS" on page 267

sysVar.formConversionTable

The system variable **sysVar.formConversionTable** contains the name of the conversion table that is used for bidirectional text conversion when an EGL-generated Java program acts as follows:

- Shows a text or print form that includes a series of Hebrew or Arabic characters; or
- Shows a text form that accepts a series of Hebrew or Arabic characters from a user.

Characteristics: The characteristics of **sysVar.formConversionTable** are as follows:

Primitive type

CHAR

Data length

8

Value saved across segments?

Yes

Related reference

"Bidirectional language text" on page 282

"Data conversion" on page 279

"Data conversion (system words)" on page 518

Date and time (system words)

The date-and-time system variables let you retrieve the system date and time in a variety of formats, as shown in the next table.

System variable	Description
sysVar.currentDate	Contains the current system date in eight-digit Gregorian format (YYYYMMDD)
sysVar.currentFormattedDate	Contains the current system date in the system default long Gregorian format
sysVar.currentFormattedJulianDate	Contains the current system date in the system default long Julian format
sysVar.currentFormattedTime	Retrieves the current system time in HH:MM:SS format
sysVar.currentJulianDate	Contains the current system date in seven-digit Julian format (YYYYDDD)
sysVar.currentShortDate	Contains the current system date in six-digit Gregorian format (YYMMDD)
sysVar.currentShortJulianDate	Contains the current system date in five-digit Julian format (YYDDD)

Related reference

"System words" on page 79

"System words in alphabetical order" on page 508

"EGL statements" on page 70

sysVar.currentDate

The system variable **sysVar.currentDate** contains the current system date in eight-digit Gregorian format (YYYYMMDD).

The **sysVar.currentDate** value is updated automatically before each reference. The value is numeric and contains no separator characters.

You can use **sysVar.currentDate** as the source in an **assignment** or **move** statement or as the argument in a **return** or **exit** statement.

The characteristics of **sysVar.currentDate** are as follows:

Primitive type

NUM

Data length

8

Value saved across segments

No

Example:

```
myDate = sysVar.currentDate
```

Related reference

"Date and time (system words)" on page 521

sysVar.currentFormattedDate

The system variable **sysVar.currentFormattedDate** contains the current system date in long Gregorian format. The value is automatically updated each time system variable is referenced by your program.

For COBOL programs, the system administrator for EGL run-time services sets the format at installation.

For Java programs, the format is in this Java run-time property:

```
vgj.datemask.gregorian.long.NLS
```

NLS

The NLS (national language support) code specified in the Java run-time property **vgj.nls.code**. The code is one of those listed for the targetNLS build descriptor option. Uppercase English (code ENP) is not supported.

For additional details on **vgj.nls.code**, see *Java run-time properties (details)*.

The format specified in **vgj.datemask.gregorian.long.NLS** includes DD (for numeric day), MM (for numeric month), and YYYY (for numeric year), with characters other than D, M, Y, or digits used as separators. You can specify the format in the **dateMask** build descriptor option, and the default format is specific to the locale.

You can use **sysVar.currentFormattedDate** as the source in an **assignment** or **move** statement or as the argument in a **return** or **exit** statement.

Make sure that this Gregorian long date format is the same as the date format specified for the SQL database manager. Matching the two formats enables **sysVar.currentFormattedDate** to produce dates in the format expected by the database manager.

The characteristics of **sysVar.currentFormattedDate** are as follows:

Primitive type

CHAR

Data length

10

Value saved across segments

No

Example:

```
myDate = sysVar.currentFormattedDate;
```

Related concepts

"Build descriptor part" on page 234

"Java run-time properties" on page 421

Related tasks

"Editing Java run-time properties in a build descriptor" on page 93

Related reference

"Date and time (system words)" on page 521

"Java run-time properties (details)" on page 423

"targetNLS" on page 267

sysVar.currentFormattedJulianDate

The system variable **sysVar.currentFormattedJulianDate** contains the current system date in long Julian format. The value is automatically updated each time the system variable is referenced by your program

For COBOL programs, the system administrator for EGL run-time services sets the format at installation.

For Java programs, the format is in this Java run-time property:

```
vgj.datemask.julian.long.NLS
```

NLS

The NLS (national language support) code specified in the Java run-time property **vgj.nls.code**. The code is one of those listed for the targetNLS build descriptor option. Uppercase English (code ENP) is not supported.

For additional details on **vgj.nls.code**, see *Java run-time properties (details)*.

The format specified in **vgj.datemask.julian.long.NLS** includes DDD (for numeric day) and YYYY (for numeric year), with characters other than D, Y, or digits used as separators. You can specify the format in the **dateMask** build descriptor option, and the default format is specific to the locale.

You can use **sysVar.currentFormattedJulianDate** as the source in an assignment or **move** statement or as the argument in a **return** or **exit** statement.

The characteristics of **sysVar.currentFormattedJulianDate** are as follows:

Primitive type

CHAR

Data length

8

Value saved across segments

No

Example:

```
myDate = sysVar.currentFormattedJulianDate;
```

Related concepts

"Build descriptor part" on page 234

"Java run-time properties" on page 421

Related tasks

"Editing Java run-time properties in a build descriptor" on page 93

Related reference

"Date and time (system words)" on page 521

"Java run-time properties (details)" on page 423

"targetNLS" on page 267

sysVar.currentFormattedTime

The system variable **sysVar.currentFormattedTime** retrieves the current system time in HH:MM:SS format. The value is automatically updated each time it is referenced by your program.

You can use **sysVar.currentFormattedTime** in these ways:

- As the source in an assignment or **move** statement
- As the argument in an **exit** or **return** statement

The characteristics of **sysVar.currentFormattedTime** are as follows:

Primitive type

CHAR

Data length

8

Value saved across segments

No

Example:

```
timeField = sysVar.currentFormattedTime;
```

Related reference

"Date and time (system words)" on page 521

sysVar.currentJulianDate

The system variable **sysVar.currentJulianDate** contains the current system date in seven-digit Julian format (YYYYDDD).

The **sysVar.currentJulianDate** value is updated automatically before each reference. The value is numeric and contains no separator characters.

You can use **sysVar.currentJulianDate** as the source in an **assignment** or **move** statement or as the argument in a **return** or **exit** statement.

The characteristics of **sysVar.currentJulianDate** are as follows:

Primitive type

NUM

Data length

7

Value saved across segments

No

Example:

```
myDay = sysVar.currentJulianDate;
```

Related reference

“Date and time (system words)” on page 521

sysVar.currentShortDate

The system variable **sysVar.currentShortDate** contains the current system date in six-digit Gregorian format (YYMMDD).

The **sysVar.currentShortDate** value is automatically updated each time it is referenced by the program. The returned value is numeric and contains no separator characters.

You can use **sysVar.currentShortDate** as the source in an **assignment** or **move** statement or as the argument in a **return** or **exit** statement.

The characteristics of **sysVar.currentShortDate** are as follows:

Primitive type

NUM

Data length

6

Value saved across segments

No

Example:

```
myDay = sysVar.currentShortDate;
```

Related reference

“Date and time (system words)” on page 521

sysVar.currentShortJulianDate

The system variable **sysVar.currentShortJulianDate** contains the current system date in five-digit Julian format (YYDDD). The value is automatically updated each time it is referenced by your program.

The value is numeric and contains no separator characters.

You can use **sysVar.currentShortJulianDate** as the source in an **assignment** or **move** statement or as the argument in a **return** or **exit** statement.

The characteristics of **sysVar.currentShortJulianDate** are as follows:

Primitive type

NUM

Data length

5

Value saved across segments

No

Example:

```
myDay = sysVar.currentShortJulianDate;
```

Related reference

“Date and time (system words)” on page 521

Exception handling and status (system words)

As shown in the next table, exception handling and status words are used to control how programs respond to exceptions and error conditions (such as arithmetic overflow), and to determine what codes are returned when exceptions occur.

System word/Usage	Description
sysLib.displayMsgNum sysLib.displayMsgNum (<i>msgNumber</i>)	Retrieves a value from the program’s message table
sysLib.setError sysLib.setError (<i>elementInError</i> , <i>msgKey</i> , <i>msgInsertList</i>) sysLib.setError (<i>msgText</i>)	Associates a message with an item in a page handler or with the page handler itself
sysLib.validationFailed sysLib.validationFailed (<i>msgNumber</i>)	Involved in message presentation on a text or print form
sysVar.errorCode sysVar.errorCode	<p>Receives a status code after any of the following events:</p> <ul style="list-style-type: none"> • The invocation of a call statement, if that statement is in a try block • An I/O operation on an indexed, MQ, relative, or serial file • The invocation of almost any system function in these cases-- <ul style="list-style-type: none"> – The invocation is within a try block; or – The program is running in VisualAge Generator Compatibility mode and sysVar.handleSysLibErrors is set to 1 <p>(For an overview that includes details on sysVar.errorCode, see <i>Exception handling</i>.)</p>
sysVar.handleHardIOErrors sysVar.handleHardIOErrors	Controls whether a program continues to run after a hard I/O error occurs on an I/O operation
sysVar.handleOverflow sysVar.handleOverflow	Controls error processing after an arithmetic overflow

System word/Usage	Description
sysVar.handleSysLibErrors sysVar.handleSysLibErrors	Specifies whether the value of the system variable sysVar.errorCode is affected by the invocation of a system function
sysVar.mqConditionCode sysVar.mqConditionCode	Contains the completion code from an MQSeries API call following an add or scan I/O operation for an MQ record
sysVar.overflowIndicator sysVar.overflowIndicator	Is set to 1 when arithmetic overflow occurs
sysVar.returnValue sysVar.returnValue	Contains the external return code to be checked when the program ends; the check is by the JCL, by the command processor, or by the calling high-level language program
sysVar.sqlca sysVar.sqlca	Contains the SQL communication area (SQLCA) returned for the last SQL I/O operation
sysVar.sqlcode sysVar.sqlcode	Contains the return code for the most recently completed SQL I/O operation
sysVar.sqlerrd sysVar.sqlerrd[index]	Is a static 6-element array, where each element contains the corresponding SQL communication area (SQLCA) value that was returned from the last SQL I/O option
sysVar.sqlerrmc sysVar.sqlerrmc	Contains the substitution variables for the error message associated with the return code in sysVar.sqlcode
sysVar.sqlstate sysVar.sqlstate	Contains the SQL state value for the most recently completed SQL I/O operation
sysVar.sqlwarn sysVar.sqlwarn[index]	Is a static 11-element array, where each element contains a warning byte returned in the SQL communications area (SQLCA) for the last SQL I/O operation and where the index is one greater than the warning number in the SQL SQLCA description; sqlwarn[1], for example, refers to SQLWARN0
sysVar.validationMsgNum sysVar.validationMsgNum	Contains the value assigned by sysLib.validationFailed

Related reference

“EGL statements” on page 70

“Exception handling” on page 75

“Function invocations” on page 316

“System words” on page 79

“System words in alphabetical order” on page 508

sysLib.displayMsgNum

The system function **sysLib.displayMsgNum** retrieves a value from the program’s message table. The message is presented the next time that a form is presented by a **converse**, **display**, **print**, or **show** statement.

If possible, the message presentation is on the form itself, in the field to which the form property **msgField** refers. If the form property **msgField** has no value, the message is displayed previous to the display of the form, on a separate, modal screen or on a printable page.

sysLib.displayMsgNum takes as its only argument a value that is compared against each cell in the first column of the program's *message table*, which is the data table to which the program's **msgTablePrefix** property refers. The message retrieved by that function is in the second column of the same row.

►► sysLib.displayMsgNum (*msgNumber*); ◄◄

msgNumber

The message is retrieved from the message table by number. The argument must be an integer literal or an item of primitive type SMALLINT or INT or the BIN equivalent.

Related reference

"System words in alphabetical order" on page 508

sysLib.setError

The system function **sysLib.setError** associates a message with an item in a page handler or with the page handler as a whole. The message is placed at the location of a JSF message or messages tag in the JSP and is displayed when the related Web page is displayed.

If a validation function invokes **sysLib.setError**, the Web page is re-displayed automatically when the function ends.

►► sysLib.setError (*itemInError* , *this* , *msgKey* , *itemInsert* , *msgText*); ◄◄

itemInError

The name of the page-handler item that is in error.

this

Refers to the page handler from which **sysLib.setError** is issued. In this case, the message is not specific to an item, but is associated with the page handler as a whole. For details on **this**, see *References to variables and constants*.

msgKey

A character item or literal (type CHAR or MBCHAR) that provides the key into the message resource bundle or properties file used at run time. If the key is blank, the message is a concatenation of message inserts.

itemInsert

The character item or literal that is included as an insert to the output message. The substitution symbol in message text is an integer surrounded by braces, as in this example:

Invalid file name {0}

msgText

The character item or literal that you can specify if you do not specify other arguments. The text is associated with the page as a whole.

You can associate multiple messages with an item or page handler. The EGL run time displays the messages when the page is re-displayed. If control is forwarded (specifically, if the page handler runs a **forward** statement), those messages are lost.

Related concepts

“PageHandler part” on page 469

“References to variables and constants” on page 34

Related tasks

“Syntax diagram” on page 506

Related reference

“forward” on page 315

sysLib.validationFailed

The system function **sysLib.validationFailed** is used in either of two ways:

- If invoked in a field-validation function in a text application, **sysLib.validationFailed** causes the re-presentation of the received text form after all validation functions are processed. The last-invoked **sysLib.validationFailed** determines what message is displayed.
If possible, the message presentation is on the form itself, in the field to which the form property **msgField** refers. If the form property **msgField** has no value, the message is displayed previous to the display of the form, on a separate, modal screen.
- If invoked outside a validation function, **sysLib.validationFailed** presents the specified message the next time that a form is presented by a **converse**, **display**, **print**, or **show** statement. The behavior in this case is like that of **sysLib.displayMsgNum**.

In any case, the value assigned to **sysLib.validationFailed** is stored in the system variable **sysVar.validationMsgNum**.

►► sysLib.validationFailed (9999
msgNumber); ◀◀

msgNumber

The number of the message to display. The argument must be an integer literal or an item of primitive type SMALLINT or INT or the BIN equivalent. This number is compared against each cell in the first column of the program's *message table*, which is the data table to which the program's **msgTablePrefix** property refers. The retrieved message is in the second column of the same row.

The message number is 9999 by default.

Related reference

"sysLib.displayMsgNum" on page 527

"sysVar.validationMsgNum" on page 536

"System words in alphabetical order" on page 508

sysVar.errorCode

The system variable **sysVar.errorCode** receives a status code after any of the following events:

- The invocation of a call statement, if that statement is in a try block
- An I/O operation on an indexed, MQ, relative, or serial file
- The invocation of almost any system function in these cases--
 - The invocation is within a try block; or
 - The program is running in VisualAge Generator Compatibility mode and **sysVar.handleSysLibErrors** is set to 1

The **sysVar.errorCode** values associated with a given system function are described in relation to the system function, not in the current topic.

You can use **sysVar.errorCode** in these ways:

- As the source or target in an assignment or **move** statement
- As a variable in a logical expression
- As the argument in a system function

sysVar.errorCode is set to 0 if the call, I/O, or system function invocation is successful.

The characteristics of **sysVar.errorCode** are as follows:

Primitive type

CHAR

Data length

8

Is value always restored after a converse?

Yes

For an overview that includes details on **sysVar.errorCode**, see *Exception handling*.

Definition considerations: If you are generating Java code, the list of possible **sysVar.errorCode** values is provided in *EGL Java run-time error codes*.

I/O errors and COBOL code: In relation to COBOL code, the following rules apply:

- If you set the build descriptor option **sysCodes** to YES and perform an I/O operation on a resource other than a database, **sysVar.errorCode** contains a return code that is specific to the type of resource; for example, specific to a VSAM file. To interpret this code, refer to the appropriate manual for the resource.
- If you set that build description option to NO, however, the file-related return codes are independent of the type of resource.

The next table indicates some of the COBOL file status codes that can be returned, along with the value that is placed in **sysVar.errorCode** if you generate COBOL output with the build descriptor option **sysCodes** set to NO. Also shown is the EGL I/O error value, which is unaffected by the value in **sysCodes**.

COBOL file status code (as placed in sysVar.errorCode when sysCodes is set to YES)	sysVar.errorCode value when sysCodes is set to NO	EGL I/O error value (a blank in this column means "not applicable")
00000000, 00000005, 00000007	00000000	
00000002	00000103	duplicate, ioError
00000004 (var record format)	00000000	
00000004 (other)	00000220	format, hardIOError, ioError
00000010, 00000014, 00000046	00000102	endOfFile, ioError
00000022	00000206	ioError, unique
00000023 (start)	00000102	endOfFile, ioError
00000023 (other)	00000205	noRecordFound, ioError
00000024, 00000034 (access method not relative or relative key not 0)	0000025A	full, hardIOError, ioError
00000035	00000251	fileNotFound, hardIOError, ioError
00000038	00000218	fileNotAvailable, hardIOError, ioError
00000039, 00000095	00000220	format, ioError
0000009D (iSeries COBOL only)	00000381	deadlock, hardIOError, ioError

The next table shows the setting for **sysVar.errorCode** when the run-time system returns other COBOL file status codes.

Type of request	sysVar.errorCode value when sysCodes is set to NO	EGL I/O error value
open	00000500	ioError, hardIOError
close or unlock	00000989	ioError, hardIOError
read or start	00000987	ioError, hardIOError
write	00000988	ioError, hardIOError

Example:

```
if (sysVar.errorCode = "00000008")
  exit program;
end
```

Related reference

"Exception handling" on page 75
 "Exception handling and status (system words)" on page 526
 "System words in alphabetical order" on page 508
 "sysVar.handleSysLibErrors" on page 533
 "try" on page 355

sysVar.handleHardIOErrors

The system variable **sysVar.handleHardIOErrors** controls whether a program continues to run after a hard error occurs on an I/O operation. The default value is 0. For background information, see *Exception handling*.

You can use **sysVar.handleHardIOErrors** in any of these ways:

- As the source or target of an assignment or **move** statement (also allowed in the "for count" of a **move** statement)
- As the variable in a logical expression used in a **case**, **if**, or **while** statement
- As the argument in a **return** or **exit** statement

The characteristics of **sysVar.handleHardIOErrors** are as follows:

Primitive type

NUM

Data length

1

Is value always restored after a converse?

Yes

Example

```
sysVar.handleHardIOErrors = 1;
```

Related reference

"Exception handling" on page 75

"Exception handling and status (system words)" on page 526

sysVar.handleOverflow

The system variable **sysVar.handleOverflow** controls error processing after an arithmetic overflow. Two types of overflow conditions are detected:

- *User variable overflow* occurs when the result of an arithmetic operation or assignment to a numeric item causes a significant value (not decimal positions) to be lost due to the length of the item.
- *Maximum value overflow* occurs when the result of an arithmetic operation is greater than 18 digits.

You can set **sysVar.handleOverflow** to one of the following values. (The default setting is 0.)

Value	Effect on user overflow	Effect on maximum value overflow
0	The program sets the system variable sysVar.overflowIndicator to 1 and continues	The program ends with an error message
1	The program ends with an error message	The program ends with an error message
2	The program sets the system variable sysVar.overflowIndicator to 1 and continues	The program sets the system variable sysVar.overflowIndicator to 1 and continues

You can use **sysVar.handleOverflow** in these ways:

- As the source or target in an assignment or **move** statement (also allowed in the "for count" of a **move** statement)
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of **sysVar.handleOverflow** are as follows:

Primitive type

NUM

Data length

1

Is value always restored after a converse?

Yes

Example:

```
sysVar.handleOverflow = 2;
```

Related reference

"Assignments" on page 296

"Exception handling and status (system words)" on page 526

"sysVar.overflowIndicator" on page 534

sysVar.handleSysLibErrors

The system variable **sysVar.handleSysLibErrors** specifies whether the value of system variable **sysVar.errorCode** is affected by the invocation of a system function. However, **sysVar.handleSysLibErrors** is available only when VisualAge Generator compatibility is in effect, as explained in *Compatibility with VisualAge Generator*.

For details and restrictions, see *Exception handling*.

You can use **sysVar.handleSysLibErrors** in these ways:

- As the source or target in an assignment or **move** statement
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of **sysVar.handleSysLibErrors** are as follows:

Primitive type

NUM

Data length

1

Is value always restored after a converse?

Only in a non-segmented text program; for details see *Segmentation*

Example:

```
sysVar.handleSysLibErrors = 1;
```

Related concepts

"Compatibility with VisualAge Generator" on page 276

"Segmentation in text applications" on page 151

Related reference

"Exception handling" on page 75

“Exception handling and status (system words)” on page 526

“System words” on page 79

“sysVar.errorCode” on page 530

sysVar.mqConditionCode

The system variable **sysVar.mqConditionCode** contains the completion code from an MQSeries API call following an **add** or **scan** I/O operation for an MQ record.

Valid values and their related meanings are as follows:

00 OK

01 WARNING

02 FAILED

You can use **sysVar.mqConditionCode** in these ways:

- As the source or target in an assignment or **move** statement (also allowed in the “for count” of a **move** statement)
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of **sysVar.mqConditionCode** are as follows:

Primitive type

NUM

Data length

2

Is value always restored after a converse?

Yes

Example:

```
add MQRecord;  
if (sysVar.mqConditionCode = 0)  
  // continue  
else  
  exit program;  
end
```

Related concepts

“MQSeries support” on page 161

Related reference

“Exception handling” on page 75

“Exception handling and status (system words)” on page 526

“System words” on page 79

sysVar.overflowIndicator

The system variable **sysVar.overflowIndicator** is set to 1 when arithmetic overflow occurs. By checking the value of this variable, you can test for overflow conditions.

After detection of an overflow condition, **sysVar.overflowIndicator** is not reset automatically. You must include code in your program to reset **sysVar.overflowIndicator** to 0 before performing any calculations that may trigger overflow checks.

You can use **sysVar.overflowIndicator** in these ways:

- As the source or target in an assignment or **move** statement (also allowed in the "for count" of a **move** statement)
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of **sysVar.overflowIndicator** are as follows:

Primitive type

NUM

Data length

1

Is value always restored after a converse?

Yes

Example:

```
sysVar.overflowIndicator = 0;
sysVar.handleOverflow = 2;
a = b;
if (sysVar.overflowIndicator = 1)
    add errorrecord;
end
```

Related reference

"Assignments" on page 296

"Exception handling and status (system words)" on page 526

"sysVar.handleOverflow" on page 532

sysVar.returnCode

The system variable **sysVar.returnCode** contains an external return code, as set by your program and made available to the operating system. It is not possible to pass return codes from one EGL program to another. A non-zero return code does not cause EGL to run an onException block, for example. The initial value of **sysVar.returnCode** is zero. The value must be in the range of 0 to 512.

In relation to Java code, **sysVar.returnCode** is meaningful only for a main text program (which runs outside of J2EE) or a main batch program (which runs either outside of J2EE or in a J2EE application client). The purpose of **sysVar.returnCode** in this context is to provide a code for the command file or exec that invokes the program. If the program ends with an error that is not under the program's control, the EGL run time ignores the setting of **sysVar.returnCode** and attempts to return the value 693.

In relation to COBOL code, the following statements apply:

- **sysVar.returnCode** contains a code that in most cases is provided to the operating system and to any caller that is not an EGL-generated program. If the program ends with an error that is not handled by your code, the EGL run time attempts to return a value greater than 512.
- **sysVar.returnCode** is implemented using the COBOL **RETURN-CODE** special register.

You can use **sysVar.returnCode** in these ways:

- As the source or target in an assignment or **move** statement (also allowed in the "for count" of a **move** statement)
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of **sysVar.returnValue** are as follows:

Primitive type

BIN

Data length

9

Is value always restored after a converse?

Yes

Example:

```
sysVar.returnValue = 6;
```

Related reference

“Exception handling and status (system words)” on page 526

sysVar.validationMsgNum

The system variable **sysVar.validationMsgNum** is available in a text application and contains the value assigned by **sysLib.validationFailed**. The value is reset to zero in each of the following cases:

- The program initializes
- The program issues a converse, display, or print statement
- The program reissues a converse statement to display a text form as the result of a validation error

You can test the value of **sysVar.validationMsgNum** to determine if a validation function reported an error.

You can use **sysVar.validationMsgNum** in these ways:

- As the source or target of an assignment or **move** statement (also allowed in the “for count” of a **move** statement)
- As the variable in a logical expression
- As the argument in a **return** or **exit** statement

The characteristics of **sysVar.validationMsgNum** are as follows:

Primitive type

INT

Is value always restored after a converse?

No

Example

```
/*Keep the first message number that was set
during validation routines */
if (sysVar.validationMsgNum > 0)
    sysLib.validationFailed(10);
end
```

Related reference

“converse” on page 306

“display” on page 309

“print” on page 341

“sysLib.validationFailed” on page 529

“System words in alphabetical order” on page 508

File and database (system words)

File and database words enable programs to access or manipulate system and database resources, as shown in the next table.

System word/Usage	Description
<i>recordName</i> .resourceAssociation <i>recordName</i> .resourceAssociation = "myFile"	Contains the system resource name associated with the record <i>recordName</i> and allows for a dynamic change to that name
sysLib.commit sysLib.commit()	Calls services to save recoverable file, database, and message queue updates since the last commit
sysLib.connect sysLib.connect (<i>database</i> , <i>userID</i> , <i>password</i> , <i>disconnectOption</i> , <i>isolationLevel</i>)	Allows a program to connect to a database at run time
sysLib.connectionService sysLib.connectionService (<i>userID</i> , <i>password</i> , <i>serverName</i> , <i>product</i> , <i>release</i> , <i>connectionOption</i>)	Allows a program to connect or disconnect to a database at run time and receives (optionally) the database product name and release level; but this function is supported only when you have requested compatibility with VisualAge Generator
sysLib.disconnect sysLib.disconnect (<i>database</i>)	Disconnects from the specified database or (if no database is specified) from the current database
sysLib.disconnectAll sysLib.disconnectAll()	Disconnects from all the currently connected databases
sysLib.queryCurrentDatabase sysLib.queryCurrentDatabase (<i>product</i> , <i>release</i>)	Returns the product and release number of the currently connected database
sysLib.rollback sysLib.rollback()	Calls system services to back out recoverable file, database, and message queue updates since the last commit point
sysLib.setCurrentDatabase sysLib.setCurrentDatabase (<i>database</i>)	Makes the specified database the currently active one
sysVar.commitOnConverse sysVar.commitOnConverse	Determines whether to cause a commit and a release of resources in a text application, before a non-segmented program issues a converse statement

System word/Usage	Description
sysVar.errorCode sysVar.errorCode	<p>Receives a status code after any of the following events:</p> <ul style="list-style-type: none"> • The invocation of a call statement, if that statement is in a try block • An I/O operation on an indexed, MQ, relative, or serial file • The invocation of almost any system function in these cases-- <ul style="list-style-type: none"> – The invocation is within a try block; or – The program is running in VisualAge Generator Compatibility mode and sysVar.handleSysLibErrors is set to 1 <p>(For an overview that includes details on sysVar.errorCode, see <i>Exception handling</i>.)</p>
sysVar.mqConditionCode sysVar.mqConditionCode	Contains the completion code from an MQSeries API call following an add or scan I/O operation for an MQ record
sysVar.sqlca sysVar.sqlca	Contains the entire SQL communication area (SQLCA) returned for the last SQL I/O operation
sysVar.sqlcode sysVar.sqlcode	Contains the return code for the most recently completed SQL I/O operation
sysVar.sqlerrd sysVar.sqlerrd[index]	Is a static 6-element array, where each element contains the corresponding SQL communication area (SQLCA) value that was returned from the last SQL I/O option
sysVar.sqlerrmc sysVar.sqlerrmc	Contains the substitution variables for the error message associated with the return code in sysVar.sqlcode
sysVar.sqlIsolationLevel sysVar.sqlIsolationLevel	Indicates the level of independence of one database transaction from another, as is meaningful only if you are generating Java output
sysVar.sqlState sysVar.sqlState	Contains the SQL state value for the most recently completed SQL I/O operation
sysVar.sqlwarn sysVar.sqlwarn[index]	Is a static 11-element array, where each element contains a warning byte returned in the SQL communications area (SQLCA) for the last SQL I/O operation and where the index is one greater than the warning number in the SQL SQLCA description; sqlwarn[2], for example, refers to SQLWARN1

Related concepts

“MQSeries support” on page 161

“SQL support” on page 171

Related reference

“EGL statements” on page 70

“Exception handling” on page 75

“Function invocations” on page 316

“System words” on page 79

“System words in alphabetical order” on page 508

recordName.resourceAssociation

When your program does an I/O operation against a record, the I/O is done on the physical file whose name is in the record-specific variable *recordName*.

resourceAssociation. The variable is initialized in accordance with the resourceAssociation part used at generation time; for details, see *Resource associations and file types*. You can change the system resource name at run time by placing a different value in **resourceAssociation**.

In most cases, you must use the syntax *recordName.resourceAssociation*. You do not need to specify a record name, however, if EGL can determine the record that you intended, as is true in each of these cases:

- I/O is only performed against one record in the program
- **resourceAssociation** is used in a function that performs I/O against only one record
- I/O is performed against multiple records in the program, but all records have the same file name; in this case, the first record that appears as an I/O object is used as the implicit qualifier.

You can use **resourceAssociation** as any of the following:

- The source or target operand of an assignment statement
- An item in a logical expression in a **case**, **if**, or **while** statement
- The argument in a **return** or **exit** statement

The characteristics of **resourceAssociation** are as follows:

Primitive type

CHAR

Data length

Varies by file type

Saved across segment?

Yes

Definition considerations: The value moved into *recordName*.

resourceAssociation must be a valid system resource name for the system and file type that were specified when the program was generated. If more than one record specifies the same file name, modification of **resourceAssociation** for any record with that file name changes the setting of **resourceAssociation** for all records in the program with the same file name.

If a system resource identified in the setting of **resourceAssociation** is open when that record-specific variable is modified, the system resource that *was* in that variable is closed in the following circumstance: an I/O option runs against a record that has the same EGL file name as the record that qualifies **resourceAssociation**.

If two programs are using the same EGL file name, each of the record-specific **resourceAssociation** variables must contain the same value. Otherwise the previously opened system resource is closed when a new one is opened.

A comparison of **resourceAssociation** with another value tests true only if the match is exact. If you initialize **resourceAssociation** with a lowercase value, for example, the lowercase value matches only a lowercase value.

The value that you place in **resourceAssociation** remains unchanged for purposes of comparison.

Files shared across programs: You can set the system resource name either at generation or at runtime:

At generation time

If two programs in the same run unit access the same logical file, you must specify the same system resource name for the logical file at generation to ensure that both programs access the same physical file at run time.

At run time

If you use *recordName*. **resourceAssociation**, each program that accesses the file must set **resourceAssociation** for the file. If two programs in the same run unit access the same logical file, each program must set **resourceAssociation** to the same system resource name to ensure that both programs access the same physical file at run time.

If a system resource is shared by multiple programs, each program that accesses the resource must set **resourceAssociation** to refer to the same resource. Also, if two programs in the same run unit access the same logical file, each program must set **resourceAssociation** to the same system resource name at generation time to ensure that both programs access the same system resource at run time.

MQ records: The system resource name for MQ records defines the queue manager name and queue name. Specify the name in the following format:

queueManagerName:queueName

queueManagerName

Name of the queue manager.

queueName

Name of the queue.

As shown, the names are separated with a colon. However, *queueManagerName* and the colon can be omitted. The system resource name is used as the initial value for the record-specific **resourceAssociation** item and identifies the default queue associated with the record. For further details, see *MQSeries support*.

Target platforms:

Platform	Compatibility considerations
iSeries COBOL	<p>The filetype must be SEQ or VSAM. The value can be moved to resourceAssociation in one of the following ways:</p> <p>LIB/FILE MEMBER Explicitly specify Library, File and Member</p> <p>LIB/FILE The first member in the file is used</p> <p>FILE MEMBER *LIBL is used to find the file</p> <p>FILE *LIBL is used to find the file, and the first member in that file is used</p> <p>When you modify the value in resourceAssociation, the iSeries OVRDBF command has this effect:</p> <ol style="list-style-type: none">1. Closes the old file2. Performs an override to the new value3. Opens the new file <p>The value set in resourceAssociation is propagated from the call level and is changed to all its subordinate call levels. The value is not propagated if the file previously was opened by the program.</p>
Java platforms	None.

Example:

```
if (process = 1)
  myrec.resourceAssociation = "myFile.txt";
else
  myrec.resourceAssociation = "myFile02.txt";
end
```

Related concepts

"MQSeries support" on page 161

"Resource associations and file types" on page 157

Related reference

"File and database (system words)" on page 537

sysLib.commit

The system function **sysLib.commit** saves updates that were made to databases and MQSeries message queues since the last commit. A generated Java program or wrapper also saves the updates done by a remote, CICS-based COBOL program (including updates to CICS recoverable files), but only when the call to the remote COBOL program involves a client-controlled unit of work, as described in *luwControl* in *callLink* element.

In most cases, EGL performs a single-phase commit that affects each recoverable manager in turn. On CICS for z/OS, however, **sysLib.commit** results in a CICS SYNCPOINT, which performs a two-phase commit that is coordinated across all resource managers.

sysLib.commit releases the scan position and the update locks in any file or databases, but an exception is in effect for COBOL programs, when the option `cursorWithHold` is used during database access; for details on the exception, see *prepare* and *open*.

When you use **sysLib.commit** with MQ records, the following statements apply:

- Message queue updates are recoverable only if the *Include message in transaction* option is selected in MQ record part.
- Both message **gets** and **adds** are affected by **commit** and **rollback** for recoverable messages. If a **rollback** is issued following a **get** for a recoverable message, the message is placed back on the input queue so that the input message is not lost when the transaction fails to complete successfully. Also, if a **rollback** is issued following an **add** for a recoverable message, the message is deleted from the queue.

You can enhance performance by avoiding unnecessary use of **sysLib.commit**. For details on when an implicit commit occurs, see *Logical unit of work*.

Special considerations for iSeries COBOL: If the program issued SQL statements, **sysLib.commit** results in an SQL COMMIT WORK. If the program has not issued SQL requests, **sysLib.commit** results in the equivalent of an iSeries COMMIT command.

Example:

```
sysLib.commit();
```

Related concepts

“Logical unit of work” on page 159
“MQSeries support” on page 161
“Run unit” on page 504
“SQL support” on page 171

Related reference

“File and database (system words)” on page 537
“luwControl in callLink element” on page 450
“prepare” on page 339
“open” on page 335
“sysVar.commitOnConverse”
“sysVar.segmentedMode” on page 626

sysVar.commitOnConverse

Setting the system variable **sysVar.commitOnConverse** to 1 causes a commit and a release of resources in a text application, before a non-segmented program issues a converse. The default value is 0 for non-segmented programs and 1 for segmented programs.

You can use **sysVar.commitOnConverse** in any of these ways:

- As the source or target of an assignment or **move** statement
- As the variable in a logical expression used in a **case**, **if**, or **while** statement
- As the argument in a **return** or **exit** statement

Other characteristics of **sysVar.commitOnConverse** are as follows:

Primitive type

NUM

Data length

1

Is value always restored after a converse?

Yes

For details on using this variable, see *Segmentation*.

Related concepts

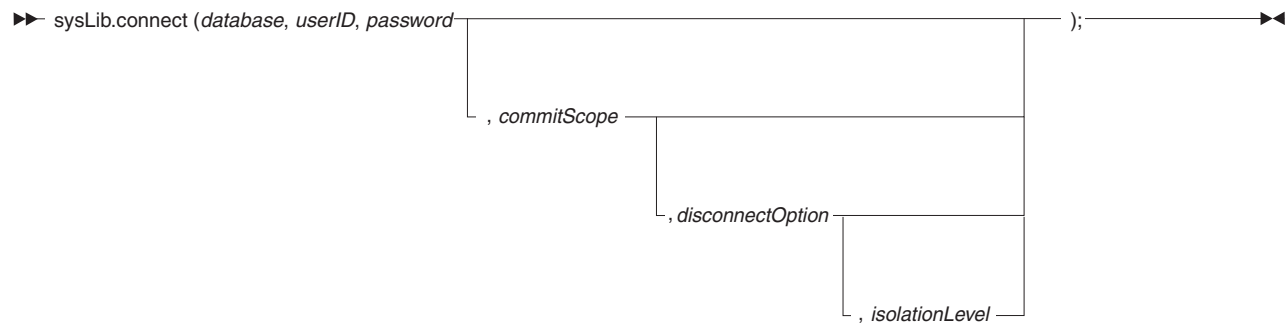
"Segmentation in text applications" on page 151

Related reference

"converse" on page 306

sysLib.connect

The system function **sysLib.connect** allows a program to connect to a database at run time. This function does not return a value.

*database*

Identifies a database.

If your code is running as a COBOL program, the following statements apply:

- Setting *database* to RESET returns to the connection status that is in effect at the beginning of a run unit, as described in *Default database*.
- Otherwise, the value of *database* must be a value in the location column in the SYSIBM.LOCATIONS table, which is in the DB2 UDB subsystem.
- In either case, the system function **sysLib.connect** closes all cursors, releases locks, ends any existing connection, and connects to the database. Despite these actions, invoke **sysLib.commit** or **sysLib.rollback** before invoking **sysLib.connect**.

If your code is running as a Java program, the following statements apply:

- Setting *database* to RESET reconnects to the default database, but if the default database is not available, the connection status remains unchanged; for further details, see *Default database*.
- Otherwise, the physical database name is found by looking up the property **vgj.jdbc.database.server**, where *server* is the name of the database specified on the **sysLib.connect** call. If this property is not defined, the database name that is specified on the **sysLib.connect** call is used as is.
- The format of the database name is different for J2EE connections as compared with non-J2EE connections:

- If you generated the program for a J2EE environment, use the name to which the datasource is bound in the JNDI registry; for example, jdbc/MyDB. This situation occurs if build descriptor option **J2EE** was set to YES.
- If you generated the program for a non-J2EE JDBC environment, use a connection URL; for example, jdbc:db2:MyDB. This situation occurs if option **J2EE** was set to NO.

userID

UserID used to access the database. The argument must be an item of type CHAR and length 8, and a literal is valid. The argument is required, but is ignored for COBOL generation. For background information, see *Database authorization and table names*.

password

Password used to access the database. The argument must be an item of type CHAR and length 8, and a literal is valid. The argument is required, but is ignored for COBOL generation.

commitScope

This parameter is meaningful only if you are generating Java output. The value is one of the following words, and you cannot use quotes and cannot use a variable:

type1 (the default)

Only a *one*-phase commit is supported. A new connection closes all cursors, releases locks, and ends any existing connection; nevertheless, invoke **sysLib.commit** or **sysLib.rollback** before making a type1 connection.

If you use type1 as the value of *commitScope*, the value of parameter *disconnectOption* must be the word *explicit*, as is the default.

type2

A connection to a database does not close cursors, release locks, or end an existing connection. Although you can use multiple connections to read from multiple databases, you should update only one database in a unit of work because only a one-phase commit is available.

twophase

Identical to type2.

disconnectOption

This parameter is meaningful only if you are generating Java output. The value is one of the following words, and you cannot use quotes and cannot use a variable:

explicit (the default)

The connection remains active after the program invokes **sysLib.commit** or **sysLib.rollback**. To release connection resources, a program must issue **sysLib.disconnect**.

If you use type1 as the value of *commitScope*, the value of parameter *disconnectOption* must be set (or allowed to default) to the word *explicit*.

automatic

A commit or rollback ends an existing connection.

conditional

A commit or rollback automatically ends an existing connection unless a cursor is open and the hold option is in effect for that cursor. For details on the hold option, see *open*.

isolationLevel

This parameter indicates the level of independence of one database transaction from another, as is meaningful only if you are generating Java output.

The following words are in order of increasing strictness, and as before, you cannot use quotes and cannot use a variable:

- **readUncommitted**
- **readCommitted**
- **repeatableRead**
- **serializableTransaction**

For details, see the JDBC documentation from Sun Microsystems, Inc.

Definition considerations: `sysLib.connect` sets the following system variables:

- `sysVar.sqlerrd[3]`
- `sysVar.sqlca`
- `sysVar.sqlcode`
- `sysVar.sqlerrmc` (available in COBOL code only)
- `sysVar.sqlwarn[2]`
- `sysVar.sqlwarn[7]` (available in COBOL code only)

Example:

```
sysLib.connect(myDatabase, myUserid, myPassword);
```

Related concepts

"Logical unit of work" on page 159

"Run unit" on page 504

"SQL support" on page 171

Related tasks

"Syntax diagram" on page 506

"Setting up a J2EE JDBC connection" on page 209

"Understanding how a standard JDBC connection is made" on page 208

Related reference

"Database authorization and table names" on page 197

"Default database" on page 198

"File and database (system words)" on page 537

"Java run-time properties (details)" on page 423

"open" on page 335

"sqlDB" on page 262

"sysLib.disconnect" on page 549

"sysVar.sqlerrd" on page 553

"sysVar.sqlca" on page 551

"sysVar.sqlcode" on page 552

"sysVar.sqlerrmc" on page 553

"sysVar.sqlwarn" on page 555

sysLib.connectionService

The system function `sysLib.connectionService` provides two benefits:

- Allows a program to connect or disconnect to a database at run time.
- Receives (optionally) the database product name and release level. You can use the received information in a **case**, **if**, or **while** statement so that run-time processing is dependent on characteristics of the database.

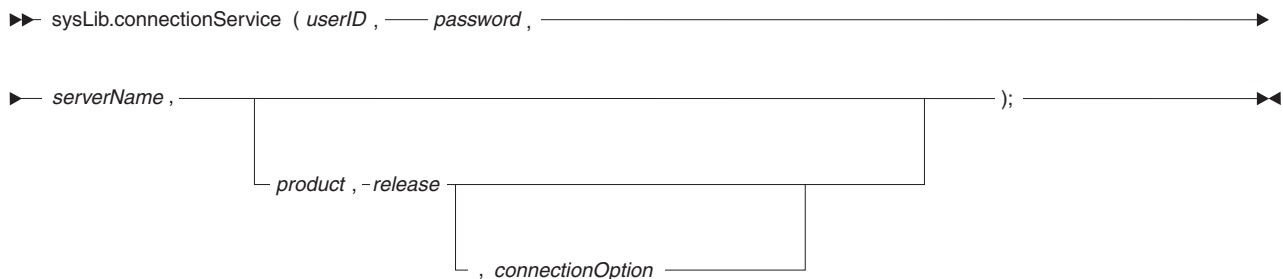
When you use **sysLib.connectionService** to create a new connection, specify the isolation level by setting the system variable **sysVar.sqlIsolationLevel**.

sysLib.connectionService is for use only in programs migrated from VisualAge Generator and EGL 5.0. The function is supported (at development time) if the EGL preference **VisualAge Generator compatibility** is selected or (at generation time) if the build descriptor option **VAGCompatibility** is set to *yes*.

For new programs, use these system functions instead:

- sysLib.connect
- sysLib.disconnect
- sysLib.disconnectAll
- sysLib.queryCurrentDatabase
- sysLib.setCurrentDatabase

sysLib.connectionService does not return a value.



userID

UserID used to access the database. The argument must be an item of type CHAR and length 8; a literal is not valid. The argument is required, but is ignored for COBOL generation. For background information, see *Database authorization and table names*.

password

Password used to access the database. The argument must be an item of type CHAR and length 8; a literal is not valid. The argument is required, but is ignored for COBOL generation.

serverName

Specifies a connection and uses that connection to assign values to the arguments *product* and *release*, if those arguments are included in the invocation of **sysLib.connectionService**.

The argument *serverName* is required and must be an item of type CHAR and length 18. Any of the following values are valid:

blanks (no content)

If a connection is in place, **sysLib.connectionService** maintains that connection. If a connection is not in place, the result (other than to assign values) is to return to the connection status that is in effect at the beginning of a run unit, as described in *Default database*.

RESET

In relation to COBOL, RESET commits changes, closes cursors, releases locks, disconnects from the current database, and returns to the connection status that is in effect at the beginning of a run unit, as described in *Default*

database. Nevertheless, if you intend to specify RESET in this case, invoke **sysLib.commit** or **sysLib.rollback** before invoking **sysLib.connectionService**.

In relation to a Java program, RESET reconnects to the default database; but if the default database is not available, the connection status remains unchanged.

For further details, see *Default database*.

serverName

Identifies a database.

If your code is running as a COBOL program, the following statements apply:

- The value of *serverName* must be a value in the location column in the SYSIBM.LOCATIONS table, which is in the DB2 UDB subsystem
- Specifying a database closes all cursors, releases locks, ends any existing connection, and connects to the database; nevertheless, if you intend to specify a value for *serverName*, invoke **sysLib.commit** or **sysLib.rollback** before invoking **sysLib.connectionService**

If your code is running as a Java program, the following statements apply:

- The physical database name is found by looking up the property **vgj.jdbc.database.server**, where *server* is the name of the server specified on the **sysLib.connectionService** call. If this property is not defined, the server name that is specified on the **sysLib.connectionService** call is used as is.
- The format of the database name is different for J2EE connections as compared with non-J2EE connections:
 - If you generated the program for a J2EE environment, use the name to which the datasource is bound in the JNDI registry; for example, jdbc/MyDB. This situation occurs if build descriptor option **J2EE** was set to YES.
 - If you generated the program for a non-J2EE JDBC environment, use a connection URL; for example, jdbc:db2:MyDB. This situation occurs if option **J2EE** was set to NO.

product

Receives the database product name. The argument, if any, must be an item of type CHAR and length 8.

To determine the string that will be received when your code connects to a particular database, review the product documentation for the database or driver; or run your code in a test environment and write the received value to a file.

release

Receives the database release level. The argument, if any, must be an item of type CHAR and length 8.

To determine the string that will be received when your code connects to a particular database, review the product documentation for the database or driver; or run your code in a test environment and write the received value to a file.

connectionOption

Valid values are as follows:

D1E

D1E is the default. The *1* in the option name indicates that only a *one*-phase commit is supported, and the *E* indicates that any disconnect must be *explicit*. In this case, a commit or rollback has no effect on an existing connection.

If you are generating a non-CICS COBOL program, you can access only one database at a time, and you must explicitly request a disconnect to release the resources from a previous connection (if any) or to connect to a different database.

If you are generating a Java program, the following statements apply:

- A connection to a database does not close cursors, release locks, or end an existing connection. If the run unit is already connected to the same database, however, the effect is equivalent to specifying DISC then D1E.
- You can use multiple connections to read from multiple databases, but you should update only one database in a unit of work because only a one-phase commit is available.

D1A

The *1* in the option name indicates that only a *one*-phase commit is supported, and the *A* indicates that any disconnect is *automatic*.

Characteristics of this option are as follows:

- You can connect to only one database at a time
- A commit, rollback, or connection to a database ends an existing connection

DISC

Disconnect from the specified database. Disconnecting from a database causes a rollback and releases locks, but only for that database.

DCURRENT

Disconnect from the currently connected database. Disconnecting from a database causes a rollback and releases locks, but only for that database.

DALL

Disconnect from all connected databases. Disconnecting from all databases causes a rollback in those database, but not in other recoverable resources.

SET

Set a connection current. (By default, the connection most recently made in the run unit is current.)

The following values are supported for compatibility with VisualAge Generator, but are equivalent to D1E: R, D1C, D2A, D2C, D2E.

Definition considerations: `sysLib.connectionService` sets the following system variables:

- `sysVar.sqlerrd`
- `sysVar.sqlca`
- `sysVar.sqlcode`
- `sysVar.sqlerrmc` (available in COBOL code only)
- `sysVar.sqlwarn`

Example:

```
sysLib.connectionService(myUserId, myPassword,  
    myServerName, myProduct, myRelease, "D1E");
```

Related concepts

"Logical unit of work" on page 159

"Run unit" on page 504

"SQL support" on page 171

Related tasks

"Syntax diagram" on page 506

"Setting up a J2EE JDBC connection" on page 209

"Understanding how a standard JDBC connection is made" on page 208

Related reference

"Database authorization and table names" on page 197

"Default database" on page 198

"File and database (system words)" on page 537

"Java run-time properties (details)" on page 423

"sqlDB" on page 262

"sysVar.sqlca" on page 551

"sysVar.sqlcode" on page 552

"sysVar.sqlerrd" on page 553

"sysVar.sqlerrmc" on page 553

"sysVar.sqlIsolationLevel" on page 554

"sysVar.sqlwarn" on page 555

sysLib.disconnect

The system function **sysLib.disconnect** disconnects from the specified database or (if no database is specified) from the current database.

```
►► sysLib.disconnect ( database ); ◀◀
```

database

A database name that was specified in **sysLib.connect** or **sysLib.connectionService**. Use a literal or variable of a character type.

Before disconnecting, invoke **sysLib.commit** or **sysLib.rollback**.

Related reference

"sysLib.commit" on page 541

"sysLib.connect" on page 543

"sysLib.connectionService" on page 545

"sysLib.rollback" on page 550

"System words in alphabetical order" on page 508

sysLib.disconnectAll

The system function **sysLib.disconnectAll** disconnects from all the currently connected databases.

Before disconnecting, invoke **sysLib.commit** or **sysLib.rollback**.

```
►► sysLib.disconnectAll (); ◀◀
```

Related reference

"sysLib.connect" on page 543

"sysLib.connectionService" on page 545

"System words in alphabetical order" on page 508

sysLib.queryCurrentDatabase

The system function **sysLib.queryCurrentDatabase** returns the product and release number of the currently connected database.

►► sysLib.queryCurrentDatabase (*product*, *release*); ◄◄

product

Receives the database product name. The argument must be an item of type CHAR and length 8.

To determine the string that will be received when your code connects to a particular database, review the product documentation for the database or driver; or run your code in a test environment and write the received value to a file.

release

Receives the database release level. The argument must be an item of type CHAR and length 8.

To determine the string that will be received when your code connects to a particular database, review the product documentation for the database or driver; or run your code in a test environment and write the received value to a file.

Related reference

"System words in alphabetical order" on page 508

sysLib.rollback

The system function **sysLib.rollback** reverses updates that were made to databases and MQSeries message queues since the last commit. That reversal occurs in any EGL-generated application.

A rollback occurs automatically when a program ends as a result of an error condition.

Definition considerations: When you use **sysLib.rollback** with MQ records, the following statements apply:

- Message queue updates are recoverable only if the *Include message in transaction* option is selected in MQ record part.
- Both message **scans** and **adds** are affected by **commit** and **rollback** for recoverable messages. If a **rollback** is issued following a **scan** for a recoverable message, the message is placed back on the input queue so that the input message is not lost when the transaction fails to complete successfully. Also, if a **rollback** is issued following an **add** for a recoverable message, the message is deleted from the queue.

Target platforms:

Platform	Compatibility considerations
iSeries, USS, Windows 2000, Windows NT	Reverses changes to relational databases and MQSeries message queues, as well as changes made to remote server programs that were called using client-controlled unit of work.

Example:

```
sysLib.rollback();
```

Related concepts

“Logical unit of work” on page 159

“MQSeries support” on page 161

“SQL support” on page 171

Related reference

“File and database (system words)” on page 537

sysLib.setCurrentDatabase

The system function **sysLib.setCurrentDatabase** makes the specified database the currently active one.

►► sysLib.setCurrentDatabase (*database*); ◄◄

database

A database name that was specified in **sysLib.connect** or **sysLib.connectionService**. Use a literal or variable of a character type.

Related reference

“sysLib.connect” on page 543

“sysLib.connectionService” on page 545

“System words in alphabetical order” on page 508

sysVar.sqlca

The system variable **sysVar.sqlca** contains the entire SQL communication area (SQLCA). As noted later, the current values of a subset of fields in the SQLCA are available to you after your code accesses a relational database.

You can use **sysVar.sqlca** in these ways:

- As the source in an assignment or **move** statement
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

In order to refer to specific fields in the SQLCA, you must move **sysVar.sqlca** to a base record. The record must have a structure as specified in the SQLCA description for your database management system. Use the base record if you pass the SQLCA contents to a remote program so that the contents will be converted correctly to the remote system data format.

For specific information about the fields that are available in **sysVar.sqlca**, refer to the following topics:

- `sysVar.sqlerrd`
- `sysVar.sqlcode`
- `sysVar.sqlerrmc` (refreshed by the database management system only in COBOL code; not in Java code or in the EGL Debugger)
- `sysVar.sqlState`
- `sysVar.sqlwarn`

The characteristics of **`sysVar.sqlca`** are as follows:

Primitive type

HEX

Data length

272 (136 bytes)

Is value always restored after a converse?

Only in a non-segmented text program; for details see *Segmentation*

Example:

```
myItem = sysVar.sqlca;
```

Related concepts

"Segmentation in text applications" on page 151

"SQL support" on page 171

Related reference

"File and database (system words)" on page 537

"`sysVar.sqlcode`"

"`sysVar.sqlerrd`" on page 553

"`sysVar.sqlerrmc`" on page 553

"`sysVar.sqlState`" on page 555

"`sysVar.sqlwarn`" on page 555

`sysVar.sqlcode`

The system variable **`sysVar.sqlcode`** contains the return code for the most recently completed SQL I/O operation. The code is obtained from the SQL communications area (SQLCA) and can vary with the relational database manager.

You can use **`sysVar.sqlcode`** in these ways:

- As the source in an assignment or **`move`** statement (also allowed in the "for count" of a **`move`** statement)
- As a variable in a logical expression
- As the argument in an **`exit`** or **`return`** statement

The characteristics of **`sysVar.sqlcode`** are as follows:

Primitive type

BIN

Data length

9

Is value always restored after a converse?

Only in a non-segmented text program; for details see *Segmentation*

Example:

```
rcitem = sysVar.sqlcode;
```

Related concepts

“Segmentation in text applications” on page 151

“SQL support” on page 171

Related reference

“File and database (system words)” on page 537

sysVar.sqlerrd

The system array **sysVar.sqlerrd** is a static 6-element array, where each element contains the corresponding SQL communication area (SQLCA) value that was returned from the last SQL I/O option. The value in **sysVar.sqlerrd[3]**, for example, is the third value and indicates the number of rows processed for some SQL requests.

Of the elements in **sysVar.sqlerrd**, only **sysVar.sqlerrd[3]** is refreshed by the database management system for Java code or at debugging time.

You can use a **sysVar.sqlerrd** element in these ways:

- As the source in an assignment or **move** statement
- As the value in the **for count** clause of a **move** statement
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of each element in the **sysVar.sqlerrd** array are as follows:

Primitive type

BIN

Data length

9

Is value always restored after a converse?

Only in a non-segmented text program; for details see *Segmentation*

Example:

```
myItem = sysVar.sqlerrd[3];
```

Related concepts

“Segmentation in text applications” on page 151

“SQL support” on page 171

Related reference

“File and database (system words)” on page 537

sysVar.sqlerrmc

The system variable **sysVar.sqlerrmc** contains the error message associated with the return code in **sysVar.sqlcode**. **sysVar.sqlerrmc** is obtained from the SQL communications area (SQLCA) and can vary with the relational database manager.

sysVar.sqlerrmc has no meaning for the JDBC environment.

You can use **sysVar.sqlerrmc** in these ways:

- As the source in an assignment or **move** statement
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of **sysVar.sqlerrmc** are as follows

Primitive type

CHAR

Data length

70

Is value always restored after a converse?

Only in a non-segmented text program; for details see *Segmentation*

Definition considerations: **sysVar.sqlerrmc** is defined as a fixed-length text string.

Example:

```
myItem = sysVar.sqlerrmc;
```

Related concepts

"Segmentation in text applications" on page 151

"SQL support" on page 171

Related reference

"File and database (system words)" on page 537

"sysVar.sqlca" on page 551

sysVar.sqlIsolationLevel

The system variable **sysVar.sqlIsolationLevel** indicates the level of independence of one database transaction from another, and is meaningful only if you are generating Java output.

For an overview of isolation level and of the phrases *repeatable read* and *serializable transaction*, see the JDBC documentation available from Sun Microsystems, Inc.

sysVar.sqlIsolationLevel is for use only in programs migrated from VisualAge Generator and EGL 5.0. The function is supported (at development time) if the EGL preference **VisualAge Generator Compatibility** is selected or (at generation time) if the build descriptor option **VAGCompatibility** is set to *yes*.

For new development, set the SQL isolation level in the **sysLib.connect**.

The following values of **sysVar.sqlIsolationLevel** are in order of increasing strictness:

0 (the default)

Repeatable read

1 Serializable transaction

You can use this variable in any of these ways:

- As the source or destination in an assignment or move statement
- As a comparison value in a logical expression
- As the value in a return statement

Characteristics of **sysVar.transactionID** are as follows:

Primitive type

NUM

Data length

1

Is value always restored after a converse?

Yes

Related reference

"sysLib.connect" on page 543

sysVar.sqlState

The system variable **sysVar.sqlState** contains the SQL state value for the most recently completed SQL I/O operation. The code is obtained from the SQL communications area (SQLCA) and can vary with the relational database manager.

You can use **sysVar.sqlState** in these ways:

- As the source in an assignment or **move** statement
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of **sysVar.sqlState** are as follows:

Primitive type

CHAR

Data length

5

Is value always restored after a converse?

Only in a non-segmented text program; for details see *Segmentation*

Example:

```
rcitem = sysVar.sqlState;
```

Related concepts

"Segmentation in text applications" on page 151

"SQL support" on page 171

Related reference

"Exception handling and status (system words)" on page 526

"File and database (system words)" on page 537

sysVar.sqlwarn

The system array **sysVar.sqlwarn** is a static 11-element array, where each element contains a warning byte returned in the SQL communications area (SQLCA) for the last SQL I/O operation and where the index is one greater than the warning number in the SQL SQLCA description. The system variable **sysVar.sqlwarn[2]**, for example, refers to SQLWARN1, which indicates whether characters in an item were truncated in the I/O operation.

Of the elements in **sysVar.sqlwarn**, only the system variable **sysVar.sqlwarn[2]** is refreshed by the database management system for Java code or at debugging time.

You can use **sysVar.sqlwarn** in these ways:

- As the source in an assignment or **move** statement
- As the value in the **for count** clause of a **move** statement
- As a variable in a logical expression

- As the argument in an **exit** or **return** statement

The characteristics of each element in the **sysVar.sqlwarn** array are as follows:

Primitive type

CHAR

Data length

1

Is value always restored after a converse?

Only in a non-segmented text program; for details see *Segmentation*

Definition considerations: **sysVar.sqlwarn[2]** contains W if the last SQL I/O operation caused the database manager to truncate character data items because of insufficient space in the program's host variables. You can use logical expressions to test whether the values in specific host variables were truncated. For details, see the references to **trunc** in *Logical expressions*.

When the host variable is a number, no truncation warning is given. Fractional parts of a number are truncated with no indication. When DB2 UDB is used, if the non-fractional part of a number does not fit into a user variable, the database manager returns -304 in **sysVar.sqlcode**.

Also when DB2 is used, **sysVar.sqlwarn[7]** contains W if an adjustment was made to correct a result that was not valid from an arithmetic operation on date or time values.

Example: In the following example, *my-char-field* is a field in the SQL row record just processed and *lost-data* is a function that sets an error message indicating that information for *my-char-field* was truncated.

```
if (sysVar.sqlwarn[2] = 'W')
  if (my-char-field is trunc)
    lost-data();
  end
end
end
```

Related concepts

"Segmentation in text applications" on page 151

"SQL support" on page 171

Related reference

"File and database (system words)" on page 537

"Logical expressions" on page 358

Java access functions

The *Java access functions* are EGL system functions that allow your generated Java code to access native Java objects and classes; specifically, to access the public methods, constructors, and fields of the native code.

This EGL feature is made possible at run time by the presence of the *EGL Java namespace*, which is a set of names and the objects to which those names refer. A single namespace is available to your generated program and to all generated Java code that your program calls locally, whether the calls are direct or by way of another local generated Java program, to any level of call. The namespace is not available in any native Java code.

To store and retrieve objects in the namespace, you invoke the Java access functions. Your invocations include use of identifiers, each of which is a string that is used to store an object or to match a name that already exists in the namespace. When an identifier matches a name, your code can access the object associated with the name.

Note: EGL code that includes a Java access function cannot be generated as a COBOL program.

Related reference

“Java access (system words)”
 “sysLib.java” on page 564
 “sysLib.javaGetField” on page 566
 “sysLib.javaIsNull” on page 568
 “sysLib.javaIsObjID” on page 569
 “sysLib.javaRemove” on page 570
 “sysLib.javaRemoveAll” on page 571
 “sysLib.javaSetField” on page 572
 “sysLib.javaStore” on page 573
 “sysLib.javaStoreCopy” on page 575
 “sysLib.javaStoreField” on page 576
 “sysLib.javaStoreNew” on page 578
 “sysLib.javaType” on page 579

Java access (system words)

This page includes the following sections:

- “List of functions”
- “Mappings of EGL and Java types” on page 558
- “Examples” on page 559
- “Error handling” on page 562

Before reviewing this page, see *Java access functions*.

List of functions: The Java access functions are listed in the next table.

Function	Description
sysLib.java	Invokes a method on a Java object or class and may return a value
sysLib.javaGetField	Returns the value of a specified field of a specified object or class
sysLib.javaIsNull	Returns a value (1 for true, 0 for false) to indicate whether a specified identifier refers to a null object
sysLib.javaIsObjID	Returns a value (1 for true, 0 for false) to indicate whether a specified identifier is in the namespace
sysLib.javaRemove	Removes the specified identifier from the namespace and, if no other identifiers refer to the object, removes the object
sysLib.javaRemoveAll	Removes all identifiers and objects from the namespace
sysLib.javaSetField	Sets the value of a field in a Java object or class

Function	Description
sysLib.javaStore	Invokes a method and places the returned object (or null) into the namespace, along with a specified identifier
sysLib.javaStoreCopy	Creates a new identifier based on another in the namespace, so that both refer to the same object
sysLib.javaStoreField	Places the value of a class field or object field into the namespace
sysLib.javaStoreNew	Invokes the constructor of a class and places the new object into the namespace
sysLib.javaType	Returns the fully qualified name of the class of an object in the namespace

Mappings of EGL and Java types: Each of the arguments you pass to a method (and each value that you assign to a field) is mapped to a Java object or primitive type. Items of EGL primitive type CHAR, for example, are passed as objects of the Java String class. A cast operator is provided for situations in which the mapping of EGL types to Java types is not sufficient.

When you specify a Java name, EGL strips single- and double-byte blanks from the beginning and end of the value, which is case sensitive. The truncation precedes any cast. This rule applies to string literals and to items of type CHAR, DBCHAR, MBCHAR, or UNICODE. No such truncation occurs when you specify either a method argument or field value (for example, the string " my data " is passed to a method as is), unless you cast the value to objID or null.

The next table describes all the valid mappings.

Category of Argument		Examples	Java Type
A string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE	No cast	"myString"	String
	Cast with objId, which indicates an identifier	(objId)"myId" x = "myId"; (objId)x	The class of the object to which the identifier refers
	Cast with null, as may be appropriate to provide a null reference to a fully qualified class	(null)"java.lang.thread" x = "java.util.HashMap"; (null)x	The specified class Note: You can't pass in a null-casted array such as (null)"int[]"
	Cast with char, which means that the first character of the value is passed (each example in the next column passes an "a")	(char)"abc" x = "abc"; (char)x	char
An item of type HEX		myHexValue	byte array

Category of Argument		Examples	Java Type
Numeric item or literal not containing decimals, with leading zeros included in the number of digits for a literal	No cast, 1-4 digits	0100	short
	No cast, 5-9 digits	00100	int
	No cast, > 9 digits	1234567890	long
Numeric item or literal containing decimals, with leading and trailing zeros included in the number of digits for a literal	No cast, 1-6 digits	3.14159	float
	No cast, >6 digits	3.14159265	double
Numeric item or literal, with or without decimals	Cast with byte, short, int, long, float, or double	X = 42; (byte)X (long)X	The specified primitive type; but if the value is out of range for that type, loss of precision occurs and the sign may change
	Cast with boolean, which means that non-zero is true, zero is false	X = 1; (boolean)X	boolean

Note: When EGL passes a float or double to Java, rounding may occur because of the way that Java approximates a real number.

For details on the internal format of items in EGL, see the help pages on *Primitive types*.

Examples: This section gives examples on how to use Java access functions.

Printing a date string: The following example prints a date string:

```
// call the constructor of the Java Date class and
// assign the new object to the identifier "date".
sysLib.javaStoreNew( (objId)"date", "java.util.Date" );

// call the toString method of the new Date object
// and assign the output (today's date) to the charItem.
// In the absence of the cast (objId), "date"
// refers to a class rather than an object.
charItem = sysLib.java( (objId)"date", "toString" );

// assign the standard output stream of the
// Java System class to the identifier "systemOut".
sysLib.javaStoreField( (objId)"systemOut",
    "java.lang.System", "out" );

// call the println method of the output
// stream and print today's date.
sysLib.java( (objId)"systemOut", "println", charItem );

// The use of "java.lang.System.out" as the first
// argument in the previous line would not have been
// valid, as the argument must either be a
// an identifier already in the namespace or a class
// name. The argument cannot refer to a static field.
```

Testing a system property: The following example retrieves a system property and tests for the absence of a value:

```
// assign the name of an identifier to an item of type CHAR
valueID = "osNameProperty"

// place the value of property os.name into the
// namespace, and relate that value (a Java String)
// to the identifier osNameProperty
sysLib.javaStore((objId)valueId, "java.lang.System",
    "getProperty", "os.name");

// test whether the property value is non-existent
// and process accordingly
myNullFlag = sysLib.javaIsNull( (objId)valueId );

if( myNullFlag = 1 )
    error = 27;
end
```

Working with arrays: When you work with Java arrays in EGL, use the Java class `java.lang.reflect.Array`, as shown in later examples and as described in the Java API documentation. You cannot use **sysLib.javaStoreNew** to create a Java array because Java arrays have no constructors.

You use the static method `newInstance` of `java.lang.reflect.Array` to create the array in the namespace. After you create the array, you use other methods in that class to access the elements.

The method `newInstance` expects two arguments:

- A Class object that determines the type of array being created
- A number that specifies how many elements are in the array

The code that identifies the Class object varies according to whether you are creating an array of objects or an array of primitives. The subsequent code that interacts with the array also varies on the same basis.

Working with an array of objects: The following example shows how to create a 5-element object array that is accessible by use of the identifier "myArray":

```
// Get a reference to the class, for use with newInstance
sysLib.javaStore( (objId)"objectClass", "java.lang.Class",
    "forName", "java.lang.Object" );

// Create the array in the namespace
sysLib.javaStore( (objId)"myArray", "java.lang.reflect.Array",
    "newInstance", (objId)"objectClass", 5 );
```

If you want to create an array that holds a different type of object, change the class name that is passed to the first invocation of **sysLib.javaStore**. To create an array of String objects, for example, pass "java.lang.String" instead of "java.lang.Object".

To access an element of an object array, use the `get` and `set` methods of `java.lang.reflect.Array`. In the following example, `i` and `length` are numeric items:

```
length = sysLib.java( "java.lang.reflect.Array",
    "getLength", (objId)"myArray" );
i = 0;

while ( i < length )
    sysLib.javaStore( (objId)"element", "java.lang.reflect.Array",
        "get", (objId)"myArray", i );
```

```

// Here, process the element as appropriate
sysLib.java( "java.lang.reflect.Array", "set",
  (objId)"myArray", i, (objId)"element" );
i = i + 1;
end

```

The previous example is equivalent to the following Java code:

```

int length = myArray.length;

for ( int i = 0; i < length; i++ )
{
  Object element = myArray[i];

  // Here, process the element as appropriate

  myArray[i] = element;
}

```

Working with an array of Java primitives: To create an array that stores a Java primitive rather than an object, use a different mechanism in the steps that precede the use of `java.lang.reflect.Array`. In particular, obtain the `Class` argument to `newInstance` by accessing the static field `TYPE` of a primitive type class.

The following example creates `myArray2`, which is a 30-element array of integers:

```

// Get a reference to the class, for use with newInstance
sysLib.javaStoreField( (objId)"intClass",
  "java.lang.Integer", "TYPE");

// Create the array in the namespace
sysLib.javaStore( (objId)"myArray2", "java.lang.reflect.Array",
  "newInstance", (objId)"intClass", 30 );

```

If you want to create an array that holds a different type of primitive, change the `Class` name that is passed to the invocation of `sysLib.javaStoreField`. To create an array of characters, for example, pass `"java.lang.Character"` instead of `"java.lang.Integer"`.

To access an element of an array of primitives, use the `java.lang.reflect.Array` methods that are specific to a primitive type. Such methods include `getInt`, `setInt`, `getFloat`, `setFloat`, and so forth. In the following example, `length`, `element`, and `i` are numeric items:

```

length = sysLib.java( "java.lang.reflect.Array",
  "getLength", (objId)"myArray2" );
i = 0;

while ( i < length )
  element = sysLib.java( "java.lang.reflect.Array",
    "getDouble", (objId)"myArray2", i );

  // Here, process an element as appropriate

  sysLib.java( "java.lang.reflect.Array", "setDouble",
    (objId)"myArray2", i, element );
  i = i + 1;
end

```

The previous example is equivalent to the following Java code:

```

int length = myArray2.length;

for ( int i = 0; i < length; i++ )
{

```

```

double element = myArray2[i];

// Here, process an element as appropriate

myArray2[i] = element;
}

```

Working with collections: To iterate over a collection that is referenced by a variable called *list*, a Java program does as follows:

```

Iterator contents = list.iterator();

while( contents.hasNext() )
{
    Object myObject = contents.next();
    // Process myObject
}

```

Assume that `hasNext` is a numeric data and that your program related a collection to an identifier called *list*. The following EGL code is then equivalent to the Java code described earlier:

```

sysLib.javaStore( (objId)"contents", (objId)"list", "iterator" );
hasNext = sysLib.java( (objId)"contents", "hasNext" );

while ( hasNext = 1 )
    sysLib.javaStore( (objId)"myObject", (objId)"contents", "next");

    // Process myObject
    hasNext = sysLib.java( (objId)"contents", "hasNext" );
end

```

Converting an array to a collection: To create a collection from an array of objects, use the `asList` method of `java.util.Arrays`, as shown in the following example:

```

// Create a collection from array myArray
// and relate that collection to the identifier "list"
sysLib.javaStore( (objId)"list", "java.util.Arrays",
    "asList", (objId)"myArray" );

```

Next, iterate over *list*, as shown in the preceding section.

The transfer of an array to a collection works only with an array of objects, not with an array of Java primitives. Be careful not to confuse `java.util.Arrays` with `java.lang.reflect.Array`.

Error handling: Many of the Java access functions are associated with error codes, as described in the function-specific help pages. If the value of the system variable **sysVar.handleSysLibErrors** is 1 when one of the listed errors occurs, EGL sets the system variable **sysVar.errorCode** to a non-zero value. If the value of **sysVar.handleSysLibErrors** is 0 when one of the errors occurs, the program ends.

Of particular interest is the **sysVar.errorCode** value "00001000", which indicates that an exception was thrown by an invoked method or as a result of a class initialization.

When an exception is thrown, EGL stores it in the namespace. If another exception occurs, the second exception takes the place of the first. You can use the identifier *caughtException* to access the last exception that occurred.

In an unusual situation, an invoked method throws not an exception but an error such as `OutOfMemoryError` or `StackOverflowError`. In such a case, the program ends regardless of the value of system variable `sysVar.handleSysLibErrors`.

The following Java code shows how a Java program can have multiple catch blocks to handle different kinds of exceptions. This code tries to create a `FileOutputStream` object. A failure causes the code to set an `errorType` variable and to store the exception that was thrown.

```
int errorType = 0;
Exception ex = null;

try
{
    java.io.FileOutputStream fOut =
        new java.io.FileOutputStream( "out.txt" );
}
catch ( java.io.IOException iox )
{
    errorType = 1;
    ex = iox;
}
catch ( java.lang.SecurityException sx )
{
    errorType = 2;
    ex = sx;
}
```

The following EGL code is equivalent to the previous Java code:

```
handleSysLibErrors = 1;
errorType = 0;

sysLib.javaStoreNew( (objId)"fOut",
    "java.io.FileOutputStream", "out.txt" );

if ( sysVar.errorCode = "00001000" )
    exType = sysLib.javaType( (objId)"caughtException" );

if ( exType = "java.io.IOException" )
    errorType = 1;
    sysLib.javaStoreCopy( (objId)"caughtException", (objId)"ex" );
else
    if ( exType = "java.lang.SecurityException" )
        errorType = 2;
        sysLib.javaStoreCopy( (objId)"caughtException", (objId)"ex" );
    end
end
end
```

Related concepts

“Java access functions” on page 556

Related reference

“Exception handling” on page 75

“Primitive types” on page 27

“sysLib.java” on page 564

“sysLib.javaGetField” on page 566

“sysLib.javaIsNull” on page 568

“sysLib.javaIsObjID” on page 569

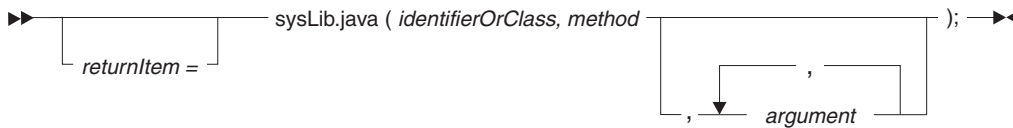
“sysLib.javaRemove” on page 570

“sysLib.javaRemoveAll” on page 571

“sysLib.javaSetField” on page 572

[“sysLib.javaType” on page 579](#)

The system function **sysLib.java** invokes a method on a native Java object or class and may return a value. **sysLib.java** is one of several Java access functions.



If the native Java method returns a value, the return item is optional.

- If the received value is a byte, short, int, long, float, or double, the return item must be of type `BIN` or an equivalent integer type (see *BIN and the integer types*). The characteristics do not need to match the value; for example, a float may be stored in a return item that is declared with no decimal digits. For details on handling overflow, see `sysVar.handleOverflow` and `sysVar.overflowIndicator`.

- If the received value is a boolean, the return item must be of a numeric primitive type. The value is 1 for true, 0 for false.
 - If the received value is a byte array, the return item must be of type HEX. For details on mismatched lengths, see *Assignments*.
 - If the received value is a String or char, the return item must be of type CHAR, DBCHAR, MBCHAR, or UNICODE--
 - If the item is of type MBCHAR or UNICODE the received value is always appropriate
 - If the item is of type CHAR, problems can arise if the received value includes Unicode characters that correspond to DBCHAR characters
 - If the item is of type DBCHAR, problems can arise if the received value includes Unicode characters that correspond to single-byte characters
- For details on mismatched lengths, see *Assignments*.
- If the native Java method does not return a value or returns a null, the following cases apply:
 - No error occurs in the absence of a return item
 - An error occurs at run time if a return item is present; the error is 00001004, as listed later

- An identifier that refers to an object in the namespace; or
- The fully qualified name of a Java class.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. If you are specifying an identifier of an object, the

identifier must be cast to objID, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

Your code cannot invoke a method on an object until you have created an identifier for the object. A later example illustrates this point with `java.lang.System.out`, which refers to a `PrintStream` object.

method

The name of the method to call.

This argument is either a string literal or an item of type `CHAR`, `DBCHAR`, `MBCHAR`, or `UNICODE`. Single- and double-byte blanks are stripped from the beginning and end of the string, which is case sensitive.

argument

A value passed to the method.

A cast may be required, as specified in *Java access (system words)*.

The Java type-conversion rules are in effect. No error occurs, for example, if you pass a short to a method parameter that is declared as an int.

The memory area in the invoking program does not change regardless of what the method does.

In the following example, the cast (objID) is required except as noted:

```
// call the constructor of the Java Date class and
// assign the new object to the identifier "date".
sysLib.JavaStoreNew( (objID)"date", "java.util.Date");

// call the toString method of the new Date object
// and assign the output (today's date) to the chaItem.
// In the absence of the cast (objID), "date"
// refers to a class rather than an object.
chaItem = sysLib.Java( (objID)"date", "toString" );

// assign the standard output stream of the
// Java System class to the identifier "systemOut".
sysLib.JavaStoreField( (objID)"systemOut", "java.lang.System", "out" );

// call the println method of the output
// stream and print today's date.
sysLib.Java( (objID)"systemOut", "println", chaItem );

// The use of "java.lang.System.out" as the first
// argument in the previous line would not have been
// valid, as the argument must either be a
// an identifier already in the namespace or a class
// name. The argument cannot refer to a static field.
```

An error during processing of `sysLib.java` can set `sysVar.errorCode` to a value listed in the next table.

Value in <code>sysVar.errorCode</code>	Description
00001000	An exception was thrown by an invoked method or as a result of a class initialization
00001001	The object was null, or the specified identifier was not in the namespace
00001002	A public method, field, or class with the specified name does not exist or cannot be loaded

Value in sysVar.errorCode	Description
00001003	The EGL primitive type does not match the type expected in Java
00001004	The method returned null, the method does not return a value, or the value of a field was null
00001005	The returned value does not match the type of the return item
00001006	The class of an argument cast to null could not be loaded
00001007	A SecurityException or IllegalAccessException was thrown during an attempt to get information about a method or field; or an attempt was made to set the value of a field that was declared final
00001009	An identifier rather than a class name must be specified; the method or field is not static

Related concepts

“Java access functions” on page 556

Related tasks

“Syntax diagram” on page 506

Related reference

“Assignments” on page 296

“BIN and the integer types” on page 32

“Exception handling” on page 75

“Java access (system words)” on page 557

“Primitive types” on page 27

“sysLib.javaGetField”

“sysLib.javaIsNull” on page 568

“sysLib.javaIsObjID” on page 569

“sysLib.javaRemove” on page 570

“sysLib.javaRemoveAll” on page 571

“sysLib.javaSetField” on page 572

“sysLib.javaStore” on page 573

“sysLib.javaStoreCopy” on page 575

“sysLib.javaStoreField” on page 576

“sysLib.javaStoreNew” on page 578

“sysLib.javaType” on page 579

“sysVar.handleOverflow” on page 532

“sysVar.overflowIndicator” on page 534

sysLib.javaGetField

The system function **sysLib.javaGetField** returns the value of a specified field of a specified object or class. **sysLib.javaGetField** is one of several Java access functions.

►► returnItem = sysLib.javaGetField (*identifierOrClass*, *field*) ; ◀◀

returnItem

The return item is required and receives the value of the field specified in the second argument. The following cases apply:

- If the received value is a byte, short, int, long, float, or double, the return item must be of type BIN or an equivalent integer type (see *BIN and the integer types*). The characteristics do not need to match the value; for example, a float may be stored in a return item that is declared with no decimal digits. For details on handling overflow, see *sysVar.handleOverflow* and *sysVar.overflowIndicator*.
- If the received value is a boolean, the return item must be of a numeric primitive type. The value is 1 for true, 0 for false.
- If the received value is a byte array, the return item must be of type HEX. For details on mismatched lengths, see *Assignments*.
- If the received value is a String or char, the return item must be of type CHAR, DBCHAR, MBCHAR, or UNICODE--
 - If the item is of type MBCHAR or UNICODE, the received value is always appropriate
 - If the item is of type CHAR, problems can arise if the received value includes Unicode characters that correspond to DBCHAR characters
 - If the item is of type DBCHAR, problems can arise if the received value includes Unicode characters that correspond to single-byte charactersFor details on mismatched lengths, see *Assignments*.
- If the native Java method does not return a value or returns a null, error 00001004 occurs, as listed later.

identifierOrClass

This argument is one of the following entities:

- An identifier that refers to an object in the namespace; or
- The fully qualified name of a Java class.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. If you are specifying an identifier of an object, the identifier must be cast to objID, as in a later example. If you intend to specify a static field in the next argument, it is recommended that you specify a class in this argument.

EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

field

The name of the field to read.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. Single- and double-byte blanks are stripped from the beginning and end of the string, which is case sensitive.

An example is as follows:

```
myItem = sysLib.javaGetField( (objId)"myID", "myField" );
```

An error during processing of **sysLib.javaGetField** can set **sysVar.errorCode** to a value listed in the next table.

Value in <code>sysVar.errorCode</code>	Description
00001000	An exception was thrown by an invoked method or as a result of a class initialization
00001001	The object was null, or the specified identifier was not in the namespace
00001002	A public method, field, or class with the specified name does not exist or cannot be loaded
00001004	The method returned null, the method does not return a value, or the value of a field was null
00001005	The returned value does not match the type of the return item
00001007	A <code>SecurityException</code> or <code>IllegalAccessException</code> was thrown during an attempt to get information about a method or field; or an attempt was made to set the value of a field that was declared final
00001009	An identifier rather than a class name must be specified; the method or field is not static

Related concepts

“Java access functions” on page 556

Related tasks

“Syntax diagram” on page 506

Related reference

“Assignments” on page 296

“BIN and the integer types” on page 32

“Exception handling” on page 75

“Java access (system words)” on page 557

“`sysLib.java`” on page 564

“`sysLib.javaIsNull`”

“`sysLib.javaIsObjID`” on page 569

“`sysLib.javaRemove`” on page 570

“`sysLib.javaRemoveAll`” on page 571

“`sysLib.javaSetField`” on page 572

“`sysLib.javaStore`” on page 573

“`sysLib.javaStoreCopy`” on page 575

“`sysLib.javaStoreField`” on page 576

“`sysLib.javaStoreNew`” on page 578

“`sysLib.javaType`” on page 579

`sysLib.javaIsNull`

The system function `sysLib.javaIsNull` returns a value (1 for true, 0 for false) to indicate whether a specified identifier refers to a null object. `sysLib.javaIsNull` is one of several Java access functions.

►► `returnItem = sysLib.javaIsNull (identifier) ;` ◄◄

returnItem

A numeric item that receives one of two values: 1 for true, 0 for false. Use of a non-numeric item causes an error at validation time.

identifier

An identifier that refers to an object in the namespace.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. The identifier must be cast to objID. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

An example is as follows:

```
// test whether an object is null
// and process accordingly
isNull = sysLib.javaIsNull( (objId)valueId );

if( isNull = 1 )
    error = 12;
end
```

An error during processing of **sysLib.javaIsNull** can set **sysVar.errorCode** to a value listed in the next table.

Value in sysVar.errorCode	Description
00001001	The specified identifier was not in the namespace

Related concepts

"Java access functions" on page 556

Related tasks

"Syntax diagram" on page 506

Related reference

"Java access (system words)" on page 557

"sysLib.java" on page 564

"sysLib.javaGetField" on page 566

"sysLib.javaIsObjID"

"sysLib.javaRemove" on page 570

"sysLib.javaRemoveAll" on page 571

"sysLib.javaSetField" on page 572

"sysLib.javaStore" on page 573

"sysLib.javaStoreCopy" on page 575

"sysLib.javaStoreField" on page 576

"sysLib.javaStoreNew" on page 578

"sysLib.javaType" on page 579

sysLib.javaIsObjID

The system function **sysLib.javaIsObjID** returns a value (1 for true, 0 for false) to indicate whether a specified identifier is in the namespace. **sysLib.javaIsObjID** is one of several Java access functions.

►► *returnItem* = sysLib.javaIsObjID (*identifier*) ; ————— ◀◀

returnItem

A numeric item that receives one of two values: 1 for true, 0 for false. Use of a non-numeric item causes an error at validation time.

identifier

An identifier that refers to an object in the namespace.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. The identifier must be cast to objID. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

An example is as follows:

```
// test whether an object is non-existent
// and process accordingly
isPresent = sysLib.javaIsObjID( (objId)valueId );

if( isPresent = 0 )
    error = 27;
end
```

No run-time errors are associated with **sysLib.javaIsObjID**.

Related concepts

"Java access functions" on page 556

Related tasks

"Syntax diagram" on page 506

Related reference

"Java access (system words)" on page 557

"sysLib.java" on page 564

"sysLib.javaGetField" on page 566

"sysLib.javaIsNull" on page 568

"sysLib.javaRemove"

"sysLib.javaRemoveAll" on page 571

"sysLib.javaSetField" on page 572

"sysLib.javaStore" on page 573

"sysLib.javaStoreCopy" on page 575

"sysLib.javaStoreField" on page 576

"sysLib.javaStoreNew" on page 578

"sysLib.javaType" on page 579

sysLib.javaRemove

The system function **sysLib.javaRemove** removes the specified identifier from the EGL Java namespace. The object related to the identifier is also removed, but only if the identifier is the only one that refers to the object. If another identifier refers to the object, the object remains in the namespace and is accessible by way of that other identifier.

sysLib.javaRemove is one of several Java access functions.

►► sysLib.javaRemove (*identifier*); ◄◄

identifier

The identifier that refers to an object. No error occurs if the identifier is not found.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. The identifier must be cast to objID, as shown in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

An example is as follows:

```
sysLib.javaRemove( (objId)myStoredObject );
```

No run-time errors are associated with **sysLib.javaRemove**.

Note: By invoking the system functions **sysLib.javaRemove** and **sysLib.javaRemoveAll**, your code allows the Java Virtual Machine to handle garbage collection in the EGL Java namespace. If you do not invoke a system function to remove an object from the namespace, the memory is not recovered during the run time of any program that has access to the namespace.

Related concepts

"Java access functions" on page 556

Related tasks

"Syntax diagram" on page 506

Related reference

"Java access (system words)" on page 557

"sysLib.java" on page 564

"sysLib.javaGetField" on page 566

"sysLib.javaIsNull" on page 568

"sysLib.javaIsObjID" on page 569

"sysLib.javaRemoveAll"

"sysLib.javaSetField" on page 572

"sysLib.javaStore" on page 573

"sysLib.javaStoreCopy" on page 575

"sysLib.javaStoreField" on page 576

"sysLib.javaStoreNew" on page 578

"sysLib.javaType" on page 579

sysLib.javaRemoveAll

The system function **sysLib.javaRemoveAll** removes all identifiers and objects from the EGL Java namespace. **sysLib.javaRemoveAll** is one of several Java access functions.

►► — sysLib.javaRemoveAll (); — ◀◀

No run-time errors are associated with **sysLib.javaRemoveAll**.

Note: By invoking the system functions **sysLib.javaRemove** and **sysLib.javaRemoveAll**, your code allows the Java Virtual Machine to handle garbage collection in the EGL Java namespace. If you do not invoke a system function to remove an object from the namespace, the memory is not recovered during the run time of any program that has access to the namespace.

Related concepts

"Java access functions" on page 556

Related tasks

"Syntax diagram" on page 506

Related reference

"Java access (system words)" on page 557

"sysLib.java" on page 564

"sysLib.javaGetField" on page 566

"sysLib.javaIsNull" on page 568

"sysLib.javaIsObjID" on page 569

"sysLib.javaRemove" on page 570

"sysLib.javaSetField"

"sysLib.javaStore" on page 573

"sysLib.javaStoreCopy" on page 575

"sysLib.javaStoreField" on page 576

"sysLib.javaStoreNew" on page 578

"sysLib.javaType" on page 579

sysLib.javaSetField

The system function **sysLib.javaSetField** sets the value of a field in a native Java object or class. **sysLib.javaSetField** is one of several Java access functions.

►► sysLib.javaSetField (*identifierOrClass*, *field*, *value*) ; ◀◀

identifierOrClass

This argument is one of the following entities:

- An identifier that refers to an object in the namespace; or
- The fully qualified name of a Java class.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. If you are specifying an identifier of an object, the identifier must be cast to objID, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

field

The name of the field to change.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. Single- and double-byte blanks are stripped from the beginning and end of the string, which is case sensitive.

value

The value itself.

A cast may be required, as specified in Java access (system words).

The Java type-conversion rules are in effect. No error occurs, for example, if you assign a short to a field that is declared as an int.

An example is as follows:

```
sysLib.javaSetField( (objID)"myId", "myField",  
  (short)myNumItem );
```

An error during processing of **sysLib.javaSetField** can set **sysVar.errorCode** to a value listed in the next table.

Value in <code>sysVar.errorCode</code>	Description
00001000	An exception was thrown by an invoked method or as a result of a class initialization
00001001	The object was null, or the specified identifier was not in the namespace
00001002	A public method, field, or class with the specified name does not exist or cannot be loaded
00001003	The EGL primitive type does not match the type expected in Java
00001007	A <code>SecurityException</code> or <code>IllegalAccessException</code> was thrown during an attempt to get information about a method or field; or an attempt was made to set the value of a field that was declared final
00001009	An identifier rather than a class name must be specified; the method or field is not static

Related concepts

“Java access functions” on page 556

“Syntax diagram” on page 506

Related reference

“Java access (system words)” on page 557

“`sysLib.java`” on page 564

“`sysLib.javaGetField`” on page 566

“`sysLib.javaIsNull`” on page 568

“`sysLib.javaIsObjID`” on page 569

“`sysLib.javaRemove`” on page 570

“`sysLib.javaRemoveAll`” on page 571

“`sysLib.javaStore`”

“`sysLib.javaStoreCopy`” on page 575

“`sysLib.javaStoreField`” on page 576

“`sysLib.javaStoreNew`” on page 578

“`sysLib.javaType`” on page 579

sysLib.javaStore

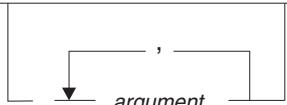
The system function **sysLib.javaStore** invokes a method and places the returned object (or null) into the EGL Java namespace, along with a specified identifier. If the identifier is already in the namespace, the action is equivalent to the following steps:

- Running **sysLib.javaRemove** on the identifier to remove the object that was related to that identifier
- Relating the **sysLib.javaStore**-returned object with the target identifier

If the method returns a Java primitive instead of an object, EGL stores an object that represents the primitive; for example, if the method returns an int, EGL stores an object of type `java.lang.Integer`.

sysLib.javaStore is one of several Java access functions.

►► sysLib.javaStore (*storeId*, *identifierOrClass*, *method*); ►►



storeId

The identifier to store with the returned object.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. The identifier must be cast to objID, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

identifierOrClass

This argument is one of the following entities:

- An identifier that refers to an object in the namespace; or
- The fully qualified name of a Java class.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. If you are specifying an identifier of an object, the identifier must be cast to objID, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

method

The method to invoke.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. If you are specifying an identifier of an object, the identifier must be cast to objID, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

argument

A value passed to the method.

A cast may be required, as specified in Java access (system words).

The Java type-conversion rules are in effect. No error occurs, for example, if you pass a short to a method parameter that is declared as an int.

The memory area in the invoking program does not change regardless of what the method does.

An example is as follows:

```
sysLib.javaStore( (objId)"storeId", (objId)"myId",
  "myMethod", 36 );
```

An error during processing of **sysLib.javaStore** can set **sysVar.errorCode** to a value listed in the next table.

Value in sysVar.errorCode	Description
00001000	An exception was thrown by an invoked method or as a result of a class initialization
00001001	The object was null, or the specified identifier was not in the namespace

Value in sysVar.errorCode	Description
00001002	A public method, field, or class with the specified name does not exist or cannot be loaded
00001003	The EGL primitive type does not match the type expected in Java
00001006	The class of an argument cast to null could not be loaded
00001007	A SecurityException or IllegalAccessException was thrown during an attempt to get information about a method or field; or an attempt was made to set the value of a field that was declared final
00001009	An identifier rather than a class name must be specified; the method or field is not static

Related concepts

"Java access functions" on page 556

Related tasks

"Java access (system words)" on page 557

Related reference

"Java access (system words)" on page 557

"sysLib.java" on page 564

"sysLib.javaGetField" on page 566

"sysLib.javaIsNull" on page 568

"sysLib.javaIsObjID" on page 569

"sysLib.javaRemove" on page 570

"sysLib.javaRemoveAll" on page 571

"sysLib.javaSetField" on page 572

"sysLib.javaStoreCopy"

"sysLib.javaStoreField" on page 576

"sysLib.javaStoreNew" on page 578

"sysLib.javaType" on page 579

sysLib.javaStoreCopy

The system function **sysLib.javaStoreCopy** creates a new identifier based on another in the namespace, so that both refer to the same object. If the source identifier is not in the namespace, a null is stored for the target identifier and no error occurs. If the target identifier is already in the namespace, the action is equivalent to the following steps:

- Running **sysLib.javaRemove** on the target identifier to remove the object that was related to that identifier
- Relating the source object with the target identifier

sysLib.javaStoreCopy is one of several Java access functions.

►► sysLib.javaStoreCopy (*sourceID*, *targetID*) ; ◄◄

sourceID

An identifier that refers to an object in the namespace or to null.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. The identifier must be cast to objId, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

targetId

The new identifier, which refers to the same object.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. The identifier must be cast to objID, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

An example is as follows:

```
sysLib.javaStoreCopy( (objId)"sourceId", (objId)"targetId" );
```

No run-time errors are associated with **sysLib.javaStoreCopy**.

Related concepts

"Java access functions" on page 556

"Syntax diagram" on page 506

Related reference

"Java access (system words)" on page 557

"sysLib.java" on page 564

"sysLib.javaGetField" on page 566

"sysLib.javaIsNull" on page 568

"sysLib.javaIsObjID" on page 569

"sysLib.javaRemove" on page 570

"sysLib.javaRemoveAll" on page 571

"sysLib.javaSetField" on page 572

"sysLib.javaStore" on page 573

"sysLib.javaStoreField"

"sysLib.javaStoreNew" on page 578

"sysLib.javaType" on page 579

sysLib.javaStoreField

The system function **sysLib.javaStoreField** places the value of a class field or object field into the EGL Java namespace. If the identifier used to store the object is already in the namespace, the action is equivalent to the following steps:

- Running **sysLib.javaRemove** on the identifier to remove the object that was related to the identifier
- Relating the new object with the identifier

If the class or object field contains a Java primitive instead of an object, EGL stores an object that represents the primitive; for example, if the field contains an int, EGL stores an object of type java.lang.Integer.

►► sysLib.javaStoreField (*storeId*, *identifierOrClass*, *field*); ◄◄

storeId

The identifier to store with the object.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. The identifier must be cast to objID, as in a later

example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

identifierOrClass

This argument is one of the following entities:

- An identifier that refers to an object in the namespace; or
- The fully qualified name of a Java class.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. If you are specifying an identifier of an object, the identifier must be cast to objID, as in a later example. If you intend to specify a static field in the next argument, it is recommended that you specify a class in this argument.

EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

field

The name of the field that refers to an object.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. Single- and double-byte blanks are stripped from the beginning and end of the string, which is case sensitive.

An example is as follows:

```
sysLib.javaStoreField( (objId)"myStoreId",  
  (objId)"myId", "myField");
```

An error during processing of **sysLib.javaStoreField** can set **sysVar.errorCode** to a value listed in the next table.

Value in sysVar.errorCode	Description
00001000	An exception was thrown by an invoked method or as a result of a class initialization
00001001	The object was null, or the specified identifier was not in the namespace
00001002	A public method, field, or class with the specified name does not exist or cannot be loaded
00001007	A SecurityException or IllegalAccessException was thrown during an attempt to get information about a method or field; or an attempt was made to set the value of a field that was declared final
00001009	An identifier rather than a class name must be specified; the method or field is not static

Related concepts

"Java access functions" on page 556

Related tasks

"Syntax diagram" on page 506

Related reference

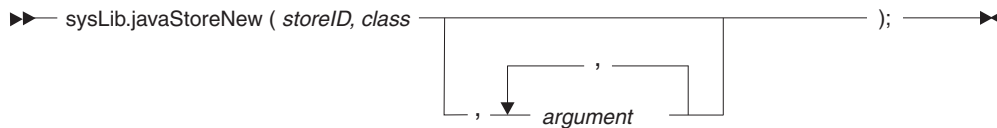
"Java access (system words)" on page 557
"sysLib.java" on page 564
"sysLib.javaGetField" on page 566
"sysLib.javaIsNull" on page 568
"sysLib.javaIsObjID" on page 569
"sysLib.javaRemove" on page 570
"sysLib.javaRemoveAll" on page 571
"sysLib.javaSetField" on page 572
"sysLib.javaStore" on page 573
"sysLib.javaStoreCopy" on page 575
"sysLib.javaStoreNew"
"sysLib.javaType" on page 579

sysLib.javaStoreNew

The system function **sysLib.javaStoreNew** invokes the constructor of a class and places the new object into the EGL Java namespace. If the identifier is already in the namespace, the action is equivalent to the following steps:

- Running **sysLib.javaRemove** on the identifier to remove the object previously associated with the identifier
- Relating the new object with the identifier

sysLib.javaStoreNew is one of several Java access functions.



storeId

The identifier to store with the new object.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. The identifier must be cast to objID, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

class

The fully qualified name of a Java class.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

argument

A value passed to the constructor.

A cast may be required, as specified in *Java access (system words)*.

The Java type-conversion rules are in effect. No error occurs, for example, if you pass a short to a constructor parameter that is declared as an int.

The memory area in the invoking program does not change regardless of what the constructor does.

An example is as follows:

```
sysLib.javaStoreNew( (objId)"storeId", "myClass", 36 );
```

An error during processing of **sysLib.javaStoreNew** can set **sysVar.errorCode** to a value listed in the next table.

Value in sysVar.errorCode	Description
00001000	An exception was thrown by an invoked method or as a result of a class initialization
00001001	The object was null, or the specified identifier was not in the namespace
00001002	A public method, field, or class with the specified name does not exist or cannot be loaded
00001003	The EGL primitive type does not match the type expected in Java
00001006	The class of an argument cast to null could not be loaded
00001007	A <code>SecurityException</code> or <code>IllegalAccessException</code> was thrown during an attempt to get information about a method or field; or an attempt was made to set the value of a field that was declared final
00001008	The constructor cannot be called; the class name refers to an interface or abstract class

Related concepts

“Java access functions” on page 556

Related tasks

“Syntax diagram” on page 506

Related reference

“Java access (system words)” on page 557

“sysLib.java” on page 564

“sysLib.javaGetField” on page 566

“sysLib.javaIsNull” on page 568

“sysLib.javaIsObjID” on page 569

“sysLib.javaRemove” on page 570

“sysLib.javaRemoveAll” on page 571

“sysLib.javaSetField” on page 572

“sysLib.javaStore” on page 573

“sysLib.javaStoreCopy” on page 575

“sysLib.javaStoreField” on page 576

“sysLib.javaType”

sysLib.javaType

The system function **sysLib.javaType** returns the fully qualified name of the class of an object in the EGL Java namespace. **sysLib.javaType** is one of several Java access functions.

►► `returnItem = sysLib.javaType (identifier);` ◄◄

returnItem

The return item is required and must be of type CHAR, DBCHAR, MBCHAR, or UNICODE--

- If the item is of type MBCHAR or UNICODE, the received value is always appropriate
- If the item is of type CHAR, problems can arise if the received value includes Unicode characters that correspond to DBCHAR characters

For details on mismatched lengths, see *Assignments*.

identifier

An identifier that refers to an object in the namespace.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. The identifier must be cast to objId, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

An example is as follows:

```
myItem = sysLib.javaType( (objId)"myId" );
```

An error during processing of **sysLib.javaType** can set **sysVar.errorCode** to a value listed in the next table.

Value in sysVar.errorCode	Description
00001001	The object was null, or the specified identifier was not in the namespace

Related concepts

"Java access functions" on page 556

Related tasks

"Syntax diagram" on page 506

Related reference

"Java access (system words)" on page 557

"sysLib.java" on page 564

"sysLib.javaGetField" on page 566

"sysLib.javaIsNull" on page 568

"sysLib.javaObjID" on page 569

"sysLib.javaRemove" on page 570

"sysLib.javaRemoveAll" on page 571

"sysLib.javaSetField" on page 572

"sysLib.javaStore" on page 573

"sysLib.javaStoreCopy" on page 575

"sysLib.javaStoreField" on page 576

"sysLib.javaStoreNew" on page 578

"sysLib.javaType" on page 579

Mathematical (system words)

Mathematical system words perform operations such as absolute value, cosine, natural log, and rounding. These words operate on items (and return values to items) of the following kinds:

- Numeric items; these are of type BIGINT, BIN, DECIMAL, INT, NUM, NUMC, PACF, SMALLINT

- Items of type HEX:
 - An item of type HEX (length 8) is assumed to be a single precision, 4-byte floating-point number that is native to the run-time environment
 - An item of type HEX (length 16) is assumed to be a double precision, 8-byte floating-point number that is native to the run-time environment

The mathematical system functions can raise exception conditions; but a **try ... onException ... end** block is necessary if you wish **sysVar.errorCode** to receive the returned code.

The mathematical functions return the following values for **sysVar.errorCode**:

- 8: The argument is not in a valid range
- 12: The intermediate or final result cannot be represented as a double-precision floating point number or with the precision of the result parameter
- 16: A function exception has occurred

The mathematical system words are listed in the next table.

System function/Invocation	Description
mathLib.abs <i>result</i> = mathLib.abs (<i>numericItem</i>)	Returns absolute value of <i>numericItem</i>
mathLib.acos <i>result</i> = mathLib.acos (<i>numericItem</i>)	Returns arccosine of <i>numericItem</i>
mathLib.asin <i>result</i> = mathLib.asin (<i>numericItem</i>)	Returns arcsine of <i>numericItem</i>
mathLib.atan <i>result</i> = mathLib.atan (<i>numericItem</i>)	Returns arctangent of <i>numericItem</i>
mathLib.atan2 <i>result</i> = mathLib.atan2 (<i>y</i> , <i>x</i>)	Computes the principal value of the arc tangent of <i>y/x</i> , using the signs of both arguments to determine the quadrant of the return value
mathLib.ceiling <i>result</i> = mathLib.ceiling (<i>numericItem</i>)	Returns smallest integer not less than <i>numericItem</i>
mathLib.compareNum <i>result</i> = mathLib.compareNum (<i>numericItem1</i> , <i>numericItem2</i>)	Returns a result (-1, 0, or 1) that indicates whether <i>numericItem1</i> is less than, equal to, or greater than <i>numericItem2</i>
mathLib.cos <i>result</i> = mathLib.cos (<i>numericItem</i>)	Returns cosine of <i>numericItem</i>
mathLib.cosh <i>result</i> = mathLib.cosh (<i>numericItem</i>)	Returns hyperbolic cosine of <i>numericItem</i>
mathLib.exp <i>result</i> = mathLib.exp (<i>numericItem</i>)	Returns exponential value of <i>numericItem</i>
mathLib.floatingAssign <i>result</i> = mathLib.floatingAssign (<i>numericItem</i>)	Returns <i>numericItem</i> as a double-precision floating-point number

System function/Invocation	Description
mathLib.floatingDifference <i>result</i> = mathLib.floatingDifference (<i>numericItem1</i> , <i>numericItem2</i>)	Returns the difference between <i>numericItem1</i> and <i>numericItem2</i>
mathLib.floatingMod <i>result</i> = mathLib.floatingMod (<i>numericItem1</i> , <i>numericItem2</i>)	Calculates the floating point remainder of <i>numericItem1</i> divided by <i>numericItem2</i> , with the result having the same sign as <i>numericItem1</i>
mathLib.floatingProduct <i>result</i> = mathLib.floatingProduct (<i>numericItem1</i> , <i>numericItem2</i>)	Returns product of <i>numericItem1</i> and <i>numericItem2</i>
mathLib.floatingQuotient <i>result</i> = mathLib.floatingQuotient (<i>numericItem1</i> , <i>numericItem2</i>)	Returns quotient of <i>numericItem1</i> divided by <i>numericItem2</i>
mathLib.floatingSum <i>result</i> = mathLib.floatingSum (<i>numericItem1</i> , <i>numericItem2</i>)	Returns sum of <i>numericItem1</i> and <i>numericItem2</i>
mathLib.floor <i>result</i> = mathLib.floor (<i>numericItem</i>)	Returns the largest integer not greater than <i>numericItem</i>
mathLib.frexp <i>result</i> = mathLib.frexp (<i>numericItem</i> , integer)	Splits a number into a normalized fraction in the range of .5 to 1 (which is the returned value) and a power of 2 (which is returned in <i>integer</i>)
mathLib.ldexp <i>result</i> = mathLib.ldexp (<i>numericItem</i> , integer)	Returns <i>numericItem</i> multiplied by 2 to the power of <i>integer</i>
mathLib.log <i>result</i> = mathLib.log (<i>numericItem</i>)	Returns the natural logarithm of <i>numericItem</i>
mathLib.log10 <i>result</i> = mathLib.log10 (<i>numericItem</i>)	Returns the base 10 logarithm of <i>numericItem</i>
mathLib.maximum <i>result</i> = mathLib.maximum (<i>numericItem1</i> , <i>numericItem2</i>)	Returns the greater of <i>numericItem1</i> and <i>numericItem2</i>
mathLib.minimum <i>result</i> = mathLib.minimum (<i>numericItem1</i> , <i>numericItem2</i>)	Returns the lesser of <i>numericItem1</i> and <i>numericItem2</i>
mathLib.modf <i>result</i> = mathLib.modf (<i>numericItem1</i> , <i>numericItem2</i>)	Splits <i>numericItem1</i> into integral and fractional parts, both with the same sign as <i>numericItem1</i> ; places the integral part in <i>numericItem2</i> ; and returns the fractional part
mathLib.pow <i>result</i> = mathLib.pow (<i>numericItem1</i> , <i>numericItem2</i>)	Returns <i>numericItem1</i> raised to the power of <i>numericItem2</i>
mathLib.precision <i>result</i> = mathLib.precision (<i>numericItem</i>)	Returns the maximum precision (in decimal digits) for <i>numericItem</i>

System function/Invocation	Description
<code>mathLib.round</code> <code>result = mathLib.round</code> (<i>numericItem</i> , <i>integer</i>) <code>result = mathLib.round</code> (<i>numericExpression</i>)	Rounds a number or expression to a nearest value (for example, to the nearest thousands) and returns the result
<code>mathLib.sin</code> <code>result = mathLib.sin</code> (<i>numericItem</i>)	Returns sine of <i>numericItem</i>
<code>mathLib.sinh</code> <code>result = mathLib.sinh</code> (<i>numericItem</i>)	Returns hyperbolic sine of <i>numericItem</i>
<code>mathLib.sqrt</code> <code>result = mathLib.sqrt</code> (<i>numericItem</i>)	Returns the square root of <i>numericItem</i> if <i>numericItem</i> is greater than or equal to zero
<code>mathLib.tan</code> <code>result = mathLib.tan</code> (<i>numericItem</i>)	Returns the tangent of <i>numericItem</i>
<code>mathLib.tanh</code> <code>result = mathLib.tanh</code> (<i>numericItem</i>)	Returns the hyperbolic tangent of <i>numericItem</i>

Related reference

“Assignments” on page 296
 “EGL statements” on page 70
 “EGL statements” on page 70
 “Exception handling” on page 75
 “Numeric expressions” on page 364
 “Primitive types” on page 27
 “System words” on page 79
 “System words in alphabetical order” on page 508

mathLib.abs

The system function **mathLib.abs** returns the absolute value of a number.

►► `result = mathLib.abs (numericItem);` ◀◀

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The absolute value of *numericItem* is converted to the format of *result* and returned in *result*.

numericItem

Any numeric item or HEX item, as described in *Mathematical (system words)*.

mathLib.abs works on every target system. In relation to Java programs, EGL uses one of the `abs()` methods in the Java `StrictMath` class so that the run-time behavior is the same for every Java Virtual Machine.

Example:

```
myItem = -5;
result = mathLib.abs(myItem); // result = 5
```

Related reference

“Mathematical (system words)” on page 580

mathLib.acos

The system function **mathLib.acos** returns the arccosine of an argument, in radians.

►► `result = mathLib.acos (numericItem);` ◄◄

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The returned value (between 0.0 and pi) is in radians and is converted to the format of *result*.

numericItem

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating point before the calculation occurs. If the value is not between -1 and 1, an error occurs.

mathLib.acos works on every target system. In relation to Java programs, EGL uses the `acos()` method in the Java `StrictMath` class so that the run-time behavior is the same for every Java Virtual Machine.

Example:

```
result = mathLib.acos(myItem);
```

Related reference

“Mathematical (system words)” on page 580

mathLib.asin

The system function **sysLib.asin** returns the arcsine of a number that is in the range of -1 to 1. The result is in radians and is in the range of -pi/2 to pi/2.

►► `result = mathLib.asin (numericItem);` ◄◄

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by the `mathLib.asin` function is converted to the format of *result* and returned in *result*.

numericItem

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating point before the `mathLib.asin` function is called.

Example:

```
result = mathLib.asin(myItem);
```

Related reference

“Mathematical (system words)” on page 580

mathLib.atan

The system function **mathLib.atan** returns the arctangent of a number. The result is in radians and is in the range of -pi/2 and pi/2.

►► `result = mathLib.atan (numericItem);` ◄◄

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by **mathLib.atan** is converted to the format of *result*.

numericItem

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating point before **mathLib.atan** is called.

Example:

```
result = mathLib.atan(myItem);
```

Related reference

“Mathematical (system words)” on page 580

mathLib.atan2

The system function **mathLib.atan2** computes the principal value of the arc tangent of y/x , using the signs of both arguments to determine the quadrant of the return value. The result is in radians and is in the range of $-\pi$ to π .

►► `result = mathLib.atan2 (numericItem1,numericItem2);` ◄◄

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by **mathLib.atan2** is converted to the format of *result* and returned in *result*.

numericItem1

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating point before **mathLib.atan2** is called. *numericItem1* is the y value.

numericItem2

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating point before **mathLib.atan2** is called. *numericItem2* is the x value.

Example:

```
myItemY = 1;
myItemX = 5;

// returns pi/2
result = mathLib.atan2(myItemY, myItemX);
```

Related reference

“Mathematical (system words)” on page 580

mathLib.ceiling

The system function **mathLib.ceiling** returns the smallest integer not less than a specified number.

►► `result = mathLib.ceiling (numericItem);` ◄◄

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The smallest integer not less than *numericItem* is converted to the format of *result* and returned in *result*.

numericItem

Any numeric or HEX item, as described in *Mathematical (system words)*.

Example:

```
myItem = 4.5;  
result = mathLib.ceiling(myItem); // result = 5
```

Related reference

“Mathematical (system words)” on page 580

mathLib.compareNum

The system function **mathLib.compareNum** returns a result (-1, 0, or 1) that indicates whether the first of two numbers is less than, equal to, or greater than the second.

►► `result = mathLib.compareNum (numericItem1 , numericItem2);` ◄◄

result

Item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places. This item receives one of the following values:

-1 *numericItem1* is less than *numericItem2*.

0 *numericItem1* is equal to *numericItem2*.

1 *numericItem1* is greater than *numericItem2*.

numericItem1

Any numeric or HEX item, as described in *Mathematical (system words)*.

numericItem2

Any numeric or HEX item, as described in *Mathematical (system words)*.

Example:

```
myItem01 = 4  
myItem02 = 7  
  
result = mathLib.compareNum(myItem01,myItem02);  
  
// result = -1
```

Related reference

“Mathematical (system words)” on page 580

mathLib.cos

The system function **mathLib.cos** returns the cosine of a number. The returned value is in the range of -1 to 1.

►► `result = mathLib.cos (numericItem);` ◄◄

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by **mathLib.cos** is converted to the format of *result* and returned in *result*.

numericItem

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before **mathLib.cos** is called.

Example:

```
result = mathLib.cos(myItem);
```

Related reference

“Mathematical (system words)” on page 580

mathLib.cosh

The system function **mathLib.cosh** returns the hyperbolic cosine of a number.

►► `result = mathLib.cosh (numericItem);` ◀◀

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by **mathLib.cosh** is converted to the format of *result* and returned in *result*.

numericItem

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before **mathLib.cosh** is called.

Example:

```
result = mathLib.cosh(myItem);
```

Related reference

“Mathematical (system words)” on page 580

mathLib.exp

The system function **mathLib.exp** returns *e* raised to the power of a number.

►► `result = mathLib.exp (numericItem);` ◀◀

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by **mathLib.exp** is converted to the format of *result* and returned in *result*.

numericItem

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before **mathLib.exp** is called.

Example:

```
result = mathLib.exp(myItem);
```

Related reference

“Mathematical (system words)” on page 580

mathLib.floatingAssign

The system function **mathLib.floatingAssign** returns *numericItem* as a double-precision floating-point number. The function assigns the value of BIN,

DECIMAL, NUM, NUMC, or PACKF items to floating-point numbers that are defined as HEX items, and vice versa.

►► `result = mathLib.floatingAssign (numericItem);` ◄◄

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The floating-point number is converted to the format of *result* and returned in *result*.

numericItem

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before being assigned to the result.

Example:

```
result = mathLib.floatingAssign(myItem);
```

Related reference

“Mathematical (system words)” on page 580

mathLib.floatingDifference

The system function **mathLib.floatingDifference** subtracts the second of two numbers from the first and returns the difference. The function is implemented using double-precision floating-point arithmetic.

►► `result = mathLib.floatingDifference (numericItem1,numericItem2);` ◄◄

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The difference is converted to the format of *result* and returned in *result*.

numericItem1

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before the difference is calculated.

numericItem2

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before the difference is calculated.

Example:

```
result = mathLib.floatingDifference(myItem01,myItem02);
```

Related reference

“Mathematical (system words)” on page 580

mathLib.floatingMod

The system function **mathLib.floatingMod** returns the floating-point remainder of one number divided by another. The result has the same sign as the numerator. A domain exception is raised if the denominator equals zero.

►► `result = mathLib.floatingMod (numericItem1,numericItem2);` ◄◄

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The floating-point remainder is converted to the format of *result* and returned in *result*.

numericItem1

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

numericItem2

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

Example:

```
result = mathLib.floatingMod(myItem01,myItem02);
```

Related reference

“Mathematical (system words)” on page 580

mathLib.floatingProduct

The system function **mathLib.floatingProduct** returns the product of two numbers. The function is implemented using double-precision floating-point arithmetic.

►► *result* = mathLib.floatingProduct (*numericItem1*,*numericItem2*); ◀◀

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The product is converted to the format of *result* and returned in *result*.

numericItem1

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

numericItem2

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

Example:

```
result = mathLib.floatingProduct(myItem01,myItem02);
```

Related reference

“Mathematical (system words)” on page 580

mathLib.floatingQuotient

The system function **mathLib.floatingQuotient** returns the quotient of one number divided by another. A domain exception is raised if the denominator equals zero. The function is implemented using double-precision floating-point arithmetic.

►► *result* = mathLib.floatingQuotient (*numericItem1*,*numericItem2*); ◀◀

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The quotient is converted to the format of *result* and returned in *result*.

numericItem1

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before the quotient is calculated.

numericItem2

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before the quotient is calculated.

Example:

```
result = mathLib.floatingQuotient(myItem01,myItem02);
```

Related reference

“Mathematical (system words)” on page 580

mathLib.floatingSum

The system function **mathLib.floatingSum** returns the sum of two numbers. The function is implemented using double-precision floating-point arithmetic.

►► `result = mathLib.floatingSum (numericItem1,numericItem2);` ◄◄

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The sum is converted to the format of *result* and returned in *result*.

numericItem1

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before the sum is calculated.

numericItem2

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before the sum is calculated.

Example:

```
result = mathLib.floatingSum(myItem01,myItem02);
```

Related reference

“Mathematical (system words)” on page 580

mathLib.floor

The system function **mathLib.floor** returns the largest integer not greater than a specified number.

►► `result = mathLib.floor (numericItem);` ◄◄

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The largest integer not greater than *numericItem* is converted to the format of *result* and returned in *result*.

numericItem

Any numeric or HEX item, as described in *Mathematical (system words)*.

Example:

```
myItem = 4.6;  
result = mathLib.floor(myItem); // result = 4
```

Related reference

“Mathematical (system words)” on page 580

mathLib.frexp

The system function **mathLib.frexp** splits a number into a normalized fraction in the range of .5 to 1 (which is returned as the *result*) and a power of 2 (which is returned in *integer*).

►► `result = mathLib.frexp (numericItem , integer);` ◀◀

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The floating-point fraction is converted to the format of *result* and returned in *result*.

numericItem

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

integer

Item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

Example:

```
result = mathLib.frexp(myItem,myInteger);
```

Related reference

“Mathematical (system words)” on page 580

mathLib.ldexp

The system function **mathLib.ldexp** returns the value of a specified number that is multiplied by the following value: two to the power of *integer*.

►► `result = mathLib.Ldexp (numericItem , integer);` ◀◀

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The calculated value is converted to the format of *result* and returned in *result*.

numericItem

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

integer

Item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

Example:

```
result = mathLib.ldexp(myItem,myInteger);
```

Related reference

“Mathematical (system words)” on page 580

mathLib.log

The system function **mathLib.log** returns the natural logarithm of a number.

►► `result = mathLib.log (numericItem);` ◀◀

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by the `mathLib.log` function is converted to the format of *result* and returned in *result*.

numericItem

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

Example:

```
result = mathLib.log(myItem);
```

Related reference

“Mathematical (system words)” on page 580

mathLib.log10

The system function **mathLib.log10** returns the base 10 logarithm of a number.

►► `result = mathLib.log10 (numericItem);` ◀◀

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by the `log10` function is converted to the format of *result* and returned in *result*.

numericItem

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

Example:

```
result = mathLib.log10(myItem);
```

Related reference

“Mathematical (system words)” on page 580

mathLib.maximum

The system function **mathLib.maximum** returns the greater of two numbers.

►► `result = mathLib.maximum (numericItem1 , numericItem2);` ◀◀

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The greater of two numbers is converted to the format of *result* and returned in *result*.

numericItem1

Any numeric or HEX item, as described in *Mathematical (system words)*.

numericItem2

Any numeric or HEX item, as described in *Mathematical (system words)*.

Example:

```
result = mathLib.maximum(myItem01,myItem02);
```

Related reference

“Mathematical (system words)” on page 580

mathLib.minimum

The system function **mathLib.minimum** returns the lesser of two numbers.

►► `result = mathLib.minimum (numericItem1 , numericItem2);` ◀◀

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The lesser of two numbers is converted to the format of *result* and returned in *result*.

numericItem1

Any numeric or HEX item, as described in *Mathematical (system words)*.

numericItem2

Any numeric or HEX item, as described in *Mathematical (system words)*.

Example:

```
result = mathLib.minimum(myItem01,myItem02);
```

Related reference

"Mathematical (system words)" on page 580

mathLib.modf

The system function **mathLib.modf** splits a number into integral and fractional parts, both with the same sign as the number. The fractional part is returned in *result* and the integral part is returned in *numericItem2*.

►► `result = mathLib.modf (numericItem1 , numericItem2);` ◀◀

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The fractional part of *numericItem1* is converted to the format of *result* and returned in *result*.

numericItem1

Any numeric or HEX item, as described in *Mathematical (system words)*.

numericItem2

Any numeric or HEX item, as described in *Mathematical (system words)*. The integral part of *numericItem1* is converted to the format of *numericItem2* and returned in *numericItem2*.

Example:

```
result = mathLib.modf(myItem01,myItem02);
```

Related reference

"Mathematical (system words)" on page 580

mathLib.pow

The system function **mathLib.pow** returns a number raised to the power of a second number. A domain exception is raised if on `pow(x,y)` the value of *x* is negative and *y* is non-integral, or the value of *x* is 0.0 and *y* is negative.

►► `result = mathLib.pow (numericItem1 , numericItem2);` ◀◀

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The result of the `mathLib.pow` function is converted to the format of *result* and returned in *result*.

numericItem1

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

numericItem2

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

Example:

```
result = mathLib.pow(myItem01,myItem02);
```

Related reference

"Mathematical (system words)" on page 580

mathLib.precision

The system function **mathLib.precision** returns the maximum precision (in decimal digits) for a number. For floating-point numbers (8-digit HEX for standard-precision floating-point number or 16-digit HEX for double-precision floating-point number), the precision is the maximum number of decimal digits that can be represented in the number for the system on which the program is running.

► `result = mathLib.precision (numericItem);` ◄

result

An item that receives the precision of *numericItem*. The *result* item is defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

numericItem

Any numeric or HEX item, as described in *Mathematical (system words)*.

Example:

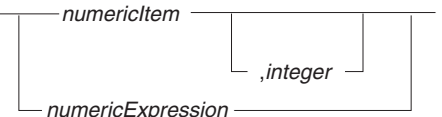
```
result = mathLib.precision(myItem);
```

Related reference

"Mathematical (system words)" on page 580

mathLib.round

The system function **mathLib.round** rounds a number or expression to a nearest value (for example, to the nearest thousands) and returns the result.

► `result = mathLib.round (`  `)` ◄

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The value produced by the rounding operation is converted to the format of *result* and returned in *result*.

The maximum supported length in this case is 17 rather than 18 because rounding occurs as follows:

- Add five to the digit in *result* at a precision one higher than the precision of the result digit
- Truncate the result

A numeric overflow occurs at run time if more than 17 digits are used in the calculation and if EGL cannot determine the violation at development time.

numericItem

Any numeric or HEX item, as described in *Mathematical (system words)*.

numericExpression

A numeric expression other than simply a numeric item. If you specify an operator, you cannot specify a value for *integer*.

You cannot use **mathLib.round** with the remainder operator (%).

integer

An integer that determines the value to which the number is rounded:

- If the integer is positive, the number is rounded to a nearest value equal to 10 to the power of *integer*. If integer is 3, for example, the number is rounded to the nearest thousands.
- The same is true if the integer is zero or negative; in that case, the number is rounded to the specified number of decimal places.

If you do not specify *integer*, **mathLib.round** rounds to the number of decimal places in *result*.

The integer is defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

Examples: In the next example, item balance is rounded to the nearest thousand:

```
balance = 12345.6789;  
rounder = 3;  
balance = mathLib.round(balance, rounder);  
// The value of balance is now 12000.0000
```

In the next example, a rounder value of -2 is used to round balance to two decimal places:

```
balance = 12345.6789;  
rounder = -2;  
balance = mathLib.round(balance, rounder);  
// The value of balance is now 12345.6800
```

Related reference

"Mathematical (system words)" on page 580

mathLib.sin

The system function **mathLib.sin** that returns the sine of a number. The result is in the range of -1 to 1.

►► `result = mathLib.sin (numericItem);` ◀◀

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by the **mathLib.sin** function is converted to the format of *result* and returned in *result*.

numericItem

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

Example:

```
result = mathLib.sin(myItem);
```

Related reference

“Mathematical (system words)” on page 580

mathLib.sinh

The system function **mathLib.sinh** returns the hyperbolic sine of a number.

►► `result = mathLib.sinh (numericItem);` ◀◀

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by the `mathLib.sinh` function is converted to the format of *result* and returned in *result*.

numericItem

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

Example:

```
result = mathLib.sinh(myItem);
```

Related reference

“Mathematical (system words)” on page 580

mathLib.sqrt

The math function **mathLib.sqrt** returns the square root of a number. The function operates on any number that is greater than or equal to zero.

►► `result = mathLib.sqrt (numericItem);` ◀◀

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by the `mathLib.sqrt` function is converted to the format of *result* and returned in *result*.

numericItem

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

Example:

```
result = mathLib.sqrt(myItem);
```

Related reference

“Mathematical (system words)” on page 580

mathLib.tan

The system function **mathLib.tan** returns the tangent of a number.

►► `result = mathLib.tan (numericItem);` ◀◀

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by the `mathLib.tan` function is converted to the format of *result* and returned in *result*.

numericItem

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

Example:

```
result = mathLib.tan(myItem);
```

Related reference

“Mathematical (system words)” on page 580

mathLib.tanh

The system function **mathLib.tanh** returns the hyperbolic tangent of a number. The result is in the range of -1 to 1.

►► `result = mathLib.tanh (numericItem);` ◀◀

result

Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by the `mathLib.tanh` function is converted to the format of *result* and returned in *result*.

numericItem

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

Example:

```
result = mathLib.tanh(myItem);
```

Related reference

“Mathematical (system words)” on page 580

String handling (system words)

String-handling system functions provide operations such as comparing or concatenating strings, each of which is a fixed-length sequence of bytes. For EGL, a string is an item with a type of CHAR, DBCHAR, HEX, MBCHAR, or UNICODE.

The meaning of a blank in a string varies by primitive type, as described in *Data initialization*.

A substring is a subset of a string and is identified by an index and a length. The index value identifies the starting byte of the substring within the item. The index value for the first byte in the item is 1. The length is the number of bytes in the substring.

To prevent substring definition from extending outside an item, the index must be a value between 1 and the number of bytes in the item, and the substring length must not extend beyond the end of the item that contains the substring. Lengths that are too long are adjusted so that the substring ends at the last byte of the item.

Functions can raise exception conditions, which your program can detect in these cases:

- The function is in a try block; or
- The function word `sysVar.handleSysLibErrors` is set to 1 when the program runs in VisualAge Generator compatibility mode.

The next table lists the string-handling system words.

System function and invocation	Description
<code>strLib.compareStr</code> <code>result = strLib.compareStr</code> <code>(target, targetSubstringIndex,</code> <code>targetSubstringLength, source,</code> <code>sourceSubstringIndex,</code> <code>sourceSubstringLength)</code>	Compares two substrings in accordance with their ASCII or EBCDIC order at run time and returns a value (-1, 0, or 1) to indicate which is greater.
<code>strLib.concatenate</code> <code>result = strLib.concatenate</code> <code>(target, source)</code>	Concatenates <i>target</i> and <i>source</i> ; places the new string in <i>target</i> ; and returns an integer that indicates whether <i>target</i> was long enough to contain the new string
<code>strLib.concatenateWithSeparator</code> <code>result = strLib.concatenateWithSeparator</code> <code>(target, source, separator)</code>	Concatenates <i>target</i> and <i>source</i> , inserting <i>separator</i> between them; places the new string in <i>target</i> ; and returns an integer that indicates whether <i>target</i> was long enough to contain the new string
<code>strLib.copyStr</code> <code>strLib.copyStr</code> <code>(target, targetSubstringIndex,</code> <code>targetSubstringLength, source,</code> <code>sourceSubstringIndex,</code> <code>sourceSubstringLength)</code>	Copies one substring to another
<code>strLib.findStr</code> <code>result = strLib.findStr</code> <code>(source,</code> <code>sourceSubstringIndex,</code> <code>sourceSubstringLength,</code> <code>searchString)</code>	Searches for the first occurrence of a substring within a string
<code>strLib.getNextToken</code> <code>result = strLib.getNextToken</code> <code>(target, source, sourceSubstringIndex,</code> <code>sourceStringLength,</code> <code>characterDelimiter)</code>	Searches a string for the next token and copies the token to <i>target</i>
<code>strLib.setBlankTerminator</code> <code>strLib.setBlankTerminator</code> <code>(target)</code>	Replaces a null terminator and any subsequent characters in a string with spaces, so that a string value returned from a C or C++ program can operate correctly in an EGL-generated program
<code>strLib.setNullTerminator</code> <code>strLib.setNullTerminator</code> <code>(target)</code>	Changes all trailing spaces in a string to nulls
<code>strLib.setSubStr</code> <code>strLib.setSubStr</code> <code>(target, targetSubstringIndex,</code> <code>targetSubstringLength, source)</code>	Replaces each character in a substring with a specified character
<code>strLib.strLen</code> <code>result = strLib.strLen</code> <code>(source)</code>	Returns the number of bytes in an item, excluding any trailing spaces or nulls

Related concepts

"Compatibility with VisualAge Generator" on page 276

Related reference

"Assignments" on page 296

"Data initialization" on page 277

"EGL statements" on page 70

"Exception handling" on page 75

"Function invocations" on page 316

"String expressions" on page 365

"System words" on page 79

"System words in alphabetical order" on page 508

"try" on page 355

strLib.compareStr

The system function **strLib.compareStr** compares two substrings in accordance with their ASCII or EBCDIC order at run time.

►► *result* = strLib.compareStr (*target* , *targetSubstringIndex* ,

► *targetSubstringLength* , *source* , *sourceSubstringIndex* ,

► *sourceSubstringLength*); ►►

result

Numeric item that receives one of the following values (defined as type INT or the equivalent: type BIN with length 9 and no decimal places) returned by the function:

- 1 The substring based on *target* is less than the substring based on *source*
- 0 The substring based on *target* is equal to the substring based on *source*
- 1 The substring based on *target* is greater than the substring based on *source*

target

String from which a target substring is derived. Can be an item or a literal.

targetSubStringIndex

Identifies the starting byte of the substring in *target*, given that the first byte in *target* has the index value 1. This index can be an integer literal. Alternatively, this index can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

targetSubStringLength

Identifies the number of bytes in the substring that is derived from *target*. The length can be an integer literal. Alternatively, this index can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

source

String from which a source substring is derived. Can be an item or a literal.

sourceSubStringIndex

Identifies the starting byte of the substring in *source*, given that the first byte in *source* has the index value of 1. This index can be an integer literal.

Alternatively, this index can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

sourceSubStringLength

Identifies the number of bytes in the substring that is derived from *source*. The length can be an integer literal. Alternatively, this index can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

A byte-to-byte binary comparison of the substring values is performed. If the substrings are not the same length, the shorter substring is padded with spaces before the comparison.

Definition considerations: The following values are returned in **sysVar.errorCode**:

- 8 Index less than 1 or greater than string length.
- 12 Length less than 1.
- 20 Invalid double-byte index. Index for a DBCHAR or UNICODE string points to middle of double-byte character
- 24 Invalid double-byte length. Length in bytes for a DBCHAR or UNICODE string is odd (double-byte lengths must always be even).

Example:

```
target = "123456";
source = "34";
result =
    strLib.compareStr(target,3,2,source,1,2);
// result = 0
```

Related reference

“String handling (system words)” on page 597

strLib.concatenate

The system function **strLib.concatenate** concatenates two strings.

►► *result* = strLib.concatenate (*target* , *source*); ◄◄

result

Numeric item that receives one of the following values (defined as type INT or the equivalent: type BIN with length 9 and no decimal places) returned by the function:

- 1 Concatenated string is too long to fit in the target item and the string was truncated, as described later
- 0 Concatenated string fits in the target item

target

Target item

source

Source item or literal

When two strings are concatenated, the following occurs:

1. Any trailing spaces or nulls are deleted from the target string.
2. The source string is appended to the string produced by step 1.

3. If the string produced by step 2 on page 600 is longer than the target string item, it is truncated. If it is shorter than the target item, it is padded with blanks.

Example:

```
phrase = "and/ "; // CHAR(7)
or      = "or";
result =
  strLib.concatenate(phrase,or);
if (result = 0)
  print phrase; // phrase = "and/or "
end
```

Related reference

“String handling (system words)” on page 597

strLib.concatenateWithSeparator

The system function **strLib.concatenateWithSeparator** concatenates two strings, inserting a separator string between them. If the initial length of the target string is zero (not counting trailing blanks and nulls), the separator is omitted and the source string is copied to the target string.

►► *result* = strLib.concatenateWithSeparator (*target* , *source* , *separator*); ◀◀

result

Numeric item that receives one of the following values (defined as type INT or the equivalent: type BIN with length 9 and no decimal places) returned by the function: :

- 0** Concatenated string fits in target item.
- 1** Concatenated string is too long to fit in the target item and the string was truncated, as described later

target

Target item.

source

Source item or literal.

separator

Separator item or literal.

Trailing spaces and nulls are truncated from *target*; then, the *separator* string and *source* are appended to the truncated value. If the concatenation is longer than the target allows, truncation occurs. If the concatenation is shorter than the target allows, the concatenated value is padded with spaces.

Example:

```
phrase = "and"; // CHAR(7)
or      = "or";
result =
  strLib.concatenateWithSeparator(phrase,or,"/");
if (result = 0)
  print phrase; // phrase = "and/or "
end
```

Related reference

“String handling (system words)” on page 597

strLib.copyStr

The system function **strLib.copyStr** copies one substring to another.

```
►► strLib.copyStr ( target , targetSubstringIndex, —————►  
  
► targetSubstringLength , source , sourceSubstringIndex , —————►  
  
► sourceSubstringLength ); —————►
```

target

String from which a target substring is derived. Can be an item or a literal.

targetSubstringIndex

Identifies the starting byte in *target*, given that the first byte in *target* has the value 1. This index can be an integer literal. Alternatively, this index can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

targetSubstringLength

Identifies the number of bytes in the substring that is derived from *target*. The length can be an integer literal. Alternatively, the length can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

source

String from which a source substring is derived. Can be an item or a literal.

sourceSubstringIndex

Identifies the starting byte of the substring in *source*, given that the first byte in *source* has the value 1. This index can be an integer literal. Alternatively, this index can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

sourceSubstringLength

Identifies the number of bytes in the substring that is derived from *source*. The length can be an integer literal. Alternatively, the length can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

If the source is longer than the target, the source is truncated. If the source is shorter than the target, the source value is padded on the right with spaces.

Definition considerations: The following values are returned in **sysVar.errorCode**:

- 8 Index less than 1 or greater than string length.
- 12 Length less than 1.
- 20 Invalid double-byte index. Index for a DBCHAR or UNICODE string points to middle of double-byte character.
- 24 Invalid double-byte length. Length in bytes for a DBCS or UNICODE string is odd (double-byte lengths must always be even).

Example:

```
target = "120056";
source = "34";
strLib.copyStr(target,3,2,source,1,2);
// target = "123456"
```

Related reference

“String handling (system words)” on page 597

strLib.findStr

The system function **strLib.findStr** searches for the first occurrence of a substring in a string.

► `result = strLib.FindStr (source , sourceSubstringIndex ,` ————— ►

`sourceSubstringLength , searchString);` ————— ►►

result

Numeric item that receives one of the following values (defined as type INT or the equivalent: type BIN with length 9 and no decimal places) returned by the function:

- 1** Search string was not found
- 0** Search string was found

source

String from which a source substring is derived. Can be an item or a literal.

sourceSubstringIndex

Identifies the starting byte for the substring in *source*, given that the first byte in *source* has the index value of 1. This index can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

sourceStringLength

Identifies the number of bytes in the substring that is derived from *source*. This index can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

searchString

String item or literal to be searched for in the source substring. Trailing blanks or nulls are truncated from the search string before searching begins.

If *searchString* is found in the source substring, *sourceSubstringIndex* is set to indicate its location (the byte of the source where the matching substring begins). Otherwise, *sourceSubstringIndex* is not changed.

Definition considerations: The following values are returned in sysVar.errorCode:

- 8** Index less than 1 or greater than string length.
- 12** Length less than 1.
- 20** Invalid double-byte index. Index for a DBCHAR or UNICODE string points to middle of double-byte character.
- 24** Invalid double-byte length. Length in bytes for a DBCHAR or UNICODE string is odd (double-byte lengths must always be even).

Example:

```

source = "123456";
sourceIndex = 1
sourceLength = 6
search = "34";
result =
    strLib.findStr(source,sourceIndex,sourceLength,"34");
// result = 0, sourceIndex = 3

```

Related reference

“String handling (system words)” on page 597

strLib.getNextToken

The system function **strLib.getNextToken** searches a substring for a token and copies that token to a target item.

Tokens are strings separated by delimiter characters. For example, if the characters space (" ") and comma (",") are defined as delimiters, the string "CALL PROGRAM ARG1,ARG2,ARG3" can be broken down into the five tokens "CALL", "PROGRAM", "ARG1", "ARG2", and "ARG3".

► *result* = strLib.getNextToken (*target* , *source* , *sourceSubstringIndex* , —————►

► *sourceStringLength* , *characterDelimiter*); —————►

result

An item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places. The value is one of these:

- +*n* Number of characters in the token. The token is copied from the substring under review to the target item.
- 0 No token was in the substring under review.
- 1 The token was truncated when copied to the target item.

target

Target item of type CHAR, DBCHAR, HEX, MBCHAR, or UNICODE.

source

Source item of type CHAR, DBCHAR, HEX, MBCHAR, or UNICODE. May be a literal of any of those types other than UNICODE.

sourceSubstringIndex

Identifies the starting byte at which to begin searching for a delimiter, given that the first byte in *source* has the value 1. *sourceSubstringIndex* can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places. If a token is found, the value in *sourceSubstringIndex* is changed to the index of the first character that follows the token.

sourceSubstringLength

Indicates the number of bytes in the substring under review. *sourceSubstringLength* can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places. If a token is found, the value in *sourceSubstringLength* is changed to the number of bytes in the substring that begins after the returned token.

characterDelimiter

One or more delimiter characters, with no characters separating one from the next. May be an item of type CHAR, DBCHAR, HEX, MBCHAR, or UNICODE. May be a literal of any of those types other than UNICODE.

You can invoke a sequence of calls to retrieve each token in a substring without resetting the values for *sourceSubstringIndex* and *sourceSubstringLength*, as shown in a later example.

Error conditions: The following values are returned in **sysVar.errorCode**:

- 8 *sourceSubstringIndex* is less than 1 or is greater than number of bytes in the substring under review.
- 12 *sourceSubstringLength* is less than 1.
- 20 The value in *sourceSubstringIndex* for a DBCHAR or UNICODE string refers to the middle of a double-byte character.
- 24 The value in *sourceSubstringLength* for a DBCHAR or UNICODE string is odd (double-byte lengths must always be even).

Example:

```
Function myFunction()
  myVar myStructurePart;
  myRecord myRecordPart;

  i = 1;
  myVar.mySourceSubstringIndex = 1;
  myVar.mySourceSubstringLength = 29;

  while (myVar.mySourceSubstringLength > 0)
    myVar.myResult = strLib.getNextToken( myVar.myTarget[i],
      "CALL PROGRAM arg1, arg2, arg3",
      myVar.mySourceSubstringIndex,
      myVar.mySourceSubstringLength, " ," );

    if (myVar.myResult > 0)
      myRecord.outToken = myVar.myTarget[i];
      add myRecord;
      set myRecord empty;
      i = i + 1;
    end
  end
end

Record myStructurePart
  01 myTarget CHAR(80)[5];
  01 mySource CHAR(80);
  01 myResult myBinPart;
  01 mySourceSubstringIndex INT;
  01 mySourceSubstringLength BIN(9,0);
  01 i myBinPart;
end

Record myRecordPart
  serialRecord:
    fileName="Output"
  end
  01 outToken CHAR(80);
end
```

Related reference

“String handling (system words)” on page 597

strLib.setBlankTerminator

The system function **strLib.setBlankTerminator** changes a null terminator and any subsequent characters to spaces. **strLib.setBlankTerminator** changes a string value

returned from a C or C++ program to a character value that can operate correctly in an EGL program.

►► `strLib.setBlankTerminator (target);` ◄◄

target

The target string item. If no null is found in *target*, the function has no effect.

Example:

```
strLib.setBlankTerminator(target);
```

Related reference

“String handling (system words)” on page 597

strLib.setNullTerminator

The system function **strLib.setNullTerminator** changes all trailing spaces in a string to nulls. You can use **strLib.setNullTerminator** to convert an item before passing it to a C or C++ program that expects a null-terminated string as an argument.

►► `strLib.setNullTerminator (target);` ◄◄

target

String to be converted

The target string is searched for trailing spaces and nulls. Any spaces found are changed to nulls.

Definition considerations: The following value can be returned in **sysVar.errorCode**:

16 Last byte of string is not a space or null

Example:

```
strLib.setNullTerminator(myItem01);
```

Related reference

“String handling (system words)” on page 597

strLib.setSubStr

The system function **strLib.setSubStr** replaces each character in a substring with a specified character.

►► `strLib.setSubStr (target , targetSubstringIndex , targetSubstringLength , source);` ◄◄

target

Item that is changed.

targetSubStringIndex

Identifies the starting byte of the substring in *target*, given that the first byte in *target* has the index value of 1. This index can be an integer literal.

Alternatively, this index can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

targetSubstringLength

Identifies the number of bytes in the substring that is derived from *target*. The

length can be an integer literal. Alternatively, the length can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

source

If the target item is CHAR, MBCHAR, or HEX, the source item must be a one-byte CHAR, MBCHAR, or HEX item or a CHAR literal. If the target is a DBCHAR or UNICODE item, the source must be a single-character DBCHAR or UNICODE item.

Definition considerations: The following values are returned in sysVar.errorCode:

- 8 Index less than 1 or greater than string length
- 12 Length less than 1
- 20 Invalid double-byte index. Index for a DBCHAR or UNICODE string points to middle of double-byte character
- 24 Invalid double-byte length. Length in bytes for a DBCHAR or UNICODE string is odd (double-byte lengths must always be even)

Example:

```
strLib.setSubStr(target,12,5," ");
```

Related reference

“String handling (system words)” on page 597

strLib.strLen

The system function **strLib.strLen** returns the number of bytes in an item, excluding any trailing spaces and nulls.

►► *length* = strLib.strLen (*source*) ; —————►◄

length

An item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

source

String item or literal to be measured.

Example:

```
length = strLib.strLen(source);
```

Related reference

“String handling (system words)” on page 597

Web-application system words

sysLib.clearRequestAttr

The system function **sysLib.clearRequestAttr** removes the argument that is associated with the specified key in the request object. This function is useful in page handlers and in programs that run in Web applications.

You can set an argument in the request object by using the system function **sysLib.setRequestAttr**. You can retrieve the argument by using the system function **sysLib.getRequestAttr**.

►► `sysLib.clearRequestAttr (key);` ————— ◄◄

Related concepts

“PageHandler part” on page 469

Related reference

“`sysLib.getRequestAttr`”

“`sysLib.setRequestAttr`” on page 609

sysLib.clearSessionAttr

The system function **sysLib.clearSessionAttr** removes the argument that is associated with the specified key in the session object. This function is useful in page handlers and in programs that run in Web applications.

You can set an argument in the session object by using the system function `sysLib.setSessionAttr`. You can retrieve the argument by using the system function `sysLib.getSessionAttr`.

►► `sysLib.clearSessionAttr (key);` ————— ◄◄

Related concepts

“PageHandler part” on page 469

Related reference

“`sysLib.getSessionAttr`” on page 609

“`sysLib.setSessionAttr`” on page 610

sysLib.getRequestAttr

The system function **sysLib.getRequestAttr** uses a specified key to retrieve an argument from the request object into a specified variable. This function is useful in page handlers and in programs that run in Web applications.

If an object is not found with the specified key, the target variable is unchanged. If the retrieved object is of the wrong type, an exception is thrown and the program or page handler terminates.

You can place an argument in the request object by using the system function **sysLib.setRequestAttr**. The argument object placed in the servlet's request collection is available for access as long as the servlet request is valid. Submitting a form from a page causes the creation of a new request.

►► `sysLib.getRequestAttr (key , variableArgument);` ————— ◄◄

key

A character literal or an item of any character type.

argument

An item, record, or array.

Related concepts

“PageHandler part” on page 469

Related reference

“sysLib.setRequestAttr”

sysLib.getSessionAttr

The system function **sysLib.getSessionAttr** uses a specified key to retrieve an argument from the session object into a specified variable. This function is useful in page handlers and in programs that run in Web applications.

If an object is not found with the specified key, the target variable is unchanged. If the retrieved object is of the wrong type, an exception is thrown and the program or page handler terminates.

You can place an argument in the session object by using the system function **sysLib.setSessionAttr**.

►► sysLib.getSessionAttr (*key* , *variableArgument*) ; ◄◄

key

A character literal or an item of any character type.

argument

An item, record, or array.

Related concepts

“PageHandler part” on page 469

Related reference

“sysLib.setSessionAttr” on page 610

sysLib.setRequestAttr

The system function **sysLib.setRequestAttr** uses a specified key to place a specified argument in the request object. This function is useful in page handlers and in programs that run in Web applications. You can retrieve the argument later by using the system function **sysLib.getRequestAttr**.

►► sysLib.setRequestAttr (*key* , *argument*) ; ◄◄

key

A character literal or an item of any character type.

argument

An item, record, or array.

In the generated Java output, item arguments are passed as primitive Java objects (String, Integer, Decimal, and so on). Record arguments are passed as record beans. Arrays are passed as an array list of the associated type. The argument object is placed in the servlet’s request collection and is available for access as long as the servlet request is valid. Submitting a form from a page causes the creation of a new request.

Related concepts

“PageHandler part” on page 469

Related reference

“sysLib.getRequestAttr” on page 608

sysLib.setSessionAttr

The system function **sysLib.setSessionAttr** uses a specified key to place a specified argument in the session object. This function is useful in page handlers and in programs that run in Web applications. You can retrieve the argument later by using the system function **sysLib.getSessionAttr**.

►► sysLib.setSessionAttr (*key*, *argument*) ; ◄◄

key

A character literal or an item of any character type.

argument

An item, record, or array.

In the generated Java output, item arguments are passed as primitive Java objects (String, Integer, Decimal, and so on). Record arguments are passed as record beans. Arrays are passed as an array list of the associated type.

Related concepts

“PageHandler part” on page 469

Related reference

“sysLib.getSessionAttr” on page 609

Miscellaneous system words

sysLib.bytes

The system function **sysLib.bytes** returns the number of bytes in a named area of memory.

►► result = sysLib.bytes (*itemOrRecord*); ◄◄

result

A numeric item that receives the number of bytes in *itemOrRecord*. Two cases are special:

- If *itemOrRecord* is an array, *result* receives the number of bytes in one element
- If *itemOrRecord* is an SQL record, *result* receives the number of bytes in the record, including the extra bytes; for details see *SQL record internals*

itemOrRecord

An array, item, or record

Example:

```
result = sysLib.bytes(myItem);
```

Related reference

“Primitive types” on page 27

“SQL record internals” on page 199

sysLib.calculateChkDigitMod10

The system function **sysLib.calculateChkDigitMod10** places a modulus-10 check digit in a character item that begins with a series of integers.

►► sysLib.calculateChkDigitMod10 (*input*, *checkLength*, *result*); ◀◀

input

A character item that begins with a series of integers. The item must include an additional position for the check digit, which goes immediately to the right of the other integers.

checkLength

An item that contains the number of characters that you want to use from the *input* item, including the position used for the check digit. This item has 4 digits and is either of type SMALLINT or is of a type BIN, with no decimal places.

result

An item that receives one of two values:

- 0, if the check digit was created
- 1, if the check digit was not created

This item has 4 digits and is either of type SMALLINT or is of a type BIN, with no decimal places.

You can use **sysLib.calculateChkDigitMod10** in a function-invocation statement.

Example: In the following example, *myInput* is an item of type CHAR and contains the value 1734289; *myLength* is an item of type SMALLINT and contains the value 7; and *myResult* is an item of type SMALLINT:

```
sysLib.verifyChkDigitMod10 (myInput, myLength, myResult);
```

An algorithm is used to derive the modulus-10 check digit, and in all cases the number at the check-digit position is not considered. The algorithm is described in relation to the example values:

1. Multiply the units position of the input number by 2 and multiply every alternate position, moving right to left, by 2:
$$\begin{array}{l} 8 \times 2 = 16 \\ 4 \times 2 = 8 \\ 7 \times 2 = 14 \end{array}$$
2. Add the digits of the products (16814) to the input-number digits (132) that were not multiplied by 2:
$$1 + 6 + 8 + 1 + 4 + 1 + 3 + 2 = 26$$
3. To get the check digit, subtract the sum from the next-highest number ending in 0:
$$30 - 26 = 4$$

If the subtraction yields 10, the check digit is 0.

In this example, the original characters in *myInput* become these:

1734284

sysLib.calculateChkDigitMod11

The system function **sysLib.calculateChkDigitMod11** places a modulus-11 check digit in a character item that begins with a series of integers.

input

A character item that begins with a series of integers. The item must include an additional position for the check digit, which goes immediately to the right of the other integers.

checkLength

An item that contains the number of characters that you want to use from the *input* item, including the position used for the check digit. This item has 4 digits and is either of type SMALLINT or is of a type BIN, with no decimal places.

result

An item that receives one of two values:

- 0, if the check digit was created
- 1, if the check digit was not created

This item has 4 digits and is either of type SMALLINT or is of a type BIN, with no decimal places.

You can use **sysLib.calculateChkDigitMod11** in a function-invocation statement.

Example: In the following example, *myInput* is an item of type CHAR and contains the value 56621869; *myLength* is an item of type SMALLINT and contains the value 8; and *myResult* is an item of type SMALLINT:

```
sysLib.verifyChkDigitMod (myInput, myLength, myResult);
```

An algorithm is used to derive the modulus-11 check digit, and in all cases the number at the check-digit position is not considered. The algorithm is described in relation to the example values:

1. Multiply the digit at the units position of the input number by 2, at the tens position by 3, at the hundreds position by 4, and so on, but let *myLength* – 1 be the largest number used as a multiplier; and if more digits are in the input number, begin the sequence again using 2 as a multiplier:

$$\begin{array}{l} 6 \times 2 = 12 \\ 8 \times 3 = 24 \\ 1 \times 4 = 4 \\ 2 \times 5 = 10 \\ 6 \times 6 = 36 \\ 6 \times 7 = 42 \\ 5 \times 2 = 10 \end{array}$$

2. Add the products of the first step and divide the sum by 11:

$$\begin{array}{l} (12 + 24 + 4 + 10 + 36 + 42 + 10) / 11 \\ = 138 / 11 \\ = 12 \text{ remainder } 6 \end{array}$$

3. To get the check digit, subtract the remainder from 11 to get the self-checking digit:

$$11 - 6 = 5$$

If the remainder is 0 or 1, the check digit is 0.

In this example, the original characters in *myInput* become these:

56621865

sysLib.clearScreen

The system function **sysLib.clearScreen** clears the screen, as is useful before the program issues a converse statement in a text application.

►► sysLib.clearScreen (); ◄◄

Related reference

“converse” on page 306

“System words in alphabetical order” on page 508

sysLib.fieldInputLength

The system function **sysLib.fieldInputLength** returns the number of characters that the user typed in the input field when the text form was last presented. That number does not include leading or trailing blanks or nulls.

If the field is at its originally defined state, the function returns a length of 0. For example, if the field contains the *value* property and it has not been modified during execution in any way, then the length is calculated as 0. The *set form initial* statement resets the field to its originally defined state. If the field is not at its originally defined state, then the length is calculated based on what was displayed or entered on the last converse statement.

►► result = sysLib.fieldInputLength (*textField*); ◄◄

textField

The name of the text field.

Related reference

“System words in alphabetical order” on page 508

sysLib.getVAGSysType

The system function **sysLib.getVAGSysType** identifies the target system in which the program is running. The function is supported (at development time) if the program property **VAGCompatibility** is selected or (at generation time) if the build descriptor option **VAGCompatibility** is set to *yes*.

If the generated output is a Java wrapper, **sysLib.getVAGSysType** is not available. Otherwise, the function returns the character value that would have been returned by the VisualAge Generator EZESYS special function word. If the current system was not supported by VisualAge Generator, the function returns the uppercase, string equivalent of the code returned by **sysVar.systemType**.

►► result = sysLib.getVAGSysType() ; ◄◄

result

A character string that contains the system type code, as shown in the next table.

sysLib.getVAGSysType returns the VisualAge Generator equivalent of the value in **sysVar.systemType**.

Value in <code>sysVar.systemType</code>	Value returned by <code>sysLib.getVAGSysType</code>
AIX	"AIX"
DEBUG	"ITF"
ISERIESC	"OS400"
ISERIESJ	"OS400"
LINUX	"LINUX"
USS	"OS390"
WIN	"WINNT"

The value returned by `sysLib.getVAGSysType` can be used only as a character string; you cannot use the returned value with the operands *is* or *not* in a logical expression, as you can with `sysVar.systemType`:

```
// valid ONLY for sysVar.systemType
if sysVar.systemType is AIX
  call myProgram;
end
```

The only place that `sysLib.getVAGSysType` can be used is as the source in an assignment or **move** statement.

The characteristics of `sysLib.getVAGSysType` are as follows:

Primitive type

CHAR

Data length

8 (padded with blanks)

Is value always restored after a converse?

Yes

It is recommended that you use `sysVar.systemType` instead of `sysLib.getVAGSysType`.

Definition considerations: The value of `sysLib.getVAGSysType` does not affect what code is validated at generation time. For example, the following **add** statement is validated even if you are generating for Windows:

```
mySystem CHAR(8);
mySystem = sysLib.getVAGSysType();
if (mySystem = "AIX")
  add myRecord;
end
```

To avoid validating code that will never run in the target system, move the statements that you do not want to validate to a second program; then, let the original program call the new program conditionally:

```
mySystem CHAR(8);
mySystem = sysLib.getVAGSysType();

if (mySystem = "AIX")
  call myAddProgram myRecord;
end
```

An alternative way to solve the problem is available, but only if you use `sysVar.systemType` instead of `sysLib.getVAGSysType`; for details, see *eliminateSystemDependentCode*.

Related reference

"eliminateSystemDependentCode" on page 249

"sysVar.systemType" on page 628

sysLib.maximumSize

The system function **sysLib.maximumSize** returns the maximum number of rows that can be in a dynamic array of data items or records; specifically, the function returns the value of the array property **maxSize**.

►► `result = sysLib.maximumSize(arrayName) ;` ◄◄

arrayName

Name of the dynamic array.

Definition considerations: The item to which the value is returned must be of type INT or the following equivalent: type BIN with length 9 and no decimal places.

The array name may be qualified by a package name, a library name, or both

An error occurs if you reference an item or record that is not a dynamic array.

Related reference

"Arrays" on page 64

"System words in alphabetical order" on page 508

sysLib.pageEject

The system function **sysLib.pageEject** advances print-form output to the top of the next page, as is useful before the program issues a print statement.

►► `sysLib.pageEject () ;` ◄◄

For other details on printing, see *Print forms*.

Related concepts

"Print forms" on page 368

Related reference

"print" on page 341

"System words in alphabetical order" on page 508

sysLib.setLocale

The system function **sysLib.setLocale** is used in page handlers. The function sets the Java locale, which determines these aspects of run-time behavior:

- The human language used for labels and messages
- The default date and time formats

You might present a list of languages on a Web page, for example, and set the Java locale based on the user's selection. The new Java locale is in use until one of the following occurs:

- You invoke **sysLib.setLocale** again; or
- The browser session ends; or

- A new Web page is presented otherwise.

In the cases mentioned, the next Web page reverts (by default) to the Java locale specified in the browser.

If the user submits a form or clicks a link that opens a new window, the Java locale in the original window is unaffected by the locale in the new window.

sysLib.setLocale conforms to the JDK 1.1 and 1.2 API documentation for class `java.util.Locale`. See ISO 639 for language codes and ISO 3166 for country codes.

```

▶▶ sysLib.setLocale ( languageCode , countryCode , variant );

```

languageCode

A two-character language code specified as a literal or contained in an item of type CHAR. Only language codes that are defined by ISO 639 are valid.

countryCode

A two-character country code specified as a literal or contained in an item of type CHAR. Only country codes that are defined by ISO 3166 are valid.

variant

A variant, which is a code specified as a literal or contained in an item of type CHAR. This code is not part of a Java specification but depends on the browser and other aspects of the user environment.

Related concepts

"PageHandler part" on page 469

sysLib.setRemoteUser

The system function **sysLib.setRemoteUser** sets the userid and password that are used on calls to remote programs from Java programs.

```

▶▶ sysLib.setRemoteUser ( userID, password );

```

userID

The user ID on the remote system.

passWord

The password on the remote system.

When the linkage option part, callLink element, property remoteComType is CICSJ2C, CICSECI, or JAVA400 on a remote call, authorization is based on the values (if non-blank) that are passed to **sysLib.setRemoteUser**. If a value is blank or not specified, it is sought in a properties file named csoidpwd.properties. Within the csoidpwd.properties file, the values of the CSOUID and CSOPWD properties are used for the username and password. If neither approach is used, EGL run-time makes the call without a username and password.

One way to provide the user ID and password is for your program to issue Java access functions that display a dialog box to prompt the user for the information.

Related concepts

"Java access functions" on page 556

Related reference

“remoteComType in callLink element” on page 455

“System words in alphabetical order” on page 508

sysLib.size

The system function **sysLib.size** returns the number of rows in the specified data table or the number of elements in the specified array. The array may be a structure-item array, a static array of data items or records, or a dynamic array of data items or records.

►► `result = sysLib.size(arrayName) ;` ◄◄

arrayName

Name of the array or data table.

Definition considerations: The item to which the value is returned must be of type INT or the following equivalent: type BIN with length 9 and no decimal places.

If the array name (*arrayName*) is in a substructured element of another array, the returned value is the number of occurrences for the structure item itself, not the total number of occurrences in the containing structure (see *Examples* section).

The array name may be qualified by a package name, a library name, or both

An error occurs if you reference an item or record that is not an array.

Examples: This example uses the value returned by **sysLib.size** to control a loop:

```
// Calculate the sum of an array of numbers
sum = 0;
i = 1;
myArraySize = sysLib.size(myArray);

while (i <= myArraySize)
  sum = myArray[i] + sum;
  i = i + 1;
end
```

Next, consider the following record part:

```
Record myRecordPart
  10 siTop CHAR(40)[3];
  20 siNext CHAR(20)[2];
end
```

Given that you create a record based on `myRecordPart`, you can use **sysLib.size(siNext)** to determine the occurs value for the subordinate array:

```
// Sets count to 2
count = sysLib.size(myRecord.siTop.siNext);
```

Related reference

“Arrays” on page 64

“System words in alphabetical order” on page 508

sysLib.startTransaction

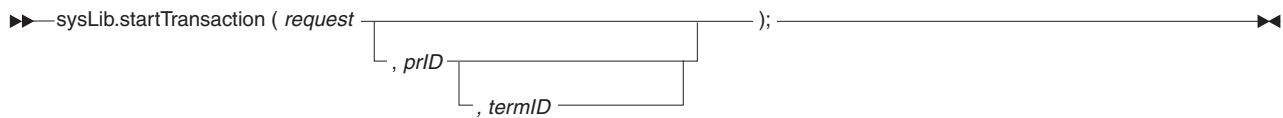
The system function **sysLib.startTransaction** invokes a main program asynchronously, associates that program with a printer or terminal device, and

passes a record. If the receiving program is generated by EGL, the record is used to initialize the input record; if the receiver is produced by VisualAge Generator, the record is used to initialize the working storage.

This function is not supported in programs that are generated as iSeries COBOL programs.

The default behavior of this function is to start a program that resides in the same Java package. To change that behavior, specify an `asynchLink` element in the linkage options part that is used to generate the invoking program.

A Java program can transfer only to another Java program on the same machine.



request

The name of a basic record, which must have the following format:

- The first 2 bytes (of type SMALLINT or of type BIN without decimals) contain the length of the data to be passed to the started transaction, plus 10 for the two fields (including this one) that are not passed. The value cannot exceed 32767 bytes. If the target is a Java program, the value cannot exceed 32767 bytes; but if the target is a COBOL program on iSeries, the value cannot exceed 4095 bytes.
- The next 8 bytes (of type CHAR) are also not passed, but contain the name of the program to be started.
- The remaining part of the request record is passed.

prID

This optional 4-byte item of type CHAR is used only for transferring to a COBOL program on iSeries. The item contains the value of the output queue used for the asynchronous job, and the default value is VGEN. The output queue must be defined before the program runs **sysVar.printerAssociation**.

termID

This optional 4-byte item of type CHAR is ignored if specified.

For COBOL programs, EGL Server for iSeries provides support for `sysLib.startTransaction` by way of two command language (CL) programs:

CREATX

Acts as follows:

- Gets the current job number
- Sends the user data to the data queue VGCREATX
- Starts the job CREATXJOB to start the CL program CREATXPP

CREATXPP

Acts as follows:

- Uses the job number as the key to retrieve data from the data queue VGCREATX
- Calls the asynchronous CL program specified in the user record bytes 3 through 11

Related reference

"asynchLink element" on page 441

"sysVar.errorCode" on page 530

"sysVar.printerAssociation" on page 623

"transfer" on page 354

sysLib.verifyChkDigitMod10

The system function **sysLib.verifyChkDigitMod10** verifies a modulus-10 check digit in a character item that begins with a series of integers.

►►— sysLib.verifyChkDigitMod10 (*input*, *checkLength*, *result*); —►►

input

A character item that begins with a series of integers. The item include an additional position for the check digit, which is immediately to the right of the other integers.

checkLength

An item that contains the number of characters that you want to use from the *input* item, including the position used for the check digit. This item has 4 digits and is either of type SMALLINT or is of a type BIN, with no decimal places.

result

An item that receives one of two values:

- 0, if the calculated check digit matches the value in *input*
- 1, if the calculated check digit does not match that value

This item has 4 digits and is either of type SMALLINT or is of a type BIN, with no decimal places.

You can use **sysLib.verifyChkDigitMod10** in a function-invocation statement; or as an item validator in a text form.

Example: In the following example, *myInput* is an item of type CHAR and contains the value 1734284; *myLength* is an item of type SMALLINT and contains the value 7; and *myResult* is an item of type SMALLINT:

```
sysLib.verifyChkDigitMod10 (myInput, myLength, myResult);
```

An algorithm is used to derive the modulus-10 check digit, and in all cases the number at the check-digit position is not considered; but when the algorithm is complete, the calculated value is compared with the number at the check-digit position.

The algorithm is described in relation to the example values:

1. Multiply the units position of the input number by 2 and multiply every alternate position, moving right to left, by 2:

$$\begin{aligned}8 \times 2 &= 16 \\4 \times 2 &= 8 \\7 \times 2 &= 14\end{aligned}$$

2. Add the digits of the products (16814) to the input-number digits (132) that were not multiplied by 2:

$$1 + 6 + 8 + 1 + 4 + 1 + 3 + 2 = 26$$

3. To get the check digit, subtract the sum from the next-highest number ending in 0:

$$30 - 26 = 4$$

If the subtraction yields 10, the check digit is 0.

In this example, the calculated check digit matches the value in the check-digit position, and the value of `myResult` is 0.

Related reference

“Validation properties” on page 59

sysLib.verifyChkDigitMod11

The system function **sysLib.verifyChkDigitMod11** verifies a modulus-11 check digit in a character item that begins with a series of integers.

►►— `sysLib.verifyChkDigitMod11 (input, checkLength, result);` —►►

input

A character item that begins with a series of integers. The item include an additional position for the check digit, which is immediately to the right of the other integers.

checkLength

An item that contains the number of characters that you want to use from the *input* item, including the position used for the check digit. This item has 4 digits and is either of type SMALLINT or is of a type BIN, with no decimal places.

result

An item that receives one of two values:

- 0, if the calculated check digit matches the value in *input*
- 1, if the calculated check digit does not match that value

This item has 4 digits and is either of type SMALLINT or is of a type BIN, with no decimal places.

You can use **sysLib.verifyChkDigitMod11** in a function-invocation statement; or as an item validator in a text form.

Example: In the following example, `myInput` is an item of type CHAR and contains the value 56621869; `myLength` is an item of type SMALLINT and contains the value 8; and `myResult` is an item of type SMALLINT:

```
sysLib.verifyChkDigitMod11 (myInput, myLength, myResult);
```

An algorithm is used to derive the modulus-11 check digit, and in all cases the number at the check-digit position is not considered; but when the algorithm is complete, the calculated value is compared with the number at the check-digit position. The algorithm is described in relation to the example values:

1. Multiply the digit at the units position of the input number by 2, at the tens position by 3, at the hundreds position by 4, and so on, but let `myLength - 1` be the largest number used as a multiplier; and if more digits are in the input number, begin the sequence again using 2 as a multiplier:

```
6 x 2 = 12
8 x 3 = 24
1 x 4 = 4
2 x 5 = 10
6 x 6 = 36
6 x 7 = 42
5 x 2 = 10
```

2. Add the products of the first step and divide the sum by 11:

$$\begin{aligned}
 & (12 + 24 + 4 + 10 + 36 + 42 + 10) / 11 \\
 & = 138 / 11 \\
 & = 12 \text{ remainder } 6
 \end{aligned}$$

3. To get the check digit, subtract the remainder from 11 to get the self-checking digit:

$$11 - 6 = 5$$

If the remainder is 0 or 1, the check digit is 0.

In this example, the calculated check digit matches the value in the check-digit position, and the value of myResult is 0.

Related reference

“Validation properties” on page 59

sysLib.wait

The system function **sysLib.wait** suspends execution for the specified number of seconds.

►► sysLib.wait (*timeInSeconds*); ◀◀

timeInSeconds

The time can be any numeric item or literal. Fractions of a second down to hundredths of seconds are honored if the number is not an integer.

You can use **sysLib.wait** when two asynchronously running programs need to communicate through a record in a shared file or database. One program might need to suspend processing until the other program updates the information in the shared record.

Example

```
sysLib.wait(15); // waits for 15 seconds
```

Related reference

“System words in alphabetical order” on page 508

sysVar.arrayIndex

The system variable **sysVar.arrayIndex** contains a number:

- The number of the first element in an array that matches the search condition of a simple logical expression with an **in** operator, as shown in a later example.
- Zero, if no array element matches the search condition.
- The number of the last element modified in the target array after a **move ... for count** statement.

You can use **sysVar.arrayIndex** as any of these:

- As an array subscript to access the matching row or array element
- As the source or target in an assignment or **move** statement
- As the count value in a **move ... for count** statement
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of **sysVar.arrayIndex** are as follows:

Primitive type

BIN

Data length

4

Is value always restored after a converse?Only in a non-segmented text program; for details see *Segmentation***Example:** Assume that the record *myRecord* is based on the following part:

```
Record mySerialRecPart
  serialRecord:
    fileName = "myFile"
  end
  10 zipCodeArray CHA(9)[100];
  10 cityStateArray CHA(30)[100];
end
```

Furthermore, assume that the arrays are initialized with zip codes and city-and-state combinations.

The following code sets the variable *currentCityState* to the city and state that corresponds to the specified zip code:

```
currentZipCode = "27540";
if (currentZipCode in myRecord.zipCodeArray)
  currentCityState = myRecord.cityStateArray[sysVar.arrayIndex];
end
```

After the **if** statement, **sysVar.arrayIndex** contains the index of the first *zipCodeArray* element that contains the value of "27540". If "27540" is not found in *zipCodeArray*, the value of **sysVar.arrayIndex** is 0.

Related concepts

"Segmentation in text applications" on page 151

Related reference

"Arrays" on page 64

"in" on page 416

"Logical expressions" on page 358

sysVar.eventKey

The system variable **sysVar.eventKey** identifies the key that the user pressed to return a form to an EGL text program. The value is reset each time that the program runs the **converse** statement.

If the EGL code has no input form, the initial value of **sysVar.eventKey** is **ENTER**.

The following values are valid (whether uppercase, lowercase, or a combination):

- **ENTER**
- **BYPASS** (which refers to any of the keys that were specified as bypass keys for the form; or if none were specified for the form, any of the keys that were specified as bypass keys for the formGroup; or if none were specified for the formGroup, any of the keys that were specified as bypass keys for the program)
- **PA1** through **PA3**
- **PF1** through **PF24** (as also used for F1 through F24)
- **PAKEY** (for any PA key)
- **PFKEY** (for any PF or F key)

Note: PA keys are always treated as bypass keys.

You can use **sysVar.eventKey** as an operand in an **if** or **while** statement.

The characteristics of this system variable are as follows:

Primitive type

CHAR

Data length

1

Value saved across segments

No

sysVar.eventKey is not valid in a batch program.

Example: The comparison operator for **sysVar.eventKey** is either *is* or *not*, as in this example:

```
if (sysVar.eventKey IS PF3)
  exit program(0);
end
```

Related reference

“Logical expressions” on page 358

sysVar.printerAssociation

The system variable **sysVar.printerAssociation** allows you to specify, at run time, the output destination when you print a print form.

You can use this variable in any of these ways:

- As the source or target in an assignment or move statement
- As a comparison value in a logical expression
- As the value in a return statement

Characteristics of **sysVar.printerAssociation** are as follows:

Primitive type

CHAR

Data length

Varies by file type

Is value always restored after a converse?

Yes

sysVar.printerAssociation is initialized to the system resource name specified during generation or for debugging. If a program passes control to another program, the value of **sysVar.printerAssociation** is set to the default value for the receiving program.

Even when multiple print jobs are allowed for a given print form, the close statement closes only the file related to the current value of **sysVar.printerAssociation**.

Details specific to Java output: For Java output, you set **sysVar.printerAssociation** to a two-part string with an intervening colon:

jobID:destination

jobID A sequence of characters (without a colon) that uniquely identifies each print job. The characters are case sensitive (*job01* is different from *JOB01*), and you can reuse *jobID* after a print job closes.

You can use different jobs to promote a different kind of output or a different ordering of output, depending on the flow of events in your code. Consider the following sequence of EGL statements, for example:

```
sysVar.printerAssociation = "job1";  
print form1;  
sysVar.printerAssociation = "job2";  
print form2;  
sysVar.printerAssociation = "job1";  
print form3;
```

When the program ends, two print jobs are created:

- form1 followed by form3
- form2 alone

destination

The printer or file that receives the output.

The string *destination* is optional and is ignored if the print job is still open. The following statements apply if the string is absent:

- You can omit the colon that precedes *destination*
- In most cases, the program shows a print preview dialog from which the user can specify a printer or a file for output. The exception occurs if the curses library is used on UNIX; in that case, the print job goes to the default printer.

The following statements apply to the setting of *destination* when you are generating for Windows 2000/NT/XP:

- To send output to the default printer, do as follows--
 - Specify a value that matches the **fileName** property in the resource associations part.
 - Change the Java run-time properties so that *spool* (rather than *seqws*) is the value of the related file type. For example, in the resource associations part, if the value of the **fileName** property is *myFile* and the value of **systemName** is *printer*, you must change the settings of Java run-time properties so that `vgj.ra.myFile.fileType` is set to *spool* rather than *seqws*. After your change, the properties are as follows:

```
vgj.ra.myFile.systemName=printer  
vgj.ra.myFile.fileType=spool
```
- To send output to a file, specify a value that matches the **fileName** property in the resource associations part, when *seqws* is the value of the related **fileType** property in the resource associations part. The **systemName** property in the resource associations part contains the name of the operating-system file that receives the output.
- Do not specify the value *printer* as the value of *destination*. If you do, the print preview dialog is displayed to the user, but that behavior may change in later versions of EGL.

The following statements apply to the setting of *destination* when you are generating for UNIX:

- To send output to the default printer (regardless of whether the curses library is in use), specify a value that matches the **fileName** property in

the resource associations part, when *spool* is the value of the related **fileType** property in the resource associations part.

- To send output to a file, specify a value that matches the **fileName** property in the resource associations part, when *seqws* is the value of the related **fileType** property in the resource associations part. The **systemName** property in the resource associations part contains the name of the operating-system file that receives the output.
- Do not specify the value *printer* as the value of *destination*. If you do (and if the curses library is not in use), the print preview dialog is displayed to the user, but that behavior may change in later versions of EGL.

Details specific to COBOL output for iSeries: In relation to iSeries COBOL, set the system variable **sysVar.printerAssociation** to the value of a **fileName** property in the resource associations part that is used at generation time. The file type must be of type SEQ and not of type SPOOL.

Multiple print jobs are not supported for COBOL programs that are generated for iSeries, and when **sysVar.printerAssociation** is set, the EGL run time closes the old file (to complete the previous output of data); uses the iSeries command OVRPRTF to override the file name; and opens the new file.

Prior to its use, the value in **sysVar.printerAssociation** is folded to uppercase; but the value in the system variable itself remains unchanged. The value of **sysVar.printerAssociation** tests true when compared against a lowercase version if the system variable was initialized with a lowercase version.

The value set in **sysVar.printerAssociation** is propagated from the call level and changed to all the subordinate call levels. The value is not propagated, however if the program opened the file previously.

sysVar.remoteSystemID

The system variable **sysVar.remoteSystemID** contains the system name for the location of a remote program. This variable does not support dynamic definition of programs, but does support dynamic selection from a predefined set of locations.

sysVar.remoteSystemID is initialized to blanks and must be set before doing any call that requires use of this variable.

If you generate a COBOL program, any value in **sysVar.remoteSystemID** is folded to uppercase. Regardless of the target language, however, any comparison of **sysVar.remoteSystemID** and a character string is case-sensitive and is based on the value assigned to the variable. The comparison in the following code resolves to false, for example:

```
sysVar.remoteSystemID = "myWin";

// resolves to false
if (sysVar.remoteSystemID = "MYWIN")
  record1.resourceAssociation = "myCorp.txt";
end
```

You can use **sysVar.remoteSystemID** in most places where an item is allowed: as the target or source in an assignment statement, as a value passed to a system function, as an item in a logical expression, or as the argument in a return statement.

The characteristics of **sysVar.remoteSystemID** are as follows:

Primitive type

CHAR

Data length

8 (padded with blanks)

Value saved across segments?

Yes

Access of remote programs: The value of **sysVar.remoteSystemID** provides access of the remote program only if the linkage options part, callLink element, property **location** is set to PROGRAMCONTROLLED. For details on the meaning of **sysVar.remoteSystemID** for remote programs, see the description of *system name* in *location in callLink element*.

Target platforms:

Platform	Compatibility considerations
iSeries COBOL	Not supported

Example:

```
sysVar.remoteSystemID = "myWIN";

// resolves to true
if (sysVar.remoteSystemID = "myWIN")
    record1.resourceAssociation = "myCorp.txt";
end
```

Related concepts

"Linkage options part" on page 439

Related tasks

"Editing the asynchLink element of a linkage options part" on page 97

"Editing the callLink element of a linkage options part" on page 96

Related reference

"asynchLink element" on page 441

"location in callLink element" on page 449

"sysLib.startTransaction" on page 617

"transferToProgram element" on page 459

sysVar.segmentedMode

The system variable **sysVar.segmentedMode** is used in a text application to change the effect of the converse statement, but the variable is ignored for this purpose in called programs. For background information, see *Segmentation*.

Values of **sysVar.segmentedMode** are as follows:

- 1 The next **converse** statement runs in segmented mode.
- 0 The next **converse** statement runs in non-segmented mode.

The default value is 0 for non-segmented programs and 1 for segmented programs. The variable is reset to the default after the **converse** statement runs.

You can use this variable in any of these ways:

- As the source or destination in an assignment or move statement
- As the count value in a **move...for count** statement
- As a comparison value in a logical expression
- As the value in a return statement

Characteristics of **sysVar.segmentedMode** are as follows:

Primitive type

NUM

Data length

1

Is value restored after a converse?

No

Related concepts

"Segmentation in text applications" on page 151

Related reference

"System words in alphabetical order" on page 508

sysVar.sessionID

In Web applications, the system variable **sysVar.sessionID** contains an ID that is specific to the Web application server session. You can use the **sysVar.sessionID** value as a key value to access file or database information shared between programs.

Outside of Web applications, the following statements apply:

- The system variable **sysVar.sessionID** contains a system-dependent user identifier or terminal identifier for your program
- **sysVar.sessionID** is supported for this use only for compatibility with products that preceded EGL (specifically, for CSP releases prior to CSP 370AD Version 4 Release 1). It is recommended that you use **sysVar.userID** or **sysVar.terminalID** instead.

You can use **sysVar.sessionID** in these ways:

- As the source in an assignment or **move** statement
- As a variable in a logical expression
- As the argument in a **return** statement

The characteristics of **sysVar.sessionID** are as follows:

Primitive type

CHAR

Data length

8 (padded with blanks if the value has less than 8 characters)

Is value always restored after a converse?

Yes

sysVar.sessionID is initialized from the Java Virtual Machine system property *user.name*; and if the property cannot be retrieved, **sysVar.sessionID** is blank.

In relation to COBOL code for iSeries, **sysVar.sessionID** is the logon user ID and equivalent to the **sysVar.userID**.

Example:

```
myItem = sysVar.sessionID;
```

Related reference

“sysVar.terminalID” on page 629

“sysVar.userID” on page 630

sysVar.systemType

The system variable **sysVar.systemType** identifies the target system in which the program is running. If the generated output is a Java wrapper, **sysVar.systemType** is not available. Otherwise, the valid values are as follows:

aix For AIX

debug For the EGL Debugger

iseriesj

For iSeries Java programs

iseriesc

For iSeries COBOL programs

linux For Linux (on Intel-based hardware)

win For Windows 2000/NT/XP

You can use **sysVar.systemType** in these ways:

- As the source in an assignment or **move** statement
- As a variable in a logical expression
- As the argument in a **return** statement

The characteristics of **sysVar.systemType** are as follows:

Primitive type

CHAR

Data length

8 (padded with blanks)

Is value always restored after a converse?

Yes

Use **sysVar.systemType** instead of **sysLib.getVAGSysType**.

Definition considerations: The value of **sysVar.systemType** does not affect what code is validated at generation time. For example, the following **add** statement is validated even if you are generating for Windows:

```
if (sysVar.systemType IS AIX)
  add myRecord;
end
```

To avoid validating code that will never run in the target system, take either of the following actions:

- Set the build descriptor option **EliminateSystemDependentCode** to YES. In the current example, the **add** statement is not validated if you set that build descriptor option to YES. Be aware, however, that the generator can eliminate

system-dependent code only if the logical expression (in this case, `sysVar.systemType IS AIX`) is simple enough to evaluate at generation time.

- Alternatively, move the statements that you do not want to validate to a second program; then, let the original program call the new program conditionally:

```
if (sysVar.systemType IS AIX)
  call myAddProgram myRecord;
end
```

Example:

```
if (sysVar.systemType is WIN)
  call myAddProgram myRecord;
end
```

Related reference

“eliminateSystemDependentCode” on page 249

“sysLib.getVAGSysType” on page 613

sysVar.terminalID

In relation to COBOL code on iSeries, **sysVar.terminalID** is initialized to blanks; and if the code is interactive, the variable is reset to the terminal device name received from a query of the attributes of the active job.

In relation to Java code, **sysVar.terminalID** (like **sysVar.sessionID**) is initialized from the Java Virtual Machine system property *user.name*, and if the property cannot be retrieved, **sysVar.terminalID** is blank.

You can use **sysVar.terminalID** in these ways:

- As the source in an assignment or **move** statement
- As a variable in a logical expression
- As the argument in a **return** statement

The characteristics of **sysVar.terminalID** are as follows:

Primitive type

CHAR

Data length

10 for iSeries COBOL, otherwise 8, and padded with blanks if the value has less than the maximum number of characters

Is value always restored after a converse?

Yes

Example:

```
myItem10 = sysVar.terminalID;
```

sysVar.transactionID

The variable is not used; but if the program was invoked by a **transfer** statement of the form *transfer to program*, the variable contains the name of the transferring program.

You can use this variable in any of these ways:

- As the source or destination in an assignment or move statement
- As a comparison value in a logical expression
- As the value in a return statement

Characteristics of **sysVar.transactionID** are as follows:

Primitive type
CHAR

Data length
8

Is value always restored after a converse?
Yes

Related concepts
“Segmentation in text applications” on page 151

Related reference

sysVar.transferName

The system variable **sysVar.transferName** allows you to specify, at run time, the name of the program or transaction to which you want to transfer.

You can use this variable in any of these ways:

- As the source or target in an assignment or move statement
- As a program or transaction name in a transfer statement
- As a comparison value in a logical expression
- As the value in a return statement

Characteristics of **sysVar.transferName** are as follows:

Primitive type
CHAR

Data length
8

Is value always restored after a converse?
Yes

Related reference
“System words in alphabetical order” on page 508
“transfer” on page 354

sysVar.userID

The system variable **sysVar.userID** contains a user identifier in environments where one is available.

You can use **sysVar.userID** in these ways:

- As the source in an assignment or **move** statement
- As a variable in a logical expression
- As the argument in a **return** statement

The characteristics of **sysVar.userID** are as follows:

Primitive type
CHAR

Data length
8 (padded with blanks if the value has less than 8 characters)

Is value always restored after a converse?

Yes

sysVar.userID is initialized from the Java Virtual Machine system property *user.name*; and if the property cannot be retrieved, **sysVar.userID** is blank.

In relation to COBOL code for iSeries, **sysVar.userID** contains the user ID specified at sign-on.

Example:

```
myItem = sysVar.userID;
```

Use declaration

This section describes the use declaration, followed by details on how to write the declaration:

- “In a program or library part” on page 632
- “In a formGroup part” on page 633
- “In a pageHandler part” on page 634

Background

The use declaration allows you to easily reference data areas and functions in parts that are separately generated. A program, for instance, can issue a use declaration that allows for easy reference to a data table, library, or form group, but only if those parts are visible to the program part. For details on visibility, see *References to parts*.

In most cases, you can reference data areas and functions from another part regardless of whether a use declaration is in effect. For example, if you are writing a program and do not have a use declaration for a library part called *myLib*, you can access the library variable called *myVar* as follows:

```
myLib.myVar
```

If you include the library name in a use declaration, however, you can reference the variable as follows:

```
myVar
```

The previous, short form of the reference is valid only if the symbol *myVar* is unique for every variable and structure item that is global to the program. (If the symbol is not unique, an error occurs.) Also, the symbol *myVar* refers to an item in the library only if a local variable or parameter does not have the same name. (A local data area takes precedence over a same-named, program-global data area.)

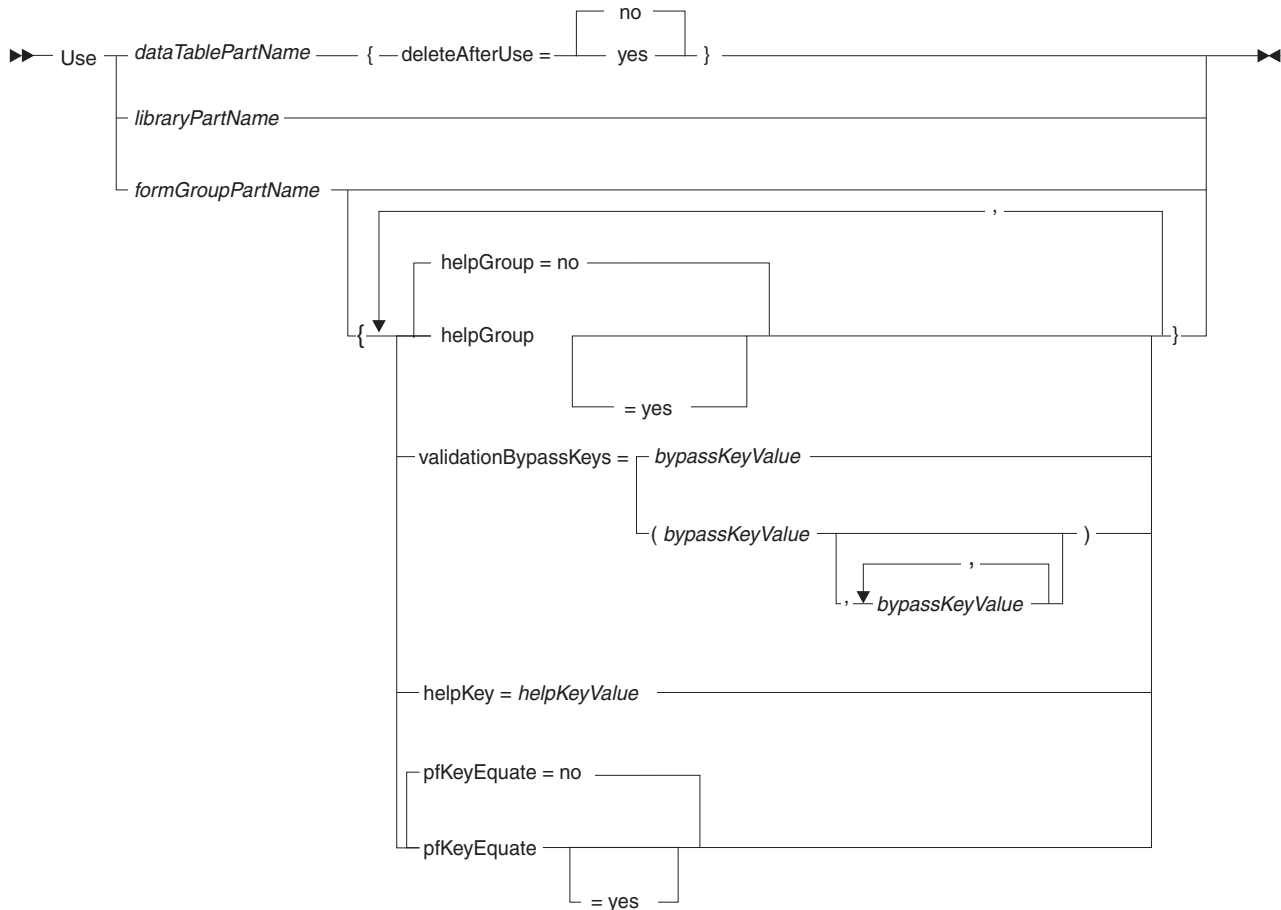
A use declaration is required in these situations:

- A program or library that uses any of the forms in a given formGroup part must have a use declaration for that formGroup part
- A formGroup part must have a use declaration for a form that is required by the program or library but is not embedded in the formGroup part

Each name specified in the use declaration may be qualified by a package name, library name, or both.

In a program or library part

Each use declaration in a program or library must be external to any function. The syntax for the declaration is as follows:



dataTablePartName

Name of a dataTable part that is visible to the program or library.

A reference in a use declaration is unnecessary for a dataTable part that is referenced in the program property **msgTablePrefix**.

You cannot override properties of a dataTable part in the use declaration.

For an overview of dataTable parts, see *DataTable part*.

libraryPartName

Name of a library part that is visible to the program or library.

You cannot override properties of the library part in the use declaration.

For an overview of library parts, see *Library part*.

formGroupName

Name of a formGroup part that is visible to the program or library. For an overview of form groups, see *FormGroup part*.

A program that uses any of the forms in a given formGroup part must have a use declaration for that formGroup part.

No overrides occur for form-level properties. If a property like **validationBypassKeys** is specified in a form, for example, the value in the

form is in effect at run time. If a form-level property is not specified in the form, however, the situation is as follows:

- EGL run time uses the value in the program's use declaration
- If no value is specified in the program's use declaration, EGL run time uses the value (if any) in the form group

The properties that follow let you change behaviors when a form group is accessed by a specific program.

helpGroup = yes, helpGroup = no

Specifies whether to use the formGroup part as a help group. The default is *no*.

validationBypassKeys = bypassKeyValue

Identifies a user keystroke that causes the EGL run time to skip input-field validations. This property is useful for reserving a keystroke that ends the program quickly. Each *bypassKeyValue* option is as follows:

pf*n*

The name of an F or PF key, including a number between 1 and 24, inclusive.

Note: Function keys on a PC keyboard are often *f* keys such as f1, but EGL uses the IBM *pf* terminology so that (for example) f1 is called pf1.

helpKey = helpKeyValue

Identifies a user keystroke that causes the EGL run time to present a help form to the user. The *helpKeyValue* option is as follows:

pf*n*

The name of an F or PF key, including a number between 1 and 24, inclusive.

Note: Function keys on a PC keyboard are often *f* keys such as f1, but EGL uses the IBM *pf* terminology so that (for example) f1 is called pf1.

pfKeyEquate = yes, pfKeyEquate = no

Specifies whether the keystroke that is registered when the user presses a high-numbered function key (PF13 through PF24) is the same as the keystroke that is registered when the user presses a function key that is lower by 12. For details, see *pfKeyEquate*.

In a formGroup part

In a formGroup part, a use declaration refers to a form that is specified outside the form group. This kind of declaration allows multiple form groups to share the same form.

The syntax for a use declaration in a formGroup part is as follows:

►► use — *formPartName* ; ◀◀

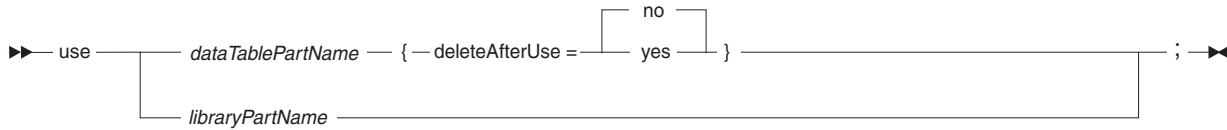
formPartName

Name of a form part that is visible to the form group. For an overview of forms, see *Form part*.

You cannot override properties of a form part in the use declaration of a formGroup part.

In a pageHandler part

Each use declaration in a pageHandler part must be external to any function. The syntax for the declaration is as follows:



dataTablePartName

Name of a dataTable part that is visible to the pageHandler part.

You cannot override properties of a dataTable part in the use declaration.

For an overview of dataTable parts, see *DataTable part*.

libraryPartName

Name of a library part that is visible to the pageHandler part.

You cannot override properties of the library part in the use declaration.

For an overview of library parts, see *Library part*.

Related concepts

“DataTable part” on page 285

“FormGroup part” on page 376

“Form part” on page 366

“Library part” on page 433

“References to parts” on page 16

Related reference

“pfKeyEquate” on page 380

EGL Java run-time error codes

At run time, EGL places error codes in the system variable `sysVar.errorCode` in the following cases:

- A failure occurs during a remote call, an EJB call, a commit, or a rollback. In those cases, the message identifier begins with CSO.
- An error occurs in a Web application. In a subset of those cases, the message identifier begins with EGL.
- An error occurs during a local call, during access of a file or database, or during execution of one the following system functions--
 - Math functions
 - String functions
 - `sysLib.convert`

In those cases, the message identifier begins with VGJ.

- An error occurs in a Java access function. In that case, the message includes only numbers.

The error codes that are assigned by the Java access functions are shown in the next table. The other error codes are shown in the next sections.

Value in <code>sysVar.errorCode</code>	Description
00001000	An exception was thrown by an invoked method or as a result of a class initialization.
00001001	The object was null, or the specified identifier was not in the namespace.
00001002	A public method, field, or class with the specified name does not exist or cannot be loaded.
00001003	The EGL primitive type does not match the type expected in Java.
00001004	The method returned null, the method does not return a value, or the value of a field was null.
00001005	The returned value does not match the type of the return item.
00001006	The class of an argument cast to null could not be loaded.
00001007	A <code>SecurityException</code> or <code>IllegalAccessException</code> was thrown during an attempt to get information about a method or field, or an attempt was made to set the value of a field that was declared final.
00001008	The constructor cannot be called; the class name refers to an interface or abstract class.
00001009	An identifier rather than a class name must be specified; the method or field is not static.

Related reference

"I/O error values" on page 418

"Java access (system words)" on page 557

"`sysVar.errorCode`" on page 530

EGL Java run-time error code CSO7000E

CSO7000E: An entry for the specified called program %1 cannot be found in the linkage properties file %2.

Explanation

The message occurs in this situation:

- When the calling program was generated, property **remoteBind** was set to **RUNTIME** in the linkage options part, in the **callLink** element for the called program; and
- An entry for the specified called program cannot be found at run time, in the linkage properties file. The reason may be one of the following:
 - The linkage properties file cannot be found.
 - The file was found, but an entry for the called program is not in that file.
 - An incorrect linkage properties file was specified.

User Response

Do as follows:

- If the program is being called from a Java wrapper, the linkage properties file must be named *link.properties*, where *link* is the name of the linkage options part used at generation. Make sure the file exists, has an entry for the called program, and is in a directory or archive specified in the CLASSPATH variable.
- If the program is being called from a program running in the J2EE environment, the linkage properties file can be identified by the `cso.linkageOptions.link` environment variable in the deployment descriptor, where *link* is the name of the linkage options part used at generation. If the environment variable is not set, the linkage properties file must be named *link.properties*, where *link* is the name of the linkage options part used at generation. Make sure the file exists, has an entry for the called program, and is in a directory or archive specified in CLASSPATH.
- If the program is being called from a program not running in the J2EE environment, the situation is as follows:
 - The linkage properties file can be identified by the `cso.linkageOptions.link` property, where *link* is the name of the linkage options part used at generation. If the property is not set, the linkage properties file may be named *link.properties*, where *link* is the name of the linkage options part used at generation. In these two cases, make sure the file exists, has an entry for the called program, and is in a directory or archive specified in CLASSPATH.
 - If the linkage properties file cannot be found, the linkage properties must be in the program properties file; in that case, make sure the program properties file includes an entry for the called program and that the program properties file is in a directory or archive specified in CLASSPATH.

For other details, see the EGL help pages on the **callLink** element, on Java run-time properties, and on setting up the environment.

If the problem persists, do as follows:

1. Record the message number and the message text.

Note: The error message includes the following important information:

- Where the error occurred

- The type of internal error
2. Record the situation in which this message occurs.
 3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code CSO7015E

CSO7015E: Cannot open the linkage properties file %1.

Explanation

The linkage properties file cannot be opened because the file is locked or cannot be found.

User Response

Make sure that the linkage properties file is not locked by another process and that the file resides in a directory or archive specified in your CLASSPATH.

EGL Java run-time error code CSO7016E

CSO7016E: The properties file csouidpwd.properties could not be read. Error: %1

Explanation

The file was found but there was an error reading from it.

User Response

Use the Error portion of the message to diagnose and correct the problem.

EGL Java run-time error code CSO7020E

CSO7020E: The conversion table %1 is not valid.

Explanation

A conversion table that handles bidirectional text is invalid or cannot be loaded.

User Response

The conversion table must reside in a directory or archive specified in the CLASSPATH. For details on developing the conversion table, see the help page on bidirectional text.

EGL Java run-time error code CSO7021E

CSO7021E: The client text attribute tag %2 in conversion table %1 is not valid.

Explanation

The conversion table file is not valid.

User Response

Correct the file and run the program again.

EGL Java run-time error code CSO7022E

CSO7022E: The server text attribute tag %2 in conversion table %1 is not valid.

Explanation

The conversion table file is not valid.

User Response

Correct the file and run the program again.

EGL Java run-time error code CSO7023E

CSO7023E: The value %3 for Arabic option tag %2 in conversion table %1 is not valid.

Explanation

The conversion table file is not valid.

User Response

Correct the file and run the program again.

EGL Java run-time error code CSO7024E

CSO7024E: The value %3 for Wordbreak option tag %2 in conversion table %1 is not valid.

Explanation

The conversion table file is not valid.

User Response

Correct the file and run the program again.

EGL Java run-time error code CSO7026E

CSO7026E: The value %3 for Roundtrip option tag %2 in conversion table %1 is not valid.

Explanation

The conversion table file is not valid.

User Response

Correct the file and run the program again.

EGL Java run-time error code CSO7045E

CSO7045E: Error obtaining the address of entry point %1 within the shared library %2. RC = %3.

Explanation

An error was encountered in obtaining the address of the entry point within the shared library.

User Response

Make sure that the referenced shared library is the correct shared library to be loaded. If so, make sure the shared library is built correctly.

EGL Java run-time error code CSO7050E

CSO7050E: An error occurred in remote program %1, date %2, time %3

Explanation

An error occurred in a called program, and the program stopped running.

User Response

Use the date and time stamp on this message to associate the message with any diagnostic messages logged at the remote location. Check those diagnostic messages for further details.

EGL Java run-time error code CSO7060E

CSO7060E: An error was encountered while loading the shared library %1. The return code is %2.

Explanation

An error was encountered while loading the shared library.

User Response

Make sure the shared library resides in a directory specified in your PATH or LIBPATH environment variable. Make sure that the shared library is built correctly.

EGL Java run-time error code CSO7080E

CSO7080E: The specified protocol %1 is not valid.

Explanation

The specified protocol in the linkage is unrecognized.

User Response

Consult the documentation and specify a valid protocol.

EGL Java run-time error code CSO7160E

CSO7160E: An error occurred in remote program %1, date %2, time %3, on system %4.

Explanation

The Java program that you are running calls a remote program on the specified system, which failed in execution at the date and time specified.

User Response

Check the remote server log for a more detailed description in problem analysis.

EGL Java run-time error code CSO7161E

CSO7161E: Run unit ended due to an application error on system %1 trying to call program %2. %3

Explanation

An error occurred at the remote server that causes the remote run unit to terminate abnormally when executing the remote program. Diagnostic messages preceding this message in the server job log explain the nature of error. If available, additional information may be included with the message text.

User Response

Check the error messages logged on the server system to determine what to do to fix the original problem.

EGL Java run-time error code CSO7162E

CSO7162E: Invalid password or user ID supplied for connecting to system %1. Java exception message received: %2.

Explanation

The password or user ID supplied to connect to the remote system is not set or not valid.

User Response

Verify that the connection is set. Verify that the user ID and password supplied to the remote system are correct, and try again.

EGL Java run-time error code CSO7163E

CSO7163E: Remote access security error to system %1 for user %2. Java exception message received: %3

Explanation

The specified user currently connecting to the system does not have sufficient authority or does not have access to the remote resource on the specified system.

User Response

Verify that the user connecting to the remote machine has the proper authority to connect to the remote machine and to execute the remote server program.

EGL Java run-time error code CSO7164E

CSO7164E: Remote connection error to system %1. Java exception message received: %2

Explanation

An error occurred when communicating or connecting to the remote system.

User Response

Check that the remote server is available; then retry. If this does not work, contact the remote host's system administrator to determine the actual problem.

EGL Java run-time error code CSO7165E

CSO7165E: Commit failed on system %1. %2

Explanation

A commit operation failed on the remote system.

User Response

Diagnose the problem by reviewing the detailed message, which is shown here as %2.

EGL Java run-time error code CSO7166E

CSO7166E: Rollback failed on system %1. %2

Explanation

A rollback operation failed on the remote system.

User Response

Diagnose the problem by reviewing the detailed message, which is shown here as %2.

EGL Java run-time error code CSO7360E

CSO7360E: AS400Toolbox execution error: %1, %2 while calling program %3 on system %4

Explanation

The Java program or applet that you are running uses the Java400 protocol to call a remote server program. An unexpected exception was caught while attempting to call the server program. The message text consists of the name of the AS400 Toolbox exception followed by the message returned with the exception.

User Response

Use the AS400 Toolbox error message provided to analyze the cause of the problem.

EGL Java run-time error code CSO7361E

CSO7361E: EGL OS/400® Host Services error. Required files not found on system %1.

Explanation

The Java program or applet that you are running uses the Java400 protocol to call a remote server program. An exception is raised when the remote catcher is not found or is not in the proper library on the server.

User Response

Check that EGL OS/400 Host Services is properly installed on the remote system. Apply the latest PTFs if available.

EGL Java run-time error code CSO7488E

CSO7488E: Unknown TCP/IP hostname: %1

Explanation

An UnknownHostException was thrown during an attempt to connect to the remote TCP/IP listener program.

User Response

Do as follows:

- Add the property `cso.serverLinkage.xxx.location` to the run-time linkage properties file, where `xxx` is the name of the called program or is an application name, as described in the EGL reference-type help page on the linkage properties file. The value of the property is a valid TCP/IP host name.
- Alternatively, set the TCP/IP host name at generation time and regenerate the program:
 - In the linkage options part, in the `callLink` element for the called program, set property **location** to the TCP/IP host name
 - If you wish to finalize linkage options only at run time, set property **remoteBind** to `RUNTIME` and generate with build descriptor option **genProperties** set to `YES`

For other details, see the EGL help pages on the `callLink` element, on the linkage properties file, and on setting up the environment.

EGL Java run-time error code CSO7489E

CSO7489E: The linkage information used to call the program is inconsistent or missing.

Explanation

The program was unable to determine how the program should be called.

User Response

Supply all required linkage information. The information that is required depends on the desired type of call. Refer to the help pages on the linkage options part, particularly on the callLink element.

EGL Java run-time error code CSO7610E

CSO7610E: An error was encountered while calling CICS ECI to commit a unit of work. The CICS return code is %1.

Explanation

A commit request was issued by the client but was not successful. An error was encountered while calling CICS External Call Interface to commit a logical unit of work.

User Response

Please refer to the appropriate CICS documentation for the corrective actions for the specified error.

EGL Java run-time error code CSO7620E

CSO7620E: An error was encountered while calling the CICS ECI to rollback a unit of work. The CICS return code is %1.

Explanation

A rollback request was issued by the client but was not successful. An error was encountered while calling CICS External Call Interface to rollback a logical unit of work.

User Response

Please refer to the appropriate CICS documentation for the corrective actions for the specified error.

EGL Java run-time error code CSO7630E

CSO7630E: An error was encountered while ending the remote procedure call to a CICS server. The CICS return code is %1.

Explanation

An attempt was made to commit all open logical units of work before ending the EGL remote procedure call to a CICS server but was not successful. This request was made via the CICS External Call Interface.

User Response

Refer to the appropriate CICS documentation for the corrective actions for the specified error.

EGL Java run-time error code CSO7640E

CSO7640E: %1 is an invalid value for the ctgport entry.

Explanation

The value of ctgport must be an integer.

User Response

Use the correct ctgport number.

EGL Java run-time error code CSO7650E

CSO7650E: An error was encountered calling program %1 using the CICS ECI. Return code: %2. CICS system identifier: %3.

Explanation

An error was returned from a CICS External Call Interface (ECI) function call when attempting to call a remote server program.

The system identifier is the name of the CICS system where the server program was to run. If blank, the system is specified in the CICS program definition for the program or in the CICS client initialization file. The return code is the CICS return code.

User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code or if the return code is not documented above, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faaecih.h or cics_eci.h.

EGL Java run-time error code CSO7651E

CSO7651E: An error was encountered calling program %1 using the CICS ECI. Return code: -3 (ECI_ERR_NO_CICS). CICS system identifier: %2.

Explanation

An error was returned from a CICS External Call Interface (ECI) function call when attempting to call a remote server program.

The system identifier is the name of the CICS system where the server program was to run. If blank, the system is specified in the CICS program definition for the program or in the CICS client initialization file. The return code is the CICS return code.

The CICS return code has the following meaning:

- -3 - ECI_ERR_NO_CICS
Client or server system not available

User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code or if the return code is not documented above, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faaecih.h or cics_eci.h.

EGL Java run-time error code CSO7652E

CSO7652E: An error was encountered calling program %1 using the CICS ECI. Return code: -4 (ECI_ERR_CICS_DIED). CICS system identifier: %2.

Explanation

An error was returned from a CICS External Call Interface (ECI) function call when attempting to call a remote server program.

The system identifier is the name of the CICS system where the server program was to run. If blank, the system is specified in the CICS program definition for the program or in the CICS client initialization file. The return code is the CICS return code.

The CICS return code has the following meaning:

- -4 - ECI_ERR_CICS_DIED
Server system no longer available

User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code or if the return code is not documented above, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faaecih.h or cics_eci.h.

EGL Java run-time error code CSO7653E

CSO7653E: An error was encountered calling program %1 using the CICS ECI. Return code: -6 (ECI_ERR_RESPONSE_TIMEOUT). CICS system identifier: %2.

Explanation

An error was returned from a CICS External Call Interface (ECI) function call when attempting to call a remote server program.

The system identifier is the name of the CICS system where the server program was to run. If blank, the system is specified in the CICS program definition for the program or in the CICS client initialization file. The return code is the CICS return code.

The CICS return code has the following meaning:

- -6 - ECI_ERR_RESPONSE_TIMEOUT
Response time out. Time limit is specified in environment variable CSOTIMEOUT.

User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code or if the return code is not documented above, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faaecih.h or cics_eci.h.

EGL Java run-time error code CSO7654E

CSO7654E: An error was encountered calling program %1 using the CICS ECI. Return code: -7 (ECI_ERR_TRANSACTION_ABEND). CICS system identifier: %2. Abend code: %3.

Explanation

An error was returned from a CICS External Call Interface (ECI) function call when attempting to call a remote server program.

The system identifier is the name of the CICS system where the server program was to run. If blank, the system is specified in the CICS program definition for the program or in the CICS client initialization file. The return code is the CICS return code.

The CICS return code has the following meaning:

- -7 - ECI_ERR_TRANSACTION_ABEND
Abnormal termination on server. Common ABEND codes are:
 - AEI0 - Server program not defined
 - AEI1 - Server transaction not defined

User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code or if the return code is not documented above, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faecih.h or cics_eci.h.

EGL Java run-time error code CSO7655E

CSO7655E: An error was encountered calling program %1 using the CICS ECI. Return code: -22 (ECI_ERR_UNKNOWN_SERVER). CICS system identifier: %2.

Explanation

An error was returned from a CICS External Call Interface (ECI) function call when attempting to call a remote server program.

The system identifier is the name of the CICS system where the server program was to run. If blank, the system is specified in the CICS program definition for the program or in the CICS client initialization file. The return code is the CICS return code.

The CICS return code has the following meaning:

- -22 - ECI_ERR_UNKNOWN_SERVER
Server system not defined

User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code or if the return code is not documented above, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faecih.h or cics_eci.h.

EGL Java run-time error code CSO7656E

CSO7656E: An error was encountered calling program %1 using the CICS ECI. Return code: -27 (ECI_ERR_SECURITY_ERROR). CICS system identifier: %2.

Explanation

An error was returned from a CICS External Call Interface (ECI) function call when attempting to call a remote server program.

The system identifier is the name of the CICS system where the server program was to run. If blank, the system is specified in the CICS program definition for the program or in the CICS client initialization file. The return code is the CICS return code.

The CICS return code has the following meaning:

- -27 - ECI_ERR_SECURITY_ERROR
User ID or password not valid

User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code or if the return code is not documented above, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faaecih.h or cics_eci.h.

EGL Java run-time error code CSO7657E

CSO7657E: An error was encountered calling program %1 using the CICS ECI. Return code: -28 (ECI_ERR_MAX_SYSTEMS). CICS system identifier: %2.

Explanation

An error was returned from a CICS External Call Interface (ECI) function call when attempting to call a remote server program.

The system identifier is the name of the CICS system where the server program was to run. If blank, the system is specified in the CICS program definition for the program or in the CICS client initialization file. The return code is the CICS return code.

The CICS return code has the following meaning:

- -28 - ECI_ERR_MAX_SYSTEMS
Maximum number of servers reached

User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code or if the return code is not documented above, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faaecih.h or cics_eci.h.

EGL Java run-time error code CSO7658E

CSO7658E: An error was encountered calling program %1 on system %2 for user %3. CICS ECI call returned RC %4 and Abend Code %5.

Explanation

A non-zero return code was returned on a CICS ECI call made from the gateway to the specified system on behalf of the user identified in the message.

User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faaecih.h or cics_eci.h.

EGL Java run-time error code CSO7659E

CSO7659E: An exception occurred on the flow of an ECI Request to CICS system %1. Exception: %2

Explanation

An unexpected exception occurred in the flow method when attempting to send the ECI Request from the gateway to the CICS system identified in the message.

User Response

Examine the exception string that was returned. If you are unable to determine the cause of the problem from the exception, please contact IBM Support for assistance.

EGL Java run-time error code CSO7669E

CSO7669E: An error was encountered when connecting to CTG. CTG Location: %1, CTG Port: %2. Exception: %3

Explanation

An unexpected exception occurred when connecting to the CICS Transaction Gateway.

User Response

Examine the exception string that was returned. If you are unable to determine the cause of the problem from the exception, please contact IBM Support for assistance.

EGL Java run-time error code CSO7670E

CSO7670E: An error was encountered when disconnecting from CTG. CTG Location: %1, CTG Port: %2. Exception: %3

Explanation

An unexpected exception occurred when disconnecting from the CICS Transaction Gateway.

User Response

Examine the exception string that was returned. If you are unable to determine the cause of the problem from the exception, please contact IBM Support for assistance.

EGL Java run-time error code CSO7671E

CSO7671E: When using CICSSSL protocol, both ctgKeyStore and ctgKeyStorePassword must be specified.

Explanation

Required values were not specified so the call cannot be completed.

User Response

Make sure that both ctgKeyStore and ctgKeyStorePassword are specified.

EGL Java run-time error code CSO7816E

CSO7816E: A socket exception occurred when the gateway attempted to connect to server with hostname %1 and port %2 for userid %4. Exception was: %3

Explanation

The socket call to create and connect a socket from the gateway to the server system identified in the message failed with the exception shown.

The EGL gateway attempted a socket call to create and connect a TCP/IP socket for a server call. The socket call failed with the exception indicated in the message.

User Response

Examine the exception information to determine a reason why a socket call from the gateway failed. If you are unable to determine the cause of the problem by examining the exception information, please contact IBM Support for assistance.

EGL Java run-time error code CSO7819E

CSO7819E: An unexpected exception occurred on function %2. Exception: %1

Explanation

The EGL gateway received an unexpected exception from the function identified in the message. An internal error may have occurred.

User Response

If you are unable to determine the source of the problem from examining the exception information, please contact IBM Support for assistance.

EGL Java run-time error code CSO7831E

CSO7831E: The client's buffer was too small for the amount of data being passed on the call. Ensure that the cumulative size of the parameters being passed does not exceed the maximum allowed which is 32567 bytes.

Explanation

The buffer established by the client cannot be made as large as the cumulative size of the parameters being passed to the remote called program.

User Response

Ensure that the cumulative size of the parameters being passed does not exceed the maximum allowed which is 32567 bytes. If they do not exceed the maximum and this error occurs, please report the error to IBM Support Center.

EGL Java run-time error code CSO7836E

CSO7836E: The client has received notification that the server is unable to start the remote called program. Reason code: %1.

Explanation

The server is unable to run the remote called program and has returned a reason code for problem determination.

User Response

Reason codes are as follows:

- 2 - Server was unable to load the class for the called program. The server trace file may show more specific information. Make sure that the class is available to the server.

This problem may result from improper conversion of the class name passed to the server. Review the help page on data conversion to verify that the correct conversion table was specified in the linkage options part, in the callLink element for the called program, in property conversionTable.

- 3 - The called program was ended because of an error. The server trace file may show more specific information.

For any reason code not listed above or if you are unable to determine the cause of the failure, contact IBM support.

EGL Java run-time error code CSO7840E

CSO7840E: The client received notification from the server that the remote called program failed with return code %1.

Explanation

The remote called program ran but ended with a non-zero return code. The problem is in the program rather than in communications.

User Response

Examine or trace the called program to determine why it completed with a non-zero return code.

EGL Java run-time error code CSO7885E

CSO7885E: A TCP/IP read function failed on a call for userid %2 to hostname %1. Exception returned was: %3

Explanation

The EGL gateway received an exception when attempting a TCP/IP read function.

User Response

Examine the exception information returned in order to determine the cause of the problem. If you are unable to determine why the failure occurred, please contact IBM Support for assistance.

EGL Java run-time error code CSO7886E

CSO7886E: A TCP/IP write function failed on a call for userid %2 to hostname %1. Exception returned was: %3

Explanation

The EGL gateway received an exception when attempting a TCP/IP write function.

User Response

Examine the exception information returned in order to determine the cause of the problem. If you are unable to determine why the failure occurred, please contact IBM Support for assistance.

EGL Java run-time error code CSO7955E

CSO7955E: %1, %2

Explanation

An unexpected Java exception was caught.

The message text shows the name of the Java exception followed by the Java message that was thrown with the exception.

User Response

Review the message and respond as appropriate.

EGL Java run-time error code CSO7957E

CSO7957E: Conversion table name %1 is not valid for Java data conversion.

Explanation

You are using a generated Java class to call a program and have incorrectly specified a conversion table to convert Java data to the format used by the called program.

User Response

Review the help page on data conversion to determine the conversion table name, which you specify in the linkage options part, in the callLink element for the called program, in property conversionTable.

EGL Java run-time error code CSO7958E

CSO7958E: The native code did not provide an object of type CSOPowerServer to the Java wrapper, as is needed to convert data between the Java wrapper and the EGL-generated program.

Explanation

The native Java code invoked the call or execute method of a Java wrapper without first instantiating an object of class CSOPowerServer and providing that object to the wrapper.

User Response

Review the help pages on the Java wrapper for details on accessing EGL middleware, as is always required for data conversion.

EGL Java run-time error code CSO7966E

CSO7966E: The code page encoding %1 was not found for the conversion table %2.

Explanation

The conversion table specified in the linkage options requires an encoding not available in the Java Virtual Machine (JVM) being used.

User Response

Review the help page on data conversion to determine the correct conversion table name, which you specify in the linkage options part, in the callLink element for the called program, in property conversionTable. If you specified the correct conversion table, make sure that the JVM that you are using is supported by the Java run-time environment of EGL.

If the previous steps do not reveal the problem, consider whether the installation of your JVM is flawed or whether your Java Virtual machine does not support all encodings. In these cases, refer to the documentation of your JVM vendor or contact the JVM vendor for assistance.

If you encountered the error when running an applet client in a browser, the error occurred at the PowerServer SessionManager used by the client applet. In this case, refer to the documentation for the JVM that the SessionManager is running on or contact the JVM vendor.

EGL Java run-time error code CSO7968E

CSO7968E: Host %1 is not known or could not be found.

Explanation

No remote system specified in the linkage.

User Response

The remote system must be specified in the linkage part's location field.

EGL Java run-time error code CSO7970E

CSO7970E: Could not load the required EGL shared library %1, reason: %2

Explanation

The shared library for is required to complete the operation, but it could not be loaded.

User Response

Make sure that the shared library is on the system. It must be included in the environment variable that specifies the shared library path, PATH or LIBPATH.

EGL Java run-time error code CSO7975E

CSO7975E: The properties file %1 could not be opened.

Explanation

The properties file required by the program could not be opened. The name of the properties file may be specified on the command line when the program is started. If no name is given when the program is started, the following name is used by default:

`tcpiplistener.properties`

Either the properties file does not exist, or it exists but could not be opened.

User Response

Ensure that the properties file exists and that the program has the proper permissions to read it, then run the program again.

EGL Java run-time error code CSO7976E

CSO7976E: The trace file %1 could not be opened. The exception is %2 The message is as follows: %3

Explanation

An exception occurred when the program tried to open the trace output file.

User Response

Correct the problem and re-run the program.

EGL Java run-time error code CSO7977E

CSO7977E: The program properties file does not contain a valid setting for the %1 property, which is required.

Explanation

The property is not defined in the program properties file.

User Response

Add the property to the program properties file and re-run the program. For details, see the help page on Java run-time properties.

EGL Java run-time error code CSO7978E

CSO7978E: An unexpected exception occurred. The exception is %1 The message is as follows: %2

Explanation

The program encountered an error.

User Response

Correct the problem and re-run the program.

EGL Java run-time error code CSO7979E

CSO7979E: Unable to create an InitialContext. Exception is %1

Explanation

The exception was thrown from the constructor of javax.naming.InitialContext. The program needs to create the InitialContext object to access the J2EE environment settings.

User Response

Use the text of the exception and the documentation of your J2EE environment to correct the problem.

EGL Java run-time error code CSO8000E

CSO8000E: The password entered to the Gateway has expired. %1

Explanation

The EGL GatewayServlet received an expired password exception when attempting to authenticate the user with the provided password.

User Response

Examine the exception information returned in order to determine the cause of the problem. Correct the problem by providing a new password.

EGL Java run-time error code CSO8001E

CSO8001E: The password entered to the Gateway is not valid. %1

Explanation

The EGL GatewayServlet received an invalid password exception when attempting to authenticate the user with the provided password.

User Response

Examine the exception information returned in order to determine the cause of the problem. Correct the problem by providing a new password.

EGL Java run-time error code CSO8002E

CSO8002E: The userid entered to the Gateway is not valid. %1

Explanation

The EGL GatewayServlet received an invalid userid exception when attempting to authenticate the user with the provided userid.

User Response

Examine the exception information returned in order to determine the cause of the problem. Correct the problem by providing a new userid.

EGL Java run-time error code CSO8003E

CSO8003E: Null entry for %1

Explanation

Null entry has been detected.

User Response

Examine the exception information returned in order to determine the cause of the problem. Correct the problem by providing required entry.

EGL Java run-time error code CSO8004E

CSO8004E: The gateway received an unknown security error.

Explanation

The EGL GatewayServlet received an unknown security exception when attempting to authenticate the user with the provided user information.

User Response

Examine the exception information returned in order to determine the cause of the problem. Correct the problem by providing new user information. If you are unable to determine why the failure occurred, please contact IBM Support for assistance.

EGL Java run-time error code CSO8005E

CSO8005E: Error occurred when changing the password. %1

Explanation

The EGL GatewayServlet received an error when attempting to change the provided password.

User Response

Examine the exception information returned in order to determine the cause of the problem. Correct the problem by providing new password. If you are unable to determine why the failure occurred, please contact IBM Support for assistance.

EGL Java run-time error code CSO8100E

CSO8100E: Unable to get a connection factory. Exception is %1

Explanation

The exception was thrown during a look-up of the connection factory that is used on a call when the value of the remoteComType property is CICSJ2C. The remoteComType property is in the linkage options part, in the callLink element for the called program.

The name of the connection factory begins java:comp/env/, followed by the value that you set in the location property of the same callLink element.

User Response

Make sure that the connection factory is defined properly in the J2EE environment and that the value for the location property is correct in the callLink element for the called program.

EGL Java run-time error code CSO8101E

CSO8101E: Unable to get a connection. Exception is: %1

Explanation

The exception was thrown by the getConnection method of the ConnectionFactory object that was used to make a call when the value of the remoteComType property is CICSJ2C. The remoteComType property is in the linkage options part, in the callLink element for the called program.

User Response

The Connection Factory or Resource Adapter may not be defined or onfigured properly. Diagnose the problem by referencing the text of the exception, the documentation of your Resource Adapter, and the documentation of your J2EE environment.

EGL Java run-time error code CSO8102E

CSO8102E: Unable to get an Interaction. Exception is: %1

Explanation

The exception was thrown by the createInteraction method of the Connection object that is used to make a call when the value of the remoteComType property is CICSJ2C. The remoteComType property is in the linkage options part, in the callLink element for the called program.

User Response

The Connection Factory or Resource Adapter may not be defined or configured properly. Diagnose the problem by using the text of the exception, the documentation of your Resource Adapter, and the documentation of your J2EE environment.

EGL Java run-time error code CSO8103E

CSO8103E: Unable to set an interaction verb. Exception is %1

Explanation

The exception was thrown by the setInteractionVerb method of the ECIInteractionSpec object that is used to make a call when the value of the remoteComType property is CICSJ2C. The remoteComType property is in the linkage options part, in the callLink element for the called program.

User Response

The Connection Factory or Resource Adapter may not be defined or configured properly. Diagnose the problem by using the text of the exception, the documentation of your Resource Adapter, and the documentation of your J2EE environment.

EGL Java run-time error code CSO8104E

CSO8104E: An error occurred during an attempt to communicate with CICS. Exception is %1

Explanation

The exception was thrown by the execute method of the Interaction object that is used to make a call when the value of the remoteComType property is CICSJ2C. The remoteComType property is in the linkage options part, in the callLink element for the called program.

User Response

The Connection Factory or Resource Adapter may not be defined or configured properly. Diagnose the problem by using the text of the exception, the documentation of your Resource Adapter, and the documentation of your J2EE environment. Additional information may be in the gateway log or in a log file on the remote system.

EGL Java run-time error code CSO8105E

CSO8105E: Unable to close an Interaction or Connection. Exception is %1

Explanation

The exception was thrown by the close method of a Connection or Interaction object that is used to make a call when the value of the remoteComType property is CICSJ2C. The remoteComType property is in the linkage options part, in the callLink element for the called program.

User Response

The Connection Factory or Resource Adapter may not be defined or configured properly. Diagnose the problem by using the text of the exception, the documentation of your Resource Adapter, and the documentation of your J2EE environment.

EGL Java run-time error code CSO8106E

CSO8106E: Unable to get a LocalTransaction for client unit of work. Exception is %1

Explanation

The exception was thrown by the getLocalTransaction method of a Connection object that is used to make a call in this situation:

- The value of the remoteComType property is CICSJ2C
- The value of the luwControl property is CLIENT

Those properties are in the linkage options part, in the callLink element for the called program.

User Response

The Connection Factory or Resource Adapter may not be defined or configured properly. Diagnose the problem by using the text of the exception, the documentation of your Resource Adapter, and the documentation of your J2EE environment.

EGL Java run-time error code CSO8107E

CSO8107E: Unable to set the timeout value on a CICSJ2C call. Exception is %1

Explanation

The exception was thrown by the `setExecuteTimeout` method of an `ECIInteractionSpec` object that is used to make a call when the value of the `remoteComType` property is `CICSJ2C`. The `remoteComType` property is in the linkage options part, in the `callLink` element for the called program.

User Response

The Connection Factory or Resource Adapter may not be defined or configured properly. Diagnose the problem by using the text of the exception, the documentation of your Resource Adapter, and the documentation of your J2EE environment.

EGL Java run-time error code CSO8108E

CSO8108E: An error occurred during an attempt to communicate with CICS.

Explanation

The `execute` method of the `Interaction` object that is used to make the call returned false. The call did not complete successfully.

User Response

The Connection Factory or Resource Adapter may not be defined or configured properly. Diagnose the problem by using the text of the exception, the documentation of your Resource Adapter, and the documentation of your J2EE environment. Additional information may be in the gateway log or in a log file on the remote system.

EGL Java run-time error code CSO8109E

CSO8109E: The timeout value %1 is invalid. It must be a number.

Explanation

An invalid value was specified for the timeout.

User Response

Either do not specify a timeout value, or specify a number.

EGL Java run-time error code CSO8110E

CSO8110E: The parmForm linkage property must be set to COMMPTR to call program %1 as there is at least one parameter that is a dynamic array.

Explanation

The parmForm must be COMMPTR because one of the parameters is a dynamic array.

User Response

Change the parmForm to COMMPTR.

EGL Java run-time error code CSO8180E

CSO8180E: The linkage specified a DEBUG call within a J2EE server. The call was not made on a J2EE server, the J2EE server is not in debug mode, or the J2EE server was not enabled for EGL debugging.

Explanation

The DEBUG call cannot be completed.

User Response

If the call is not being made on a J2EE server, the TCP/IP hostname of the machine running the EGL debugger must be specified in the location field of the linkage. If the call is being made on a J2EE server, make sure that it was started in debug mode and make sure that the EGL Debugger jar files were added to it.

EGL Java run-time error code CSO8181E

CSO8181E: Cannot contact the EGL debugger at hostname %1 and port %2. Exception is %3

Explanation

The DEBUG call cannot be completed because the EGL debugger could not be contacted.

User Response

Make sure an EGL Listener is running in the EGL debugger at the specified hostname and port.

EGL Java run-time error code CSO8182E

CSO8182E: An error occurred while communicating with the EGL debugger at hostname %1 and port %2. Exception is %3

Explanation

Communication between the EGL debugger and the calling program failed.

User Response

Use the information in the exception message to correct the problem.

EGL Java run-time error code CSO8200E

CSO8200E: Array wrapper %1 cannot be expanded beyond its maximum size.
The error occurred in method %2.

Explanation

The maximum size of the array was exceeded.

User Response

Check the size and maximum size of the array before attempting to add to it.

EGL Java run-time error code CSO8201E

CSO8201E: %1 is an invalid index for array wrapper %2. Maximum size: %3.
Current[®] size: %4

Explanation

The index is outside the bounds of the array.

User Response

Use a valid index.

EGL Java run-time error code CSO8202E

CSO8202E: %1 is not a valid maximum size for array wrapper %2.

Explanation

The property maxSize must be greater than or equal to zero.

User Response

Do not set the property maxSize to a negative number.

EGL Java run-time error code CSO8203E

CSO8203E: %1 is an invalid object type to add to an array wrapper of type %2.

Explanation

The contents of the array must match its definition.

User Response

Change the type of objects that the array stores, or do not attempt to store that type of object in the array.

EGL Java run-time error code EGL0650E

EGL0650E: The %1RequestAttr function failed with key, %2. Error: %3

Explanation

The EGL GetRequestAttr or SetRequestAttr function failed when invoked with the given key.

User Response

Use the Error part of this message to diagnose and correct the problem. Make sure the function is used within a page handler function.

EGL Java run-time error code EGL0651E

EGL0651E: The %1SessionAttr function failed with key, %2. Error: %3

Explanation

The EGL GetSessionAttr or SetSessionAttr function failed when invoked with the given key.

User Response

Use the Error part of this message to diagnose and correct the problem. Make sure the function is invoked within a page handler function.

EGL Java run-time error code EGL0652E

EGL0652E: The forward statement failed with label, %1. Error: %2

Explanation

Control could not be forwarded to the given label.

User Response

Use the Error part of this message to diagnose and correct the problem. Make sure the the EGL object which associated with the label is generated correctly and that the label is defined in the application configuration file.

EGL Java run-time error code EGL0653E

EGL0653E: Failed to create Bean from EGL object, %1. Error: %2

Explanation

Could not create an access bean from EGL record or page handler definition.

User Response

Use the Error part of this message to diagnose and correct the problem.

EGL Java run-time error code EGL0654E

EGL0654E: The SetError function failed with item, %1, key, %2. Error: %3

Explanation

The SetError function failed when invoked with the given message key.

User Response

Use the Error part of this message to diagnose and correct the problem. Make sure the item has an error entry in the JSP and the key is defined in the message resource file.

EGL Java run-time error code EGL0655E

EGL0655E: Failed to copy data from Bean to EGL record, %1. Error: %2

Explanation

An attempt to move data from the form bean to the record failed.

User Response

Use the Error part of this message to diagnose and correct the problem. Make sure the bean definition matches with the record definition.

EGL Java run-time error code EGL0656E

EGL0656E: Cannot assign array of size %1 to static array of size %2.

Explanation

The sizes of the arrays must match.

User Response

Check the EGL array definitions and make sure the array sizes are the same.

EGL Java run-time error code EGL0657E

EGL0657E: Processing of an onPageLoad parameter failed. Error: %1.

Explanation

An error occurred when EGL tried to receive values into the parameters of the onPageLoad function.

User Response

Use the Error part of this message to diagnose and correct the problem. Make sure the type definition of the passed value matches the type defined for the parameter in the onPageLoad function.

EGL Java run-time error code VGJ0001E

VGJ0001E: Maximum value overflow from %1.

Explanation

During an arithmetic calculation, either a value was divided by zero or an intermediate result exceeded 18 significant digits. The program ends unless system variable **handleOverflow** is set to 2.

User Response

Perform one or more of the following actions:

- Correct the logic of your program to avoid the error.
- Define the program logic to handle the overflow condition; use the system variables **handleOverflow** and **overflowIndicator**.

EGL Java run-time error code VGJ0002E

VGJ0002E: Error %1 occurred. The message text for this error could not be found in the message file %2.

Explanation

The message file may be corrupt or from an older release of EGL.

User Response

Complete one of the following instructions:

- If you extracted class files from the file `fda.jar`, verify that the classes you have are at the same release or maintenance level as the classes in `fda.jar`. If you find a mismatch, replace the older classes with the correct version.
- Reinstall `fda.jar` from EGL.

If the problem persists, do as follows:

1. Record the message number and the message text.

Note: The error message includes the following important information:

- Where the error occurred
 - The type of internal error
2. Record the situation in which this message occurs.
 3. For further instructions on how to report possible defects to the IBM Support Center, refer to the product's installation manual.

EGL Java run-time error code VGJ0003E

VGJ0003E: An internal error occurred at location %1.

Explanation

This error can occur only when system constraints or requirements were not satisfied or when EGL program parts were used improperly. The location specified in the error is used only for IBM diagnostic purposes.

User Response

Check the program setup and restart the system. If the problem persists, do as follows:

1. Record the message number and the message text.

Note: The error message includes the following important information:

- Where the error occurred
- The type of internal error

2. Record the situation in which this message occurs.
3. For further instructions on how to report possible defects to the IBM Support Center, refer to the product's installation manual.

EGL Java run-time error code VGJ0004I

VGJ0004I: The error occurred in %1, function %2.

Explanation

This message accompanies another message when an error occurs. It identifies the program or record where the error occurred, as well as the function that was executing at the time.

User Response

None.

EGL Java run-time error code VGJ0005I

VGJ0005I: The error occurred in %1.

Explanation

This message accompanies another message and identifies the program or record where an error occurred.

User Response

None.

EGL Java run-time error code VGJ0006E

VGJ0006E: An error occurred during an I/O operation. %1

Explanation

An I/O operation failed, and the EGL statement has no try statement to deal with the error.

User Response

If you want the program to handle the error, set `handleHardIOErrors` to 1 and put the I/O statement in a try statement, as in the following example:

```
handleHardIoErrors = 1;  
  
if (userRequest = "A")  
  try  
    add record1  
  onException  
    myErrorHandler(12);  
  end  
end
```

EGL Java run-time error code VGJ0050E

**VGJ0050E: An exception occurred while loading program %1. Exception: %2
Message: %3**

Explanation

The program's class could not be loaded.

User Response

Use the exception message to diagnose and fix the problem. The most common cause of this error is that the jar file or the directory containing the program's class file is not listed in the CLASSPATH environment variable.

EGL Java run-time error code VGJ0055E

VGJ0055E: An error occurred on a call to program %1. The error code was %2 (%3).

Explanation

The error occurred during a call to a local Java program.

User Response

Use the exception message to diagnose and fix the problem.

EGL Java run-time error code VGJ0056E

VGJ0056E: Called program %1 expected %2 parameters but was passed %3.

Explanation

The wrong number of parameters was passed to a called program.

User Response

Rewrite the calling program or the called program so that both expect the same number of parameters to be passed.

EGL Java run-time error code VGJ0057E

VGJ0057E: An exception occurred while passing parameters to called program %1. Exception: %2 Message: %3

Explanation

An error occurred during a call to a Java program. The error may have happened before or after the program began.

User Response

Use the exception and its message to diagnose and fix the problem.

EGL Java run-time error code VGJ0058E

VGJ0058E: Properties file %1 could not be loaded.

Explanation

The program's properties file could not be loaded. The name of the properties file is obtained from the system property `vgj.properties.file`.

User Response

Ensure that `vgj.properties.file` has the correct file name and that the properties file is in a Jar file or directory listed in the `CLASSPATH` environment variable.

EGL Java run-time error code VGJ0060E

VGJ0060E: StartTransaction to class %1 failed. The exception is %2.

Explanation

The exception was thrown while the program was attempting to start a new JVM to run the specified server class as a new transaction. The property `vgj.java.command` specifies the command used to start a new JVM. The default command is `java`.

User Response

Ensure that the property `vgj.java.command` has the correct value, and that your program has permission to create a new process.

Put the `startTransaction` statement inside a try statement to prevent this from being a fatal error. When the `startTransaction` fails within a try statement, an error code will be stored in the `errorCode` system variable.

EGL Java run-time error code VGJ0062E

VGJ0062E: One or more parameters passed to MQ program %1 was of the wrong type. %2

Explanation

An exception was thrown while attempting to call the MQ program. The parameters are incorrect.

User Response

Consult the MQ program's documentation and the exception's message to correct the error.

EGL Java run-time error code VGJ0064E

VGJ0064E: Program %1 expected text form %2 but it was given text form %3 on a show statement.

Explanation

Both programs must use the same text form.

User Response

Modify the programs to use the same text form and regenerate.

EGL Java run-time error code VGJ0100E

VGJ0100E: The data of %1 is not in %2 format.

Explanation

The data in the item is in an unexpected format. Another item may have written over the specified item.

User Response

Correct the program logic to avoid the error.

EGL Java run-time error code VGJ0104E

VGJ0104E: %1 is not a valid index for subscript %2 of %3.

Explanation

One of the subscripts used with a multidimensional array is invalid. A subscript value must be between one and the number of occurrences defined for the subscripted item.

User Response

Ensure that the index value is a valid subscript for the subscripted item.

EGL Java run-time error code VGJ0105E

VGJ0105E: %1 is not a valid index for %2.

Explanation

A subscript value must be between one and the number of occurrences defined for the subscripted item.

User Response

Ensure that the index value is a valid subscript for the subscripted item.

EGL Java run-time error code VGJ0106E

VGJ0106E: User overflow during assignment of %1 to %2.

Explanation

The target of an assignment is not large enough to hold the result without truncating significant digits. The value of system variable `handleOverflow` is 1, which causes the program to end.

User Response

Do as follows:

- Increase the number of significant digits in the target; or
- Define the program logic to handle the overflow condition; use the system variables `handleOverflow` and `overflowIndicator`.

EGL Java run-time error code VGJ0108E

VGJ0108E: HEX item %1 was assigned nonhexadecimal value %2.

Explanation

HEX items can receive only hexadecimal digits.

User Response

Make sure that the source value includes only hexadecimal digits.

EGL Java run-time error code VGJ0109E

VGJ0109E: HEX item %1 was assigned nonhexadecimal value from %2: %3.

Explanation

HEX items can receive only hexadecimal digits.

User Response

Make sure that the source in the assignment contains only hexadecimal digits.

EGL Java run-time error code VGJ0110E

VGJ0110E: HEX item %1 was compared to nonhexadecimal value: %2.

Explanation

HEX items can be compared only to hexadecimal digits.

User Response

Make sure that the comparison value includes only hexadecimal digits.

EGL Java run-time error code VGJ0111E

VGJ0111E: HEX item %1 was compared to nonhexadecimal value from %2: %3.

Explanation

HEX items can be compared only to hexadecimal digits.

User Response

Make sure that the comparison value contains only hexadecimal digits.

EGL Java run-time error code VGJ0112E

VGJ0112E: NUM item %1 was assigned nonnumeric value: %2.

Explanation

NUM items can be assigned only numeric values. Such values contain digits and may have leading and trailing spaces, a decimal point, and a leading sign. The decimal point is allowed in between two digits, immediately before the first digit, or immediately after the last digit.

User Response

Make sure that the source value is numeric.

EGL Java run-time error code VGJ0113E

VGJ0113E: NUM item %1 was assigned nonnumeric value from %2: %3.

Explanation

NUM items can be assigned only numeric values. Such values contain digits and may have leading and trailing spaces, a decimal point, and a leading sign. The decimal point is allowed in between two digits, immediately before the first digit, or immediately after the last digit.

User Response

Make sure that the source value is numeric.

EGL Java run-time error code VGJ0114E

VGJ0114E: The value of item %1 (%2) is not valid as a subscript.

Explanation

The value has too many digits to be a subscript for any element in the array. A subscript value must be between one and the number of occurs declared for the structure item.

User Response

Make sure that the index value is a valid subscript for the array.

EGL Java run-time error code VGJ0115E

VGJ0115E: %1 cannot be assigned a string. The string was %2.

Explanation

This is an internal error. The item cannot be assigned a string.

User Response

Complete the following steps:

1. Record the message number and the message text.

Note: The error message includes the following important information:

- Where the error occurred
 - The type of internal error
2. Record the situation in which this message occurs.
 3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code VGJ0116E

VGJ0116E: %1 cannot be assigned a number. The number was %2.

Explanation

This is an internal error. The item cannot be assigned a number.

User Response

Complete the following steps:

1. Record the message number and the message text.

Note: The error message includes the following important information:

- Where the error occurred
 - The type of internal error
2. Record the situation in which this message occurs.
 3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code VGJ0117E

VGJ0117E: %1 cannot be converted to a long.

Explanation

The item cannot be converted to a long.

User Response

Complete the following steps:

1. Record the message number and the message text.

Note: The error message includes the following important information:

- Where the error occurred
 - The type of internal error
2. Record the situation in which this message occurs.
 3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code VGJ0118E

VGJ0118E: %1 cannot be converted to a VGJBigNumber.

Explanation

This is an internal error. The item cannot be converted to a VGJBigNumber.

User Response

Complete the following steps:

1. Record the message number and the message text.

Note: The error message includes the following important information:

- Where the error occurred
 - The type of internal error
2. Record the situation in which this message occurs.
 3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code VGJ0119E

VGJ0119E: %1 is not a valid number.

Explanation

While using the debugger, the user attempted to set the value of a numeric item, but the new value is not a number.

User Response

Use a numeric value.

EGL Java run-time error code VGJ0140E

VGJ0140E: Array function %1 failed because there was an attempt to expand array %2 beyond its maximum size.

Explanation

The array cannot hold any more values.

User Response

Modify the program to check the size of the array before attempting to add to it.

EGL Java run-time error code VGJ0141E

VGJ0141E: %1 is an invalid index for array %2. Current size: %3. Max size: %4

Explanation

The index is out for range for the array.

User Response

Modify the program to use a valid array index.

EGL Java run-time error code VGJ0142E

VGJ0142E: The `maximumSize` of array %1 cannot be changed. Expected %2 got %3.

Explanation

The array was passed on a call statement. The corresponding array in the called program had a different `maximumSize`.

User Response

Change one of the programs so that both use an array with the same `maximumSize`.

EGL Java run-time error code VGJ0143E

VGJ0143E: %1 is not a valid size for array %2.

Explanation

The array was passed on a call statement. The called program changed the array's size to a value that is less than zero or larger than the value of the property `maxSize`.

User Response

Change the programs so they use the same value for the property `maxSize`.

EGL Java run-time error code VGJ0160E

VGJ0160E: Math function %1 failed with error code 8 (domain error).

Explanation

An argument to the function is not valid.

User Response

Do as follows:

- Change the program logic to ensure that the arguments to the function are valid, as per the function's documentation; or
- Call the function in a try statement, or set `handleSysLibErrors` to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0161E

VGJ0161E: Math function %1 failed with error code 8 (domain error).

Explanation

The argument must be between -1 and 1.

User Response

Do as follows:

- Change the program logic to ensure that the argument passed to the function is between -1 and 1; or
- Call the function in a try statement, or set `handleSysLibErrors` to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0162E

VGJ0162E: Math function atan2 failed with error code 8 (domain error).

Explanation

Both arguments cannot be zero.

User Response

Do as follows:

- Change the program logic to ensure that at least one argument passed to the function is not zero; or
- Call the function in a try statement, or set `handleSysLibErrors` to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0163E

VGJ0163E: Math function %1 failed with error code 8 (domain error).

Explanation

The second argument must not be zero.

User Response

Do as follows:

- Change the program logic to ensure that the second argument is not zero; or
- Call the function in a try statement, or set `handleSysLibErrors` to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0164E

VGJ0164E: Math function %1 failed with error code 8 (domain error).

Explanation

The argument must be greater than zero.

User Response

Do as follows:

- Change the program logic to ensure that the argument passed to the function is greater than zero; or
- Call the function in a try statement, or set `handleSysLibErrors` to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0165E

VGJ0165E: Math function pow failed with error code 8 (domain error).

Explanation

If the first argument is zero, the second must be greater than zero.

User Response

Do as follows:

- Change the program logic to ensure that if the first argument passed to the function is zero, the second argument is greater than zero; or
- Call the function in a try statement, or set `handleSysLibErrors` to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0166E

VGJ0166E: Math function pow failed with error code 8 (domain error).

Explanation

If the first argument is less than zero, the second must be an integer.

User Response

Do as follows:

- Change the program logic to ensure that if the first argument passed to the function is less than zero, the second argument is an integer; or
- Call the function in a try statement, or set `handleSysLibErrors` to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0167E

VGJ0167E: Math function `sqrt` failed with error code 8 (domain error).

Explanation

The argument must be greater than or equal to zero.

User Response

Do as follows:

- Change the program logic to ensure that the argument passed to the function is greater than or equal to zero; or
- Call the function in a try statement, or set `handleSysLibErrors` to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0168E

VGJ0168E: Math function `%1` failed with error code 12 (range error).

Explanation

An intermediate or final result cannot be represented as a double precision floating point number or with the precision of the result item.

User Response

Do as follows:

- Change the program logic to ensure that the target item is large enough to hold the result value; or
- Change the program logic so that the arguments to the function have values that do not cause this problem; or
- Call the function in a try statement, or set `handleSysLibErrors` to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0200E

VGJ0200E: String function `%1` failed with error code 8.

Explanation

The index must be between 1 and the length of the string.

User Response

Do as follows:

- Change the program logic to ensure that the index-related argument to the function ranges between 1 and the length of the string; or
- Call the function in a try statement, or set `handleSysLibErrors` to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0201E

VGJ0201E: String function %1 failed with error code 12.

Explanation

The length must be greater than zero.

User Response

Do as follows:

- Change the program logic to ensure that the length arguments passed to the function have values that are greater than zero; or
- Call the function in a try statement, or set `handleSysLibErrors` to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0202E

VGJ0202E: String function `setNullTerminator` failed with error code 16.

Explanation

The last byte of the target string must be a blank or null character.

User Response

Do as follows:

- Change the program logic to ensure that the last byte of the target string is a blank or null character; or
- Call the function in a try statement, or set `handleSysLibErrors` to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0203E

VGJ0203E: String function %1 failed with error code 20.

Explanation

The index of a `DBCHAR` or `UNICODE` substring must be odd so that the index identifies the first byte of a character.

User Response

Do as follows:

- Change the program logic to ensure that the index arguments passed to the function are valid; or
- Call the function in a try statement, or set `handleSysLibErrors` to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0204E

VGJ0204E: String function %1 failed with error code 24.

Explanation

The length of a DBCHAR or UNICODE substring must be even to refer to a whole number of characters.

User Response

Do as follows:

- Change the program logic to ensure that the length arguments passed to the function have valid values; or
- Call the function in a try statement, or set `handleSysLibErrors` to 1 before calling the function, so that the program can handle the error.

EGL Java run-time error code VGJ0215E

VGJ0215E: %1 was passed the nonnumeric string %2.

Explanation

Every character in the portion of the string defined by the length argument must be numeric.

User Response

Change the program logic so the characters in the portion of the string defined by the length argument are numeric.

EGL Java run-time error code VGJ0216E

VGJ0216E: %1 is not a valid date mask for %2.

Explanation

The date mask defined in the properties file for use with the function is not valid.

The valid characters for a date mask are as follows:

D, M, Y

D for Day, M for Month, Y for Year

Separator character

Any nonnumeric, single-byte character except D, M, or Y.

Valid date masks can be in any of the following formats:

- Long Gregorian

The long version of the Gregorian mask must contain the following parts in any order:

YYYY 4-digit year

MM 2-digit numeric month

DD 2-digit numeric day of month

The mask parts must be separated by any nonnumeric single-byte character except D, M, or Y.

For example, a mask of YYYY/MM/DD is used to display August 25, 1997 as 1997/08/25.

- Long Julian

The long version of the Julian mask must contain the following parts in any order:

YYYY 4-digit year

DDD 3-digit numeric day of year

The mask parts must be separated by any single-byte nonnumeric character except D, M, or Y.

For example, a mask of DDD-YYYY can be used to display August 25, 1997 as 237-1997.

User Response

Change the date mask property to a valid value and restart the program. If no date mask property is defined, a default date mask will be used.

The date masks can be set using the properties `vgl.datemask.gregorian.long.NNN` and `vgl.datemask.julian.long.NNN`, where *NNN* is the current NLS code.

EGL Java run-time error code VGJ0217E

VGJ0217E: An error occurred in the convert function with argument %1: %2

Explanation

The attempt to convert the data of the argument failed. The reason for the failure is included in the message.

User Response

Use the error message to diagnose and correct the problem.

EGL Java run-time error code VGJ0250E

VGJ0250E: Could not retrieve item %1 from containing part %2.

Explanation

An internal error occurred. An attempt was made to access an item with the specified index in the record or table.

User Response

Do as follows:

1. Record the message number and the message text.

Note: The error message includes the following important information:

- Where the error occurred
- The type of internal error

2. Record the situation in which this message occurs.

3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code VGJ0300E

VGJ0300E: Table file for table %1 could not be loaded. Could not find a file named either %2 or %3.

Explanation

Neither of the named files could be found in any of the resource locations. All resource locations are searched for the first file. If no such file exists, all resource locations are searched for the second file.

Resource locations differ depending on the mechanism that was used to locate the table file.

If the error was encountered in an applet, resource locations refer to locations on the server machine and can vary depending on the implementation of the Java Virtual Machine. However, all implementations should search the directory on the server specified by the CODEBASE value. This value is set by the APPLET tag in the HTML file containing the applet. If no CODEBASE value is specified, it defaults to the directory on the web server containing the HTML file.

If the error was encountered in an application, valid resource locations are as follows:

- The directory that the Java Virtual Machine was started in (the working directory for the executable).
- Any directory specified in the CLASSPATH for the application being run. Specification of this value is system-dependent. On some systems, it can be specified as an environment variable. All systems allow it to be specified when invoking the Java Virtual Machine using the -classpath option. See the documentation that came with your copy of the Java Virtual Machine for more information on the value of CLASSPATH.

User Response

First, locate the table file and make sure the permissions necessary to access it are set.

If the error occurred from within an applet or if the error occurred from within an application and you do not want to modify the existing set of resource locations, copy the table file into a valid resource location.

Otherwise, complete one of the following instructions:

- If the Java interpreter will use the value of the CLASSPATH environment variable, add the directory containing the table file to the current value of CLASSPATH.
- Specify the directory containing the table file by using the -classpath option when invoking the Java interpreter. If specifying the -classpath option overrides the value of the CLASSPATH environment variable, you need to specify the path to the Java run-time classes (e.g. classes.zip or rt.jar) in addition to any directories you add as resource locations.

Additional diagnostic information may become available if you enable program trace.

EGL Java run-time error code VGJ0301E

VGJ0301E: Table file %1 for table %2 could not be loaded because an incorrect number of bytes was returned during the read operation on the table header.

Explanation

One of the following conditions exists:

- The table file has become corrupt.
- The table file was not generated with EGL or VisualAge Generator.

User Response

Regenerate the table.

Additional diagnostic information may become available if you enable program trace.

If the problem persists, do as follows:

1. Record the message number and the message text.

Note: The error message includes the following important information:

- Where the error occurred
 - The type of internal error
2. Record the situation in which this message occurs.
 3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code VGJ0302E

VGJ0302E: Table file %1 for table %2 could not be loaded because an unexpected magic number was encountered during inspection of the table header.

Explanation

One of the following conditions exists:

- The table file has become corrupt.
- The table file was not generated with EGL or VisualAge Generator.

User Response

Regenerate the table.

Additional diagnostic information may become available if you enable program trace.

If the problem persists, do as follows:

1. Record the message number and the message text.

Note: The error message includes the following important information:

- Where the error occurred
 - The type of internal error
2. Record the situation in which this message occurs.
 3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code VGJ0303E

VGJ0303E: Table file %1 for table %2 could not be loaded because an internal I/O error occurred during a read or close operation.

Explanation

One of the following conditions exists:

- The table file has become corrupt.
- The table file was not generated with EGL or VisualAge Generator.

User Response

Regenerate the table.

Additional diagnostic information may become available if you enable program trace.

If the problem persists, do as follows:

1. Record the message number and the message text.

Note: The error message includes the following important information:

- Where the error occurred
 - The type of internal error
2. Record the situation in which this message occurs.
 3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code VGJ0304E

VGJ0304E: Table file %1 for table %2 could not be loaded because an incorrect number of bytes was returned during the read operation on the table data.

Explanation

One of the following conditions exists:

- The table file was regenerated after its columns were changed but the program that is attempting to load the table was not regenerated. Generating only the table after changing the column definition causes an inconsistency to exist between the definition in the table file and the definition in the table class file, which is only generated during run-time code generation.
- The table file has become corrupt.
- The table file was not generated with EGL or VisualAge Generator.

User Response

Do as follows:

- If the column definition has not been changed, regenerate the table.
- If the column definition has been changed, either remove the change and regenerate the table or regenerate the run-time code for the program that uses the table.

Additional diagnostic information may become available if you enable program trace.

If the problem persists, do as follows:

1. Record the message number and the message text.

Note: The error message includes the following important information:

- Where the error occurred
 - The type of internal error
2. Record the situation in which this message occurs.
 3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code VGJ0305E

VGJ0305E: Table file %1 for table %2 could not be loaded. The data encountered in the table file for item %3 is not in the correct format. The corresponding data format error is: %4

Explanation

One of the following conditions exists:

- The table file was regenerated after its columns were changed but the applet or application that is attempting to load the table was not regenerated. Generating only the table after changing the column definition causes an inconsistency to exist between the definition in the table file and the definition in the table class file, which is only generated during run-time code generation.
- The table file has become corrupt.
- The table file was not generated with WebSphere Studio Enterprise Developer or VisualAge Generator.

User Response

Do as follows:

- If the column definition has not been changed, regenerate the table.
- If the column definition has been changed, either remove the change and regenerate the table or regenerate the run-time code for the program that uses the table.

If the problem persists, do as follows:

1. Record the message number and the message text.

Note: The error message includes the following important information:

- Where the error occurred

- The type of internal error
- 2. Record the situation in which this message occurs.
- 3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code VGJ0306E

VGJ0306E: Table file %1 for table %2 could not be loaded because the data in the table file is for a different type of table than table %2.

Explanation

One of the following conditions exists:

- The table file was regenerated after its columns were changed but the applet or application that is attempting to load the table was not regenerated. Generating only the table after changing the column definition causes an inconsistency to exist between the definition in the table file and the definition in the table class file, which is only generated during run-time code generation.
- The table file has become corrupt.
- The table file was not generated with EGL or VisualAge Generator.

User Response

Do as follows:

- If the table type has not been changed, regenerate the table.
- If the table type has been changed, either edit the table definition so that it is of the correct type and regenerate the table or regenerate the run-time code for the program that uses the table.

Additional diagnostic information may become available if you enable program trace.

If the problem persists, do as follows:

1. Record the message number and the message text.

Note: The error message includes the following important information:

- Where the error occurred
 - The type of internal error
2. Record the situation in which this message occurs.
 3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code VGJ0307E

VGJ0307E: Table file %1 for table %2 could not be loaded because table file %1 is a VisualAge Generator C++ table file and is not in big-endian format.

Explanation

Table files generated by the VisualAge Generator C++ generator can only be used with Java programs if the byte-ordering used to encode numeric data within the table is big-endian.

User Response

Regenerate the table in big-endian format or as a Java platform-independent table.

To regenerate the table in big-endian format, use VisualAge Generator to generate the table for a C++ target system that is big-endian (e.g. AIX). To regenerate the table as a Java platform-independent table, generate the table for a Java target system with VisualAge Generator or EGL.

Additional diagnostic information may become available if you enable program trace.

EGL Java run-time error code VGJ0308E

VGJ0308E: Table file %1 for table %2 could not be loaded. Table file %1 is a VisualAge Generator C++ table file, and the character encoding used in the table (%3) is not supported on the run-time system.

Explanation

Table files generated by the VisualAge Generator C++ generator can be used with Java programs only if the type of character encoding used for data within the table is the same type of encoding used by the run-time system.

User Response

Do as follows:

1. Determine the character encoding used on your system. Java programs use either the ASCII or EBCDIC character encodings. Most workstations use the ASCII encoding. Most host platforms use the EBCDIC encoding. If you do not know the encoding used on your system, contact your system administrator.
2. Regenerate the table using the correct character encoding or as a Java platform-independent table.

To regenerate the table using the correct character encoding, use VisualAge Generator to generate the table for your target system or another C++ target system that uses the same character encoding. To regenerate the table as a Java platform-independent table, generate the table for a Java target system with VisualAge Generator or EGL.

Additional diagnostic information may become available if you enable program trace.

EGL Java run-time error code VGJ0315E

VGJ0315E: A shared table entry for table %1 could not be found during the table unloading process.

Explanation

An internal error occurred.

User Response

Do as follows:

1. Record the message number and the message text.

Note: The error message includes the following important information:

- Where the error occurred
 - The type of internal error
2. Record the situation in which this message occurs.
 3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code VGJ0320E

VGJ0320E: An edit routine with table %1 failed while comparing the table column %2 and the field %3.

Explanation

The table column and the field have types that are not valid for comparison.

User Response

Do one of the following:

- Ensure that the types of the column and the field are valid for comparison by doing the following:
 1. Correct either the type of the column or the type of the field so that the comparison will be valid.
 2. Regenerate the program.
 3. Run the program.
- Modify your program to use a different table for the edit routine such that the comparison of the column and the field will be valid.

Refer to the trace output for more information.

EGL Java run-time error code VGJ0330E

VGJ0330E: Could not find a message with ID %1 in the message table %2.

Explanation

This error can occur during the following operations:

- Lookup of the the value for a form's msgField.
- Lookup of the value with the identifier specified as an edit message.

One of the following conditions exists:

- A message with this ID does not exist in the message table.
- The table file or message resource bundle for the table has become corrupt.

User Response

Do one of the following:

- Ensure that a message with the message ID exists by doing the following:
 1. Add a message to the table with the message ID if it does not already exist.
 2. Regenerate the table.
 3. Run the program.

- Modify your program to use a different message that is already defined in the table.
- Modify your program to use a different message table that contains a message with the message ID.

EGL Java run-time error code VGJ0331E

VGJ0331E: Message table file %1 could not be loaded.

Explanation

The class for the program's message table could not be loaded, or an instance of the class could not be created.

User Response

Ensure the message table has been generated.

EGL Java run-time error code VGJ0350E

VGJ0350E: An error occurred on a call to program %1. The error code was %2.

Explanation

A remote or EJB call to the specified program failed.

User Response

Additional diagnostic information may become available if you enable program trace.

EGL Java run-time error code VGJ0351E

VGJ0351E: commit failed: %1

Explanation

The resources could not be committed.

User Response

Additional diagnostic information may become available if you enable program trace.

EGL Java run-time error code VGJ0352E

VGJ0352E: rollBack failed: %1

Explanation

The resources could not be rolled back.

User Response

Additional diagnostic information may become available if you enable program trace.

EGL Java run-time error code VGJ0400E

VGJ0400E: An invalid parameter index, %1, was used for function %2.

Explanation

This is an internal error.

User Response

Contact IBM support.

EGL Java run-time error code VGJ0401E

VGJ0401E: An invalid parameter descriptor was detected for function %1, parameter %2.

Explanation

This is an internal error.

User Response

Contact IBM support.

EGL Java run-time error code VGJ0402E

VGJ0402E: The type of the value used for parameter %1 of function or program %2 is invalid.

Explanation

The value cannot be passed as a parameter, because the type of the value is incompatible with the type of the parameter.

User Response

Do one of the following:

- Change the definition of the parameter to match the type of the value.
- Change the type of the value to match the definition of the parameter.

EGL Java run-time error code VGJ0403E

VGJ0403E: An error occurred while running script %1. The exception text is %2.

Explanation

The script caused an exception to be thrown.

User Response

Correct the program logic to avoid the error.

EGL Java run-time error code VGJ0416E

VGJ0416E: An error occurred on a call to program %1. The error code was %2 (%3).

Explanation

An exception was thrown during an attempt to run the called program. The problem may be due to one of the following conditions:

- The program may not have permission to create a new process.
- The called program may not exist.
- The called program may not be found in the system path.

User Response

Do as follows:

1. Verify that the program has permission to create a new process.
2. Verify that the called program exists.
3. Verify that the called program can be found in the system path.

If the problem persists, do as follows:

1. Record the message number and the message text.

Note: The error message includes the following important information:

- Where the error occurred
 - The type of internal error
2. Record the situation in which this message occurs.
 3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code VGJ0450E

VGJ0450E: I/O operation %1 with I/O object %2 failed for this reason: %3.

Explanation

An EGL I/O statement failed outside of a try statement, or when the value of system variable `handleHardIoErrors` was zero.

User Response

Review the error message and respond as appropriate.

EGL Java run-time error code VGJ0500E

VGJ0500E: No input received for required field - enter again.

Explanation

No data was typed in the field. The field is defined as required.

User Response

Enter data in the field, or press a bypass edit key to bypass the edit check. Blanks will not satisfy the data input requirement for any type of field. In addition, zeros will not satisfy the data input requirement for numeric fields. The program continues.

EGL Java run-time error code VGJ0502E

VGJ0502E: Data type error in input - enter again.

Explanation

The data in the field is not valid numeric data. The field was defined as numeric.

User Response

Enter only numeric data in this field, or press a bypass edit key to bypass the edit check. In either situation, the program continues.

EGL Java run-time error code VGJ0503E

VGJ0503E: Number of allowable significant digits exceeded - enter again.

Explanation

Data was entered into a numeric field that is defined with decimal places, a sign, currency symbol, or numeric separator edits. The input data exceeds the number of significant digits that can be displayed within the editing criteria. The number entered is too large. The number of significant digits cannot exceed the field length, minus the number of decimal places, minus the places required for editing characters.

User Response

Enter a number with fewer significant digits.

EGL Java run-time error code VGJ0504E

VGJ0504E: Input not within defined range - enter again.

Explanation

The data in the field is not within the range of valid data defined for this item.

User Response

Enter data that is within the defined range or press a bypass edit key to bypass the edit check. In either case, the program will continue.

EGL Java run-time error code VGJ0505E

VGJ0505E: Input minimum length error - enter again.

Explanation

The data in the field does not contain enough characters to meet the required minimum length.

User Response

Enter the required number of characters to meet the minimum length or press a bypass edit key to bypass the edit check. In either case, the program will continue.

EGL Java run-time error code VGJ0506E

VGJ0506E: Table edit validity error - enter again.

Explanation

The data in the field does not meet the table edit requirement defined for the variable field.

User Response

Enter data that conforms to the table edit requirement or press a bypass edit key to bypass the edit check. In either case, the program will continue.

EGL Java run-time error code VGJ0507E

VGJ0507E: Modulus check error on input - enter again.

Explanation

The data in the field does not meet the modulus check requirement defined for the variable field.

User Response

Enter data that conforms to the modulus check defined for the variable field or press a bypass edit key to bypass the edit check. In either case, the program will continue.

EGL Java run-time error code VGJ0508E

VGJ0508E: Input not valid for defined date or time format %1.

Explanation

The data in the field, defined with a date edit, does not meet the requirements of the format specification.

User Response

Enter the date in the correct format shown in the message.

EGL Java run-time error code VGJ0510E

VGJ0510E: Input not valid for boolean field.

Explanation

The value typed in the field does not conform to the boolean check. Input into a boolean field must be either 'Y' or 'N' for character fields and either 1 or 0 for numeric fields.

User Response

Enter a 'Y' or 'N' for a character field or a 1 or 0 for a numeric field, or press the bypass edit key to bypass the edit check. In either case, the program will continue.

EGL Java run-time error code VGJ0511E

VGJ0511E: Edit table %1 is not defined for %2.

Explanation

A user message was requested but a user message table prefix was not defined for the program.

User Response

Have the program developer do one of the following:

- Add the message table prefix to the program specification and generate the program again.
- Remove the user message number from the field edit and generate again.

EGL Java run-time error code VGJ0512E

VGJ0512E: Hexadecimal data is not valid.

Explanation

The data in the variable field must be in hexadecimal format. One or more of the characters you entered does not occur in the following set: a b c d e f A B C D E F
0 1 2 3 4 5 6 7 8 9

User Response

Enter only hexadecimal characters in the variable field. The characters are left-justified and padded with the character 0. Embedded blanks are not permitted.

EGL Java run-time error code VGJ0600E

VGJ0600E: Unable to get linkage for program, %1.

Explanation

An entry for the specified program cannot be found in the CSO properties file because of one of the following reasons:

- An incorrect properties file was specified in the GatewayServlet configuration.
- The entry for the program was not specified in the CSO properties file.
- The CSO properties file is not in the directory specified in the GatewayServlet configuration.

User Response

Contact the web server administrator to make sure that the following are performed:

- Make sure the GatewayServlet configuration specifies the correct CSO properties file using the linkageTable initialization parameter.
- Make sure that the program is defined in the CSO properties file.

EGL Java run-time error code VGJ0601E

VGJ0601E: An exception occurred while attempting to call entry point program, %1. Exception: %2. Message: %3.

Explanation

An unexplained error occurred while attempting to call the entry point program. The exception and message will define the error further. An entry point page or program gives the user a menu of programs which can be started using the GatewayServlet.

User Response

Contact the web server administrator to make sure that the entry point page or the entry program are specified correctly in the GatewayServlet configuration.

EGL Java run-time error code VGJ0603E

VGJ0603E: The bean, %1, is invalid.

Explanation

The Page Bean or the bean name is invalid.

User Response

Contact the web server administrator to make sure that the bean name is correct and that the Page Bean and the Java Server Page are deployed and made available to the GatewayServlet.

EGL Java run-time error code VGJ0604E

VGJ0604E: An exception occurred while attempting to load bean, %1. Exception: %2. Message: %3.

Explanation

An unexplained error occurred while trying to load the Page Bean. The exception and message will define the error further.

User Response

Contact the web server administrator to make sure that the bean name is correct and that the Page Bean and the Java Server Page are deployed and made available to the GatewayServlet.

EGL Java run-time error code VGJ0607E

VGJ0607E: A version mismatch has occurred between the server, %1, and bean, %2.

Explanation

The version of the User Interface Record Bean does not match the version of the User Interface Record used by the server program. For proper operation, the versions must be compatible.

User Response

Contact the program developer and generate both the program and user interface record beans. Contact the web server administrator to make sure that the user interface record bean is deployed to the proper location.

EGL Java run-time error code VGJ0608E

VGJ0608E: An error occurred while attempting to set data in the bean, %1. Exception: %2. Message: %3.

Explanation

An exception occurred while trying to set the record data from the server application into the User Interface Record Bean. The exception and message are included to help determine the problem.

User Response

Use the exception and message included in the message for problem determination.

EGL Java run-time error code VGJ0609I

VGJ0609I: A gateway session is being bound for user, %1.

Explanation

This informational message appears on the application server's stdout or stderr. The message appears whenever a web session is created for the user.

User Response

No response is required.

EGL Java run-time error code VGJ0610I

VGJ0610I: A gateway session is being unbound for user, %1.

Explanation

This informational message appears on the application server's stdout or stderr. The message appears whenever a web session has ended for the user. A session will end after a period of inactivity or if a severe error occurs that terminates the session.

User Response

No response is required.

EGL Java run-time error code VGJ0611E

VGJ0611E: Unable to establish a connection with the SessionIDManager.

Explanation

The GatewayServlet was unable to connect to the SessionIDManager. The SessionIDManager is the component which gives session ids for gateway users. A session id is obtained for each active session and is used by the server program for saving and restoring application data.

The SessionIDManager is a separate application which listens for connects and requests for ids. When a session ends, the SessionIDManager will make the session id available to other sessions. The SessionIDManager must be active in order to run the GatewayServlet.

User Response

Contact your web server administrator to start the SessionIDManager. If already started, the location of the SessionIDManager must be set in the GatewayServlet's configuration.

EGL Java run-time error code VGJ0612I

VGJ0612I: A gateway session is connected to the SessionIDManager for user, %1.

Explanation

This informational message appears in the web server's stdout or stderr. A session has connected to the SessionIDManager successfully in order to obtain a session id. The session id is used by the server program to save and restore program data.

User Response

No response is required.

EGL Java run-time error code VGJ0614E

VGJ0614E: A required parameter, %1, is missing from the GatewayServlet configuration.

Explanation

A required parameter was not specified in the servlet configuration. The GatewayServlet will not run without these parameters.

User Response

Contact the web server administrator to make sure that the GatewayServlet is properly configured. Reference your application server documentation to determine how to configure servlet parameters.

EGL Java run-time error code VGJ0615E

VGJ0615E: Web transaction %1 is not allowed to run on this instance of the EGL Action Invoker.

Explanation

There was a problem creating or retrieving the GatewayRequestHandler for the program.

User Response

Ensure that the named application has been generated and deployed to the server.

EGL Java run-time error code VGJ0616E

VGJ0616E: The gateway parameter %1 does not specify valid class: %2

Explanation

The class identified in the specified gateway property could not be loaded or instantiated.

User Response

Ensure that the named class has been deployed to the server and specified correctly in the gateway properties file.

EGL Java run-time error code VGJ0617E

VGJ0617E: Please provide valid public user information in the gateway properties file.

Explanation

The public user name or password specified in the gateway properties file is invalid.

User Response

Ensure that the public user name and password values in the gateway properties file are correct.

EGL Java run-time error code VGJ0700E

VGJ0700E: An error occurred during database connection: %1.

Explanation

An error occurred during an attempt to connect to a database. The error message ends with text from the database management system.

User Response

Review the error message and respond as appropriate. Additional diagnostic information may become available if you enable program trace.

EGL Java run-time error code VGJ0701E

VGJ0701E: A database connection must be established prior to an SQL I/O operation.

Explanation

An SQL I/O operation was attempted before a database connection was established.

User Response

An SQL I/O operation is valid only after the program creates a database connection. The program can create a default connection based on a program property and can override the default by running the connect system function. Review the EGL help pages for details on program properties and on setting up database access.

EGL Java run-time error code VGJ0702E

VGJ0702E: An error occurred during SQL I/O operation %1. %2.

Explanation

An error occurred during the specified SQL I/O operation. The message ends with text from the database management system.

User Response

Review the message and respond as appropriate.

Additional diagnostic information may become available if you enable program trace.

EGL Java run-time error code VGJ0703E

VGJ0703E: An error occurred during setup for SQL I/O operation %1. %2.

Explanation

An error occurred during setup for the specified SQL I/O operation.

User Response

Review the message and respond as appropriate.

Additional diagnostic information may become available if you enable program trace.

EGL Java run-time error code VGJ0705E

VGJ0705E: An error occurred while disconnecting database %1. %2.

Explanation

An error occurred during an attempt to disconnect from the specified database. The error message ends with text from the database management system.

User Response

Review the message and respond as appropriate.

Additional diagnostic information may become available if you enable program trace.

EGL Java run-time error code VGJ0706E

VGJ0706E: Cannot set connection to database %1. The connection does not exist.

Explanation

An error occurred during an attempt to set the connection to the specified database. The connection can be set only to an active database connection within the transaction.

User Response

Make sure that the name of the database matches one of the active database connections established for the transaction.

Additional diagnostic information may become available if you enable program trace.

EGL Java run-time error code VGJ0707E

VGJ0707E: An SQL I/O sequence error occurred on %1.

Explanation

A sequence error may occur in these cases:

- An EGL replace or delete occurs but was not preceded by a setupd or update statement against the same SQL record
- An EGL scan occurs but was not preceded by a setupd or setinq statement against the same SQL record

The message identifies the last I/O operation that the program attempted, whether replace, delete, or scan.

User Response

Make sure that the order of EGL statements is correct.

Additional diagnostic information may become available if you enable program trace.

EGL Java run-time error code VGJ0708E

VGJ0708E: Error while loading the JDBC driver classes: %1

Explanation

An error occurred while loading the JDBC driver classes, which are necessary for SQL I/O.

User Response

Ensure that the JDBC driver classes are specified correctly in the property `vgj.jdbc.drivers`. If more than one is needed, separate their names with a semicolon. Also ensure that the classes can be found somewhere in the classpath.

EGL Java run-time error code VGJ0709E

VGJ0709E: A statement (%1) used a prepared statement that has not been prepared.

Explanation

The prepared statement named in the error message does not exist. Prepared statements are created by calling the EGL prepare statement.

User Response

Correct the program logic by adding a prepare before the prepared statement is used.

EGL Java run-time error code VGJ0710E

VGJ0710E: A %1 statement used a result set that is closed or does not exist.

Explanation

The result set used by the statement cannot be used because it is not open or does not exist.

User Response

Correct the program logic to avoid using invalid result sets.

EGL Java run-time error code VGJ0711E

VGJ0711E: An error occurred while connecting to database %1: %2

Explanation

A connection could not be established to the database named in the message.

User Response

Use the Error part of this message to diagnose and correct the problem.

EGL Java run-time error code VGJ0712E

VGJ0712E: Cannot connect to the default database. The name of the default database was not specified.

Explanation

The name of the default database was not specified, so the program cannot connect to it.

User Response

The name of the default database can be specified in several ways. One of the properties `vgj.jdbc.default.database.programName` (where `programName` is the name of the program) and `vgj.jdbc.default.database` must be set. The value of that property may be the actual name of the default database, or it may be the default database's logical name. When a logical name is used, another property must be set: `vgj.jdbc.database.logicalName`. The value of this property must be the actual name of the default database.

EGL Java run-time error code VGJ0750E

VGJ0750E: The I/O driver for file %1 could not be created. %2

Explanation

A failure occurred during creation of the I/O driver for the specified file. This error can occur at the following times:

- On the first I/O operation for a record that is related to the specified file; or
- On the first access of the system variable resourceAssociation for a record that is related to the specified file.

The end of the message indicates the reason for the failure.

User Response

Review the error message and respond as appropriate.

EGL Java run-time error code VGJ0751E

VGJ0751E: The fileType property for file %1 could not be found in the Java run-time property vgj.ra.fileName.fileType.

Explanation

You need to set the following run-time property to a valid file type:

`vgj.ra.fileName.fileType`

fileName

Name of the file specified in the message. This file name is a logical file name that is associated with an EGL record.

For an MQ record, the value is mq; for a serial record, the value is seqws. The source of the value is the generation-time resource associations part; specifically, the association element for the file, property **fileType**.

User Response

Do as follows:

- Add the run-time fileType property to the run-time properties file or deployment descriptor; or
- Set the fileType value at generation time and regenerate the program:
 - In the file-name-specific association element of the resource associations part, set the property **fileType**
 - In the build descriptor used at generation, set the option **genProperties** to GLOBAL

For other details, see the EGL help pages on the association element, on Java run-time properties, and on setting up the environment.

EGL Java run-time error code VGJ0752E

VGJ0752E: An invalid fileType %1 was specified for file %2 in the resource associations part.

Explanation

You need to set the following run-time property to a valid file type:

`vgj.ra.fileName.fileType`

fileName

Name of the file specified in the message. This file name is a logical file name that is associated with an EGL record.

For an MQ record, the value is mq; for a serial record, the value is seqws. The source of the value is the generation-time resource associations part; specifically, the association element for the file, property **fileType**.

User Response

Do as follows:

- Change the run-time fileType property in the run-time properties file or deployment descriptor; or
- Reset the fileType value at generation time and regenerate the program:
 - In the file-name-specific association element of the resource associations part, change the property **fileType**
 - In the build descriptor used at generation, set the option **genProperties** to GLOBAL

For other details, see the EGL help pages on the association element, on Java run-time properties, and on setting up the environment.

EGL Java run-time error code VGJ0754E

VGJ0754E: The record length item must contain a value that splits non-character data at item boundaries.

Explanation

The record has a variable length. When its data is written out, the record length item indicates how many bytes to write. The last byte of data must be the last byte of an item, unless the item is a char.

User Response

Change the program so that the record length item's value points to the last byte of an item, or falls within a char item.

EGL Java run-time error code VGJ0755E

VGJ0755E: The value in the occursItem or lengthItem is too big.

Explanation

The record has a variable length. An attempt has been made to write out more bytes than the record currently contains.

User Response

Change the program so that the value of the lengthItem or occursItem is within the size of the record.

EGL Java run-time error code VGJ0770E

VGJ0770E: An error occurred while creating the InitialContext or looking up the java:comp/env environment. The error was %1

Explanation

The exception was either thrown from the constructor of javax.naming.InitialContext, or from invoking the lookup method with the value "java:comp/env". The program needs to create the InitialContext object and look up "java:comp/env" in order to access the J2EE environment settings.

User Response

Use the text of the exception and the documentation of your J2EE environment to correct the problem.

EGL Java run-time error code VGJ0800E

VGJ0800E: The assignment of %1 to %2 is invalid.

Explanation

While using the debugger, you attempted to set a system variable to an invalid value.

User Response

Choose a valid value, as described in the help page for the system variable.

EGL Java run-time error code VGJ0801E

VGJ0801E: %1 cannot be modified or does not exist.

Explanation

When using the debugger, you attempted to set the value of a system variable that cannot be set or does not exist.

User Response

Review the help pages for a list of system variables and for a description of each.

EGL Java run-time error code VGJ0802E

VGJ0802E: Error debugging %1: %2

Explanation

An error occurred while attempting to debug a page handler.

User Response

Use the Error part of the message to diagnose and correct the problem.

EGL Java run-time error code VGJ1000E

VGJ1000E: %1 failed. Invoking a method or accessing a field called %2 resulted in an unhandled error. The error message is %3

Explanation

The error occurred in a Java access function. Either an Exception was thrown and the function was not called within a try statement or handleSysLibErrors is 0, or something other than an Exception was thrown, such as an Error.

User Response

Use information in the error message to correct the problem. If some kind of Exception was thrown, change the program logic to handle the error by calling the Java access function within a try statement, or by setting handleSysLibErrors to 1 before invoking the Java access function.

EGL Java run-time error code VGJ1001E

VGJ1001E: %1 failed. %2 is not an identifier, or it is the identifier of a null object.

Explanation

The error occurred in a Java access function. The identifier cannot be used because it does not refer to a non-null object.

User Response

Use an identifier of a non-null object.

EGL Java run-time error code VGJ1002E

VGJ1002E: %1 failed. A public method, field, or class named %2 does not exist or cannot be loaded, or the number or types of parameters are incorrect. The error message is %3

Explanation

The method, field, or class used by a Java access function could not be found.

User Response

Do as follows:

- Make sure the target is a public method, field, or class.
- Make sure the name of the method, field, or class is correct. Class names must be qualified with the name of their package.
- If the problem is a missing class and the name is correct, make sure the directory or archive containing the class is in the Java classpath.

- If the problem is a missing method and the name is correct, make sure the types and number of parameters are correct. Compare the values passed to the Java access function with the values expected by the method.

EGL Java run-time error code VGJ1003E

VGJ1003E: %1 failed. The type of a value in EGL does not match the type expected in Java for %2. The error message is %3

Explanation

The type of a value passed to a Java access function is not correct.

User Response

Values assigned to fields, and parameters passed to methods and constructors, must have the proper type. An exact match is not required as long as the conversion between the types is valid in Java. For example, a subclass may be used instead of its superclass, and a smaller primitive type, such as short, may be used instead of a larger one, such as int.

EGL Java run-time error code VGJ1004E

VGJ1004E: %1 failed. The target is a method that returned null, a method that does not return a value, or a field whose value is null.

Explanation

The Java access function expected the result of the operation to be a non-null object, but did not get one.

User Response

To call a method that may return null or does not return a value, either use `javaStore`; or use the `java system` function and do not assign the result to an item. To get the value of a field that may be null, use `javaStoreField`.

EGL Java run-time error code VGJ1005E

VGJ1005E: %1 failed. The returned value does not match the type of the return item.

Explanation

The value returned by the Java access function cannot be assigned to the return item because of a type mismatch.

User Response

Change the program logic to use a return item of an appropriate type.

EGL Java run-time error code VGJ1006E

VGJ1006E: %1 failed. The class %2 of an argument cast to null could not be loaded. The error message is %3

Explanation

The class of the argument passed to the Java access function could not be found.

User Response

Do as follows:

- Make sure the name of the class is correct. Class names must be qualified with the name of a package.
- If the name is correct, make sure the directory or archive containing the class is in the Java classpath.

EGL Java run-time error code VGJ1007E

VGJ1007E: %1 failed. Could not get information about the method or field named %2, or an attempt was made to set the value of a field declared final. The error message is %3

Explanation

A `SecurityException` or `IllegalAccessException` was thrown while trying to get information about the method or field, or an attempt was made to set the value of a field declared final. Fields declared final cannot be modified.

User Response

Do as follows:

- If the problem happened when setting a value, change the program logic so the code does not try to set the value of a field declared final; alternatively, change the declaration of the field.
- If the problem was access to information, ask a system administrator to update the security policy file of the Java Virtual Machine so that your program has the necessary permission. The administrator probably needs to grant the `ReflectPermission "suppressAccessChecks"`.

EGL Java run-time error code VGJ1008E

VGJ1008E: %1 failed. %2 is an interface or abstract class, so the constructor cannot be called.

Explanation

The constructor of an interface or abstract class cannot be called.

User Response

Change the program logic to call the constructor of a class that is not abstract.

EGL Java run-time error code VGJ1009E

VGJ1009E: %1 failed. The method or field %2 is not static. An identifier must be used instead of a class name.

Explanation

When a method or field is not declared static, it exists only in a specific instance of a class, not the class itself. An identifier of the object must be used in this case.

User Response

Change the program logic to use an identifier instead of a class name.

EGL Java run-time error code VGJ9900E

VGJ9900E: An error has occurred. The error was %1. Unable to load the error description.

Explanation

The program either could not locate or load both the default message class file and the message class file for your locale. One or both of these message class files may be missing or corrupt.

Note: During run time, this message can only be displayed in U.S. English because of the problem in loading message files.

User Response

If you have extracted class files from the file `fda.jar`, verify that the classes you have are at the same release or maintenance level as the classes in that jar file. If you are using older classes, replace them with the correct version. Also, you can reinstall `fda.jar` from EGL.

If the problem persists, do as follows:

1. Record the message number and the message text.

Note: The error message includes the following important information:

- Where the error occurred
 - The type of internal error
2. Record the situation in which this message occurs.
 3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

EGL Java run-time error code VGJ9901E

VGJ9901E: An error has occurred. The error was %1. The message text for %1 could not be found in the message class file %2. The message text for VGJ0002E also could not be found.

Explanation

The message class file does not contain the run-time message for the message ID or for message ID VGJ0002E. The message class file is either corrupt or from a previous release of EGL.

Note: During run time, this message can only be displayed in U.S. English because of the problem in loading message files.

User Response

If you have extracted class files from the file `fda.jar`, verify that the classes you have are at the same release or maintenance level as the classes in that jar file. If you are using older classes, replace them with the correct version. Also, you can reinstall `fda.jar` from EGL.

If the problem persists, do as follows:

1. Record the message number and the message text.

Note: The error message includes the following important information:

- Where the error occurred
 - The type of internal error
2. Record the situation in which this message occurs.
 3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

Appendix. Notices

Note to U.S. Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Lab Director
IBM Canada Ltd. Laboratory
8200 Warden Avenue
Markham, Ontario, Canada L6G 1C7

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to

IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 2000, 2004. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

Programming interface information is intended to help you create application software using this program.

General-use programming interfaces allow you to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and service marks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

- AIX
- AS/400
- CICS
- CICS/ESA
- ClearCase
- Cloudscape
- DB2
- DB2 Connect
- DB2 Universal Database
- Everyplace
- IBM
- ibm.com
- iSeries
- OS/390
- IMS
- Informix
- MQSeries
- MVS
- OS/400

- RACF
- Rational
- Rational Unified Process
- SP2
- Support Pac
- SystemView
- VisualAge
- WebSphere
- z/OS

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Intel is a trademark of Intel Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product or service names, may be trademarks or service marks of others.

Index

A

- add statement 293
- alias callLink element property 445
- alias names
 - COBOL 464
 - Java 465
 - Java wrappers 466
 - overview 463, 464
- alias transfer-related element
 - property 462
- arrays 64
- assignments 296
- associations elements 231
- asynchLink elements
 - description 441
 - package 442
 - recordName 442

B

- basic program parts 479
- basic record parts 491
- batch interface for generation 126, 127, 128
- bidiConversionTable build descriptor
 - option 243
- bidirectional language text
 - conversion 282
- bindings 135
- breakpoints 118
- build descriptor parts
 - adding 92
 - COBOL options 132
 - description 234
 - editing general options 93
 - editing Java run-time properties 93
 - Java options 129
 - master build descriptors 413
 - options, alphabetic list 237
 - setting the default 237
- build files
 - adding import statements 89
 - creating 88, 89, 90
 - description 9
 - format 269
- build parts
 - build descriptor 234, 237
 - linkage options 439
 - resource associations 157
- build paths, EGL
 - editing 91
 - overview 409
- build plans
 - description 392
 - invoking after generation 123
- build project menu option 121
- build scripts
 - delivered with EGL 271
 - description 271
 - predefined symbolic parameters 273

- build scripts (*continued*)
 - required options 271
 - symbolic parameters 272
- build servers
 - description 275
 - starting on AIX, Linux, or Windows 2000/NT/XP 129
- buildPlan build descriptor option 243

C

- call statement 299
- callLink elements
 - alias 445
 - conversionTable 445
 - ctgKeyStore 446
 - ctgKeyStorePassword 446
 - ctgLocation 447
 - ctgPort 447
 - description 443
 - JavaWrapper 447
 - library 448
 - linkType 448
 - location 449
 - luwControl 450
 - package 451
 - parmForm 452
 - pgmName 453
 - providerURL 453
 - refreshScreen 454
 - remoteBind 454
 - remoteComType 455
 - remotePgmType 457
 - serverID 458
 - type 458
- case statement 302
- checkNumericOverflow build descriptor
 - option 243
- checkType build descriptor option 244
- CICSJ2C call setup 213
- cicsj2cTimeout build descriptor
 - option 244
- clientCodeSet build descriptor
 - option 245
- close statement 303
- COBOL alias names 464
- COBOL reserved words 275, 276
- code generation, types 2
- command files 405
- commentLevel build descriptor
 - option 245
- comments 305
- ConnectionFactory, CICSJ2C 213
- constants, references to 34
- content assist
 - description 103
 - using 103
- converse statement 306
- conversion
 - bidirectional language text 282
 - data 279

- conversionTable callLink element
 - property 445
- ctgKeyStore callLink element
 - property 446
- ctgKeyStorePassword callLink element
 - property 446
- ctgLocation callLink element
 - property 447
- ctgPort callLink element property 447
- currencySymbol build descriptor
 - option 245
- curses library, UNIX 228
- cursors, SQL 174, 178

D

- data codes, SQL 200
- data conversion 279
- data conversion system words 518
 - sysLib.connect 543
 - sysLib.convert 518
 - sysVar.callConversionTable 519
 - sysVar.formConversionTable 521
- data initialization 277
- data parts
 - basic record 491
 - dataItem 284
 - dataTable 285, 287
 - indexed record 492
 - MQ record 493
 - relative record 495
 - serial record 497
 - SQL record 498
- database authorization 197
- database connection preferences 188
- dataItem parts
 - description 284
 - EGL source format 284
 - Page Designer 138
- dataTable parts
 - creating 86
 - description 285
 - EGL source format 287
- date and time system words 521
 - sysVar.currentDate 522
 - sysVar.currentFormattedDate 522
 - sysVar.currentFormattedJulianDate 523
 - sysVar.currentFormattedTime 524
 - sysVar.currentJulianDate 524
 - sysVar.currentShortDate 525
 - sysVar.currentShortJulianDate 525
- dbms build descriptor option 246
- debugger, EGL
 - build descriptors 110
 - call statements 111
 - commands 107
 - creating a launch configuration 116
 - invocation from generated code 112
 - overview 107
 - preparing a server 116
 - recommendations 113

- debugger, EGL (*continued*)
 - setting preferences 114
 - SQL database access 111
 - starting a program 115
 - starting a server 117
 - starting a Web session 117
 - stepping through a program 118
 - system type preference 112
 - using breakpoints 118
 - viewing variables 119
- debugTrace build descriptor option 246
- decimalSymbol build descriptor option 246
- default database, SQL 198
- definition files, EGL Web service 147, 148
- delete statement 307
- deployment descriptors
 - setting values 207
 - updating 216, 224
- deployment setup, J2EE
 - ConnectionFactory, CICSJ2C 213
 - descriptor values 207, 216, 224
 - JDBC connections 209
 - run-time environment 206
 - TCP/IP listeners 210
- deployment, Java applications outside of J2EE 228
- destDirectory build descriptor option 246
- destHost build descriptor option 247
- destLibrary build descriptor option 248
- destPassword build descriptor option 248
- destPort build descriptor option 248
- destUserID build descriptor option 249
- development process 1
- directories, generating into 220
- display statement 309
- dynamic arrays 64
- dynamic SQL statements 180

E

- ear files, eliminating duplicate jar files 219, 227
- editors
 - content assist 103
 - EGL 103
 - locating source files 88
 - opening a part 87
 - preferences, EGL 104
- EGL build file format 269
- EGL build paths
 - editing 91
 - overview 409
- EGL command files 405
- EGL debugger
 - breakpoints 118
 - build descriptors 110
 - call statements 111
 - commands 107
 - creating a launch configuration 116
 - invocation from generated code 112
 - overview 107
 - preparing a server 116
 - recommendations 113

- EGL debugger (*continued*)
 - setting preferences 114
 - SQL database access 111
 - starting a program 115
 - starting a server 117
 - starting a Web session 117
 - stepping through a program 118
 - system type preference 112
 - viewing variables 119
- EGL editor
 - content assist 103
 - overview 103
 - preferences 104
- EGL Java run-time error codes 635
- EGL overview 1
- EGL reserved words 290
- EGL run-time code for Java, installing 205
- EGL SDK (EGL Software Development Kit) 128
- EGL source format 292
- EGL Web services
 - creating 143
 - definition files 147, 148
 - overview 139
- EGL_GENERATORS_PLUGINDIR variable 219
- EGLCMD 126, 127, 407
- eglmaster.properties 412
- eglpsh 409
- EGLSDK 410
- EJB projects
 - deployment code generation 227
 - setting the JNDI name 228
- EJB sessions
 - components 393
 - description 393
- eliminateSystemDependentCode build descriptor option 249
- enableJavaWrapperGen build descriptor option 250
- environment files, J2EE
 - description 394
 - updating 224
- exception handling and status system words 526
 - sysLib.displayMsgNum 527
 - sysLib.setError 528
 - sysLib.validationFailed 529
 - sysVar.errorCode 530
 - sysVar.handleHardIOErrors 532
 - sysVar.handleOverflow 532
 - sysVar.handleSysLibErrors 533
 - sysVar.mqConditionCode 534
 - sysVar.overflowIndicator 534
 - sysVar.returnValue 535
 - sysVar.validationMsgNum 536
- exceptions
 - handling of 75
 - I/O error values 418
 - system words 526
 - try blocks 76
- execute statement 309
- exit statement 313
- explicit SQL statements 195, 196
- expressions
 - description 357

- expressions (*continued*)
 - logical 70, 358
 - numeric 70, 364
 - string 70, 365
- externallyDefined transferToTransaction element property 463

F

- field-presentation properties 45
- file and database system words 537
 - recordName.resourceAssociation 539
 - sysLib.commit 541
 - sysLib.connectionService 545
 - sysLib.disconnect 549
 - sysLib.disconnectAll 549
 - sysLib.queryCurrentDatabase 550
 - sysLib.rollback 550
 - sysLib.setCurrentDatabase 551
 - sysVar.commitOnConverse 542
 - sysVar.sqlca 551
 - sysVar.sqlcode 552
 - sysVar.sqlerrd 553
 - sysVar.sqlerrmc 553
 - sysVar.sqlIsolationLevel 554
 - sysVar.sqlState 555
 - sysVar.sqlwarn 555
- files
 - associations with record types 159
 - build 9, 88
 - creating 85, 88
 - EGL command 405
 - J2EE environment 394
 - linkage properties 404, 431
 - program properties 404
 - results 405
 - source 8, 85
 - Web service definition 9
- fillWithNulls build descriptor option 250
- folders, creating 84
- form parts
 - description 366
 - EGL source format 370
 - field-presentation properties 45
 - formatting properties 47
 - print 368
 - text 367
 - validation properties 59
- formatting properties 47
- formGroup parts
 - creating 154
 - description 376
 - EGL source format 377
 - pfKeyEquate property 380
 - use declarations 633
- forward statement 315
- fromPgm transferToProgram element property 460
- function invocations 316
- function parts 380, 381
- functions, Java access 556

G

- genDataTables build descriptor
 - option 250
- genDDSFile build descriptor option 251
- genDirectory build descriptor
 - option 251
- generation
 - batch interface 126, 127, 128
 - COBOL load module 132
 - COBOL options 132
 - COBOL output 388
 - directory target 220
 - EGL command files 126, 127
 - EGL SDK 128
 - EGLCMD 126, 127, 407
 - eglpth 409
 - EGLSDK 127, 410
 - EJB projects, deployment code 227
 - Java options 129
 - Java output 388, 394
 - Java wrappers 131
 - Java wrapper output 389
 - library parts 403
 - output types 386, 387
 - overview 123
 - project target 214
 - Results view 126
 - setting
 - EGL_GENERATORS_PLUGINDIR 219
 - wizard 125
 - workbench 124
- genFormGroup build descriptor
 - option 252
- genHelpFormGroup build descriptor
 - option 252
- genProject build descriptor option 252
- genProperties build descriptor
 - option 254
- get next statement 324
- get previous statement 328
- get statement 318
- goTo statement 332

H

- host variables, SQL 200

I

- I/O error values 418
- if, else statement 332
- implicit SQL statements 193, 194, 195, 196, 197
- import 26
- in operator 416
- indexed record parts 492
- initialization, data 277
- initIORecords build descriptor
 - options 255
- initNonIOData build descriptor
 - options 255
- input forms 486
- input records 487
- installation, EGL run-time code for
 - Java 205

items

- description 43
- properties 40
- properties, page 53
- properties, SQL 57
- structure 505

J

- J2EE build descriptor option 255
- J2EE deployment setup
 - ConnectionFactory, CICSJ2C 213
 - descriptor values 207, 216, 224
 - JDBC connections 209
 - run-time environment 206
 - TCP/IP listeners 210
- J2EE environment files
 - description 394
 - updating 224
- J2EE JDBC connections 209
- jar files, run-time
 - eliminating duplicates from ear files 219, 227
 - providing access to 217, 225
- Java access
 - functions 556
 - system words 557
 - sysLib.java 564
 - sysLib.javaGetField 566
 - sysLib.javaIsNull 568
 - sysLib.javaObjId 569
 - sysLib.javaRemove 570
 - sysLib.javaRemoveAll 571
 - sysLib.javaSetField 572
 - sysLib.javaStore 573
 - sysLib.javaStoreCopy 575
 - sysLib.javaStoreField 576
 - sysLib.javaStoreNew 578
 - sysLib.javaType 579
- Java alias names 465
- Java run-time properties 421, 423
- Java wrappers
 - alias names 466
 - classes 395
 - description 395
 - generating 131
 - generation output 389
 - using 3
- JavaWrapper callLink element
 - property 447
- JDBC connections
 - J2EE 209
 - standard 208
- JNDI name, setting for EJB projects 228
- JSPs 135

K

- keyword statements
 - add 293
 - alphabetic list 72
 - call 299
 - case 302
 - close 303
 - converse 306
 - delete 307

keyword statements (continued)

- display 309
- execute 309
- exit 313
- forward 315
- get 318
- get next 324
- get previous 328
- goTo 332
- if, else 332
- move 333
- MQSeries-related 163
- open 335
- prepare 339
- print 341
- replace 341
- return 344
- set 345
- show 353
- transfer 354
- try 355
- while 356

L

- launch configurations 116
- leftAlign build descriptor option 256
- library callLink element property 448
- library parts
 - creating 87
 - description 433
 - EGL source format 434
 - generated output 403
 - use declarations 632
- linkage build descriptor option 256
- linkage options parts
 - adding 95
 - description 439
 - editing asynchLink elements 97
 - editing callLink elements 96
 - editing transfer-related elements 98
- linkage properties files
 - deploying 223
 - description 404
 - details 431
- linkType callLink element property 448
- linkType transferToProgram element
 - property 460
- location callLink element property 449
- logic parts
 - basic program 479
 - function 380, 381
 - library 433, 434
 - pageHandler 469, 472
 - textUI program 481
- logical expressions 358
- logical unit of work 159
- luwControl callLink element
 - property 450

M

- master build descriptors
 - eglmaster.properties 412
 - overview 413
 - plugin.xml 414

- math build descriptor option 256
- mathematical system words 580
 - mathLib.abs 583
 - mathLib.acos 584
 - mathLib.asin 584
 - mathLib.atan 584
 - mathLib.atan2 585
 - mathLib.ceiling 585
 - mathLib.compareNum 586
 - mathLib.cos 586
 - mathLib.cosh 587
 - mathLib.exp 587
 - mathLib.floatingAssign 587
 - mathLib.floatingDifference 588
 - mathLib.floatingMod 588
 - mathLib.floatingProduct 589
 - mathLib.floatingQuotient 589
 - mathLib.floatingSum 590
 - mathLib.floor 590
 - mathLib.frexp 591
 - mathLib.Idexp 591
 - mathLib.log 591
 - mathLib.log10 592
 - mathLib.maximum 592
 - mathLib.minimum 593
 - mathLib.modf 593
 - mathLib.pow 593
 - mathLib.precision 594
 - mathLib.round 594
 - mathLib.sin 595
 - mathLib.sinh 596
 - mathLib.sqrt 596
 - mathLib.tan 596
 - mathLib.tanh 597
- message queues
 - MQ options records 169
 - MQ record properties 169
 - MQSeries direct calls 165
 - MQSeries support 161
 - MQSeries-related EGL keywords 163
 - remote 165
- miscellaneous system words 610
 - sysLib.bytes 610
 - sysLib.calculateChkDigitMod10 611
 - sysLib.calculateChkDigitMod11 611
 - sysLib.clearScreen 613
 - sysLib.fieldInputLength 613
 - sysLib.getVAGSysType 613
 - sysLib.maximumSize 615
 - sysLib.pageEject 615
 - sysLib.setLocale 615
 - sysLib.setRemoteUser 616
 - sysLib.size 617
 - sysLib.startTransaction 617
 - sysLib.verifyChkDigitMod10 619
 - sysLib.verifyChkDigitMod11 620
 - sysLib.wait 621
 - sysVar.arrayIndex 621
 - sysVar.eventKey 622
 - sysVar.printerAssociation 623
 - sysVar.remoteSystemID 625
 - sysVar.segmentedMode 626
 - sysVar.sessionID 627
 - sysVar.systemType 628
 - sysVar.terminalID 629
 - sysVar.transactionID 629
 - sysVar.transferName 630

- miscellaneous system words *(continued)*
 - sysVar.userID 630
- modified data tags 152
- move statement 333
- MQ record parts
 - EGL source format 493
 - options records 169
 - properties 169
- MQSeries
 - direct calls 165
 - MQ options records 169
 - MQ record properties 169
 - related EGL keywords 163
 - support 161
- multidimensional arrays 64

N

- names
 - aliases 463, 464, 465, 466
 - conventions 468
- nextBuildDescriptor build descriptor
 - option 257
- null 179
- numeric expressions 364

O

- one-dimensional arrays 64
- oneFormItemCopybook build descriptor
 - option 257
- open statement 335
- operators
 - in 416
 - precedence 39
- options for generation
 - COBOL 132
 - Java 129
- output
 - build project menu option 121
 - building 122
 - COBOL generation 388
 - generated types 386, 387
 - Java generation 388, 394
 - Java wrapper generation 389
 - rebuild all menu option 121
 - rebuild project menu option 121

P

- package asynchLink element
 - property 442
- package callLink element property 451
- packages
 - creating 84
 - description 8
 - recommendations for 10
- Page Designer
 - bindings 135
 - data items 138
 - primitive types 138
 - Quick Edit view 137
 - records 139
 - support 135
- page item properties 53

- pageHandler parts
 - description 469
 - EGL source format 472
 - use declarations 634
- parameters, program 483
- parmForm callLink element
 - property 452
- parts
 - description 11
 - opening 87
 - properties 40
 - references to 16
- perspectives 81
- pfKeyEquate property 380
- pgmName callLink element
 - property 453
- plugin.xml 414
- positiveSignIndicator build descriptor
 - option 257
- predefined symbolic parameters 273
- preferences
 - EGL 81
 - EGL debugger 114
 - EGL editor 104
 - source styles 105
 - SQL database connections 188
 - SQL retrieve 190
 - templates 105
- prep build descriptor option 258
- prepare statement 339
- primitive types
 - BIN 32
 - CHAR 29
 - DBCHAR 30
 - DECIMAL 32
 - description 27
 - HEX 30
 - MBCHAR 31
 - NUM 33
 - NUMC 33
 - PACF 34
 - Page Designer 138
 - UNICODE 31
- print forms 368
- print statement 341
- program calls 3
- program parts
 - basic 479
 - COBOL generation 388
 - creating 85
 - description 477
 - EGL source format 478
 - input forms 486
 - input records 487
 - Java generation 388
 - Java program generation 394
 - Java wrapper generation 389
 - non-parameter data 487
 - parameters 483
 - properties 485
 - textUI 481
 - use declarations 632
- program properties files 404
- program transfers 5
- projects
 - creating 82
 - description 7

- projects (*continued*)
 - EJB, deployment code generation 227
 - EJB, JNDI name 228
 - generating into 214
 - specifying database options 83
- properties
 - field-presentation 45
 - formatting 47
 - items 40
 - Java run-time 421, 423
 - MQ record 169
 - page item 53
 - parts 40
 - program parts 485
 - SQL item 57
 - validation 59
 - variable-length records 502
- providerURL callLink element
 - property 453

Q

- Quick Edit view 137

R

- rebuild all menu option 121
- rebuild project menu option 121
- recommendations, development
 - build descriptors 9
 - packages 10
 - part assignment 10
- record internals, SQL 199
- record parts
 - basic 491
 - description 490
 - indexed 492
 - MQ 493
 - Page Designer 139
 - properties, variable-length 502
 - relative 495
 - serial 497
 - SQL 176, 192, 193, 498
- record types
 - associations with file types 159
 - description 13
- recordName asynchLink element
 - property 442
- referencing
 - constants 34
 - parts 16
 - variables 34
- refreshScreen callLink element
 - property 454
- relative record parts 495
- remoteBind callLink element
 - property 454
- remoteComType callLink element
 - property 455
- remotePgmType callLink element
 - property 457
- replace statement 341
- reserved words
 - COBOL 275, 276
 - EGL 290

- reservedWord build descriptor
 - option 259
- resource associations parts
 - adding 99
 - associations elements 231
 - description 157
 - editing 100
- resourceAssociations build descriptor
 - option 259
- result-set processing, SQL 174, 181
- results files 405
- Results view, generation 126
- resultSetID 181
- retrieve feature, SQL 179, 191
- retrieve preferences, SQL 190
- return statement 344
- run units 504
- run-time environment, J2EE setup 206

S

- segmentation
 - text applications 151
- serial record parts 497
- serverCodeSet build descriptor
 - option 259
- serverID callLink element property 458
- sessionBeanID build descriptor
 - option 259
- set statement 345
- setItemFull build descriptor
 - option 261
- show statement 353
- Software Development Kit, EGL (EGL SDK) 127, 128
- source files
 - content assist 103
 - creating 85
 - description 8
 - format 292
 - locating in the Project Navigator 88
- source styles 105
- spacesZero build descriptor option 261
- SQL
 - constructing a PREPARE statement 194
 - creating data item parts 192, 193
 - cursors 174, 178
 - data codes 200
 - database authorization 197
 - database connection preferences 188
 - default database 198
 - dynamic statements 180
 - EGL statements 172
 - examples 181
 - explicit statements 178, 195, 196
 - host variables 200
 - implicit statements 177, 178, 193, 194, 195, 196, 197
 - null 179
 - record internals 199
 - record parts 176
 - result-set processing 174, 181
 - retrieve feature 179, 191, 199
 - retrieve preferences 190
 - support 171, 199
- SQL item properties 57

- SQL record parts 498
- sqlDB build descriptor option 262
- sqlErrorTrace build descriptor
 - option 262
- sqlID build descriptor option 263
- sqlIOTrace build descriptor option 263
- sqlJDBCClass build descriptor
 - option 263
- sqlJNDIName build descriptor
 - option 264
- sqlPassword build descriptor option 264
- sqlValidationConnectionURL build
 - descriptor option 265
- standard JDBC connections 208
- statements
 - assignment 70, 296
 - constant declaration 70
 - function invocation 70, 316
 - keyword 70
 - null 70
 - SQL 172
 - variable declaration 70
- static arrays 64
- string expressions 365
- string handling system words 597
 - strLib.compareStr 599
 - strLib.concatenate 600
 - strLib.concatenateWithSeparator 601
 - strLib.copyStr 602
 - strLib.findStr 603
 - strLib.getNextToken 604
 - strLib.setBlankTerminator 605
 - strLib.setNullTerminator 606
 - strLib.setSubStr 606
 - strLib.strLen 607
- structures 19
- symbolic parameters 272
- syntax diagrams 506
- sysCodes build descriptor option 266
- system build descriptor option 266
- system limits 356
- system words
 - alphabetic list 508
 - data conversion 518
 - date and time 521
 - description 79
 - exception handling and status 526
 - file and database 537
 - Java access 557
 - mathematical 580
 - miscellaneous 610
 - string handling 597
 - Web application 607

T

- targetNLS build descriptor option 267
- TCP/IP listeners 210
- templateDir build descriptor option 267
- templates 105
- text applications
 - formGroup parts 154
 - modified data tags 152
 - segmentation 151
- text forms 367
- textUI program parts 481

- toPgm transfer-related element
 - property 461
- trademarks 713
- transfer of control across programs 74
- transfer statement 354
- transferToProgram elements
 - alias 462
 - description 459
 - fromPgm 460
 - linkType 460
 - toPgm 461
- transferToTransaction elements
 - alias 462
 - description 462
 - externallyDefined 463
 - toPgm 461
- try statement 355
- type callLink element property 458
- type definitions 20
- typedefs 20

Web-application system words (*continued*)

- sysLib.setRequestAttr 609
- sysLib.setSessionAttr 610
- while statement 356
- workbench, generation in 124, 125

U

- UNIX curses library 228
- use declarations 631
- user interface (UI) parts
 - form 366, 370
 - formGroup 376, 377
 - page item properties 53

V

- VAGCompatibility build descriptor
 - option 268
- validateMixedItems build descriptor
 - option 268
- validateOnlyIfModified build descriptor
 - option 269
- validateSQLStatements build descriptor
 - option 269
- validation properties 59
- variables, references to 34
- VisualAge Generator
 - EGL compatibility 276
 - migration from 5
- VSAM
 - access prerequisites 202
 - support 202
 - system names 203

W

- Web applications
 - EGL Web services 139, 143
 - Page Designer 135
 - support 135
- Web service definition files 9
- Web services, EGL
 - creating 143
 - definition files 147, 148
 - overview 139
- Web-application system words 607
 - sysLib.clearRequestAttr 607
 - sysLib.clearSessionAttr 608
 - sysLib.getRequestAttr 608
 - sysLib.getSessionAttr 609



Part Number: 000000000
Program Number: 5724-D46

Printed in USA