IBM ILOG Views
Data Access V5.3
User's Manual

June 2009

© Copyright International Business Machines Corporation 1987, 2009.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Copyright notice

© Copyright International Business Machines Corporation 1987, 2009.

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

Trademarks

IBM, the IBM logo, ibm.com, Websphere, ILOG, the ILOG design, and CPLEX are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Notices

For further information see <*installdir*>/license/notices.txt in the installed product.

Table of Contents

Preface **About This Manual** 11

What You Need To Know. 11

Manual Organization 12

Notation. 13

Typographic Conventions 13

Naming Conventions 13

Related Documentation and Bibliography. 13

IBM ILOG Manuals. 13

C++ Programming Language Publications 14

Database Publications 14

Part I **IBM ILOG Views Data Access Common Framework. . . 15**

Chapter 1 **Introducing Data Access** 17

What is Data Access? 17

Supported Databases 18

Distribution Structure 19

Chapter 2 **Data Access Basics** 21

Overview 21
IBM ILOG Views Interface 22
IlvApplication 22
Containers 23
Gadgets 24
Callbacks 24
Data Access Concepts 25
Values 26
Tables 26
Data Sources 27
Data-Source-Aware Gadgets 27
Integrating with IBM ILOG Views Advanced Graphics 28
Chapter 3 Tables 31
Introduction to Tables 32
One-Tier and Two-Tier Tables 33
The Role of a Table Object 34
Schemas 35
Schema Properties 35
Defining the Schema of a Table Object 37
Managing Rows in a Table 38
Basic Techniques 38
Techniques for Two-Tier Tables 40
Error Catching 41
Changing Error Messages 43
Table Hook 43
Copying and Serializing Table Objects 44
Specialized Table Subclasses 44
IliSQLTable 45
IliMemoryTable 45
IliStringsTable 45
IliMapTable 46

Subclassing IliTable. 46

Guidelines 46

Subclassing Example. 49

Directory Class Example 51

Persistence 52

Table Properties. 53

Scoped Properties 53

Property-Aware Gadgets 54

Chapter 4 Data Sources and Gadgets 57

Data Sources 57

Creating a Data Source Gadget. 58

Connecting Data-Source-Aware Gadgets 59

The Scope of a Data Source 59

Managing Rows in a Data Source 60

Customizing a Data Source 62

Error Handling 66

The Repository. 67

Data-Source-Aware Gadgets. 69

Interface to Data-Source-Aware Gadgets 69

IliTableGadget 72

IliDbField 76

IliEntryField 76

IliTableComboBox 77

IliDbText. 77

IliDbToggle. 78

IliToggleSelector 78

IliDbNavigator. 78

IliDbTimer. 80

IliHTMLReporter. 80

IliXML 80

IliDbPicture. 81

IliDbOptionMenu 81
IliDbStringList 82
IliDbTreeGadget 82
IliChartGraphic 83
IliDbGrapher 85
IliDbGantt 86
Global Callbacks 88

Chapter 5 Handling Values in Data Access 91

The IliValue Class 91
Constructing a Value Object 92
Null Value 92
Data Types 92
Checking the Data Type of an Object 93
Converting a Data Access Data Type to a C++ Type 93
Formatting an IliValue Object 95
Structured Types 96

Chapter 6 Hints and Tips for Using Data Access 99

Working with DbFields in Data Access 99
The Style of a DbField 100
Creating a Form Using the Forms Assistant 103
Foreign Tables 105
Specifying a Foreign Table in IBM ILOG Views Studio 105
Using a Foreign Table to Convert Values 107
Using a Foreign Table to Constrain Values 109
Using the Forms Assistant with Foreign Tables 109
Setting the Table Look 110
Column Geometry 110
Read-Only Settings 111
Fixed Columns 112
Troubleshooting 112

Avoiding Common Names in Foreign Tables. 112
Matching Types with a Foreign Table 113

Part II **Data Access and SQL 115**

Chapter 7 **SQL Tables 117**

Introduction 118
Structural Definition. 118
Creating the Definition Using IBM ILOG Views Studio. 119
Creating the Definition in C++ 120
A Shortcut C++ Definition. 122
The SQL Session of an SQL Table 122
Run-Time Options 123
Concurrency Control 123
Auto-Commit Mode 124
Fetch Policy 125
Auto-Refresh Mode 125
Inserting-Nulls Mode 126
Dynamic-SQL Mode. 126
Bound Variables Mode. 126
Cursor Buffering. 127
Auto-Row Locking Mode 127
Parameters 128
Transaction Manager. 128
Structured Types 131
Asynchronous Mode 134

Chapter 8 **SQL Data Sources 137**

Query Mode 137
Parameters 139
Defining a Parameter 139
Defining a Parameter That Accepts User Input 139

Working with an SQL Data Source . . . 144
Defining Columns . . . 144
Forcing the Name of a Column . . . 145
The Table Primary Key . . . 147
Chapter 9 Connecting to a Database . . . **149**
SQL Sessions and Cursor Objects . . . 149
Connecting to a Database System . . . 150
Cursors . . . 151
Database Drivers . . . 153
The Connect Dialog Box . . . 154
Registered Sessions . . . 154
Part III IBM ILOG Views Data Access Gadgets . . . **157**
Chapter 10 IBM ILOG Views Studio Data Access Gadgets . . . **159**
The Palettes Panel . . . 159
Data Access and SQL Gadgets . . . 160
Charts, Grapher and Gantt Chart Gadgets . . . 162
SQL Tables . . . 163
Notebook Pages Common to Data Access Gadgets Inspectors . . . 165
Callbacks Notebook Page . . . 167
Dialog Boxes Common to Data Access Gadgets Inspectors . . . 168
Font Chooser Dialog Box . . . 168
Color Chooser Dialog Box . . . 169
File Chooser Dialog Box . . . 170
Chapter 11 Data Source Gadgets Reference . . . **171**
IliSQLDataSource . . . 171
IliSQLDataSource Inspector Panel . . . 172
IliSQLDataSource Menus . . . 172
General Elements . . . 174
SELECT Section Notebook Pages . . . 174

Dialog Boxes 184
IliMemoryDataSource 192

Chapter 12 Display Gadgets Reference 201

IliTableGadget 202
Table Gadget Inspector Panel 202
IliDbField 209
DbField Inspector Panel 209
IliEntryField 213
Entry Field Inspector Panel 213
IliTableComboBox 216
Table Combo Box Inspector Panel 216
IliDbText 223
DbText Inspector Panel 224
IliDbToggle 226
DbToggle Inspector Panel 227
IliToggleSelector 231
ToggleSelector Inspector Panel 231
IliDbNavigator 234
DbNavigator Inspector Panel 234
IliDbTimer 237
DbTimer Inspector Panel 237
IliHTMLReporter 238
HTMLReporter Inspector Panel 238
IliXML. 243
XML Inspector Panel 243
IliDbPicture 245
DbPicture Inspector Panel 245
IliDbOptionMenu 247
DbOptionMenu Inspector Panel 247
IliDbStringList. 250
DbStringList Inspector Panel 250

IliDbTreeGadget **256**
DbTreeGadget Inspector Panel 257
IliChartGraphic **265**
ChartGraphic Inspector Panel 265
IliDbGrapher **269**
DbGrapher Inspector Panel 270
IliDbGantt **276**
DbGantt Inspector Panel 276
Appendix A Utility Classes **291**
The IliString Class **291**
The IliDecimal Class **292**
The IliDate Class **292**
The IliFormat Class **293**
The IliInputMask Class **295**
Appendix B Format Syntax **297**
String Formats **297**
Number Formats **298**
Date Formats **300**
Literal Characters **302**
Appendix C Mask Syntax **303**
Placeholders **304**
Predefined Masks **305**
Appendix D Error Messages **307**
Index **309**

About This Manual

Welcome to IBM® ILOG® Views Data Access, referred to as Data Access, a library dedicated to the development of client-server database applications. Data Access is fully integrated with IBM ILOG Views, therefore allowing you to build graphical user interfaces and to link them to data sources to provide intuitive data.

What You Need To Know

The guide assumes that you are familiar with the UNIX® or PC environment in which you are going to use Data Access, including its specific windowing system. Since Data Access is written for C++ developers, this guide also assumes that you can write C++ code and that you are familiar with your C++ development environment so as to manipulate files and directories, use a text editor, and compile and run C++ programs.

Finally, as this product is an add-on to IBM® ILOG® Views Controls, you must be familiar with how to use IBM ILOG Views Controls, and its graphical editor IBM ILOG Views Studio.

Manual Organization

This manual consists of a table of contents, preface, 12 chapters, 4 appendixes and a glossary.

Part I, *IBM ILOG Views Data Access Common Framework*

- Chapter 1, *Introducing Data Access* contains a brief introduction of the product.
- Chapter 2, *Data Access Basics* describes the basic objects in IBM ILOG Views that are necessary to build an application with Data Access. The second part of this chapter is an overview of Data Access and the main objects that are available in the API.
- Chapter 3, *Tables* describes one of the most important objects in Data Access, the table object (`IliTable` class and its subclasses).
- Chapter 4, *Data Sources and Gadgets* describes the data source object (`IliDataSource` and its subclasses) and data source aware gadgets.
- Chapter 5, *Handling Values in Data Access* contains information on how values are handled in Data Access, without having to take into account their actual type until run time. This feature is implemented by the `IliValue` class.
- Chapter 6, *Hints and Tips for Using Data Access* contains some examples of the types of situations that you may encounter when using Data Access (and IBM ILOG Views Studio), and provides you with some useful tips on how to handle them.

Part II, *Data Access and SQL*

- Chapter 7, *SQL Tables* contains more detailed information on one of the most important table subclasses in Data Access, `IliSQLTable`. This is the class used to connect with a relational database management system.
- Chapter 8, *SQL Data Sources* tells you how to define parameters in an SQL table and provides hints on using the SQL data source.
- Chapter 9, *Connecting to a Database* contains information on how Data Access objects are used to implement a connection to a database.

Part III, *IBM ILOG Views Data Access Gadgets*

- Chapter 10, *IBM ILOG Views Studio Data Access Gadgets* introduces the Data Access gadgets found on the Palettes panel.
- Chapter 11, *Data Source Gadgets Reference* describes the two data source creation gadgets: `IliSQLDataSource` and `IliMemoryDataSource`.
- Chapter 12, *Display Gadgets Reference* describes the display gadgets listed in the Data Access menu in the Palettes Panel.

- Appendix A, *Utility Classes* provides information on a few of the useful additional classes in Data Access, `IliString`, `IliDate`, `IliFormat`, and `IliInputMask`.
- Appendix B, *Format Syntax*, describes the syntax used to specify the data format.
- Appendix C, *Mask Syntax*, describes the syntax used to specify the mask format.
- Appendix D, *Error Messages*, describes the error messages.

Notation

Typographic Conventions

The following typographic conventions apply throughout this manual:

- Code extracts and file names are written in `courier` typeface.
- Entries to be made by the user are written in `courier` typeface.
- Some words appear in *italics* when seen for the first time.

Naming Conventions

Throughout the documentation, the following naming conventions apply to the API.

- The names of classes defined in the IBM ILOG Views library begin with `Ilv`, for example `IlvDisplay`.
- The names of classes as well as global functions are written as concatenated words with each initial letter capitalized, for example `IlvGadgetContainer`.

Related Documentation and Bibliography

Certain IBM ILOG manuals can help you get started with Data Access, while various books found in the marketplace can be a good source of information to create SQL database applications.

IBM ILOG Manuals

These IBM ILOG manuals can help you use Data Access:

- To get started with Data Access, see the IBM ILOG Views *Data Access Getting Started* manual.

- The IBM ILOG Views *Data Access Reference Manual* describes the C++ classes for Data Access.
- For more help in using the graphical user interface, see the manual IBM ILOG Views *Studio User's Manual* provided with IBM ILOG Views.
- The IBM ILOG Views *Foundation User's Manual* provides helpful information and numerous examples to help you quickly get proficient in the use of IBM ILOG Views.
- For information on C++ classes of other packages of IBM ILOG Views, refer to the appropriate reference manuals.
- For information on IBM ILOG Views DB Link, see the IBM ILOG Views *DB Link Reference Manual*.

C++ Programming Language Publications

The following books provide information on the C++ programming language:

- Lippman, Stanley B. *C++ Primer*. Reading, MA: Addison-Wesley, 1989.
- Stroustrup, Bjarne. *The C++ Programming Language*. Reading, MA: Addison-Wesley, 1986.
- Stroustrup, Bjarne. *The Design and Evolution of C++*. Reading, MA: Addison-Wesley, 1994.

Database Publications

The following books contain some helpful, general information on databases:

- Date, C.J. *A Guide to the SQL Standard.* Reading, Mass.:Addison Wesley Publishing Company.
- Date, C.J. *An Introduction to Database Systems.* Reading, Mass.:Addison Wesley Publishing Company.

Part I

IBM ILOG Views Data Access Common Framework

This part describes how to use the common features of Data Access, including the Data Access basics, the use of table objects in Data Access, data sources and gadgets, and the handling of values in Data Access. It also provides some hints and tips for using Data Access.

C H A P T E R

1

Introducing Data Access

This chapter introduces you to the Data Access package of IBM® ILOG® Views Studio. You can find information on the following topics:

- *What is Data Access?*
- *Supported Databases*
- *Distribution Structure*

What is Data Access?

Data Access is a visual environment for graphic-intensive database applications. Using IBM® ILOG® Views, it lets you create graphical business objects and link them to data sources to provide intuitive data access.

Data Access is organized as a set of C++ class libraries. These classes are to be used in conjunction with the IBM ILOG Views C++ class libraries. Data Access is also accompanied by a schema editor (SQL Schema Editor).

Libraries

The IBM ILOG Views libraries provide the API needed to implement the graphical part of your application. IBM ILOG Views handles the drawing and management of gadgets and

graphics. The Data Access libraries provide the additional functionality required to handle data from an external data source.

For more information on the IBM ILOG Views class libraries, refer to the appropriate IBM ILOG Views *Reference Manual*.

Editors

The graphical editor provided with IBM ILOG Views, called IBM ILOG Views Studio, is a powerful editor that enables you to build a portable graphical user interface. IBM ILOG Views Studio allows you to construct your interface from predefined "gadgets"—that is, buttons, scroll bars, menus, and other interface objects—using simple drag-and-drop operations, while generating C++ code for you to program your application. IBM ILOG Views Studio is fully documented in the IBM ILOG Views *Studio User's Manual*.

The graphical editor provided with Data Access is based on IBM ILOG Views Studio, but has been adapted to be used with Data Access. This editor is referred to as IBM ILOG Views Studio and contains an additional "Data Access" palette, which contains all those predefined gadgets that may interact with an external source of data. There is also a special interface that allows you to set up the connection with the external data source, in a simple graphical way.

Also included with Data Access is the SQL Schema Editor. This editor is provided should you need a simple editor to create tables in a database.

The schema is the table-form structure in which the data is stored. The schema editor is therefore used to edit the table definitions and the data. The schema editor is also used to drop a table in a database. This editor is located into the SQL Tables palette from the Data Access palette.

Supported Databases

You can use Data Access with the following databases:

- Oracle
- Informix
- Sybase
- OLE DB (only for Windows)
- ODBC (only for Windows)
- Microsoft SQL Server (only for Windows)
- DB2

Note: Check the compatibility of your particular database version directly with IBM support.

Distribution Structure

When Data Access is installed on your machine, several directories are created, some of them accompanied with a dedicated README file that you are advised to read. The following main directories are created:

- **`bin`** and its subdirectories, provide some basic tools (IBM ILOG Views Studio with static Data Access libraries, IBM ILOG Views Prototype Studio with static Data Access libraries and other tools). In the directory of each tool you will find <systems> and <database> directories for your specific target systems.
- **`data`** and its subdirectories provide panel description files (suffixed `.ilv`) used by the delivered Data Access samples and editors, as well as the message description files. You should avoid modifying them.
- **`include`** and its subdirectories provide all Data Access class header files.
- **`inform30`** and its subdirectories provide the compatibility with InForm 3.0.
- **`lib`** and its subdirectories contain the Data Access libraries.
- **`samples`** and its subdirectories provide sample coding to let you see particular aspects of the classes provided by Data Access.
- **`studio`** and its subdirectories contain the Data Access libraries for Studio.

C H A P T E R

2

Data Access Basics

This chapter briefly describes the IBM® ILOG® Views objects that are required to create a basic Data Access application. It then continues with an overview of the basic concepts of Data Access and the C++ classes that are provided in the Data Access API.

You can find information on the following topics:

- *Overview*
- *IBM ILOG Views Interface*
- *Data Access Concepts*

Overview

Data Access is a library dedicated primarily to the development of client-server database applications. These applications generally consist of forms, which contain a set of fields (text fields, check boxes, and so on). The values shown in these fields are the result of a mapping between the fields and the data from an external data system.

The mapping between the fields in the graphical user interface and the external data system is bidirectional: data can be retrieved from the database and displayed in the fields, and can be modified by the user and updated in the external data source for long term storage.

The gadgets that enable you to build the user interface for your application in Data Access are provided by IBM® ILOG® Views. Some of these gadgets are the same as in IBM ILOG Views, whereas others have been slightly modified to enable them to connect to an external data source.

Data Access applications can be programmed either in C++ or in IBM ILOG Script, an IBM ILOG implementation of the JavaScript language. This user's manual shows how things can be done in C++.

IBM ILOG Views Interface

Data Access is an add-on to IBM® ILOG® Views so, therefore, the complete functionality of the IBM ILOG Views API is available to users of Data Access. Some IBM ILOG Views classes are essential to an application created in Data Access. These are briefly described in this section. For more information, refer to the IBM ILOG Views documentation.

IlvDisplay

Any application that is constructed using Data Access must have an `IlvDisplay` object before anything else can be created. This object manages all aspects of the communication with the display system (such as drawing primitives, event handling, and so on).

The following code sample shows how a display object can be created:

```
// --- Display ---
int main (int argc, char* argv[]) {
  ...
  IlvDisplay* display = new IlvDisplay("sample", "", argc, argv);
  ...
}
```

Note that when IBM ILOG Views Studio generates source code for your application it will create an `IlvApplication` object, instead of creating a display object in this way. This application object then creates the display.

IlvApplication

Applications built with IBM ILOG Views Studio contain an instance of a subclass of the `IlvApplication` class. This class manages the creation of the `IlvDisplay` object along with the creation of the application panels (that is, containers). Assuming the name of the

application is “MyApp”, the following is a sample of the code that IBM ILOG Views Studio would generate:

```
class MyApp: public IlvApplication {
     ...
     virtual void makePanels();
     ...
  };
  void MyApp::makePanels() {
     // Create all the panels defined in the Application.
     ...
  }
  int main (int argc, char* argv[]) {
     ...
     MyApp* appli = new MyApp("myapp", 0, argc, argv);
     ...
  }
```

Containers

Applications interact with the end user through windows that appear on the computer screen. A container is a window that may hold a given number of graphic objects (such as charts, gauges, buttons, and so on). Most of the interaction between an application and the end user takes place through containers and the graphic objects they contain.

> ***Note:*** *In IBM ILOG Views Studio, containers are referred to as panels.*

Data Access typically uses the IBM ILOG Views `IlvGadgetContainer` class as a base class for the panels of the application.

Three different techniques can be used to set up a container. You can:

1. Code completely in C++.
   - Create the container.
   - Create the graphic objects.
   - Put the graphic objects into the container.
   - Set their positions and any other properties as required (font, color, and so on).

   Since this technique requires the most coding, it is seldom used except in situations where a great deal of flexibility is required (such as creating graphic objects that depend on run-time information).

2. Design the panel using IBM ILOG Views Studio and save it in an `.ilv` data file. Then create the container by coding in C++ and initialize it by reading the `.ilv` data file. Although this technique is more convenient than the previous one, it still has one

shortcoming: in order to manipulate a graphic object in a container through coding, you need to call the `IlvContainer::getObject` member function and cast its result into the appropriate type.

3. Design the panel using IBM ILOG Views Studio and generate the source code for the corresponding panel class. This technique combines the benefits of the previous technique with those gained from the fact that IBM ILOG Views Studio generates a custom subclass of the `IlvGadgetContainer` class that corresponds to the panel being generated. This panel class will have appropriate member functions to retrieve the objects contained in the panel and will define virtual member functions to handle callbacks.

Note that these techniques can be combined. For example, it is possible to design a panel with IBM ILOG Views Studio, generate its source code, and then, at run time, create additional objects and put them into the container.

For more information regarding the code generated by IBM ILOG Views Studio, refer to the IBM ILOG Views *Gadgets - User's Manual*.

Gadgets

Among the graphic objects that can be used in containers, one category of graphic objects is especially relevant for Data Access, namely gadgets.

All gadget classes inherit from the `IlvGadget` class, which is a subclass of the `IlvGraphic` class. Gadgets are specially designed graphic objects that are used to build data entry forms.

IBM® ILOG® Views provides a variety of gadgets (text fields, buttons, menu bars, and so on) for creating objects in your graphical user interface. These gadgets can be accessed in the IBM ILOG Views Studio Palettes panel. See Chapter 1 of the IBM ILOG Views *Data Access Getting Started.*

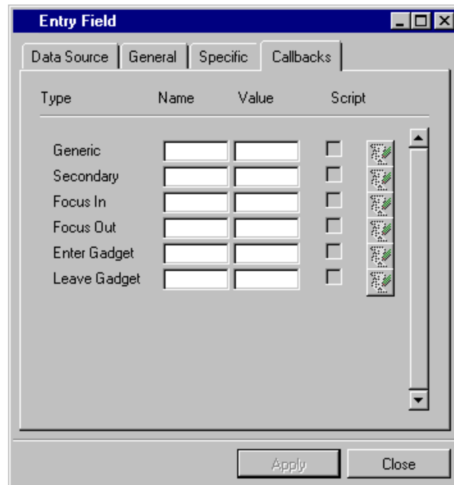
Data Access provides a certain number of additional gadgets that are designed to facilitate the seamless integration of different types of data sources with graphical user interfaces. These gadgets are generally referred to as *data-source-aware* gadgets. Data-source-aware gadgets are described in more detail later. See *Data-Source-Aware Gadgets* on page 69.

Callbacks

The behavior of a gadget can be customized by defining a callback function and attaching this function to one of the callback types that the gadget is able to trigger.

At the bottom of the IBM ILOG Views Studio main window, you will find a callback field that lets you define the *primary callback* of the selected gadget. There are, however, other callback types available with certain gadget classes. These other callback types can generally be defined through the last page of the various gadget inspectors. (The only

exception is the SQL Data Source inspector where the callbacks panel is accessed through the Callbacks item in the Tools menu.)



Note that a check box lets you choose whether the callback is coded in IBM ILOG Script or C++.

The callback types that are supported by each gadget class are described in the IBM ILOG Views and *Data Access Reference Manuals*. Additional information for classes specific to Data Access can be found in the rest of this manual.

Data Access Concepts

This section summarizes the fundamental concepts of Data Access. The information in this introduction is brief. Each section, however, contains a reference to a later chapter where you will find more detailed information.

You can find information on the following topics:

- *Values*
- *Database Connection*
- *Tables*
- *Data Sources*
- *Data-Source-Aware Gadgets*
- *Formats*

- *Masks*
- *Integrating with IBM ILOG Views Advanced Graphics*

Values

The C++ language provides different types that can be used to represent values (for example, `int`, `double`, `char*`, and so on). Data Access can deal with data in a uniform way, independently of its type. It uses the `IliValue` class to represent data whose real type is not known at compile time. See Chapter 5, *Handling Values in Data Access*.

The `IliDatatype` class defines objects that are used to represent the dynamic type of an `IliValue` object.

Database Connection

All communication between a Data Access program and a remote database system goes through the `IliSQLSession` and `IliSQLCursor` classes. These classes provide a high-level interface to all the database access functions needed by Data Access. They are themselves implemented with the IBM ILOG Views DB Link library. See Chapter 9, *Connecting to a Database*.

Tables

The `IliTable` abstract class defines an object that resembles a table. See Chapter 3, *Tables*.

A table is a data structure that is defined by an ordered collection of columns. Each column has a name, a data type, and other properties that define the way values in the column should be handled in Data Access. The ordered collection of columns of a table is known as the table *schema*.

Once the schema of a table is defined, the table can manage a set of rows, each row being an ordered collection of values. The values in a row conform to the data types of the corresponding columns in the schema.

The `IliTable` is abstract in the sense that although it provides the *interface* needed to manipulate the rows in a table, it does not, itself, provide any useful implementation for this interface.

Instead, more specialized subclasses of `IliTable` provide implementations that are specific to different types of data stores. For instance, the `IliSQLTable` class manages rows that are located in a remote relational database server. See Chapter 7, *SQL Tables*. The `IliMemoryTable` class manages rows that are located in the process memory space.

The Data Access API also enables you to define new types of table objects that can deal with other types of data stores or data feeds.

The `IliSQLTable` class (and its instances) should not be confused with the tables that are located in a remote database server. These two entities are referred to as *table objects* and *database tables,* respectively. The `IliSQLTable` class represents Data Access objects that are located in the process address space and that serve as a bridge to tables (or SQL queries) that are located (and executed) remotely in a database system.

Data Sources

Generally speaking, a data source indicates a particular source of information, such as a data feed or a database system.

Data Access provides an `IliDataSource` gadget class. This class serves as a bridge between the `IliTable` class and the *data-source-aware* gadget classes. Data-source-aware gadgets are dedicated to handling user input and displaying data in different styles. They are described in the next section.

From now on in this manual, the term “data source” is used to refer to instances of the `IliDataSource` class (or one of its subclasses). This should not be confused with the general meaning of this term.

Although the `IliDataSource` class is a gadget class, its instances are not visible to the user of an Data Access application. They are, however, visible in IBM ILOG Views Studio so that they can be edited.

Like other gadgets, a data source has a name and supports a set of callback types through which it is possible to customize its behavior for specific needs, such as business rules, and so on. See Chapter 4, *Data Sources and Gadgets*.

Data-Source-Aware Gadgets

In Data Access, you will find a set of gadget classes that can be seamlessly integrated with the data sources described in the previous section. These gadgets are known as *data-source-aware gadgets*.

A data-source-aware gadget class is a class that inherits through multiple inheritance paths from both the `IlvGadget` class (or one of its subclasses) and the `IliFieldItf` class.

The `IliFieldItf` class defines the interface common to all data-source-aware gadgets. It has member functions that deal with such operations as connecting to a data source, querying or changing the value of the gadget, and so on.

For instance, the `IliEntryField` class is the data-source-aware version of the IBM ILOG Views `IlvTextField` class. It has two superclasses, `IlvTextField` and `IliFieldItf`. See Chapter 4, *Data Sources and Gadgets*.

Formats

The IliFormat class is used to format values according to different rules. For instance, it is possible to define a format that specifies that floating-point numbers should be displayed with three digits after the decimal point. See the description of the IliFormat class in Appendix A, *Utility Classes*.

Once a format has been defined, it can be used in C++ code to convert an IliValue object into a character-string representation. Alternatively, it can be used to configure a data-source-aware gadget so that the values displayed in the gadget are formatted according to a predefined format.

Formats are defined using the format specification language described in Appendix B, *Format Syntax*.

Masks

The IliInputMask class is similar to the IliFormat class except that it also manages user input in addition to the format. Masks can be used to:

- Check application-defined constraints on the values entered by the end user.
- Permit the end user to enter values according to customized syntax.

A specification language lets you specify masks from within IBM ILOG Views Studio. This format specification language for masks is described in Appendix C.

It is possible to specify a mask in C++ instead of using the specification language. The IliInputMaskIpl class needs to be subclassed for this purpose.

Integrating with IBM ILOG Views Advanced Graphics

One of the interesting features of Data Access is that its complete integration with IBM ILOG Views enables the creation of customized graphic objects that are linked to data originating from an Data Access data source. The way in which this can be set up is shown in the following figure:

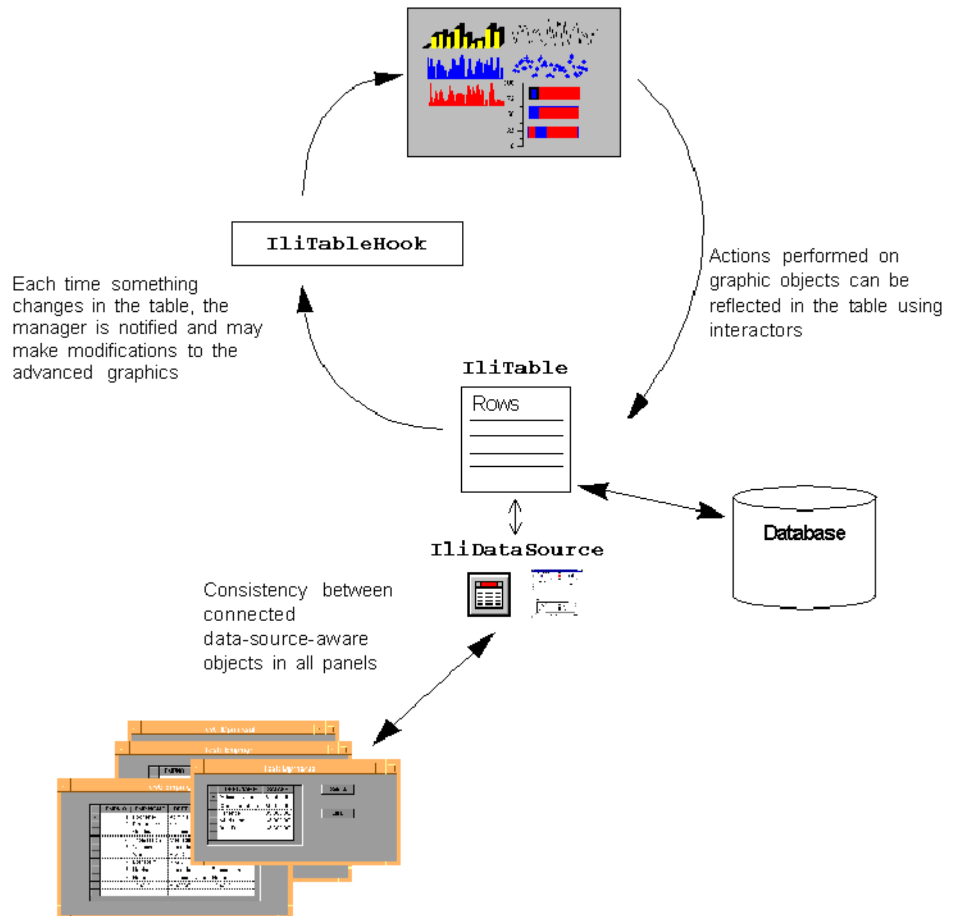


Figure 2.1 Data Access with IBM ILOG Views Advanced Graphics

C H A P T E R

3

Tables

This chapter describes the most important object in Data Access, the table object. The table is created and manipulated via the `IliTable` class and its subclasses. This chapter also discusses the `IliSchema` class and its relationship to table objects.

You can find information on the following topics:

- *Introduction to Tables*
- *One-Tier and Two-Tier Tables*
- *The Role of a Table Object*
- *Schemas*
- *Managing Rows in a Table*
- *Table Hook*
- *Copying and Serializing Table Objects*
- *Specialized Table Subclasses*
- *Subclassing IliTable*
- *Table Properties*

Introduction to Tables

The IliTable class plays a central role within Data Access since it serves two fundamental purposes:

- **Modelling**

  Tables are used as a structuring tool for user interface intensive applications. Before tables, user interfaces were designed as a set of unrelated entry fields and the relationship between these fields and the application data was coded in a programming language. Now, the data model of the application can be graphically defined in terms of tables, with gadgets in the panels being connected to table columns.

- **Connectivity**

  Data can be seamlessly exchanged with external data stores using specialized subclasses of the IliTable class. For instance, the IliSQLTable class is dedicated to data exchange with a relational database system.

> ***Note:*** *In database terminology, the term "table" designates a data structure that is stored in and managed by a database system. From the graphical user interface point of view, the term "table" designates a graphic object (or gadget). In this document, the terms "database table" and "table gadget" are used to refer to these two different table types. The term "table" or "table object" will be used to designate an instance of a subclass of the* IliTable *class.*

A table is implemented by the IliTable class in Data Access. This class is an abstract class that defines objects capable of managing a collection of rows. Each table has a schema that is defined by the IliSchema class, from which the IliTable class inherits. The rows that a table manages must conform to its schema.

As explained in Chapter 2, *Data Access Basics*, the IliTable class defines an interface for managing rows in a table but it does not provide any useful implementation for this interface. Instead, Data Access provides a set of subclasses of the IliTable class that implement this interface with a specific storage policy.

The IliTable class defines a protocol for creating, editing, and inspecting a table object, and is designed to be subclassed. The subclasses of IliTable (that is, IliSQLTable, IliMemoryTable, IliStringsTable, and IliMapTable) implement this protocol for the different types of data sources with which they are associated.

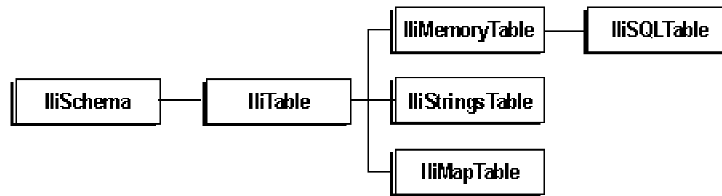


Figure 3.1 *The* `IliSchema` *Hierarchy*

One-Tier and Two-Tier Tables

The subclasses of `IliTable` manage tables in different ways. `IliMemoryTable` and `IliStringsTable` manage rows in the process memory space. These tables are called *one-tier* tables. Only the local process is involved to manage these types of tables.

Other classes such as `IliSQLTable`, however, manage rows that are located in a remote database. These types of tables are referred to as *two-tier* tables since they interact with another process (in this case, the database server).

In the case of two-tier tables, a local row cache is implemented, which stores copies of some of the remote rows which the table is tied to, at any particular time. This cache reduces the communication overhead with the remote database engine. It also provides random access to the rows, even if the database system lacks this capability.

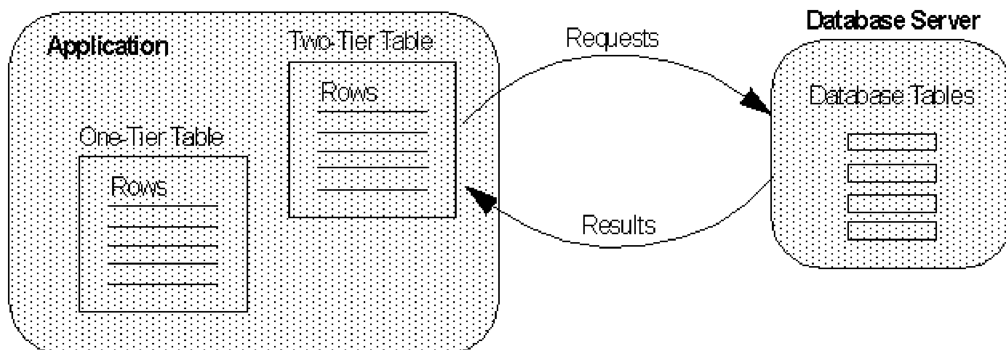


Figure 3.2 *One-Tier and Two-Tier Tables*

The Role of a Table Object

The primary role of a table object is to manage data. The data is managed in the form of rows of values. But before a table can manage rows, it needs to be properly defined.

The way in which a table object is defined depends on its class. For instance, an `IliSQLTable` object needs to know its own schema and how this schema relates to the schema of the associated relational database. This contrasts with `IliMemoryTable` objects that do not need any information other than their own schema.

A table object can be defined either by coding in C++ through the API or interactively by using an appropriate inspector in IBM® ILOG® Views Studio.

> ***Note:*** *In IBM ILOG Views Studio, inspectors are only available for the* `IliSQLTable` *and* `IliMemoryTable` *classes.*

Once defined, a table object can be used directly by coding its member functions in C++ to inspect, add to, or modify its rows. However, it is mainly used by being attached to a data source gadget that will manage it on behalf of the end user. A data source gadget will usually have one or more gadgets (such as table gadgets, entry fields, combo boxes, and so on) connected to it.

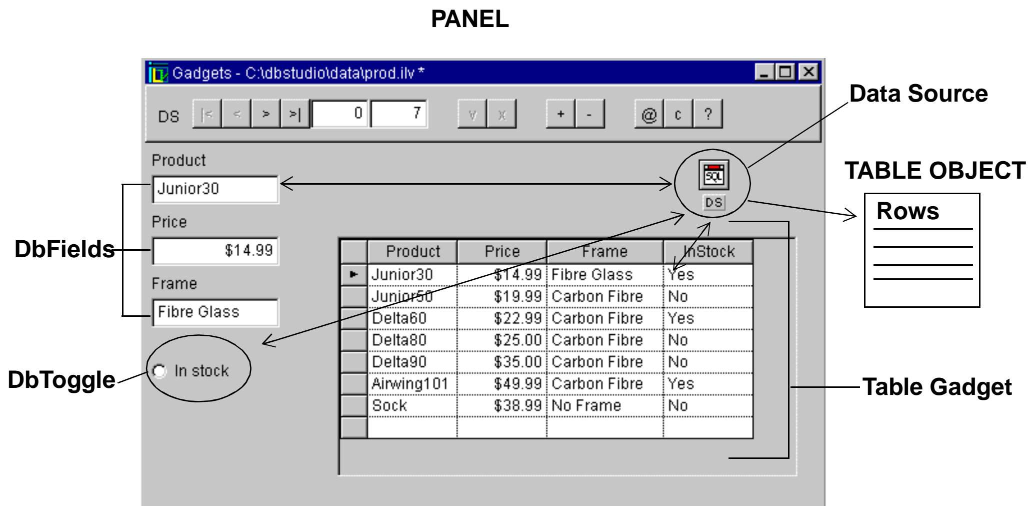


Figure 3.3 *The Links Between a Data Source and Data-Source-Aware Gadgets and a Table Object*

Schemas

This section describes the properties of a table that relate to its columns. These properties are defined by the `IliSchema` and the `IliTable` objects.

You can find information on the following topics:

- *Schema Properties*
- *Defining the Schema of a Table Object*

Schema Properties

The schema of a table is an ordered collection of columns. Most of the properties relating to the schema of a table are defined by the `IliSchema` class, from which the `IliTable` class inherits. However, the "mapping" properties are defined by the `IliTable` class. Each column in a schema has the following properties:

Identification

- Index — Indicates the position of the column within the schema (starting from `0`). Note that the index of a column may change when other columns are inserted or removed from the schema.
- Name — Allows other components of the Data Access library (such as data-source-aware gadgets) to refer to a column by its name.
- Token — Is a "magic cookie" (an `IlInt`) that is assigned to the column at creation time. It is guaranteed to remain constant even across program executions and is unique among all the columns that belong to a given schema. It is used mostly by subclasses of the `IliTable` class that need to identify columns independently of name or index changes. This property is not accessible from within IBM ILOG Views Studio.

Column Type

- Datatype — Specifies the data type of all values in this column.
- Maximum Length — Applies only when the data type of the column assumes values of varying size (typically, the `IliStringType` data type).
- Nullable — Specifies whether a column allows null values or not.
- Part Of Key — Specifies whether the column belongs to the primary key of the table or not. The primary key of a schema is a subset of the columns such that the table will reject any update or insertion that would result in the table having two rows whose values are equal over the columns belonging to the primary key. In other words, the primary key is a set of columns that can serve to identify rows in the table uniquely.

- Default Value — Indicates a value that will be displayed when the user inserts a new row in the table.

Look

- Format — Specifies the format that will be used to display values. See the `IliFormat` class.
- Mask — Specifies the mask used to enter values. See the `IliInputMask` class.
- Alignment — Specifies how values in this column will be displayed. Usually, character string values are left-aligned and date and numeric values are right-aligned.
- Display Width — Indicates the width in pixels of the column when it is displayed.
- Visibility — Indicates whether a column is visible to the end users. Note that in this case the column and the values it contains can still be accessed by the API.
- Title — Specifies the caption of the column when it is displayed in a table gadget. By default, the name of the column is used.
- Label — Is the caption of the column when it is displayed in a `DbField` gadget. By default, the name of the column is used.
- Read Only — Specifies whether the column is read-only.

For more information on how the look of a column applies to gadgets that are connected to it, see *Setting the Table Look* on page 110.

Mapping

A column can be mapped onto a column that belongs to another table. This table is referred to as the *foreign table*. See *Foreign Tables* on page 105. In this situation, when the column is displayed, the value shown is not the original column value. Instead, a value from the foreign table is displayed. The foreign table is therefore used as if it were a dictionary. In addition, the user can modify the column value by selecting a value from a pull-down menu that contains a list of possible values. The foreign table provides the domain of values for the column.

The properties relating to the mapping of a column are defined by the `IliTable` class.

There are two ways in which a foreign table can be defined: either the name of a data source or the name of a table object can be specified. The latter is an API-only option.

- Foreign Data Source Name — Specifies the name of the data source from which the foreign table is obtained.
- Foreign Table — Indicates the foreign table (this property is not accessible from within IBM ILOG Views Studio).
- Value Column — Indicates the name of the column in the foreign table that defines the domain.

- Display Column — Indicates the name of the column in the foreign table that will be displayed in place of the original column value.
- Constrained — Indicates whether the column rejects any values that do not belong to the value column of the foreign table.
- Completion — Indicates whether any incomplete user input will be automatically completed by the gadget on validation, when the column is being edited in a table combo box.

Defining the Schema of a Table Object

This section shows how a table can be created and its schema defined. Since the `IliTable` class is abstract and therefore cannot be instantiated, the `IliMemoryTable` class is used in the example below.

The `IliMemoryTable` class implements the table interface by storing rows in the process memory space. Therefore, this class is suitable for transient tables that do not retain their states across program executions.

A number of member functions let you access or modify the schema of a table. Most of these can be found in the description of the `IliSchema` class. See IBM ILOG Views *Data Access Reference Manual*. The `IliTable` class defines those member functions that deal specifically with the mapping of columns.

Here is a list of some of the schema member functions :

```
class IliSchema {
   ...
   IlInt getColumnsCount() const;
   const char* getColumnName(IlInt colno) const;
   const IliDatatype* getColumnType(IlInt colno) const;
   IlBoolean isColumnPartOfKey(IlInt colno) const;
   void setColumnPartOfKey (IlInt colno, IlBoolean partOfKey);
   IlBoolean isColumnNullable(IlInt colno) const;
   void setColumnNullable(IlInt colno, IlBoolean nullable);
   IlBoolean insertColumn(IlInt colno,
                          const char* colname,
                          const IliDatatype* type,
                          IlInt maxlen = -1);
};
```

The following code shows how a memory table can be defined:

```
enum ColumnTag { IdColumn, NameColumn, SalaryColumn };
  IlvDisplay* display;
  ...
  IliMemoryTable* tbl = new IliMemoryTable(display);
  tbl->lock();
  tbl->insertColumn(IdColumn, "Id", IliIntegerType);
  tbl->insertColumn(NameColumn, "Name", IliStringType);
  tbl->insertColumn(SalaryColumn, "Salary", IliDoubleType);
  tbl->setColumnPartOfKey(IdColumn, IlvTrue);
  tbl->unLock();
```

In this example, a memory table is created and its schema is defined. The schema has three columns, one of which serves as a key for the table. The `IliSchema::insertColumn` and `IliSchema::setColumnPartOfKey` member functions are used.

Note that the `IliMemoryTable` class like all classes derived from the `IliSchema` class is reference counted. This means that it is necessary to lock instances of these classes when they are used. An instance of these classes is implicitly deleted when its reference count reaches 0.

The next section contains information on how to access or edit the rows of a table. The `IliTableBuffer` class is the main class required to carry out these actions.

Managing Rows in a Table

The `IliTable` class provides a set of virtual member functions that can be used to manage the rows of a table. Basic row management techniques that apply to all table types are described first. Then, the special case of two-tier tables is described.

You can find information on the following topics:

- *Basic Techniques*
- *Techniques for Two-Tier Tables*
- *Error Catching*
- *Changing Error Messages*

Basic Techniques

The main operations that can be performed on a row are:

- Insert a row (see `IliTable::insertRow`)
- Modify a row (see `IliTable::updateRow`)
- Delete a row (see `IliTable::deleteRow`)

◆ Inspect a row (see `IliTable::getValue` and `IliTableBuffer::rowToBuffer`)

These member functions do not give access to the actual row implementation. Instead, they make use of the `IliTableBuffer` class that can store a copy of the values in a row.

This means that any updates to a table are always carried out on a complete row. The user edits one complete row at a time, therefore avoiding — in the case of `IliSQLTable` objects — time consuming network activity by validating changes in a complete row.

In addition, it avoids any problems with data coherency that the end user may have. The data in individual columns in a row may apply constraints to each other. For example, having a particular value in one column may limit the values allowed in another column. If the end user must validate a row one column at a time, this problem may occur. When a whole row is validated at one time, the problem is avoided.

A table buffer is created using the `IliTable::getBuffer` method. The table buffer must be released using the `IliTable::releaseBuffer` method when it is no longer needed.

Note that rows are identified by their position within the table object. This is represented by an integer, starting with zero for the first row.

The following code sample shows how to insert a new row into the memory table created in the previous example:

```
IliTableBuffer* buf = tbl->getBuffer();
buf->at(IdColumn).importInteger(1);
buf->at(NameColumn).importString("Smith");
buf->at(SalaryColumn).importDouble(255.00);
if (tbl->appendRow(buf) < 0) {
  IlvPrint("Append row failed");
}
tbl->releaseBuffer(buf);
```

The `IliTableBuffer::at` member function returns a reference to an `IliValue` object that stores the value of the column, whose index is given. The `IliValue::importInteger` and `IliValue::importString` member functions are then used to assign the `id` and `name` to the buffer values. Finally, `IliTable::appendRow` is called to insert a new row into the table.

The following code sample shows how a row in a table can be modified:

```
IliTableBuffer* buf = tbl->getBuffer();
IlInt rowno = 10;
if (!buf->rowToBuffer(rowno)) {
   IlvPrint("Invalid row number : %ld", (long)rowno);
}
IlDouble salary = buf->at(SalaryColumn).asDouble();
buf->at(SalaryColumn).importDouble(salary * 1.1);
if (!tbl->updateRow(rowno, buf)) {
   IlvPrint("Update row failed");
}
tbl->releaseBuffer(buf);
```

The `IliTableBuffer::rowToBuffer` member function copies the row at the specified index into the buffer.

The following code sample shows how a row can be removed from a table by calling the `IliTable::deleteRow` member function:

```
IlInt rowno = 15;
tbl->deleteRow(rowno);
```

Additional member functions are provided in the `IliTable` class that allow you to move a row in a table, sort a table, or search for a given value in a table. For more information, refer to the `moveRow`, `findRow`, `findFirstRow`, and `sortRows` member functions in the `IliTable` class documented in the IBM ILOG Views *Data Access Reference Manual*.

Techniques for Two-Tier Tables

The `IliTable` class also provides a set of member functions dedicated to managing two-tier tables. A two-tier table is tied to rows that are managed by some external process or system; therefore serving as a bridge between Data Access and this external system. For instance, the `IliSQLTable` class (described in Chapter 6, *Hints and Tips for Using Data Access*) serves as a bridge to relational database systems. The `DirectoryTable` sample class serves as a bridge to the file system. See *Subclassing Example* on page 49.

A two-tier table is usually defined by specifying some sort of criteria that will be used to identify a result set extracted from the remote or external system. Precisely how this is done depends on the subclass of `IliTable` being used. For an `IliSQLTable`, for example, an SQL SELECT statement has to be specified.

The `IliTable::select` member function retrieves the data from the external system identified by the above mentioned criteria. This data is then copied into a local row cache managed by the table object.

With a two-tier table, the basic row management member functions (described in section *Managing Rows in a Table* on page 38) perform specific actions, the details of which depend on the specific subclass being used. For more information, refer to the appropriate class in the IBM ILOG Views *Data Access Reference Manual*.

A two-tier table can retrieve rows in one of the following ways:

- The `select` member function immediately retrieves all rows identified by the selection criteria and stores them in the local row cache.
- The `select` member function locates the rows identified by the selection criteria in the external system, but delays their retrieval until they are required.

The `IliTable::getRowsCount` member function returns the number of rows that are located in the local row cache. However, when a two-tier table implements the delayed row retrieval, the value returned by `IliTable::getRowsCount` corresponds to the number of

rows that have been retrieved to date. It does not take into account those rows that have not been retrieved yet.

The `IliTable::fetchCompleted` member function returns `true` when all rows have been retrieved and stored in the local row cache. This member function can therefore be used to check that the row count (as returned by `getRowsCount`) is definitive.

The `IliTable::fetchNext` and `IliTable::fetchAll` functions are used to retrieve explicitly a fixed number of rows or all remaining rows from the result set. However, when any of the `insertRow`, `updateRow`, `deleteRow`, and `getValue` member functions are called and given a row number outside of the rows in the local row cache, all missing rows up to this row are retrieved. In this way, the delayed retrieval feature is transparent except for the rows count.

The `insertRowInCache`, `updateRowInCache`, and `deleteRowInCache` member functions are similar to their non-`inCache` counterparts except that they simply act on the local row cache, leaving the remote data store unaffected.

The `IliTable::clearRows` clears the row cache. For two-tier tables, the external system to which the `IliTable` object is tied is not affected by the `clearRows` member function. A subsequent select would retrieve the same rows again.

In a one-tier table, however, the `IliTable::clearRows` effectively deletes all rows in the `IliTable` object.

Error Catching

All operations that are carried out on the rows of a table can fail for a variety of reasons. If an operation fails, an `IliErrorMessage` object that describes the error is created. The member function that triggered the error returns an error status.

An instance of the `IliErrorMessage` class contains the following information:

- Origin — an enumeration tag that identifies the library from which the error originates. It can be any of the following libraries: `DbmsServer, DbmsClientApi, DbLink, Data Access` or `Application`.
- Code — an integer whose interpretation depends on the origin.
- Message — a character string that contains a description of the error.

Error messages are caught by `IliErrorSink` objects. An error sink is an object to which errors can be forwarded.

```
class IliErrorSink {
public:
   ...
   virtual void addError(const IliErrorMessage&) {}
};
//The IliErrorSink class is intended to be subclassed. Here is an example:
class MyErrorSink : public IliErrorSink {
public:
   virtual void addError(const IliErrorMessage& msg) {
      IlvPrint("Error: code=%ld, message='%s'",
               (long)msg.getCode(),
                msg.getMessage());
   }
};
```

Alternatively, the `IliErrorList` class can be used. This class inherits from `IliErrorSink`, overloading the `addError` member function so that all errors caught are recorded and made available for inspection.

Once a particular type of error sink has been chosen, the `IliTable::addErrorSink` member function can be used to indicate that all subsequent error messages be forwarded to it.

Here is an example of how the `addErrorSink` member function can be used:

```
IliErrorList errors;
IliTable* tbl;
...
tbl->addErrorSink(&errors);
if (!tbl->deleteRow(10)) {
  for(IlInt i = 0; i < errors.getErrorsCount(); ++i) {
    IlvPrint("Error: %s",
             errors.getErrorAt(i).getMessage());
}
tbl->removeErrorSink(&errors);
```

Note that when table objects are acted upon through the default Data Access interaction mechanisms (such as the `DbNavigator` or the table gadget), any errors that occur are automatically reported to the end user.

However, when table objects are acted upon by custom C++ or IBM ILOG Script code that executes on behalf of user interface callbacks, it is the custom code that bears the responsibility to catch any errors that may occur (in an error list object, for example) and to explicitly report these errors to the end user.

The distinction should be made between user interface callbacks (such as the callback of a button gadget or of a menu item) and other more specialized callbacks (such as the data source `ValidateRow` callback).

Catching and reporting errors has to be done by user interface callbacks. It does not need to be done by the more specialized callbacks because the latter callbacks always execute in the context of a user interface callback.

Errors are reported by the `IliErrorReporter` class, which has a virtual `reportErrors` member function. It is possible to provide a custom error reporter on a data source or table gadget basis, or, more globally, the default error reporter may be overridden.

Changing Error Messages

Data Access and IBM® ILOG® Views DB Link error messages are translated in message database files.

- Data Access error messages are located in:

  `$ILVHOME/data/dataccess/dataccess.dbm`

- IBM ILOG Views DB Link error messages are located in:

  `$ILVHOME/data/dataccess/dblink.dbm`

The following code sequence is necessary to ensure that error messages are correctly translated:

```
IliFormat::ReadMessageDatabase(display, "dataccess/dataccess.dbm");
IliErrorMessage::ReadMessageDatabase(display);

IliFormat::ReadMessageDatabase(display, "dataccess/dblink.dbm");
IliSQLSession::ReadMessageDatabase(display);
```

Table Hook

The `IliTableHook` class can be used to monitor the changes that a table object undergoes. The `IliTableHook` class has a number of virtual member functions that can be overloaded in subclasses to monitor different events that occur within a table object.

In the following example, a table hook is used to print a message each time a row is inserted in a table object:

```
class CustomHook: public IliTableHook {
   virtual void rowInserted (IlInt rowno) {
       IlvPrint("A rows has been inserted at position %d",
                  (int)rowno);
   }
};
int main () {
   IliTable* tbl;
   ...
   CustomHook* hook = new CustomHook;
   tbl->addHook(hook);
   ...
```

```
    tbl->removeHook(hook);
    delete hook;
    return 0;
}
```

Copying and Serializing Table Objects

A table object can be copied with the `IliTable::copyTable` member function.

```
IliTable* origTable;
...
IliTable* cloneTable = origTable->copyTable();
```

Note that, in the case of two-tier tables, the row cache is not copied.

A table object can be written to a stream with the `IliTable::writeTable` member function:

```
IliTable* tbl;
ostream& os;
...
tbl->writeTable(os);
```

At a later date, the table object can be rebuilt by reading from a stream:

```
istream& is;
...
IliTable* tbl = IliTable::ReadTable(is);
```

Note that, in the case of two-tier tables, the row cache is not written to the stream.

Specialized Table Subclasses

The Data Access library provides subclasses of the `IliTable` class. Each one is dedicated to a specific row management policy.

You can find information on the following topics:

- *IliSQLTable*
- *IliMemoryTable*
- *IliStringsTable*
- *IliMapTable*

IliSQLTable

The `IliSQLTable` class implements the table interface by managing rows that are located in a remote relational database system. It handles all communication aspects with the database system, such as the generation of SQL statements, error checking, and so on.

The `IliSQLTable` class defines two-tier tables.

Instances of this class can be defined and used either through the C++ API or interactively in IBM ILOG Views Studio through an `IliSQLDataSource` object.

See Chapter 7, *SQL Tables* for more information on the `IliSQLTable` class.

IliMemoryTable

A memory table is a table that is managed locally in memory. You would use this type of table for data that is required only temporarily and therefore is not stored in a database. The SQL query language cannot be used with memory tables.

The `IliMemoryTable` class defines one-tier tables.

Objects of this class can be defined and used either through the C++ API or interactively in IBM ILOG Views Studio through an `IliMemoryDataSource` object.

Examples of how a memory table is defined and used can be found in *Managing Rows in a Table* on page 38.

IliStringsTable

The `IliStringsTable` class defines a one-tier table with a single column of type `string`. It is similar to the `IliMemoryTable` class, with the following exceptions:

- Its schema is fixed and cannot be changed.
- It provides a custom interface to manage rows. This interface uses the `const char*` C++ type instead of the `IliValue` and `IliTableBuffer` classes.

Here is an example of how it is used:

```
IlvDisplay* dpy;
IliDataSource* ds;
...
IliStringsTable* tbl = new IliStringsTable(dpy);
// No need to define the schema.
tbl->lock();
tbl->appendString("One");
tbl->appendString("Two");
tbl->appendString("Three");
ds->setTable(tbl);
tbl->unLock();
```

Objects of this class can be defined only through the C++ API.

IliMapTable

The `IliMapTable` class defines one-tier tables with two columns, the first column being of type `integer` and the second of type `string`. It is similar to the `IliMemoryTable` class, with the following exceptions:

- Its schema is fixed and cannot be changed.
- It is read-only. Its rows are given at construction time and cannot be changed afterwards.
- It supports IBM ILOG Views messages so that the values in the second column can be automatically translated before being displayed. See the `IliMapTable::setLanguageSensitive` member function.

Here is an example of how the `IliMapTable` class can be used:

```
IlvDisplay* dpy;
IliDataSource* ds;
static IliMapEntry entries[] = {
   1, "red",
   2, "green",
   3, "blue",
   0, NULL
};
...
IliMapTable* tbl = new IliMapTable(dpy, entries);
ds->setTable(tbl, IlTrue);
```

Objects of this class can be defined only through the C++ API.

Subclassing IliTable

This section describes how to subclass the `IliTable` class to create your own custom table classes.

You can find information on the following topics:

- *Guidelines*
- *Subclassing Example*
- *Directory Class Example*
- *Persistence*

Guidelines

When subclassing `IliTable`, the following guidelines should be respected:

- The following virtual member functions may be overloaded for both one- and two-tier tables:

Member Function	Overload
`getRowsCount`	mandatory
`getValue`	mandatory
`updateRow`	optional
`insertRow`	optional
`deleteRow`	optional
`moveRow`	optional
`allowRowMove`	optional
`updateRowInCache`	optional
`insertRowInCache`	optional
`deleteRowInCache`	optional

- The following virtual member functions may be overloaded for two-tier tables:

Member Function	Overload
`clearRows`	mandatory
`select`	mandatory
`isSelectDone`	mandatory
`fetchCompleted`	optional
`fetchNext`	optional
`fetchAll`	optional

- The subclass should be designed to notify the Data Access library when certain events occur. Notification is performed by calling the appropriate function from the following:

Member Function	When Called
`allRowsDeleted`	Called when the `clearRows` member function is called.
`tableChanged`	Called when the `IliTable` object has undergone a significant number of changes.

Member Function	When Called
`rowInserted`	Called just after a new row has been inserted in the table.
`rowsInserted`	Called just after a sequence of rows has been inserted. Note that instead of calling this member function, the `rowInserted` member function may be called repeatedly, once for each row.
`rowToBeChanged`	Called just before a row is changed.
`rowChanged`	Called just after a row has changed.
`rowToBeDeleted`	Called just before a row is removed.
`rowDeleted`	Called just after a row has been removed.
`rowMoved`	Called just after a row has moved to another position.
`rowsExchanged`	Called just after two rows have exchanged positions.
`rowFetched`	Called just after a new row has been fetched from a remote database and inserted into the local row cache (the `rowInserted` member function must also be called).
`rowsFetched`	Called just after a sequence of rows has been fetched. Note that instead of calling this member function, the `rowFetched` member function may be called repeatedly, once for each row.
`cellChanged`	Called just after a cell has changed. If more than one cell has changed in a row, it is preferable to call the `rowChanged` member function once, instead of calling `cellChanged` many times.
`raiseError`	Called each time an error occurs. The error is described by an `IliErrorMessage` object.

Note that in many instances, the implementor of an `IliTable` object can choose to notify certain events by calling one or another member function.

For instance, if two cells in a given row are changed, the `IliTable` object implementor can choose to do one of the following:

- Call `cellChanged` twice.

or

- Call `rowChanged` once.

Similarly, the insertion of two or more consecutive rows in the table, can be notified in one of the following ways:

- By repeatedly calling `rowInserted`, once for each row.

or

- By calling `rowsInserted` once.

As a consequence, when a given event can be notified either by calling one member function or another, an `IliTable` object using a table hook to monitor changes undergone by the table should overload **both** member functions, otherwise some events may be missed.

Subclassing Example

Here is an example that defines a `DirectoryTable` class. This class manages files in a directory. Within this example, the member functions that are used to notify the Data Access library when certain events occur appear in bold type.

```
#include <dirent.h>
#include <string.h>
#include <stdio.h>
#include <limits.h>
#include <ilviews/dataccess/table.h>

class DirectoryTable : public IliTable {
public:
   enum ColumnTags { FileName = 0 };
   DirectoryTable(IlvDisplay* dpy, const char* directory)
     : IliTable(dpy)
   {
      _rowsCount = 0;
      _files = NULL;
      _directory = dupString(directory);
      insertColumn(FileName, "FileName", IliStringType);
   }
   ~DirectoryTable() {
      tidy();
      delete [] _directory;
   }
   void clearRows() {
      tidy();
      allRowsDeleted();
      tableChanged();
   }
   IlvBoolean select() {
      readDir();
   }
   IlvBoolean isSelectDone() const {
      return _files != NULL;
   }
```

```
   IlvInt getRowsCount() const {
      return _rowsCount;
   }
   IlvBoolean getValue(IlInt rowno, IlvInt colno,
                       IliValue& value) const {
      if (rowno >= 0 && rowno < _rowsCount && colno == 0) {
         value = _files[rowno];
         return IlTrue;
      }
      return IlFalse;
   }
   IlvBoolean updateRow(IlInt rowno,
                        IliTableBuffer* buf) {
      const IliValue& value = buf->at(FileName);
      if (rowno >= 0 && rowno < _rowsCount
          && !value.isNull() && value.getType() == IliStringType) {
         const char* newname = value.asString();
         char oldpath[_POSIX_MAX_PATH];
         char newpath[_POSIX_MAX_PATH];
         sprintf(oldpath, "%s/%s", _directory, _files[rowno]);
         sprintf(newpath, "%s/%s", _directory, newname);
         if (rename(oldpath, newpath) == 0) {
           delete _files[rowno];
           _files[rowno] = dupString(newname);
           rowChanged(rowno);
           return IlTrue;
         } else {

           IliErrorMessage msg;
           msg.setApplicationError(strerror(errno));
           raiseError(msg);
         }
      }
      return IlFalse;
   }
   IlvBoolean insertRow(IlInt rowno,
                        IliTableBuffer* buf) {
      IliErrorMessage msg;
      msg.setApplicationError("Insertion not supported in "
                              "DirectoryTables");
      raiseError(msg);
      return IlFalse;
   }
   IlvBoolean deleteRow (IlInt rowno) {
      IliErrorMessage msg;
      msg.setApplicationError("Deletion not supported in "
                              "DirectoryTables");
      raiseError(msg);
      return IlFalse;
   }

private:
   IlInt _rowsCount;
   char* _directory;
   char** _files;

   char* dupString(const char* str) const {
      char* d = new char[strlen(str) + 1];
```

```
      strcpy(d, str);
      return d;
   }
   void tidy() {
     for (IlInt i = 0; i < _rowsCount; ++i)
        delete [] _files[i];
     delete [] _files;
     _files = NULL;
     _rowsCount = 0;
   }
   IlBoolean readDir() {
     DIR *dir = opendir(_directory);
     if (dir != NULL) {
       tidy();
       struct dirent* entry;
       while ((entry = readdir(dir)) != NULL)
          _rowsCount++;
       _files = new char*[_rowsCount];
       rewinddir(dir);
       _rowsCount = 0;
       while ((entry = readdir(dir)) != NULL)
          _files[_rowsCount++] = dupString(entry->d_name);
       closedir(dir);
       tableChanged();
       return IlTrue;
     }
     IliErrorMessage msg;
     msg.setApplicationError(strerror(errno));
     raiseError(msg);
     IlFalse;
   }
};
```

Note: *This example applies only to systems where the* `opendir`*,* `readdir`*, and* `closedir` *functions are defined. For other systems, you may have to make some changes in order to call the appropriate functions.*

Directory Class Example

The directory class could be used in the following way:

```
IlvDisplay* dpy;
IliDataSource* ds;
...
DirectoryTable* tbl = new DirectoryTable(dpy, "/usr/home/me");
ds->setTable(tbl, IlTrue);
```

In this example, the directory table is created and attached to an existing data source.

Persistence

Having created a custom table class, you may want to make it *persistent*. Making it persistent means that it is integrated in IBM® ILOG® Views Studio. It can appear in the Data Access palette of the Palettes panel and can be used in the same way as other Data Access classes. For example, you may want to make a custom data source that uses your persistent custom table.

To make your custom table class persistent, you should ensure that the following items are declared in the header file for the class:

- a copy constructor
- a stream based constructor
- the `IliDeclareDTypeInfo` macro in the class declaration
- a `write` virtual member function
- an `operator==`
- the `IliDeclareTypeInit` macro is used in the header file

```
#include <ilviews/dataccess/table.h>

class DirectoryTable : public IliTable {
public:
   ...
   DirectoryTable(const DirectoryTable&);
   DirectoryTable(IlvDisplay*, istream&);

   IliDeclareDTypeInfo(DirectoryTable);
   virtual void write (ostream&) const;
   int operator == (const DirectoryTable&) const;
   ...
};
IliDeclareTypeInit(DirectoryTable);
```

In the source file, the implementation should do the following:

- Use the `IliRegisterDClass` macro.
- Implement all the constructors and member functions mentioned above.

Here is an outline of what the implementation may look like. The details have been left for you to fill in:

```
DirectoryTable::DirectoryTable(const DirectoryTable& o)
 : IliTable(o)
{
   ...
}

DirectoryTable::DirectoryTable(IlvDisplay* dpy, istream& is)
  : IliTable(dpy, is)
```

```
{
  ...
}

void DirectoryTable::write (ostream& os) const {
   IliTable::write(os);
   ...
}

int DirectoryTable::operator == (const DirectoryTable& o) const {
   if (!IliTable::operator == (o))
      return 0;
   ...
}

IliRegisterDClass(DirectoryTable,IliTable);
```

Table Properties

The `IliTable` class supports annotating parts of a table with properties. In contrast to the primary content of the table (the table's rows), the properties are not constrained by the table schema.

A property has a name (an `IlSymbol*` object) and a value (an `IliValue` object).

The parts of an `IliTable` object that can have properties are:

- The whole table
- Any column
- Any row
- Any cell

Each part can have any number of properties attached to it as long as the property names are unique for each part. Two different parts (two cells or a cell and a row) can have properties with the same name.

The `IliTable` class does not manage properties itself, instead it delegates property management to the `IliTablePropertyManager` class.

An `IliTable` object has a default property manager, but it can manage additional property managers if needed. The requirement that a given part of a table cannot have two properties with the same name applies to each property-manager. Among different property managers, a given `IliTable` part can have properties with the same name, one for each property manager.

Scoped Properties

There is a containment relationship between the different types of parts of a table:

- A given row may contain a given cell.
- A given column may contain a given cell.
- The whole table contains all columns, all rows, and all cells.

Consequently, in addition to the properties that are attached to a given part, there may be properties attached to containing parts.

A given part is said to have a given scoped property if it has this property or if one of the parts in which it is contained has a scoped property with this name.

Note that the scoped property value of a part is the value of the property closest to the part. In the case of a conflict, if a given cell does not have a property but both the row and the column of the cell have a property with that name, the property of the row takes precedence.

The order of precedence for scoped properties lookup is as follows:

- Cell
- Row
- Column
- The whole table

Property-Aware Gadgets

Data-source-aware gadgets can be sensitive to given properties. Each gadget specifies the property names to which it is sensitive and what values are expected for these properties. It is then possible to change the behavior or look of the gadget by changing the property value of the table part to which the gadget is connected.

As a consequence, if a given table part is simultaneously displayed through different gadgets, some of its graphical attributes (for example, the font or the color) will automatically be identical in all gadgets. The application code that decides of the color needs only assign the "font" property to that part. It does not need to know in which gadgets that part is displayed nor does it need to access these gadgets and call member functions specific to them.

In addition, a property-aware gadget can use a different property manager than the default property manager of the table on demand.

Currently, the following gadgets are property-aware:

- `IliTableGadget`
- `IliDbField`
- `IliEntryField`
- `IliDbText`
- `IliTableComboBox`

- `IliDbStringList`
- `IliToggleSelector`
- `IliDbToggle`
- `IliDbOptionMenu`

These gadgets support the following properties (when applicable):

Property Name	Property Value Type	Property Value
font	String	font name
background	String	color name
foreground	String	color name
readOnly	Boolean	1 or 0
format	String	format name or specification
mask	String	mask name or specification

The following code example shows how table properties are used:

```
void MakePrimaryColumnsReadOnly(IliTable* table) {
  IlvInt count = table->getColumnsCount();
  const IlInt allRows = -1;
  const IlInt allColumns = -1;
  const IlInt insertRow = -2;
  const IlvSymbol* readOnlyName = IlvGetSymbol("readOnly");
  IliValue trueVal = (IlInt)1;
  IliValue falseVal = (IlInt)0;

  for (IlInt colno = 0; colno < count; ++colno) {
    if (table->isColumnPartOfKey(colno)) {
      table->setProperty(allRows,
                         colno,
                         readOnlyName,
                         trueVal);
    }
  }
  table->setProperty(insertRow,
                     allColumns,
                     readOnlyName,
                     falseVal);
}
```

Note that a value of `-2` can be used for the row index to designate the insertion row.

The `MakePrimaryColumnsReadOnly` function in the previous example works for the following reason. Since rows have precedence over columns, the `readOnly` property will be false for all cells contained in the insertion row (whatever the column), whereas it will be true for all primary key column cells contained in other rows.

C H A P T E R

4

Data Sources and Gadgets

This chapter describes data sources and the gadgets, called *data-source-aware gadgets*, that can be connected to data sources.

You can find information on the following topics:

◆ *Data Sources*

◆ *Data-Source-Aware Gadgets*

Data Sources

Data Access provides an `IliDataSource` class that “glues” table objects (the `IliTable` class and its subclasses) and gadgets used for data entry.

A *data source* is a gadget. Like other gadgets, it can have defined callbacks. The data source appears in the Data Access palette of the IBM ILOG Views Studio Palettes panel and can be inspected in IBM ILOG Views Studio.

Each data source manages an `IliTable` object and a current row.

Data Access provides other types of gadgets, such as *data-source-aware gadgets*, that are connected to a data source. These gadgets connect to a particular column and display the value of the current row in this column. Data-source-aware gadgets are discussed later in this chapter. See *Data-Source-Aware Gadgets* on page 69.

The `IliDataSource` class inherits from the `IliDataGem` class. `IliDataGem` is a special class that defines gadgets that are only visible during the design phase. Data source gadgets can be moved and inspected using the Selection mode of IBM ILOG Views Studio. However, when the panel is being tested during panel construction or when the application is being run, the data source gadget is no longer visible.

You can find information on the following topics:

- *Creating a Data Source Gadget*
- *Connecting Data-Source-Aware Gadgets*
- *The Scope of a Data Source*
- *Managing Rows in a Data Source*
- *Customizing a Data Source*
- *Error Handling*
- *The Repository*

Creating a Data Source Gadget

A data source is a gadget that manages an `IliTable` object. The following code sample shows how a data source can be created and set up to manage a memory table:

```
IlvDisplay* display;
IlvGadgetContainer* panel;
...
// Create a data source gadget.
IliDataSource* ds = new IliDataSource(display,
                                      IlvPoint(10, 10));

// Create and define a memory-table.
IliMemoryTable* tbl = new IliMemoryTable(display);
tbl->lock();
tbl->appendColumn("Id", IliIntegerType);
tbl->appendColumn("Name", IliStringType);

// Assign the memory-table to the data source.
ds->setTable(tbl);
tbl->unLock();

// Put the data source in a panel.
panel->addObject(ds);
panel->setObjectName(ds, "EMP");
```

Data Access provides the following subclasses of the `IliDataSource` class:

- `IliMemoryDataSource`—this class instantiates an `IliMemoryTable`.
- `IliSQLDataSource`—this class instantiates an `IliSQLTable`.

The only difference between these two classes and their base class is that they automatically instantiate the corresponding type of table. These two subclasses are provided for convenience, however, the `IliDataSource` class can manage any type of `IliTable` object.

> ***Note:*** *Another slight difference between these subclasses and their base classes is that each subclass has a specific bitmap that appears in IBM ILOG Views Studio.*

Connecting Data-Source-Aware Gadgets

A *data-source-aware gadget* is a gadget that can be connected to a data source. The following code sample shows how an `IliEntryField` gadget is created.

```
// Create an entry-field and put it in the panel.
IliEntryField* ef = new IliEntryField(display,
                                   IlvRect(25, 50, 55, 22));
panel->addObject(ef);

// Connect the entry field to the data source.
ef->f_setDataSourceName("EMP");
ef->f_setDataSourceColumnName("Id");
```

The newly created entry field is connected to the “Id” column of the “EMP” data source.

Similarly, the next example shows how to create another entry field gadget and connect it to the “Name” column.

```
// Create an entry-field and put it in the panel.
ef = new IliEntryField(display, IlvRect(25, 80, 155, 22));
panel->addObject(ef);

// Connect the entry-field to the data source.
ef->f_setDataSourceName("EMP");
ef->f_setDataSourceColumnName("Name");
```

The Scope of a Data Source

A data source can be accessed either by the gadgets in its own panel or by gadgets in other panels. A data source can be accessed by gadgets located in other unrelated panels only if it has global scope. The `IliDataGem` class has two member functions, `hasGlobalScope` and `setGlobalScope`, that let you determine and set the scope of a data source. See the `IliDataGem` class in the IBM ILOG Views *Data Access Reference Manual*.

The precise rules by which a data source is accessed are found in the section *The Repository* on page 67.

Managing Rows in a Data Source

Once a data source gadget has been defined and gadgets have been connected to it, the row management member functions of the data source can be used. These row management functions differ from the IliTable row management functions in that they are based on the concept of a current row. Changing the current row of a data source has a direct result on the user interface. The IliTable row management functions should be used when you want to manage rows without changing the current row in the user interface. Changing the current row means moving to another row. With regard to editing the values in a row (current or not), the user interface is updated in both cases. See *Managing Rows in a Table* on page 38.

From the end user's point of view, Data Access provides an IliDbNavigator gadget that connects to a data source. With this gadget, the end user can perform actions on the data source.

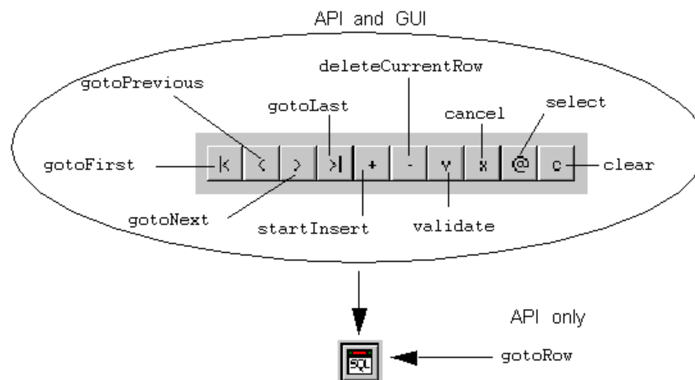


Figure 4.1 *An IliDbNavigator Gadget and the API Member Functions that It Calls*

Each button in the IliDbNavigator calls the corresponding member functions in IliDataSource directly.

If the table object managed by the data source is a two-tier table (for example, an IliSQLTable), the select member function can be used to re-evaluate the SQL table query.

The "goto" set of member functions enable you to move the current row of the data source. You can move to any row using the gotoPrevious, gotoNext, gotoFirst, gotoLast, gotoRow. The gotoRow member function takes a row index parameter.

When the current row of a data source changes, the value displayed by any gadget connected to the data source (for example, entry fields), is automatically adjusted.

If you want to modify a row in the underlying table of a data source, you can proceed in the following way:

```
IliDataSource* ds;
...
// We want to modify the 2nd row.
ds->gotoRow(1);
// Make any changes to the columns of the data source.
ds->setValue("Name", IliValue("Smith"));
...
// Validate changes.
if(!ds->validate())
   IlvPrint("Update failed");
```

The `setValue` member function is used to modify a value contained in the data source buffer. The data source buffer retains the changes until the `validate` member function is called. When there are changes pending in a validation, the `isInputModified` member function returns `true`.

During the period of time that it takes for the `isInputModified` member function to return `true` on a data source, any pending changes can be canceled using the `cancel` member function.

In all situations, the underlying table is modified only when `validate` is called, not each time the `setValue` member function is called.

The following code sample shows how a new row can be inserted:

```
IliDataSource* ds;
...
// We want to insert a new row.
ds->startInsert();
// Assign values to the columns of the data source.
ds->setValue("Id", IliValue(32));
ds->setValue("Name", IliValue("Jones"));
...
// Validate insertion.
if(!ds->validate())
   IlvPrint("Insert failed");
```

The main difference between this example and the previous one is that the `startInsert` member function is used instead of using the `gotoRow` member function to move to an existing row.

> ***Note:*** *You can disable the insertion through a data source by calling the* `enableInsert` *member function with the parameter set to false.*

The `deleteCurrentRow` member function can be used to remove the current row.

Customizing a Data Source

The behavior of a data source can be customized using callbacks. A callback is a C++ function that will be called when a given event occurs. A callback can be defined by calling one of the `IlvGraphic::setCallback` or `IlvGraphic::addCallback` member functions.

Here is an example:

```
void ILVCALLBACK MyEnterRow(IlvGraphic* g, IlAny) {
  IliDataSource* ds = (IliDataSource*)g;
  IlvPrint("Enter Row %ld in data source '%s'",
           (long)ds->getCurrentRow(),
           ds->getName());
}

int main(int argc, char** argv) {
   IliDataSource* ds;
   ...
   ds->setCallback(IliDataSource::EnterRowSymbol(),
                   MyEnterRow);
   ...
}
```

This example defines the `EnterRow` callback of a data source. This callback will be called each time a new row becomes the current row of the data source.

For each callback type (such as `EnterRow`), the `IliDataSource` class provides:

- A static member function that returns the name of the callback type in the form of an `IlSymbol*` (for example, `EnterRowSymbol`).
- A virtual member function (such as `onEnterRow`) that is called by the data source when the corresponding event occurs. This virtual member function calls in turn the corresponding callback (if any).

Monitoring the Selected Row

The following callback types can be used to monitor the selected row:

- `EnterRow`
- `QuitRow`

Update Validation

The following callback types can be used to customize the way rows are updated:

- `EnterUpdateMode`
- `ValidateRow`
- `PrepareUpdate`
- `QuitUpdateMode`

- `CancelEdits`

These callbacks are called in a particular order.

The first time the end user starts to modify a row (by typing a key on the keyboard), the `EnterUpdateMode` callback is called. At this point, the `IliDataSource::isInputModified` member function returns `true`.

When the end user has finished editing the row, validation will usually be triggered. Validation proceeds as follows:

- The `ValidateRow` callback is called.
- The `PrepareUpdate` callback is called.

These two callbacks are designed to allow you to code custom checks that depend on the application logic, and make on-the-fly adjustments to the row being updated.

Both of these callbacks have the same purpose. However, the `ValidateRow` callback is also called when a row is inserted (as you will see in the next section), therefore enabling you to specify a single function that is called in both cases.

The following example shows how custom checks can be coded:

```
void ILVCALLBACK MyValidateRowCallback(IlvGraphic* g, IlAny) {
    IliDataSource* ds = (IliDataSource*)g;
    if (ds->getValue("Qty").asInteger() > 15) {
        ds->dontValidateRow();
        ds->addErrorMessage("Invalid quantity");
    }
}
...
IliDataSource* ds = ...;
ds->setCallback(IliDataSource::ValidateRowSymbol(),
                MyValidateRowCallback);
```

If the check criterion is not satisfied, the callback calls the `dontValidateRow` member function to stop validation and it calls the `addErrorMessage` member function to provide an appropriate error message.

If both of these callbacks agree on validation (that is, they do not call the `dontValidateRow` member function), the row updates are transmitted to the underlying table through the `IliTable::updateRow` member function. If this call succeeds, the `QuitUpdateMode` callback is called.

The `CancelEdits` callback is called if the end user cancels the modifications instead of validating.

Insert Validation

The following callback types can be used to customize the way rows are inserted:

- `EnterInsertMode`
- `ValidateRow`

- PrepareInsert
- QuitInsertMode
- CancelEdits

These callbacks work in the same way as the update callbacks (see the previous section).

The following example shows how a `PrepareInsert` callback can be defined to compute a unique identifier:

```
void ILVCALLBACK MyInsertRowCallback(IlvGraphic* g, IlAny) {
   IliSQLDataSource* ds = (IliSQLDataSource*)g;
   IliSQLTable* tbl = ds->getSQLTable();
   if (ds->getValue("ID").isNull()) {
      IliSQLSession* session = tbl->getEffectiveSQLSession();
      IliSQLCursor* curs = session->newCursor();
      if (curs->execute("SELECT NEXTID FROM COUNTER FOR UPDATE")
          && curs->fetchNext()) {
        IlInt id = curs->getIntegerValue(0);
        curs->execute("UPDATE COUNTER SET NEXTID = NEXTID + 1"));
        ds->setValue("ID", IliValue(id));
      }
      else {
        ds->dontValidateRow();
        ds->addErrorMessage(curs->getErrorMessage());
      }
      session->releaseCursor(curs);
   }
}
...
IliSQLDataSource* ds = ...;
ds->setCallback(IliDataSource::PrepareInsertSymbol(),
                MyInsertRowCallback);
```

This example assumes that the data source is tied to a table with an “ID” column. When the end user inserts a row through the data source, the value of the “ID” column is computed by incrementing a value found in the COUNTER database table.

Computed Columns

The `FetchRow` callback can be used to compute the value of one or more columns. This callback, which applies only to two-tier tables, is called each time a row is retrieved from the remote system and stored in the local row cache.

The following example illustrates the use of the `FetchRow` callback. Assume you have a database table, named HEATER, with a MAXTEMP column that describes the maximum temperature in degrees Celsius.

To display this table and show the MAXTEMP column in degrees Fahrenheit, do the following:

◆ Ensure that the SQL data source has, at least, the following two columns:

	MAXTEMP	FAHRENHEIT	
Select:	MAXTEMP	1	
From:	HEATER		
Operation:			
Order:			
Where:			

Select | Having | Datatype | Look | Mapping | Parameters

Apply | Close

	MAXTEMP	FAHRENHEIT	
Name:		FAHRENHEIT	
Type:	Double	Double	
Length:			
Null:	No	Yes	
Default:			
Retrieve:	Yes	Yes	

Select | Having | Datatype | Look | Mapping | Parameters

Apply | Close

◆ Define a `FetchRow` callback in the following way:

```
void ILVCALLBACK MyFetchRowCallback(IlvGraphic* g, IlAny) {
   IliSQLDataSource* ds = (IliSQLDataSource*)g;
   IliSQLTable* tbl = ds->getSQLTable();
   IlInt rowno = ds->getFetchedRow();
   IliTableBuffer* buf = tbl->getBuffer();
   buf->rowToBuffer(rowno);
   IlDouble celsius = buf->at("MAXTEMP").asInteger();
   IlDouble fahrenheit = CelsiusToFahrenheit(celsius);
   buf->at("FAHRENHEIT") = IliValue(fahrenheit);
   tbl->updateRowInCache(rowno, buf);
   tbl->releaseBuffer(buf);
}
...
IliSQLDataSource* ds = ...;
ds->setCallback(IliDataSource::FetchRowSymbol(),
                MyFetchRowCallback);
```

Note that the `updateRowInCache` member function is called instead of `updateRow` because only the local row cache needs to be changed.

Deleted Rows

The following callback types can be used to customize the way rows are deleted:

◆ `PrepareDeleteRow`

◆ `DeleteRow`

The PrepareDeleteRow callback can be used to prohibit row deletions through a given data source. If the callback calls dontDeleteRow, the user will not be able to delete the current row. As with the ValidateRow callback, the addErrorMessage member function may be called to provide an error message.

The DeleteRow callback can be used to monitor row deletion events. Here is an example:

```
void ILVCALLBACK MyRowDeleted(IlvGraphic* g, IlAny) {
  IliDataSource* ds = (IliDataSource*)g;
  IlvPrint("Row %ld in data source '%s' has been deleted",
           (long)ds->getDeletedRow(),
           ds->getName());
}

int main(int argc, char** argv) {
   IliDataSource* ds;
   ...
   ds->setCallback(IliDataSource::DeleteRowSymbol(),
                   MyRowDeleted);
   ...
}
```

Error Handling

This section provides information on three topics concerning the handling of errors for a data source: error catching, error reporting, and error raising.

Error Catching

Errors can be caught through error sink objects using a similar technique to the one described for IliTable objects. See the addErrorSink and removeErrorSink member functions. Note that all errors raised by the underlying table object are forwarded to the data source error sinks. In addition, the data source itself can raise specific errors.

The following example shows how a data source should be set up in order to catch and report errors:

```
IliDataSource* ds;
...
// Set up an error sink.
IliErrorList errors;
ds->addErrorSink(&errors);
// Act on the data source.
ds->gotoRow(10);
ds->setValue("NAME", IliValue("Smith"));
ds->validate();
...
ds->removeErrorSink(&errors);
// Check for errors.
if (errors.getErrorsCount() > 0) {
   ds->reportErrors(errors);
}
```

Error Reporting

Errors are reported through an instance of the `IliErrorReporter` class. The `setErrorReporter` member function can be used to provide a custom error reporter.

The following example shows how the error reporter of a data source can be redefined:

```
class MyErrorReporter: public IliErrorReporter {
public:
   virtual void reportErrors (IlvDisplay* dpy,
                              IlvAbstractView* anchor,
                              const IliErrorList& errors) const {
      for (IlInt i = 0; i < errors.getErrorsCount(); ++i) {
         IlvPrint("Error: %s", errors.at(i).getMessage());
      }
   }
};

int main(int argc, char** argv) {
   IliDataSource* ds;
   ...
   MyErrorReporter* rep = new MyErrorReporter;
   ds->setErrorReporter(rep);
   ...
}
```

Error Raising

Errors can be raised within a validation callback such as `ValidateRow`, `PrepareUpdate` or `PrepareInsert`. In such cases, the `dontValidateRow` member function should be called to stop the validation process and errors can be raised by calling the `addErrorMessage` member function. For more information on Error Messages, see Appendix D, *Error Messages*.

The Repository

There are two ways in which a data source can be retrieved when its name is known:

- If the container in which it is located is known, the `IlvContainer::getObject` member function can be used as follows:

```
IlvContainer* container;
...
IliDataSource* ds = (IliDataSource*)container->getObject("EMP");
```

- Alternatively, the `IliRepository` class provides static member functions that allow you to determine the data sources that are registered in the repository.

Each data source gadget is automatically registered in the repository when it is added to a container or manager. When it is removed from its holder, it is unregistered from the repository.

Here are the `IliRepository` rules by which a gadget can connect to a data source based on the data source name and on the location of the gadget:

- Same holder

  If a data source is found in the same container as the gadget, it is chosen.

- Same scope class

  Otherwise, if a data source is found in another container that belongs to the same scope class as the gadget container, it is chosen.

- Same container hierarchy

  Otherwise, if a data source is found in another container that belongs to the same container hierarchy as the gadget container, it is chosen.

- Global scope

  Otherwise, if a global data source is found, it is chosen.

- Otherwise, the look-up operation fails and the gadget does not connect to any data source.

Enumerating All Data Sources Accessible from the Repository

The following code fragment iterates through all the data source gadgets that are registered in a repository.

```
IlvInt count = IliRepository::GetDataSourcesCount();
for (IlInt i = 0; i < count; ++i) {
   IliDataSource* ds = IliRepository::GetDataSource(i);
   ...
}
```

Finding a Data Source Using Its Name

The `IliRepository::FindDataSource` static member function can be used to retrieve a data source gadget using its name.

```
IlvGadget* g;
...
IliDataSource* ds;
ds = IliRepository::FindDataSource("EMP", g->getHolder());
```

Subscribing to a Given Data Source

The `IliRepository` supports a “subscription” mechanism. This mechanism allows you to specify a C++ function that should be called whenever a data source with a given name becomes available.

For more information, see the `IliRepository::SubscribeToDataSource` member function in the IBM ILOG Views *Data Access Reference Manual*.

Data-Source-Aware Gadgets

This section describes the gadgets that can connect to a data source. These gadgets inherit from both the `IlvGadget` and the `IliFieldItf` classes.

You can find information on the following topics:

- *Interface to Data-Source-Aware Gadgets*
- *IliTableGadget*
- *IliDbField*
- *IliEntryField*
- *IliTableComboBox*
- *IliDbText*
- *IliDbToggle*
- *IliToggleSelector*
- *IliDbNavigator*
- *IliDbTimer*
- *IliHTMLReporter*
- *IliXML*
- *IliDbPicture*
- *IliDbOptionMenu*
- *IliDbStringList*
- *IliDbTreeGadget*
- *IliChartGraphic*
- *IliDbGrapher*
- *IliDbGantt*
- *Global Callbacks*

Interface to Data-Source-Aware Gadgets

Data-source-aware gadgets have a common interface that is defined by the `IliFieldItf` class.

For example, the `IliEntryField` class has the following hierarchy:

Figure 4.2 The IliEntryField Hierarchy and Example Gadget

As you can see, `IliEntryField` inherits from both the `IliFieldItf` and `IlvTextField` classes. `IlvTextField` is an IBM ILOG Views gadget class.

The `IliIsAField` global function can be used to test whether a graphic object is a data-source-aware gadget. The `IliGraphicToField` global function can be used to convert a pointer to a graphic object into a pointer to an `IliFieldItf`.

```
IlvGraphic* g;
...
if (IliIsAField(g)) {
   IliFieldItf* fld = IliGraphicToField(g);
   ...
}
```

Objects can be converted in the opposite direction using the `f_getGraphic` member function.

```
IliFieldItf* fld;
...
IlvGraphic* g = fld->f_getGraphic();
```

Connecting to a Data Source

The main feature of data-source-aware gadgets is their ability to connect to a data source and stay “tuned” with the value in the current row of a given column.

Staying tuned involves the following:

- When the current row of the data source changes, the data-source-aware gadget is assigned the new current row value for the given column.
- When the end user edits the value in the data-source-aware gadget, this value is sent to the data source.

Here is an example that shows how a data-source-aware gadget is connected to a data source:

```
IliEntryField* ef;
...
ef->f_setDataSourceName(“EMP”);
ef->f_setDataSourceColumnName(“DEPTNO”);
```

The data source is specified by name. However, if a data source with this name does not exist when `f_setDataSourceName` is called, the gadget remains unconnected, but remembers

the name of the data source it must connect to. The `f_isConnectedToDataSource` member function can be used to check whether a gadget is connected to a data source.

The actual connection takes place whenever a data source with the given name enters into the gadget scope. A data source is in gadget scope when it is either in the same panel as the gadget or in another panel and with global scope (see `IliDataGem::setGlobalScope`).

Managing Gadget Values

The value of a data-source-aware gadget can be managed with the `f_getValue` and `f_setValue` member functions.

The `f_getValue` member function can be used to retrieve the value of the gadget:

```
IliFieldItf* fld;
...
const IliValue& val = fld->f_getValue();
IlvPrint("Current value is : %s", val.getFormatted());
```

The `f_setValue` member function can be used to assign a new value to a gadget:

```
IliValue newval = "A New val";
fld->f_setValue(newval);
```

Gadget Look

Some aspects of the look of a gadget can be accessed and set with the following member functions:

Accessor	Mutator
f_isReadOnly	f_setReadOnly
f_getFormat	f_setFormat
f_getAlignment	f_setAlignment
f_getLabel	f_setLabel
f_getMask	f_setMask

The effect of these functions depends on the actual gadget class being used and sometimes there is no effect at all. For example, calling `f_setFormat` for an `IliDbToggle` has no effect.

Foreign Table

A gadget can have a mapping associated with it. See *Mapping* on page 36. To specify a mapping, you must provide the following information:

- Foreign table
- Value column
- Display column

A mapping is used to convert the value of the gadget (its *internal* value) to another value that is substituted in the display.

```
IliFieldItf* fld;
...
fld->f_setForeignDataSourceName("DEPT");
fld->f_setForeignValueColumnName("DEPTNO");
fld->f_setForeignDisplayColumnName("NAME");
```

For more information on foreign tables, see *Foreign Tables* on page 105.

Subclassing IliFieldItf

New data-source-aware gadget classes can be defined. This typically involves defining a new subclass of both `IliFieldItf` and some other existing gadget class. For an example of such a subclass, see the `IliFieldItf` class in the IBM ILOG Views *Data Access Reference Manual*.

IliTableGadget

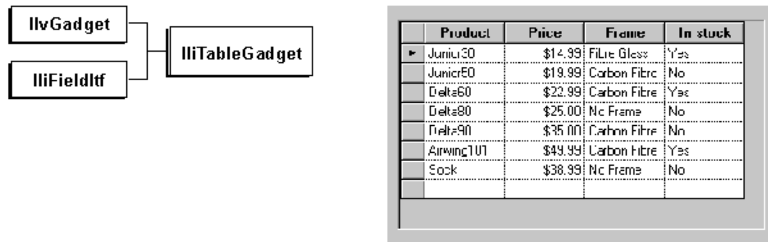


Figure 4.3 *The IliTableGadget Hierarchy and Example Gadget*

The `IliTableGadget` class enables you to display an entire table. It also lets the end user edit table values, and add or delete rows.

A table gadget can be connected to a data source by calling `f_setDataSourceName`.

```
IliTableGadget* tg = new IliTableGadget(display,
                                        IlvRect(20, 30, 300, 450));
panel->addObject(tg);
tg->f_setDataSourceName("EMP");
```

The `IliTableGadget` class provides many member functions that allow you to control the look of the gadget.

Selection

The `IliTableSelection` class is used to describe the highlighted area in a table gadget at a given point in time. This area can be any of the following:

- a cell (identified by column and row indices)
- one or more rows (identified by their indices)
- one or more columns (identified by their indices)
- the complete table gadget
- empty

The `getSelection` member function returns the current selection in a table gadget and the `setSelection` changes this selection.

By default, a table gadget is *bound* to its data source. This means that the row of the selection in the table gadget remains synchronized with the current row of the data source. The `bindToDataSource` member function controls this.

The `pointToSelection` member function can be used to identify which part of a table gadget contains a given geometrical point.

Column Geometry

The table gadget event handler lets the end user resize the columns and change their order. If there is more than one table gadget connected to the same data source, resizing a column in one of them will also resize the column in the others (by default). This is because the size and order of columns are stored in the table object of the data source and the data source is shared by the table gadgets.

The table gadget supports a special mode in which column size, visibility, and order are local to the table gadget itself and independent of other table gadgets connected to the same data source. The `setColumnsGeometryLocal` member function can be used to activate this mode. See *Setting the Table Look* on page 110.

Note that when column geometry is local, the index of a column can be different in the table gadget and the underlying table. The `getRealColno` and `getVisualColno` member functions can be used to convert a column index between the table gadget order and the underlying table object order.

Cell Editor

Table gadget cells can be edited with editors that are managed by the table gadget. Each column in a table gadget has an editor. By default, a table gadget creates the editor for each column depending on the column data type and mapping.

For columns with a foreign column, the table gadget creates an `IliTableComboBox`. For other columns, it creates an `IliEntryField`.

NAME	DEPT	SALARY
Rochette	Administration	6000.0
Fernandez	Administration	7500.0
Goodman	Documentation	6300.0
Thomasson	Marketing	8000.0
Williams	Marketing	4000.0
Wong	R & D	7700.0
Bornstein	R & D	8650.0
Harrison	Marketing	4575.0
Horner	Documentation	6375.0
Tanaka	Finance	9000.0

A table cell with an IliTableComboBox as **editor**

A table cell with an **IliEntryField as editor**

Figure 4.4 *Different Types of Editors within a Table Gadget*

The `setColumnEditor` can be used to define a custom editor for a given column.

Customizing a Table Gadget

Table gadgets can be customized with callbacks.

The `GetCellPalette` callback can be used to change the foreground and background colors, and the font, cell by cell.

Here is an example:

```
void ILVCALLBACK MyGetCellPalette(IlvGraphic* g, IlAny) {
   IliTableGadget* tg = (IliTableGadget*)g;
   IliCellPaletteStruct* cell = tg->getCellPaletteStruct();
   if (cell->getRowno() == 3 && cell->getTableColno() == 2) {
   // Change the background color of cell(3,2).
   cell->setFillPalette(tg->getDisplay()->getPalette("Highlight"));
}

int main() {
   IliTableGadget* tg;
   ...
   tg->setCallback(IliTableGadget::GetCellPaletteSymbol(),
                   MyGetCellPalette);
   ...
}
```

Note that colors defined through a `GetCellPalette` callback are dynamic. The table gadget does not keep a record of the colors for a cell. Instead, it calls the `GetCellPalette` callback each time it needs to draw the cell.

Note that another technique is available to define colors on a row, a column, or a cell basis. See *Table Properties* on page 53 for more information.

The `DrawCell` callback can be used to provide a custom draw procedure for some of the cells in a table gadget. Here is an example:

```
void ILVCALLBACK MyDrawCell(IlvGraphic* g, IlAny) {
   IliTableGadget* tg = (IliTableGadget*)g;
   IliDrawCellStruct* cell = tg->getDrawCellStruct();
```

```
   if (cell->tblColno == 2) {
      // Draw cells in column 2.
      IlvDisplay* dpy = tg->getDisplay();
      IlvPalette* pal = dpy->defaultPalette();
      IlvInt value;

      //---- compute the position gauge ----
      value = tg->at(cell->rowno, cell->tblColno).asInteger();
      value = (value > 100L) ? 100L : value;
      value = (value < 0L) ? 0L : value;
      value = (cell->bbox.w() * value) / 100L;

      // Draw the cell.
      IlvRect rect;
      rect.x(cell->bbox.x());
      rect.w((IlvDim)value);
      rect.y(cell->bbox.y()+2);
      rect.h(cell->bbox.h()-4);
      dpy->fillRectangle(cell->dst, pal, rect);
   }
   else
      tg->defaultDrawCell();
}

int main() {
   IliTableGadget* tg;
   ...
   tg->setCallback(IliTableGadget::DrawCellSymbol(),
                   MyDrawCell);
   ...
}
```

Customizing the Column Editor of a Table Gadget with a Toggle

You can change the column editor of a table gadget. For example, columns with a data type like Boolean, can replace the Yes/No combo box by a toggle without a label (class `IliSimpleToggle`).

```
// In the header file:
#include <inform/gadgets/dbsimtog.h>

class MyPanel:
public IlvGadgetContainer {
...
protected:
IliSimpleToggle* _toggle;
...
};
// In the source file:
MyPanel::MyPanel(...):IlvGadgetContainer(...) {
  ...
     _toggle = new IliSimpleToggle(getDisplay(), IlvPoint(0,0));
     IliTableGadget*
                  tbl=(IliTableGadget*)getObject("tableGadgetName");
     tbl->setColumnEditor(_toggle);
...
}
```

Note: If you work under Windows 95, Windows NT 4, or Motif, you should change the column background color (do not keep white) to see the toggle relief.

IliDbField

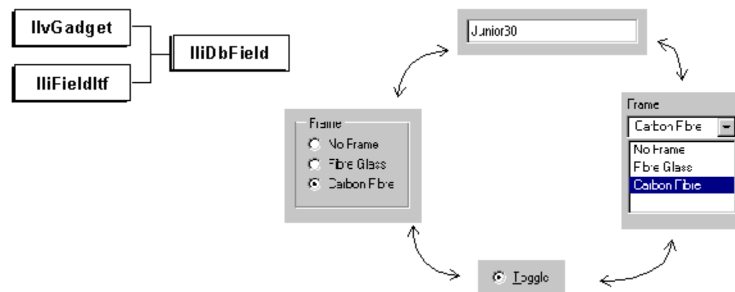


Figure 4.5 The IliDbField Hierarchy and Different Gadget Looks

The `IliDbField` gadget is very flexible. It has different styles that can determine its look and feel. The style can be changed using the `setStyle` member function. For more information on the `DbField` gadgets, see *Working with DbFields in Data Access* on page 99.

A form created using the Forms Assistant (IBM ILOG Views Studio) is made up of a set of `DbField`, each of which is connected to a column of the underlying table object. Initially, each `DbField` has a default style that can subsequently be changed, if required.

The flexibility of the `DbField` look and feel lets you modify the styles of each of the `DbField` contained in the form to suit your requirements. Without this flexibility, you would have to replace a `DbField` object with the appropriate object to change its style.

IliEntryField

Figure 4.6 The IliEntryField Hierarchy and Example Gadget

The `IliEntryField` class defines a text field gadget that can be connected to a data source.

Note that this gadget does not take the foreign table into account.

IliTableComboBox

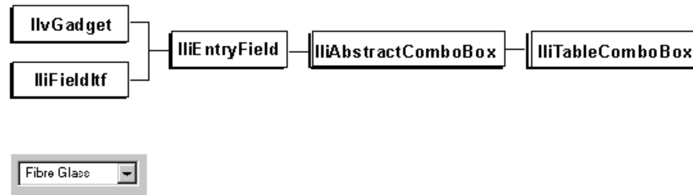


Figure 4.7 The IliTableComboBox Hierarchy and Example Gadget

The `IliTableComboBox` defines a combo box gadget that opens a pull-down menu when the user clicks on the combo button. The pull-down menu displays a column of the foreign table of the field.

```
IliTableComboBox* combo;
combo = new IliTableComboBox(display, IlvRect(20, 30, 150, 21));
panel->addObject(combo);

// Connect the combo to EMP(DEPTNO).
combo->f_setDataSourceName("EMP");
combo->f_setDataSourceColumnName("DEPTNO");

// Specify the mapping as DEPT(ID -> NAME).
combo->f_setForeignDataSourceName("DEPT");
combo->f_setForeignValueColumnName("ID");
combo->f_setForeignDisplayColumnName("NAME");
```

IliDbText

Figure 4.8 The IliDbText Hierarchy and Example Gadget

The `IliDbText` class defines a gadget that edits multi-line character strings. This gadget is not designed to be used with a foreign table.

IliDbToggle

Figure 4.9 The IliDbToggle Hierarchy and Example Gadget

The `IliDbToggle` class defines a gadget that manages Boolean values. A Boolean value can be either `0`, `1`, or `null`. Note that it is possible to specify another set of values through the use of a foreign table. For example, the set of values could be the three strings "on", "off", and "out Of Order".

IliToggleSelector

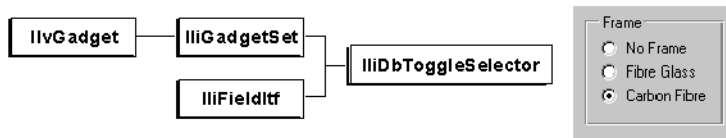


Figure 4.10 The IliToggleSelector Hierarchy and Example Gadget

The `IliToggleSelector` class defines a gadget that contains a set of toggle gadgets. Each toggle gadget corresponds to a row of the foreign table. Within these toggle gadgets, only one gadget can be checked at any particular time. See *IliToggleSelectorStyle* on page 102.

IliDbNavigator

`IliDbNavigator` is a gadget that lets the end user carry out certain actions on the data source. When a table is being edited via a form or table, the current row (record) is stored in a buffer (specified by `IliTableBuffer`). The end user is free to edit any of the values shown in the current record.

Figure 4.11 The IliDbNavigator Hierarchy and Example Gadget

Using the DbNavigator gadget, you can do the following:

- Change the current record (|< < > >| back to the first record in the table, back to the previous record, forward to the next record, or forward to the last record in the table).
- Display the current row position in the data source or, if in query mode, display the current row position of the query [3] .
- Display the number of rows in the data source or, if in query mode, display the number of rows of the query [11] .
- Validate [v] any entries that have changed in the current record (create in database) or apply query mode.
- Cancel [x] any entries that have been made in the current record (create in database) or cancels query mode.
- Insert a new record [+] , provided Allow Insert is activated.
- Delete current record [-] .
- Clear the data source cache, update it by querying the database, and refresh the display [@].
- Clear the data source cache [c] .
- Put the data source in query mode [?] .

***Note:** Until you validate the changes that you have made in the current record, the values will not be updated in the database. The new values are actually stored at the data source in the table buffer. This leaves you the possibility of editing the complete record.*

Adding a User Button to the Navigator

The following sample code shows how to add buttons to your DbNavigator to customize it for your needs. The new buttons here are called Print and Quit.

```
MyPanel::initialize()
{
  IliDbNavigator* navig = ...;
```

```
  // Add the buttons.
  Navig->addButton("P", MyPrintCallback, "Print", 2);
  Navig->addButton("Q", MyQuitCallback, "Quit", 1);

  // Compute the position and width of the buttons.
  Navig->adjustButtonsSize();
}
```

IliDbTimer

Figure 4.12 The IliDbTimer Hierarchy and Example Gadget

The `IliDbTimer` gadget lets you specify a time interval and a callback that will be called repeatedly.

IliHTMLReporter

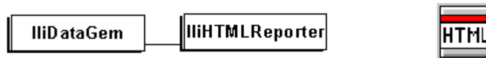


Figure 4.13 The IliHTMLReporter Hierarchy and Example Gadget

The `IliHTMLReporter` gadget generates an HTML document from the contents of a data source.

IliXML

Figure 4.14 The IliXML Hierarchy and Example Gadget

The `IliXML` gadget manages the communication between a data source and an `XML` stream. It manages the import and export of notification and modification.

IliDbPicture

Figure 4.15 The IliDbPicture Hierarchy and Example Gadget

The `IliDbPicture` class defines a gadget that displays bitmaps. The column to which it is connected must contain bitmap names. This gadget is not designed to be used with a foreign table.

IliDbOptionMenu

Figure 4.16 The IliDbOptionMenu Hierarchy and Example Gadget

The `IliDbOptionMenu` class defines an option menu gadget that can be connected to a data source. This gadget opens a pull-down menu when the user clicks the gadget. The pull-down menu displays a column of foreign tables of the field.

```
IliDbOptionMenu* opt;
opt = new IliDbOptionMenu(display, IlvRect(20, 30, 150, 21));
panel->addObject(opt);

// Connect the Option Menu to EMP(DEPTNO)
opt->f_setDataSourceName(“EMP”);
opt->f_setDataSourceColumnName(“DEPTNO”);

// Specify the mapping as DEPT(ID -> NAME)
opt->f_setForeignDataSourceName(“DEPT”);
opt->setForeignValueColumnName(“ID”);
opt->setForeignDisplayColumnName(“NAME”);
```

IliDbStringList

Figure 4.17 The IliDbStringList Hierarchy and Example Gadget

The `IliDbStringList` class defines a string list gadget that can be connected to a data source. This gadget displays a list of strings and/or pictures that come from the field of a foreign table column.

```
IliDbStringList* lst;
lst = new IliDbStringList*(display, IlvRect(20, 30, 150, 150));
panel->addObject(lst);
// Connect the string list to EMP(DEPTNO)
lst->f_setDataSourceName("EMP");
lst->f_setDataSourceColumnName("DEPTNO");

// Specify the mapping as DEPT(ID -> NAME
lst->f_setForeignDataSourceName("DEPT");
lst->setForeignValueColumnName("ID");
lst->setForeignDisplayColumnName("NAME");
lst->setForeignBitmapColumnName("PICTURE");
```

IliDbTreeGadget

Figure 4.18 The IliDbTreeGadget Hierarchy and Example Gadget

The `IliDbTreeGadget` class defines a tree gadget that works with a data source or several data sources (depending on the model). This gadget displays the links between the children and their parents. Each data source row defines a tree gadget item.

To provide the data to the tree gadget, you can use the following models:

- Recursive: All items of the tree gadget are defined by only one data source.
- Structural: Each tree gadget level has its own data source with its columns.

The Parent column is not used for the first level.

IliChartGraphic

Figure 4.19 *The IliChartGraphic Hierarchy and Example Gadget*

The `IliChartGraphic` class defines a chart graphic that works with a data source or several data sources (depending on the data model). With an `IliChartGraphic` it is also possible to display a pie chart.

Here is an example:

Let's take a data source, named DATA_MS, with the following schema:

- two columns

  NAME and VALUE(double)

- data

  Beer 150.0

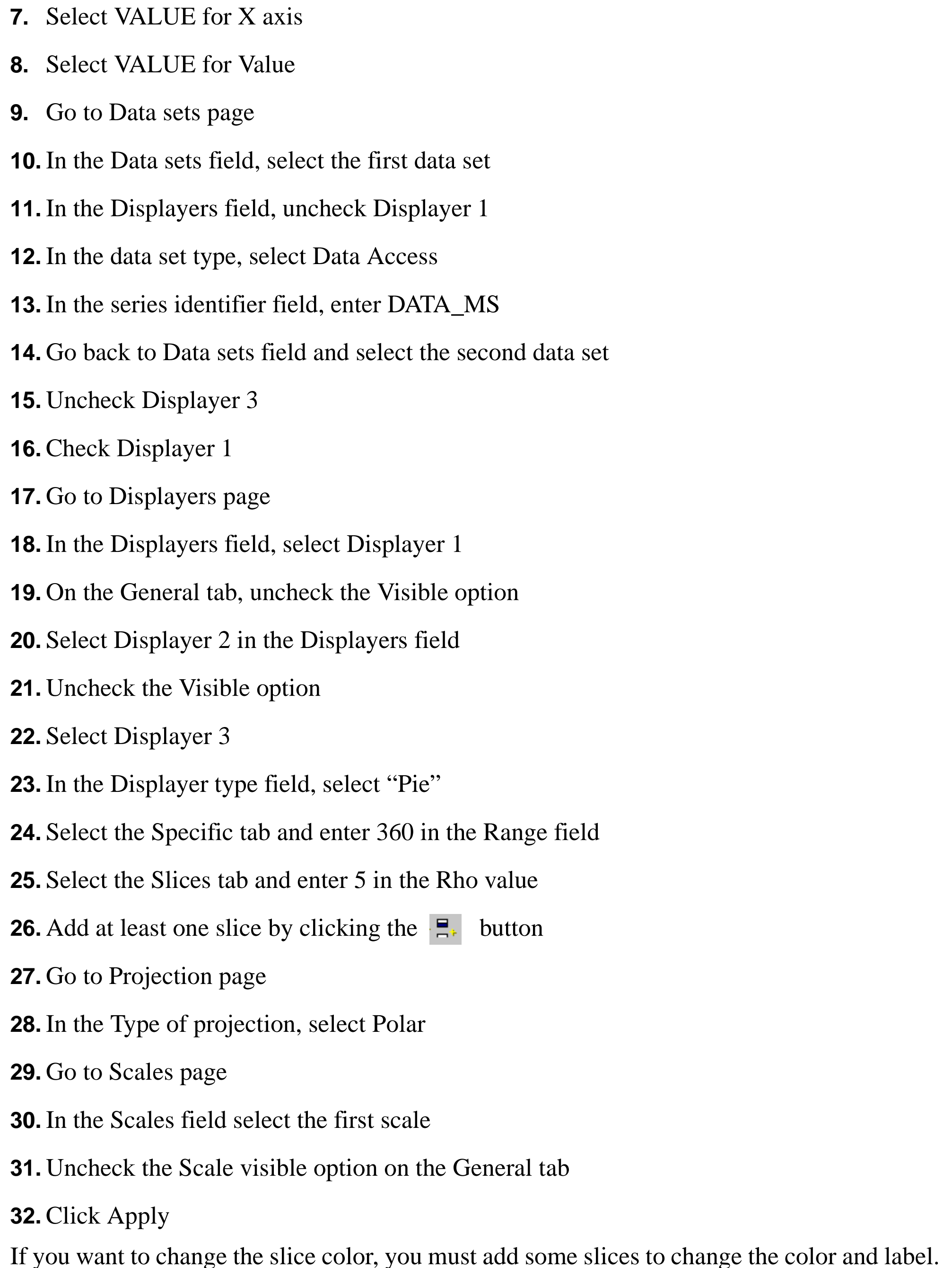
  Soda 300.0

  Water 600.0

  Wine 350.0

To connect an `IliChartGraphic` to this data source, you must go through the following steps:

1. Drag and drop an `IliChartGraphic`
2. Inspect the `IliChartGraphic`
3. Go to Specific page
4. In the Data model field, select "By data sources"
5. Go to Data Source page
6. Select DATA_MS for data source

7. Select VALUE for X axis
8. Select VALUE for Value
9. Go to Data sets page
10. In the Data sets field, select the first data set
11. In the Displayers field, uncheck Displayer 1
12. In the data set type, select Data Access
13. In the series identifier field, enter DATA_MS
14. Go back to Data sets field and select the second data set
15. Uncheck Displayer 3
16. Check Displayer 1
17. Go to Displayers page
18. In the Displayers field, select Displayer 1
19. On the General tab, uncheck the Visible option
20. Select Displayer 2 in the Displayers field
21. Uncheck the Visible option
22. Select Displayer 3
23. In the Displayer type field, select “Pie”
24. Select the Specific tab and enter 360 in the Range field
25. Select the Slices tab and enter 5 in the Rho value
26. Add at least one slice by clicking the  button
27. Go to Projection page
28. In the Type of projection, select Polar
29. Go to Scales page
30. In the Scales field select the first scale
31. Uncheck the Scale visible option on the General tab
32. Click Apply

If you want to change the slice color, you must add some slices to change the color and label.

IliDbGrapher

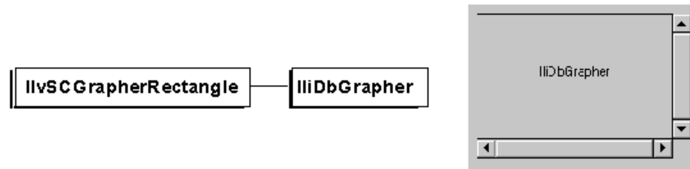


Figure 4.20 The IliDbGrapher Hierarchy and Example Gadget

The `IliDbGrapher` gadget lets you specify a nodes data source and a links data source. It displays the information contained in these data sources in the form of a graph.

Customizing a DbGrapher

The DbGrapher can be customized with callbacks.

The `setDefineObject` member function lets you specify a callback that is called when the end user attempts to create a new node. The callback is expected to fill in a table buffer with the values for the node to be created. It can open a dialog box to obtain information from the end user if needed. A similar callback for defining links can also be specified.

Here is an example:

```
static IlvBoolean
MyDefineNodeCallback(IliTableBuffer* buf, IlvGraphic*, IlAny any) {
  static int stId = 1000;

  if (buf->at("IDENTIFIER").isNull()) {
     buf->at("IDENTIFIER").importInteger(stId);
     stId++;
  }
  return IlvTrue;
}

int main() {
   IliDbGrapher* gr;
   ...
   gr->setDefineObjectCallback(MyDefineNodeCallback, NULL, IlvTrue);
   ...
}
```

The `setCreateObject` member function lets you specify a callback that is called when a new row has been inserted in the nodes data source. This callback is needed to create and configure the graphic object that will represent this row. A similar callback for defining links can also be specified.

Here is an example:

```
static IlvGraphic*
```

```
MyCreateNodeCallback(IliTableBuffer* buf, IlAny any) {
   IliDbGrapher* gr = (IliDbGrapher*)any;
   const char* picture;
   switch (buff->at("TYPE").asInteger()) {
      case TypeNodeCenter  : picture = "center.xpm"; break;
      case TypeNodeParabol : picture = "parabol.xpm"; break;
      default              : picture = "terminal.xpm"; break;
   }
   gr->setBitmapName((const char*)picture);
   IlvGraphic* obj  = gr->createDefaultObjectNode(buf);
   if (obj->isSubtypeOf(IliLabeledBitmap::ClassInfo())) {
      IliLabeledBitmap* node = (IliLabeledBitmap*)obj;
      node->setLabelName(buff->at("NAME").getFormatted());
   }
   return node;
}

int main() {
   IliDbGrapher* gr;
   ...
   gr->setCreateObjectCallback(MyCreateNodeCallback, gr,IlvTrue);
   ...
}
```

There are also a `NodeDoubleClicked` callback and a `LinkDoubleClicked` callback.

Here is an example of using the `NodeDoubleClicked` callback:

```
static void
MyDoubleClickNodeCallback(IlvGraphic* g, IlAny any) {
  IliDbGrapher* gr = (IliDbGrapher*)g;
  const char* s = gr->getObjectNameDoubleClicked();
  IlvPrint ("Node %s double clicked", (const char*)s);
}

int main() {
  IliDbGrapher* gr;
  ...
  gr->addCallback(IliDbGrapher::NodeDoubleClickedSymbol(),
                  MyDoubleClickNodeCallback,myPanel)
  ...
}
```

IliDbGantt

Figure 4.21 The IliDbGantt Hierarchy and Example Gadget

The `IliDbGantt` gadget lets you specify data sources for resources, activities, constraints, precedences, breaks and work load curve. It displays the information contained in these data sources in the form of a Gantt chart.

Customizing a DbGantt Gadget

A DbGantt gadget can be customized with callbacks (or corresponding virtual functions).

The `getScaleNumericLabel` member function lets you specify a callback that is called when the end user needs to compute a label for a numeric scale label.

Here is an example:

```
static void
MyComputeLabel(IlvGraphic* g,IlAny any) {
  IliDbGantt* dbg = (IliDbGantt*)g;
  IliString s;

  s << dbg->getScaleNumericValue();
  s << "$";
  dbg->setScaleNumericLabel(s);
}

int main() {
  IliDbGantt* dbg;
  ...

  dbg->addCallback(IliDbGantt::ScaleNumericLabelSymbol,
                   MyComputeLabel,0);
  ...
}
```

The `isActivePeriod` member function lets you specify a callback that is called when the end user needs to determinate if a period is active or not. By default, a period is active. This is why this callback is used to indicate the periods which are not active.

Here is an example:

```
static void
MyComputePeriod(IlvGraphic* g,IlvAny any) {
  IliDbGantt* dbg = (IliDbGantt*)g;

  if (dbg->getActivePeriodInfo(IliScaleDateWeekDay) == IliDbGanttSunday)
  dbg->setInactivePeriod();
}

int main() {
  IliDbGantt* dbg;
  ...
  dbg->addCallback(IliDbGantt::IsActivePeriodSymbol,
                   MyComputePeriod,0);
  ...
}
```

Global Callbacks

Gadget behavior can be customized using callbacks. Data Access supports four ways of defining callbacks.

- **Callback function**—The callback is a C++ function directly attached to the gadget. The following example shows how a callback function can be attached directly to a gadget.

```
void CustomCallback(IlvGraphic*g, IlvAny userdata) {
   ...
}

int main(int argc, char* argv[]) {
   ...
   IlvSymbol* callbackType;
   IlvGadget* g;
   IlvAny userData;
   ...
   g->setCallback(callbackType,
                  CustomCallback,
                  userData);
   ...
}
```

- **Named callback**—The `IlvContainer::registerCallback` member function is used to associate a name with a C++ function. This name can then be used as a callback for any gadgets in the container. The following example shows how this can be done:

```
void CustomCallback(IlvGraphic*g, IlvAny userdata) {
     ...
  }

  int main() {
     ...
     // Register the callback.
     IlvContainer* cont;
     ...
     cont->registerCallback(“MyCallbackName”,
                            CustomCallback);
     ...

     // Use this callback.
     IlvGadget* g;
     IlvSymbol* callbackType;
     ...
     g->setCallback(callbackType,
                    “MyCallbackName”);
     ...
  }
```

- **IBM ILOG Script callback**—The callback is an IBM ILOG Script function. These callbacks are usually defined from within the IBM ILOG Views Studio environment.

◆ **Global callback**— Data Access adds a way to define and use callbacks globally. The `IliCallbackManager` class can be used to register callbacks that are globally available and that can have parameters. The callback manager is the unique instance of the `IliCallbackManager` class and it can be retrieved by calling the `IliGetCallbackManager` function. For the callback manager to be active, the file `<ilviews/dataccess/gcallbak.h>` must be included in at least one of the application modules. If this is not the case, the global callbacks will not be available. The following example shows how a callback can be set up as global:

```
#include <ilviews/dataccess/gcallbak.h>
 // Define the callback function.
 void MessageBoxCallbak(IlvGraphic* g,
                        IlAny arg,
                        IlInt paramsCount,
                        const char* const* params) {
  IlvContainer* view = IlvContainer::getContainer(g);
  if (paramsCount == 1 && views != NULL) {
    IlvIMessageDialog msgBox(view->getDisplay(),
                             params[0],
                             NULL,
                             IlvDialogOk,
                             view->getSystemView());
    msgBox.show();
  }
 }
 int main() {
   ...
   // Register the global callback.
   IlvSymbol* callbackName = IlvGetSymbol("MsgBox");
   IliGetCallbackManager().registerCallback(callbackName,
                                            MessageBoxCallback);
   ...
 }
```

Once a global callback has been registered as shown above, it can be used by prefixing its name with the “@” character.

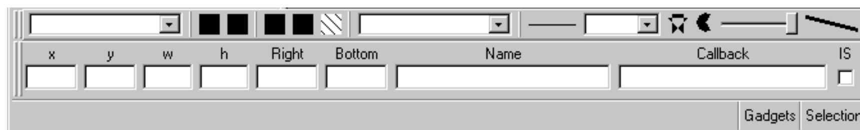


Figure 4.22   Using a Global Callback in IBM ILOG Views Studio

Guidelines for Defining Global Callbacks

Some general guidelines can help you determine which method to use for defining global callbacks:

◆ For C++ programmers:

- In a pure C++ application where IBM ILOG Views Studio is not used, the *callback function* approach is the most convenient as it results in less coding.

- When IBM ILOG Views Studio is used to design the panels of the application, the *named callback* and the *global callback* approaches are recommended since they enable you to enter the callback name directly in the required inspector in IBM ILOG Views Studio.
- If a callback is needed in only one panel, the *named callback* approach is the most suitable. However, if the same callback is required in many panels, the *global callback* approach is the most appropriate.

◆ For IBM ILOG Script programmers:

- Global callbacks are of little help to IBM ILOG Script programmers since it is as convenient and straightforward to define an IBM ILOG Script function and use it as a callback as it would be to use a global callback. In effect, we consider that global callbacks have been superseded by the capability to program in IBM ILOG Script.

Predefined Global Callbacks

The following is a list of the predefined global callbacks that are available in Data Access:

◆ `@Quit()`

◆ `@ShowPanel(panelName)`

◆ `@HidePanel(panelName)`

◆ `@Validate(dataSourceName)`

◆ `@Cancel(dataSourceName)`

◆ `@Clear(dataSourceName)`

◆ `@Select(dataSourceName)`

◆ `@StartInsert(dataSourceName)`

◆ `@Commit(sessionName)`

◆ `@Rollback(sessionName)`

◆ `@Connect(sessionName)`

◆ `@QueryConnect(sessionName)`

◆ `@Disconnect(sessionName)`

C H A P T E R

5

Handling Values in Data Access

This chapter describes how Data Access handles values. They are implemented by the `IliValue` class and the following sections describe how to use an `IliValue` object.

You can find information on the following topics:

- *The IliValue Class*
- *Data Types*
- *Structured Types*

The IliValue Class

The `IliValue` class supports polymorphism for the most primitive data types, such as character strings, integers, and float values. An `IliValue` object can hold a value belonging to any of these primitive types. Moreover, the type of a value can be changed dynamically. This class works in conjunction with the `IliDatatype` class, which is used to represent the dynamic type of an `IliValue` object.

The `IliValue` class is essential to Data Access. It enables values to be handled in a completely transparent way, so that table objects and data-source-aware gadgets can deal with values without having to take into account their type.

Each of the data-source-aware gadgets has an `f_getValue` member function that returns the gadget value in the form of an `IliValue` object.

Constructing a Value Object

There are two ways to construct a value. It can be specified with an initial value:

```
IliValue initVal = 3;
IliValue stringVal = "Good morning."
```

Also, it can be constructed by specifying the data type explicitly:

```
IliValue boolVal(IliBooleanType);
```

In this case, the value has an initial value of `null`.

Null Value

The null value is a special value that can be used to denote an unspecified value. You can check for the null value with the `IliValue::isNull` method.

```
IliValue value;
...
if (value.isNull()) {
    ...
}
```

An object can have a null value in two situations: either it has been constructed with no initial value just by specifying its data type (shown above) or it has been set as null using the `IliValue::setNull` method.

The `IliValue::setNull` method also allows you to change the object data type. This can be seen in the following code sample:

```
IliValue value = "aString";
value.setNull(); // value set to null,
                 // datatype remains IliStringType
value.setNull(IliIntegerType); //datatype is now IliIntegerType
```

Data Types

The `IliValue` class works in close conjunction with the `IliDatatype` class. This class specifies the type of an `IliValue` object. An instance of the `IliDatatype` class exists for each supported type. The possible data types are:

- `IliNullType`
- `IliStringType`
- `IliBooleanType`

- `IliByteType`
- `IliIntegerType`
- `IliFloatType`
- `IliDoubleType`
- `IliDecimalType`
- `IliDateType`
- `IliTimeType`
- `IliBinaryType`
- `IliAnyType`

The data type of an `IliValue` object constrains the set of values that it can hold. For example, an `IliBooleanType` object can hold three values: `IlvTrue`, `IlvFalse`, and `null`.

In addition to these predefined types, the Data Access library can dynamically synthesize new types. For a description of when this can occur, see *Parameters* on page 128. These types are collectively called structured types and fall in two categories: object types and table types. The section *Structured Types* at the end of this chapter explains how these types are used.

Checking the Data Type of an Object

The data type of an object can be checked using the `IliValue::getType` method in the following way:

```
IliValue value;
...
if (value.getType() == IliDateType) {
    ...
}
```

Converting a Data Access Data Type to a C++ Type

To manipulate the data in an application, you will need to work with values in the standard C++ types:

- `const char*`
- `IliByte`
- `IlvInt`
- `IlvFloat`
- `IlvDouble`
- `IliDecimal`
- `IliDate`

- IliBinary
- IliTable*
- IlvAny

Any IliValue object can be easily converted into one of the standard C++ types. However, a conversion of this type will only return a meaningful value if the data type of the IliValue object is compatible with the target C++ standard type.

Table 5.1   Type Conversions That Return a Meaningful Value

Data Type	C++ Cast Operator	as<Type>() Method
IliStringType	const char*	asString(const char* nv1)
IliStringType IliBooleanType IliByteType IliIntegerType IliFloatType IliDoubleType IliDecimalType	IliByte IlvInt IlvFloat IlvDouble IliDecimal	asBoolean(IlvBoolean nv1) asByte(IliByte nv1) asInteger(IlvInt nv1) asFloat(IlvFloat nv1) asDouble(IlvDouble nv1) asDecimal(const IliDecimal& nv1)
IliStringType IliDateType IliTimeType	IliDate IliTime	asDate(const IliDate& nv1) asTime(const IliTime& nv1)
IliBinaryType	IliBinary	asBinary(const IliBinary& nv1)
Object type *Table type*	IliTable*	asTable(const IliTable* nvl)
IliAnyType	IlAny	asAny(IlAny nv1)

IliValue objects can be converted in one of two ways:

- Using a C++ cast operator (implicit or explicit).
- By calling one of the as<Type>() methods available in the IliValue class.

This is demonstrated in the next example:

```
IliValue stringValue = “Hello world!”;
const char* str1 = stringValue; // implicit cast
const char* str2 = stringValue.asString(“Null”);
```

The main difference between these two conversions is that the as<Type> method accepts an extra parameter. In the following example, the asString method returns the string value of the object for which it is called unless the object is null or its data type is not IliStringDataType. In this case, it returns the value of the nv1 parameter.

```
class IliValue {
```

```
public:
    operator const char*() const;
    const char* asString(const char* nv1 = 0) const;
};
```

The cast operator returns `0` if the object is `null` or its type is not `IliStringDataType`.

```
IliValue stringValue(IliStringType); //initial value is null
IliValue integerValue = 6;

const char* str1 = stringValue; //str1 == 0
const char* str2 = integerValue; //str2 == 0
const char* str3 = integerValue.asString("Undefined"); //str3 != 0
```

You should remember to use the character string returned by the `const char*` operator or the `asString()` method as soon as possible. If not, the character string can become invalid the next time the `IliValue` object is modified.

Numeric data types can be converted to any of the numeric C++ types. However, the conversion can cause a loss of precision and numbers can even be truncated.

```
IliValue integerValue = 5;
IliValue doubleValue = 9.8;
IlInt anInt = doubleValue; //loss of precision
IlDouble aDouble = integerValue;
```

The `IliStringType` data type can also be converted to any of the numeric types as shown in the following example:

```
IliValue stringValue = "3.14";
IlInt i = stringValue; // i set to 3
IlDouble d = stringValue; // d set to 3.14

IliValue stringValue = "Not A Number";
IlInt i = stringValue; // i set to 0
IlDouble d = stringValue; // d set to 0.0
i = stringValue.asInt(-1); // i set to -1
d = stringValue.asDouble(-2.0); // i set to -2.0
```

An `IliValue` object can also be changed using one of the = operators:

```
IliValue value = "One";
value = "Two";
value = 3; //Data type changed to IliIntegerType
value = 4.0; //Data type changed to IliDoubleType
```

Formatting an IliValue Object

An `IliValue` object can be formatted using the `IliValue::format` member function. This is shown in the following example:

```
IliString stringBuf;
IliFormat fmt("#,##0.00 Yens");
IliValue value = 5.677;
value.format(stringBuf, fmt);
```

```
IlvPrint("Price: %s", (const char*)stringBuf);
```

You can also use the `IliValue::getFormatted` member function, which does not require an `IliString` parameter to store the result. An example is shown in the following code:

```
IliFormat fmt("#,##0.00 Yens");
IliValue value = 5.677;
const char* result = value.getFormatted(fmt);
IlvPrint("Price: %s, result);
```

Note that the `getFormatted` member function returns its result in a static character buffer. This result must therefore be used immediately after the call.

Structured Types

The Data Access type system can be extended beyond the basic data types described in the previous sections. This occurs automatically when structured values are obtained from a database system such as Oracle 9i (or later) or Informix 9.x.

Each structured type has an associated schema that can be obtained as follows:

```
void DescribeStructuredType(const IliDatatype* type) {
  if (type->isStructuredType()) {
    const IliSchema* schema = type->getNestedSchema();
    IlvPrint("Type %s manages tables with %ld columns",
            schema->getName(),
            (long)schema->getColumnsCount());
  }
  else
    IlvPrint("Not a structured type !");
}
```

A structured type manages `IliValue` objects that contain a pointer to an `IliTable` object.

The following code sample shows how an `IliValue` object can be initialized with an `IliTable` object:

```
IliValue MakeStructuredValue(const IliDatatype* type) {
  if (type->isStructuredType()) {
    IliValue value(type);
    IliTable* table = type->makeTable();
    table->lock();
    value.importTable(table);
    table->unLock();
    return value;
  }
  else {
    IlvPrint("Not a structured type !");
    return IliValue();
  }
}
```

This example uses the `IliDatatype::makeTable` member function to create an `IliTable` object whose schema is identical to the schema returned by the `IliDatatype::getNestedSchema` member function.

The table returned by `IliDatatype::makeTable` is really an `IliMemoryTable` object. Consequently, all row management member functions (`insertRow`, `updateRow`, and so on) can be called to fill this table. Note, however, that structured types fall into two categories: object types and table types. They differ in that an object types expects to manage a table that contains at most one row, whereas table types manage tables with many rows. The `isObjectType` and `isTableType` member functions can be used to distinguish between object and table types.

It is important to understand how the `IliValue` class manages `IliTable` objects. When an `IliValue` object is copied into another, the nested `IliTable` object is shared between both `IliValue` objects as shown by the following code excerpt:

```
const IliDatatype* type = ...;
IliValue firstVal = MakeStructuredValue(type);
IliValue secondVal = firstVal;
assert(firstVal.asTable() == secondVal.asTable());
```

As a consequence, you should not directly alter the table contained in an `IliValue` object. Instead, a copy of the table should be made, altered, and then assigned back to the `IliValue` objects in the following code excerpt:

```
IliValue value = ...;
const IliTable* table = value.asTable();
if (table != NULL) {
  IliTable* tempTable = table->copyTable();
  tempTable->lock();
  // alter tempTable in some way ...
  tempTable->sortRows();
  value.importTable(tempTable);
  tempTable->unLock();

}
```

C H A P T E R

6

Hints and Tips for Using Data Access

This chapter contains some useful hints for using Data Access. The most common user requirements are described, along with tips on how to use the features of Data Access to satisfy these requirements. The chapter primarily discusses the way to set up your application using IBM ILOG Views Studio, rather than using the Data Access API.

You can find information on the following topics:

- *Working with DbFields in Data Access*
- *Foreign Tables*
- *Setting the Table Look*
- *Fixed Columns*
- *Troubleshooting*

Working with DbFields in Data Access

This section describes the `IliDbField` class and the different forms that it can take. A `DbField` gadget is an entry field that can be connected to the column of a table object. The `DbField` gadget then displays the value of this column for the current record. A form is made up of `DbField` objects with different styles.

The Style of a DbField

A IliDbField object can dynamically change its look and feel. Its style can be any of the following:

- IliEntryFieldStyle
- IliTextStyle
- IliOptionMenuStyle
- IliTableComboBoxStyle
- IliToggleStyle
- IliToggleSelectorStyle
- IliStringListStyle

To change the style of an IliDbField gadget in IBM ILOG Views Studio, you must select the required style from the Style field in the corresponding DbField inspector.

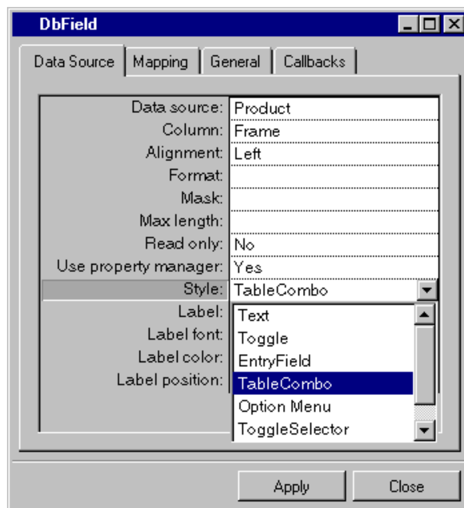


Figure 6.1 Changing the DbField Style in the DbField Inspector

The following sections describe the different styles of a DbField gadget, giving you the best situations to use them in.

IliEntryFieldStyle

This style can be used to display or edit a value of any type. It is not designed to be used with a column that accepts a Boolean value or with a column that has a foreign table. Other styles are better adapted to these two situations.

IliTextStyle

This style is a scrolling text area that can be used to display or edit multiple lines of data. It is not designed to be used with a column that accepts a Boolean value or with a column that has a foreign table. Other styles are better adapted to these two situations.

IliOptionMenuStyle

This style is designed to be used with a column that has a foreign table or with a column that accepts a Boolean value (see *Foreign Tables* on page 105). It consists of a label and a button. The button accesses a menu that contains values to be selected and displayed in the label.

If the `DbField` is linked to a Boolean value column, the menu contains the options Yes and No, and will not accept any other input.

However, if the column has a foreign table, only the values contained in the menu can be selected in the field.

IliTableComboBoxStyle

This style is designed to be used with a column that has a foreign table or a column that accepts a Boolean value (see *Foreign Tables* on page 105). It consists of an entry field and a button. The button accesses a menu that contains values that can be entered in the field.

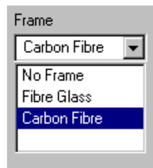


Figure 6.2 *A DbField with the IliTableComboBoxStyle*

If the `DbField` is linked to a Boolean value column, the menu contains the options Yes and No, and will not accept any other input.

However, if the column has a foreign table and if the Constrained property in the `DbField` inspector is set to Yes, only the values contained in the menu can be entered in the field.

However, if the Constrained property in the `DbField` inspector is set to No, values other than those in the menu can be entered directly into the field.

Finally, if the Completion property in the `DbField` inspector is set to Yes, when one or more characters are entered in the `DbField`, the option is completed by the appropriate menu entry (provided the characters uniquely define a menu entry).

IliToggleStyle

This style is very similar to the `IliDbToggle` object. It is a graphical object that enables you to display a state. A toggle usually includes a state marker and a label. The state marker

is to the left of the label and indicates whether the state is on or off (according to its Boolean value).

The look of the button depends on the look that has been selected in IBM ILOG Views Studio (Windows, Windows XP, or Motif).

Figure 6.3 *A Toggle in Windows95 Look and Feel and a Toggle in Motif Look and Feel*

This style is specifically designed to be connected to a table column that has a Boolean value. In this way it can be used to turn the value on or off in the column as required. It should not be used with columns that contain values of other types or with a column that has a foreign table.

There are two ways to create a toggle that is linked to a table object column: either the style of an `IliDbField` instance can be changed or an instance of `IliDbToggle` can be created directly.

The `IliDbToggle` class provides more flexibility regarding the look of the toggle state marker. However, changing the style of an existing `IliDbField` enables you to create a form using the *Forms Assistant* and to customize the style of each of the `DbField` contained in the form.

IliToggleSelectorStyle

This style is specifically designed to be used with a column that has a foreign table mapping. It is a set of toggles that are contained in a frame. Only one of the toggles can be turned on at a time.

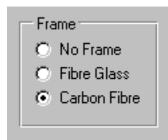


Figure 6.4 *A DbField with the IliToggleSelectorStyle*

> ***Note:*** *This* `IliToggleSelectorStyle` *should not be used when the list of values is long since the style becomes difficult to manage.*

IliStringListStyle

This style is designed to be used with a column that has a foreign table. It consists of a string list and a string in the list can be selected using the mouse or the keyboard. The strings in the list are read from the foreign table. See *Foreign Tables* on page 105.

Creating a Form Using the Forms Assistant

A form in Data Access is a set of gadgets that lets you display all the values in a table object row, thus providing easier access than with a table gadget. An instant form can be obtained from the Forms Assistant command in the Data Access menu in IBM® ILOG® Views Studio.

If you create a form using the Forms Assistant command, a set of fields (`IliDbField` objects) are displayed. Each field is connected to one of the columns of the data source table object and has a label that is the title of its associated column. There is an `IliDbNavigator` across the top of the panel to help you navigate through the table object records.

The Forms Assistant can also create a form based on a table gadget instead of on a set of `DbField` gadgets.

You may notice that the `DbField` gadget automatically adopts the `IliTableComboBoxStyle` for columns that either contain a Boolean value or have a foreign table mapping (see *Foreign Tables* on page 105). In the case of a Boolean, the pop-up menu has two entries, Yes and No.

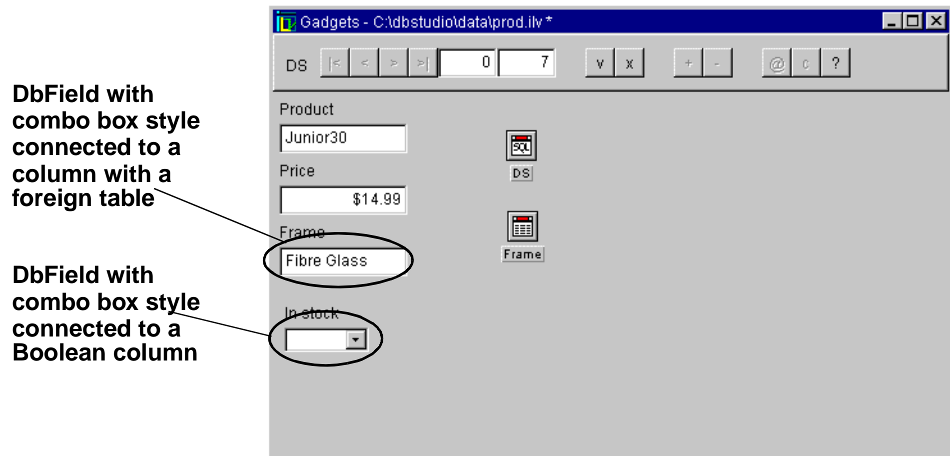


Figure 6.5 A Form Showing the Automatic Combo Box for Columns with a Limited Value Domain

It can be useful to change the style of the `DbField` with a Boolean value to a toggle gadget. See *The Style of a DbField* on page 100 for information on how to change the style.

Forms Assistant Pages

The Forms Assistant has four notebook pages that let you change various presentations.

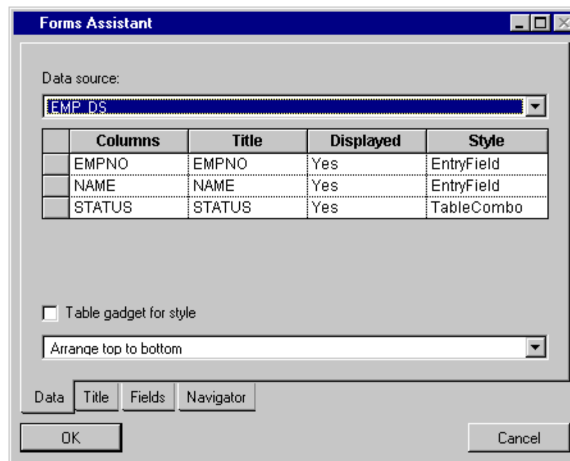


Figure 6.6 *The Data Page of the Forms Assistant*

The Data page is used to select the data source using the data source combo box. In the table column, you can specify which data source columns will have a `DbField` and choose the type of `DbField` used to display the information (text, toggle, combo box, and so on). The Table gadget style check box is used to create a form with a table gadget. You can also choose the presentation of the `DbField` in the panel (Arrange top to bottom or left to right) using the combo box at the bottom of the page.

The Title page is used to change the title name and its presentation (color and font).

The Fields page lets you change the Labels and Fields presentation.

The Navigator page lets you activate or de-activate various navigator and the position of the navigator.

To apply changes and display information in the Main window, click OK.

Columns with a Foreign Table

As shown in Figure 6.6, the `DbField` that is linked to a column with a foreign table is set automatically to the `IliTableComboBoxStyle`. In addition, it automatically has the same foreign table as the table column (this can be seen in its inspector). This occurs only when the forms assistant is activated on a data source that already has a foreign table mapping.

If you create a `DbField` and link it to a data source column yourself, the foreign table will not automatically be set for the `DbField` gadget.

For more information on foreign tables and how they can be linked to table objects and data-source-aware gadgets, see *Foreign Tables* on page 105.

Foreign Tables

Any table object in Data Access can be used as a foreign table; that is, the `IliTable` subclasses, `IliSQLTable`, `IliMemoryTable`, and so on. A foreign table is identified as such from another object. Each column of a table object or of the `IliDbField`, `IliTableComboBox`, and `IliToggleSelector` gadgets can have a foreign table.

A foreign table has two main purposes. It can be used to map the values in a column to another set of values or it can be used to define the domain of values for the column.

There are two approaches for applying a foreign table to a table column.

- A foreign table can be applied to the column of a table object using the data source inspector in IBM ILOG Views Studio. The foreign table then applies directly to any table gadget, or form created with the Forms Assistant that is subsequently connected to the data source.
- A foreign table can be applied to a particular data-source-aware gadget. In this case, the foreign table applies only to the gadget.

Foreign tables are specified in the appropriate inspector in IBM ILOG Views Studio. In the Data Access API, the methods found in the `IliTable` (see *Schema Properties* on page 35) and `IliFieldItf` classes (see *Foreign Table* on page 71) can be used to specify the foreign table.

Specifying a Foreign Table in IBM ILOG Views Studio

A foreign table can be set up for the data source or for an individual gadget. The `IliDbField`, `IliTableComboBox`, and `IliToggleSelector` gadgets can have a foreign table. The foreign table is set up for a particular column in the appropriate inspector.

To specify a foreign table for a particular table column, you must select the required column and specify the following characteristics of the foreign table:

- the data source with which the foreign table is associated
- the column in the foreign table that corresponds to the *value column*
- the column in the foreign table that corresponds to the *display column*

The value column is a column in the foreign table in which values will be located. A value from the table column is matched with a value in the foreign table value column. The display column is the column from which a value will be extracted to be displayed in place of the original value.

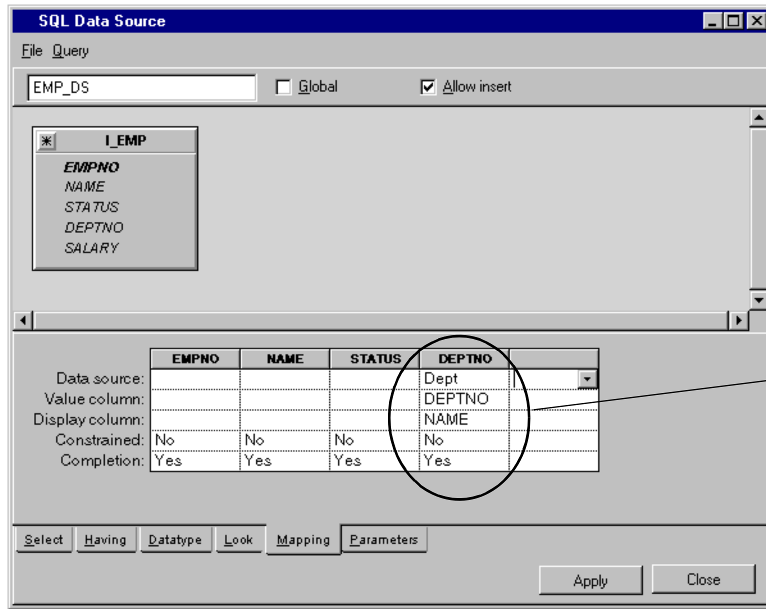


Figure 6.7 Specifying a Foreign Table for a Column of an SQL Table

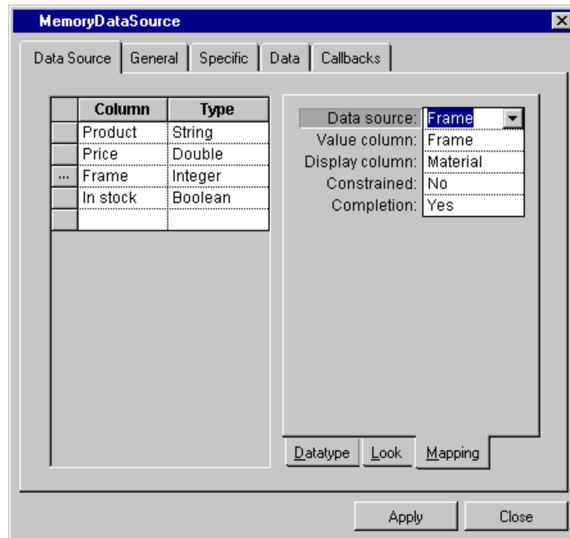


Figure 6.8 Specifying a Foreign Table for the "Frame" Column of a Memory Table

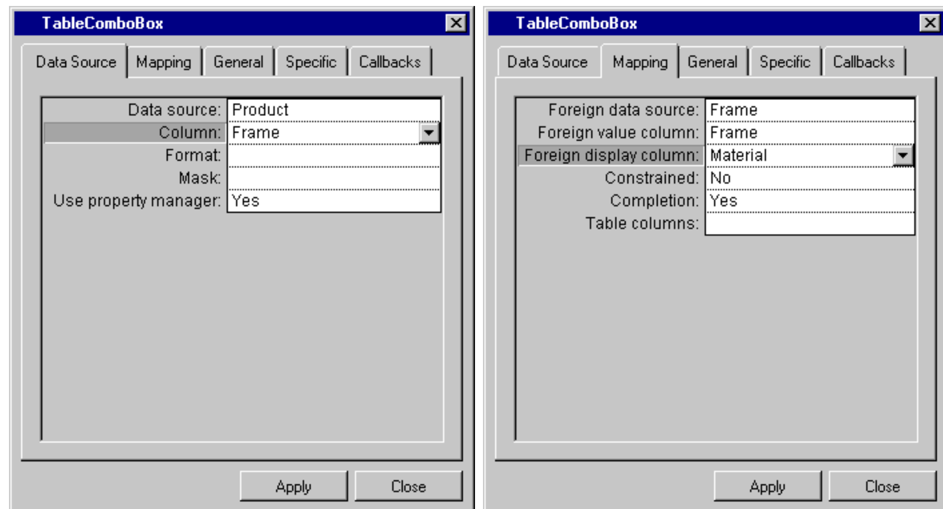


Figure 6.9 *Specifying a Foreign Table for an IliTableComboBox*

Using a Foreign Table to Convert Values

If you are using the foreign table to convert values, the value and display columns will be different columns in the foreign table. The values in a table object column are converted in the following way.

For each value in the column:

1. The value is searched for in the foreign table value column.
2. The row in which it is found is identified.
3. The foreign table value that is in the display column and the identified row is returned and displayed in place of the original column value.

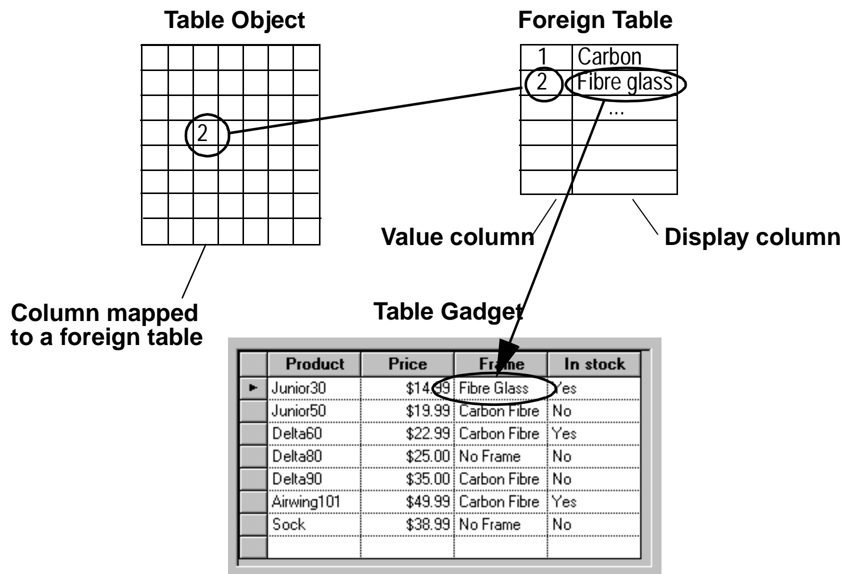


Figure 6.10 Using a Foreign Table to Convert Values

In addition, a combo box is automatically displayed on a cell in a column that is connected to a foreign table. The combo box menu contains all the possible values that the cell can accept, that is, all the values that are contained in the foreign table display column.

This mechanism is completely reversed if the user edits a value in the display gadget.

1. The new value is searched for in the foreign table display column.
2. The row in which it is found is identified.
3. The value in the value column and the identified row, of the foreign table is returned to the table object (replacing the display value).

> ***Note:*** *A table column that uses a foreign table to convert values is automatically set as Constrained. This prevents any inconsistency that can arise from the user entering a new value that is not in the foreign table display column.*

Using the Toggle Selector

When a `DbField` has a foreign table, it is possible to display the value domain. This can be done by selecting a particular style for the `IliDbField`.

You can use table combo box style to show the value domain in a pop-up menu. Another useful style, however, is the `IliToggleSelectorStyle`. This style enables you to display a set of toggle gadgets that show the domain of values.

Each toggle in the toggle selector corresponds to a foreign table row. Only one toggle can be checked at a time.

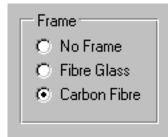


Figure 6.11 *A Toggle Selector Connected to a Column with a Foreign Table*

Using a Foreign Table to Constrain Values

A foreign table can also be used to constrain values. To use a foreign table in this way, specify a value and a display column that are the same column. In addition, set the Constrained option in the appropriate inspector to `Yes`.

This ensures that only the values contained in the foreign table display (value) column can be entered by the user.

Using the Forms Assistant with Foreign Tables

When you create an automatic form using the Forms Assistant option in the Data Access menu of IBM ILOG Views Studio, a column that already has a foreign table will automatically have a combo box for a field.

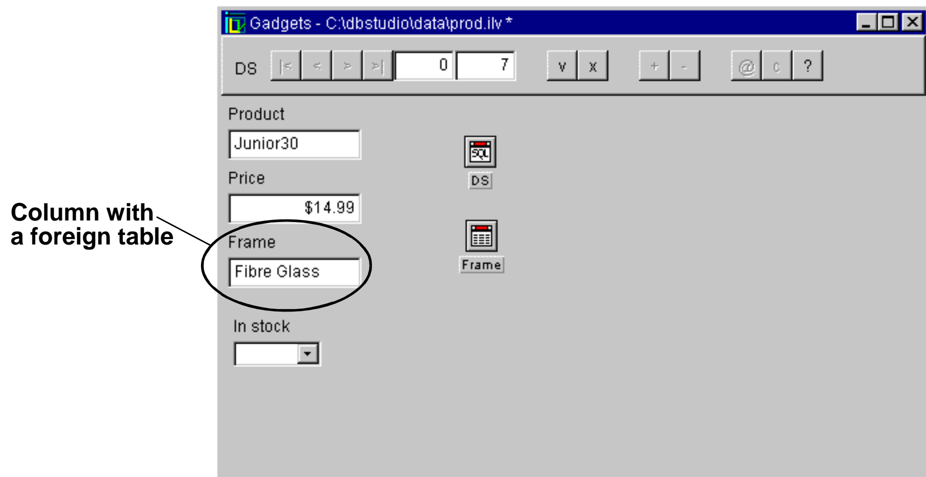


Figure 6.12 A Form Showing a Field Connected to a Column with a Foreign Table

This happens only when the foreign table has been tied to a column via the data source.

Setting the Table Look

The SQL Data Source and the Memory Data Source inspectors provide a Look page that allows you to set several properties. These properties are special in that they apply to any table gadgets or forms (created using the Forms Assistant) that are connected to the data source. They include the text alignment in columns, the column width, the column read/write permissions, and the column visibility.

Column Geometry

When a number of table gadgets are connected to the same data source, column geometry is controlled globally via the data source. This means that if you manually change the column width in one table gadget, the change will automatically be reflected in any other table gadget that is connected to the same data source.

If you want two table gadgets that are connected to the same data source to have different column geometry, you can disconnect them from each other via the Columns Geometry property of the table gadget inspector.

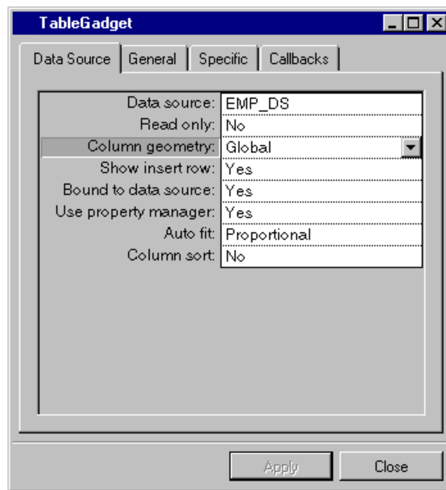


Figure 6.13 *Setting Columns Geometry in the Table Gadget Inspector*

By default, this property is set to Global, which means that the table gadget inherits the properties of the data source table. If you want to work on a table gadget without your changes affecting other table gadgets, you must set the Column Geometry to Local. You can work on this table gadget locally.

If, at any time, you reset Column Geometry to Global, the table gadget is immediately set to the geometry specified for the data source table.

> ***Note:*** *If you want two table gadgets that are connected to the same data source to have a different set of visible columns, set up the SQL data source so that the appropriate columns for one of the table gadgets are visible. Then disconnect one of the table gadgets by setting it with a local column geometry. You can then set up the columns just for the second table gadget.*

The column geometry properties that can be controlled on a global or local level are column width, column visibility, and column position.

If a table gadget has local column geometry, a column can be picked up and dragged to a new horizontal position in the table gadget. However, if it has a global column geometry, this same procedure can only be carried out in the SQL Data Source inspector. In this case, it applies to all table gadgets with a global column geometry.

Read-Only Settings

A table object column can be set as read-only. This means that the values in the column can only be consulted and not modified. If you set up a column as being read-only in the data source inspector, this will apply to any connected table gadget or form that is subsequently

created with the Forms Assistant. However, in the `DbField` that make up the form, the read-only property of a column can be changed locally.

Fixed Columns

A table gadget can be set so that when you horizontally scroll the table a set number of the columns on the far left side of the table always stay in position. This is useful for columns that contain key information, such as the record name.

To set the columns that will be fixed, you must access the Table Gadget inspector and use the Fixed Column field of the inspector panel to set the number of columns that you require. These are counted from the column the farthest to the left in the table schema. Therefore, you should make sure that the columns that you want fixed are on the left of the data source table.

For example, if you enter 2 in the Fixed Columns field, the two columns farthest to the left in the data source table will always be displayed in the table gadget.

The methods in the API that implement this feature are `getFixedColumnsCount` and `setFixedColumnsCount` in the `IliTableGadget` class. Because this is a direct property of the `IliTableGadget` class, any other table gadget that is linked to the same data source can have a different number of fixed columns (or no fixed columns at all).

Troubleshooting

This section describes some of situations that you should be careful to avoid as they can cause unexpected results in your application.

You can find information on the following topics:

- *Avoiding Common Names in Foreign Tables*
- *Matching Types with a Foreign Table*

Avoiding Common Names in Foreign Tables

When a gadget is tied to a table column and has a foreign table, you must ensure that the Data Source and Column gadget values are never the same as its Foreign Data Source and Foreign Value Column/Foreign Display Column.

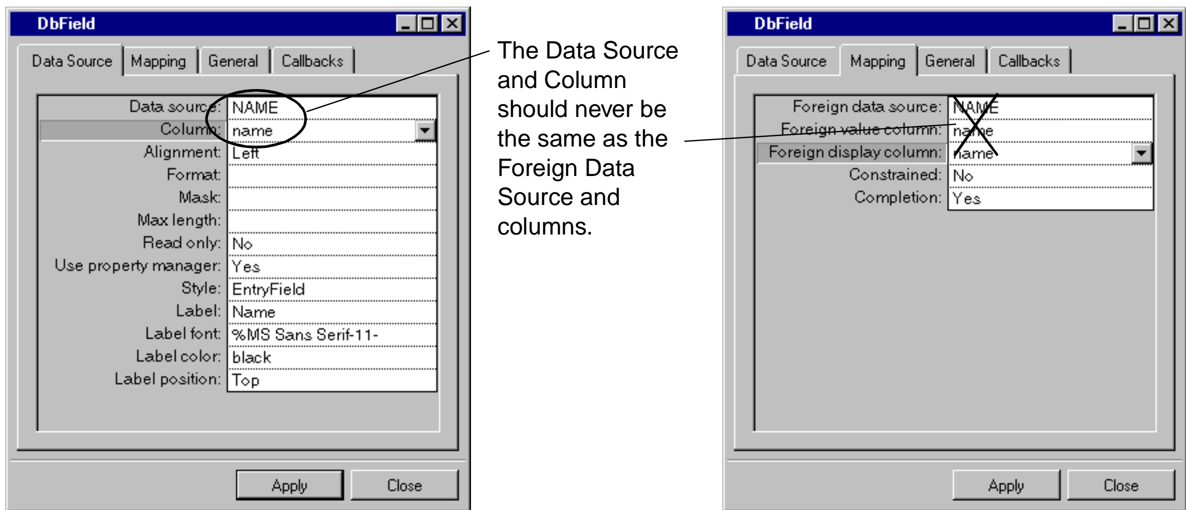


Figure 6.14 A DbField with the Same Data Source and Foreign Data Source

You can be tempted to set up a combo box this way so that you can select the current row based on values in one of the columns. This step has a more disturbing result.

Data Source and Column are used to specify the data source column to which the gadget is tied. This means that values from this column can be displayed and edited via the gadget. Foreign Data Source and Foreign Value Column/Foreign Display Column specify the foreign table of a gadget.

Take the case of a gadget that is tied to a column displaying values from that column and that also has a combo box. The combo box menu will contain the set of possible values that the column can take, which will be the same set of values. If you then select a new value for a row, you will end up with a column that starts to have repeating values.

To successfully set up a combo box that enables you to select the current row in a form, you must use a parameter. See *Parameters* on page 139.

Matching Types with a Foreign Table

When a gadget has a foreign table, the actual value of the column to which it is tied is compared with a value column in the foreign table to identify a value to be displayed from the foreign table display column. It is therefore important to ensure that the type of gadget column and the foreign table value column are the same.

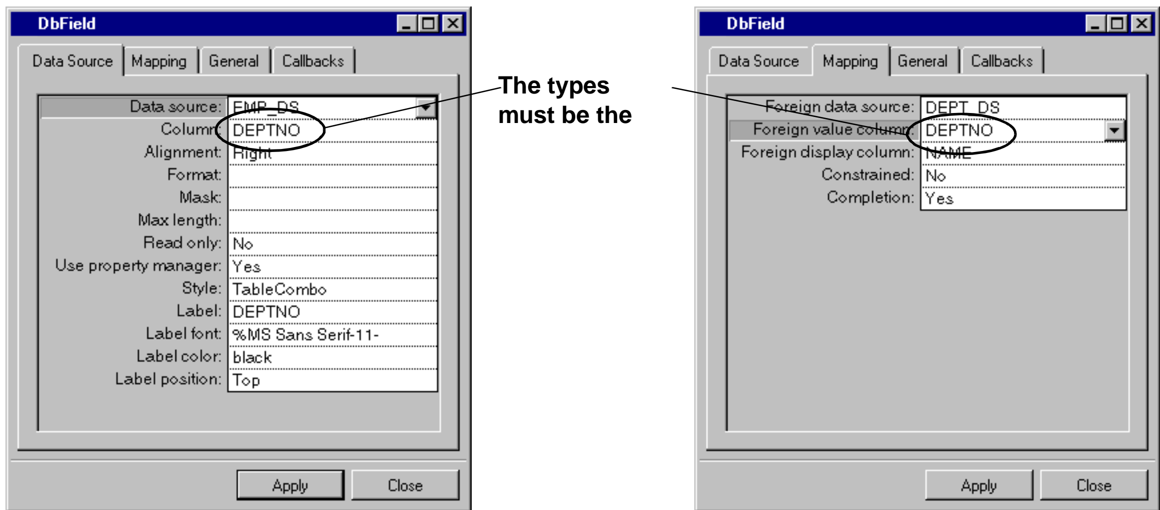


Figure 6.15 A DbField Inspector with a Foreign Table

If the type is not the same, the comparison between the actual value and the value column is not possible. Values must be of the same type for a comparison to produce a useful result.

Part II

Data Access and SQL

This part describes how to use Data Access when connected to an SQL database. It describes SQL tables and data sources, and how to connect to a database.

CHAPTER

7

SQL Tables

You were given an overview of table objects in Chapter 3, *Tables*. An SQL table is a table object that is used when Data Access is connected to an SQL database. This chapter describes the SQL table object in more detail.

You can find information on the following topics:

- *Introduction*
- *Structural Definition*
- *The SQL Session of an SQL Table*
- *Run-Time Options*
- *Parameters*
- *Transaction Manager*
- *Structured Types*
- *Asynchronous Mode*

Introduction

The `IliSQLTable` class defines an object that manages rows on a remote relational database management system. It is created automatically when an `IliSQLDataSource` is created. Each `IliSQLDataSource` object has an associated `IliSQLTable`.

This chapter explains how `IliSQLTable` objects are defined and subsequently used.

Structural Definition

Each `IliSQLTable` object must have an associated SQL SELECT statement that is submitted to the database system whenever the rows of the SQL table object need to be recomputed.

The following shows an example of an SQL SELECT statement:

```
SELECT EMP.ID, EMP.NAME, EMP.DEPTNO
FROM EMP
WHERE EMP.DEPTNO = 3
ORDER BY 2
```

When you submit this statement, the result is an ordered collection of rows, each of which has three values (EMP.ID, EMP.NAME and EMP.DEPTNO). These rows are extracted from the database table named EMP and they are sorted by NAME (`ORDER BY 2`). Not all rows are extracted from table EMP: only the rows whose EMP.DEPTNO is equal to 3. In other words, this SQL statement returns the ID, NAME, and DEPTNO of all employees working in department number 3.

An SQL SELECT statement can have more than one database table referenced in the FROM clause. Here is an example:

```
SELECT EMP.ID, EMP.NAME, DEPT.NAME
FROM EMP, DEPT
WHERE EMP.DEPTNO = DEPT.ID
```

The rows produced by this query are obtained by combining rows from the database tables EMP and DEPT through a *join* operation.

An `IliSQLTable` object can be defined so that its SQL query resembles one of the statements shown above. Two different techniques can be used to achieve this. The following sections describe these techniques: creating the definition interactively using IBM ILOG Views Studio and creating the definition in pure C++.

Creating the Definition Using IBM ILOG Views Studio

The `IliSQLTable` can be defined interactively in IBM ILOG Views Studio through the `IliSQLDataSource` object inspector.

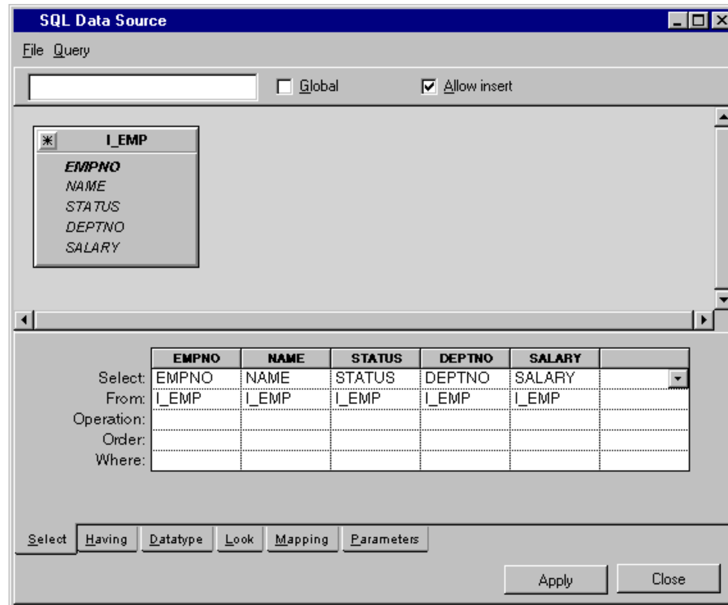


Figure 7.1 The SQL Data Source Inspector

Within SQL Data Source inspector, you can:

- Specify the database tables to be referenced by the FROM clause (the Add Tables menu item).
- Specify the joins between tables when multiple database tables are added. These joins can be specified by dragging a column from one table to a column in another table. This results in a condition of the form `Table1.Column1 = Table2.Column2` being added to the `WHERE` clause of the SQL SELECT statement.
- Specify the columns of the `IliSQLTable` object. These columns are derived from the columns of the database tables in the following way: drag a column belonging to a database table and drop it over the SELECT area in the inspector. This will create a new column in the schema of the `IliSQLTable` object.

  Alternatively, the SELECT area in the inspector contains an undefined column that allows you to type in the definition of a new column. The rows labeled Select and From are used to specify how the `IliSQLTable` column relates to the database table. In the From row, enter the name of the database table from which the column is derived. In the

Select row, enter the name of the database table column from which the column is derived.

If you want to define an `IliSQLTable` column that is derived from multiple database columns, leave the From row empty and, in the Select row, enter an SQL expression that computes the column value (for example, `PRICE * QTY`).

- ◆ Specify the `WHERE` clause by typing the selection criteria in the Where row. If you enter conditions for multiple columns, these conditions will be combined with an `AND` logical operator in the SELECT statement. Once you enter a condition in the Where row, a new row labeled Or appears. In the Or row, you can type a new set of conditions that will be combined, via an `OR` logical operator, with the Where row conditions. There can be any number of Or rows.
- ◆ If the SELECT statement contains a `GROUP BY` clause or if it contains aggregate functions such as `COUNT` or `SUM`, enter the appropriate operations in the Operation row. Note that in this case, all columns must have a specified operation and that conditions should be entered in the Having row instead of the Where row.
  - Specify the `ORDER BY` clause in the "Order By" row.
  - Specify other column properties in the Datatype, Look, and Mapping notebook pages.
  - Specify the parameters used in the `WHERE` clause in the Parameters page.
  - If duplicate rows should be removed from the result produced by the SELECT statement, the Distinct property should be set to Yes.
  - In addition to performing an SQL SELECT statement, an `IliSQLTable` object is able to forward user updates to the database. This involves generating SQL UPDATE, INSERT, and DELETE statements on-the-fly. Note that when more than one database table is added to a data source, only one of the database tables can be updated in this way. Ensure that the updatable Table property contains the database table name that will be changed through the `IliSQLTable` object.

At this point, the `IliSQLTable` is structurally defined. This means that its schema is defined and the process that it should use to compute its rows from the tables in the database is also defined.

Creating the Definition in C++

All of the steps described in the previous section can also be carried out by coding in C++. This following example shows how this can be done:

- ◆ Create the `IliSQLTable` object

```
IlvDisplay* display;
...
IliSQLTable* sqlTbl = new IliSQLTable(display);
sqlTbl->lock();
```

◆ Specify the database tables

```
IliSQLTableRef refEMP("EMP", "SCOTT");
IliSQLTableRef refDEPT("DEPT", "SCOTT");
IlInt tblEMP = sqlTbl->addTable(refEMP);
IlInt tblDEPT = sqlTbl->addTable(refDEPT);
```

◆ Specify the joins

```
sqlTbl->addJoin(tblEMP, "DEPTNO", tblDEPT, "ID");
```

◆ Specify the columns

```
IlInt cID = sqlTbl->appendColumn("ID", IliIntegerType);
sqlTbl->setColumnPartOfKey(cID, IlvTrue);
sqlTbl->setColumnSQLText(cID, "ID");
sqlTbl->setColumnTable(cID, tblEMP);

IlInt cNAME = sqlTbl->appendColumn("NAME", IliStringType);
sqlTbl->setColumnSQLText(cNAME, "NAME");
sqlTbl->setColumnTable(cNAME, tblEMP);

IlInt cDEPT = sqlTbl->appendColumn("DEPT", IliStringType);
sqlTbl->setColumnSQLText(cDEPT, "NAME");
sqlTbl->setColumnTable(cDEPT, tblDEPT);
```

A computed column could be defined in the following way (note that the following code excerpt is not part of the example).

```
IlInt cTOTAL = sqlTbl->appendColumn("TOTAL", IliIntegerType);
sqlTbl->setColumnSQLText(cTOTAL, "PRICE * QTY");
```

◆ Specify the criteria

```
IlInt where = 0;
sqlTbl->insertConjunct(where,IlvTrue);
sqlTbl->setColumnPredicat(cNAME, where, "<> 'Smith'", IlvTrue);
```

◆ Specify the sort

```
sqlTbl->setColumnOrderBy(cNAME, IliSQLAscending);
```

◆ Specify the updatable table

```
sqlTbl->setTableUpdatable(tblEMP, IlTrue);
```

◆ Generate the SQL SELECT statement

```
sqlTbl->makeQuery();
```

At this point, the `IliSQLTable` object is defined and ready for use. Calling the `IliSQLTable::getQuery` member function generates the following SQL statement:

```
SELECT EMP.ID, EMP.NAME, DEPT.NAME
FROM SCOTT.EMP, SCOTT.DEPT
WHERE EMP.DEPTNO = DEPT.ID
```

```
  AND EMP.NAME <> 'Smith'
ORDER BY 2
```

A Shortcut C++ Definition

The `IliSQLTable::setQueryFrom` member function allows you to define an `IliSQLTable` that has only one database table. Here is an example:

```
IlvDisplay* display;
...
// Create the IliSQLTable object.
IliSQLTable* sqlTbl = new IliSQLTable(display);
sqlTbl->lock();

// Define its session.
IliSQLSession* session;
session  = new IliSQLSession("oracle", "scott/tiger@orasrv");
sqlTbl->setSQLSession(session);

// Define the IliSQLTable object.
IliSQLTableRef tblRef("EMP", "SCOTT");
sqlTbl->setQueryFrom(tblRef);
```

The `setQueryFrom` member function reads the schema of the given database table and defines the `IliSQLTable` object accordingly.

The SQL Session of an SQL Table

An `IliSQLTable` object must have a properly defined SQL session before any real work can be done. The SQL session handles all requests sent to a database server. See *SQL Sessions and Cursor Objects* on page 149.

When defining an `IliSQLDataSource` gadget in IBM ILOG Views Studio through the inspector, the SQL session can be edited through the `Connection` property. Editing this property causes a connection dialog box to be displayed in which the user can choose one of the following options:

- Creating a custom session by entering all required connection parameters.
- Selecting the name of an application-wide session.

An SQL session can be shared among several `IliSQLTable` objects by selecting the same application-wide session for each of them.

The rest of this section describes how these actions can be carried out in C++.

A custom session can be defined as follows:

```
IliSQLTable* sqlTbl;
...
IliSQLSession* session;
```

```
session  = new IliSQLSession("oracle", "scott/tiger@orasrv");
sqlTbl->setSQLSession(session);
```

An application-wide session can be selected as follows:

```
IliSQLTable* sqlTbl;
...
IliSQLSession* session;
session = IliSQLSession::GetRegisteredSession("Main");
sqlTbl->setSQLSession(session);
```

The SQL session that an `IliSQLTable` object uses to communicate with the database server can be retrieved in the following way:

```
IliSQLSession* session = sqlTbl->getEffectiveSQLSession();
```

Run-Time Options

This section contains information on the options that affect the way an `IliSQLTable` object behaves. These options can be set through the API using the appropriate member functions in the `IliSQLTable` class.

You can find information on the following topics:

- *Concurrency Control*
- *Auto-Commit Mode*
- *Fetch Policy*
- *Auto-Refresh Mode*
- *Inserting-Nulls Mode*
- *Dynamic-SQL Mode*
- *Bound Variables Mode*
- *Cursor Buffering*
- *Auto-Row Locking Mode*

Concurrency Control

In a client-server environment, there can be multiple client programs simultaneously accessing the same data in the database server. When concurrency control is enabled, any row updates or deletions in an SQL table will succeed only if the rows concerned have not changed in the database since the last time they were fetched and stored in the local memory cache. In other words, concurrency control obliges the SQL table to protect the work carried out through it from any changes made by other database users.

This is achieved by adding extra conditions in the SQL statements that are generated when a row is updated or deleted.

For example, assume that the SQL SELECT statement of an `IliSQLTable` object is the following and that `ID` is the primary key of the `EMP` table:

```
SELECT ID, NAME
FROM EMP
```

If the name of an employee whose ID is 6 is changed from “Smith” to “Jones”, the resulting SQL UDPATE statements will be as follows:

- Without concurrency control

```
UPDATE EMP
SET NAME = ‘Jones’
WHERE ID = 6
```

- With concurrency control enabled

```
UPDATE EMP
SET NAME = ‘Jones’
WHERE ID = 6
  AND NAME = ‘Smith’
```

The latter statement will fail if the name of the employee has been changed by another user.

> ***Note:*** *The technique used to deal with concurrency control is referred to as optimistic concurrency control. It does not explicitly lock rows at read time. For more information on how rows can be locked at read time using a technique known as pessimistic concurrency control, see* *Auto-Row Locking Mode on page 127.*

Auto-Commit Mode

When the auto-commit mode is enabled, the `IliSQLTable` object commits the transaction immediately after an INSERT, UPDATE or DELETE operation. When auto-commit is disabled, the transaction must be committed (through the effective SQL session) as required.

Here is an example:

```
IliSQLTable* sqlTbl;
...
// Delete two rows and then commit.
sqlTbl->setAutoCommit(IlvFalse);
sqlTbl->deleteRow(10);
sqlTbl->deleteRow(9);
sqlTbl->getEffectiveSQLSession()->commit();
```

Fetch Policy

Data Access supports two ways of fetching rows from the database after a successful SELECT operation: either all rows are fetched at once at select time or the retrieval of rows is delayed until required. The fetch policy of an SQL table can be set using the `IliSQLTable::setFetchPolicy` member function.

Here is an example where the fetch policy is *immediate*:

```
IliSQLTable* sqlTbl;
...
sqlTbl->setFetchPolicy(IliFP_Immediate);
sqlTbl->select();
// All rows have been fetched and are now available locally.
IlInt rowsCount = sqlTbl->getRowsCount();
// The rows count is accurate.
```

The advantage of the *immediate* fetch policy is that the row count is accurate, since the `IliTable::select` member function retrieves all the database rows at once. However, the disadvantage is that if the SQL SELECT statement retrieves ten thousand rows, the `IliTable::select` function will incur serious overhead.

Alternatively, the `IliSQLTable` object can delay the retrieval of rows until necessary. Here is an example:

```
IliSQLTable* sqlTbl;
...
sqlTbl->setFetchPolicy(IliFP_AsNeeded);
// Auto-commit must be disabled.
sqlTbl->setAutoCommit(IlFalse);
sqlTbl->select();
// Rows have not yet been fetched.
IlInt initialRowsCount = sqlTbl->getRowsCount();
// The initial rows count equals 0.
sqlTbl->fetchNext(10);
IlInt halfWayRowsCount = sqlTbl->getRowsCount();
// Now, 10 rows are available locally.
sqlTbl->fetchAll();
IlInt totalRowsCount = sqlTbl->getRowsCount();
// All rows have been fetched and are now available locally.
```

Note that the auto-commit mode must be disabled for this way of retrieving rows to be effective.

Auto-Refresh Mode

When auto-refresh mode is enabled, an INSERT or UPDATE operation on the database is followed immediately by a request to the database for the inserted or updated row. The row obtained via this request replaces the inserted or updated row in the `IliSQLTable` object.

This is useful in either of the following situations:

- The database contains triggers that can alter inserted or modified rows.
- The selected list of the SQL SELECT statement contains formulas (for example, `PRICE * QTY`) that need to be recalculated each time the row changes.

Inserting-Nulls Mode

When inserting a row in a database table, the row can contain one or more NULL values.

Here is an example of the SQL statement when the ADDRESS of employee Williams is NULL and the *insert-nulls* property is enabled:

```
INSERT INTO EMP(ID, NAME, ADDRESS) VALUES(7, 'Williams', NULL)
```

Alternatively, the following example shows the SQL statement when the insert-nulls property is disabled:

```
INSERT INTO EMP(ID, NAME) VALUES(7, 'Williams')
```

Dynamic-SQL Mode

When a row is edited, it can contain some modified values and some values that remain unchanged. For instance, assuming the address of employee Williams is modified, the following example shows the SQL statement that would be generated if dynamic-SQL is enabled:

```
UPDATE EMP
SET ADDRESS = '16, Chocolate Street'
WHERE ID = 7
```

Here is the SQL statement if dynamic-SQL is disabled:

```
UPDATE EMP
SET ID = 7,
    NAME = 'Williams',
    ADDRESS = '16, Chocolate Street'
WHERE ID = 7
```

In the latter case, the values for all the columns are sent back to the database each time an update takes place.

Bound Variables Mode

Most database systems support the use of *bound variables* in SQL statements. When bound variables are used, the SQL statements contain variable markers. Here is an example:

```
UPDATE EMP
SET NAME = ?
WHERE ID = ?
```

The question marks represent variables. The value for a variable is provided separately. The advantage of using bound variables is that the same SQL statement can be reused even though the values involved change each time. Using the same SQL statement each time instead of having many SQL statements can offer a serious boost in performance. This is because the time spent in parsing an SQL statement and selecting an access plan on the database server can be significant.

Cursor Buffering

Cursor buffering is a way to obtain better throughput when fetching a large number of rows. It consists of having the lower layers of the database library fetch more than one row at a time from the database server.

By default, rows are obtained one at a time from the database server. This can seriously slow down the application when many rows have to be fetched since a network round-trip will be required for each row.

The `IliSQLTable::setCursorBufferedRowsCount` member function lets you specify how many rows can be fetched at one time.

```
IliSQLTable* sqlTbl;
...
sqlTbl->setCursorBufferedRowsCount(15);
```

Note that this will result in better throughput only with database servers that support this feature, which currently are Oracle and Sybase.

Auto-Row Locking Mode

The `IliSQLTable` class has a `refreshAndLockRow` member function that rereads a given row and attempts to acquire a lock on this row in the database. This is useful if you want to implement the “pessimistic concurrency control policy”.

The `IliDataSource` class has “auto-row locking” mode in which it automatically calls the `refreshAndLockRow` member function of the underlying table whenever the end user starts modifying the current row.

Here is how the “auto-row locking” mode can be enabled:

```
IliSQLDataSource* sqlDs;
...
sqlDs->enableAutoRowLocking(IlvTrue);
```

> ***Note:*** *When using an Oracle database system, it is possible to request that the* `IliSQLTable::refreshAndLockRow` *member function includes the "NOWAIT" clause in the SQL SELECT statement it uses to lock the row. By doing this, the end user will not wait when locks are held by other database users. See the* `IliSQLTable::enableNoWaitOnLockRow` *member function for more information.*

Parameters

The SQL SELECT statement of an `IliSQLTable` object can contain references to parameters. See *Parameters* on page 139. Here is an example:

```
SELECT ID, NAME, ADDRESS
FROM EMP
WHERE NAME = :name_p
```

In this example `name_p` is the name of a parameter and as such it must be preceded by a colon ":" in the SQL statement.

In addition, the parameter must be declared in the `IliSQLTable` object. This can be done either interactively through the SQL Data Source inspector in IBM ILOG Views Studio or in C++ as follows:

```
IliSQLTable* sqlTbl;
...
sqlTbl->appendParameter("name_p", IliStringType);
```

A value should be assigned to this parameter before the `IliTable::select` member function is called. This code extract only shows how the parameter is defined in C++. For a complete description of the code required to generate this SQL statement, see *Creating the Definition in C++* on page 120.

```
IliValue v("Smith");
sqlTbl->setParameterValue("name_p", v);
sqlTbl->select();
```

The parameter can subsequently be assigned other values as required.

Transaction Manager

The `IliTransactionManager` class manages so-called local transactions in which the propagation of changes of one or more `IliTable` objects is deferred. This is especially useful when using `IliSQLTable` objects. In effect, each time a row is inserted, updated, or

deleted in an IliSQLTable object, one or more corresponding SQL statements are submitted to the database server to which the SQL table object is connected.

Although it is necessary to send SQL statements immediately when the IliSQLTable object is in auto-commit mode, it can be useful to retain these statements when not in auto-commit mode. Instead, the user could be allowed to make a number of changes to one or more IliSQLTable objects. The changes would be effective in local row cache so that they are reflected in all connected gadgets, but the corresponding SQL statements are retained. Later, when the user validates the changes, all SQL statements are submitted to the database server.

If all statements are accepted by the server, the transaction has succeeded and the user can issue a database commit. However, if at least one of the statements fails, the user should issue a database rollback to cancel any statement that would have succeeded before it. The user can then either make some additional changes and retry the validation process or cancel all changes locally so that the local row cache(s) of the IliSQLTable object(s) revert their state(s) to what they were before the first change.

The set of changes that have been effected in the local row cache(s) of the IliSQLTable object(s) and the corresponding SQL statements that have been retained are called a local transaction.

To participate in a local transaction, an IliTable object must be managed by an instance of the IliTransactionManager class.

A transaction manager is anonymous or named. The name space for a transaction manager is global. There can be at most one transaction manager with a given name.

The simplest way to assign a transaction manager to an IliTable object is to assign its transactionManagerName property. If a name is assigned to the property and there is no transaction manager with that name, a transaction manager is automatically created and is assigned to the IliTable object. Other table objects can then be made to share this transaction manager by assigning the same name to their transactionManagerName property. The transaction manager can then be obtained by accessing the transactionManager property of one of the table objects.

The IliSQLTable and IliMemoryTable classes are currently transaction-manager-aware classes.

Note that the following restrictions apply to the use of a transaction manager:

- The IliTable::select and IliTable::clearRows member functions should not be called on table objects involved in a pending local transaction.
- Nested tables should not be managed by a transaction manager. See the *Parameters* on page 128 for more information.

The following code sample shows how a transaction manager is used:

```
class MyPanel: public IlvGadgetContainer {
public:
  IliSQLTable* getTableEMP() const { ... }
  IliSQLTable* getTableDEPT() const { ... }

  MyPanel(IlvDisplay* dpy)
    : IlvGadgetContainer(dpy, ...)
    { ... }

void initPanel() {
    // Could be done in IBM ILOG Views Studio without coding,
    // shown here for exposition.
    // Ensure both tables are using the same transaction manager.
    getTableEMP()->setTransactionManagerName("TRANS_MGR");
    getTableDEPT()->setTransactionManagerName("TRANS_MGR");

    // And the same session.
    IliSQLSession* session = getTableEMP()->getSQLSession();
    getTableDEPT()->setSQLSession(session);

    // Do not auto-commit.
    getTableEMP()->setAutoCommit(IlvFalse);
    getTableDEPT()->setAutoCommit(IlvFalse);
  }

  IliTransactionManager* getTransMgr() const
    { return getTableEMP()->getTransactionManager(); }

  IliSQLSession* getSQLSession() const
    { return getTableEMP()->getEffectiveSQLSession(); }


  IlBoolean isTransactionStarted() const
    { return getTransMgr()->isStarted(); }

  void startTransaction()
    { getTransMgr()->start(); }

  IlBoolean acceptTransaction() {
    if (getTransMgr()->accept()) {
      getTransMgr()->stop();
      getSQLSession()->commit();
      return IlTrue;
    }
    else {
      getSQLSession()->rollback();
      return IlFalse;
    }
  }

  void cancelTransaction() {
    getTransMgr()->cancel();
    getTransMgr()->stop();
  }
};
```

Structured Types

Some database systems (for example, Oracle 9i or later, Informix 9.x) support extensible type systems in which the type of a database table column is not limited to the traditional scalar data types. It can also be either a structured type or a collection type.

Data Access represents values of the structured types as `IliMemoryTable` objects.

For example, a database table called PRODUCT can be defined in an Oracle 8.x database system as follows:

```
CREATE TYPE PART_T AS OBJECT(
  PARTNO INTEGER NOT NULL,
  PARTNAME VARCHAR(20) NOT NULL);

CREATE TYPE PART_TABLE_T AS TABLE OF PART_T;

CREATE TABLE PRODUCT(
  PRODNO INTEGER NOT NULL PRIMARY KEY,
  PRODNAME VARCHAR2(50) NOT NULL,
  PARTS PART_TABLE_T);
```

An SQL data source named PRODUCT_DS based on the PRODUCT table can then be created. All three columns of the PRODUCT table should be included in the PRODUCT_DS data source.

The following code extract shows how the contents of the PARTS column can be used in IBM ILOG Script.

```
IliSQLDataSource prodDS = ...;
IliSQLTable* prodTable = prodDS->getSQLTable();
for (IlInt prodIdx = 0; prodIdx < prodTable->getRowsCount(); ++prodIdx) {
  const IliTable* partsTable = prodTable->at(prodIdx, "PARTS").asTable();
  const char* prodName = prodTable->at(prodIdx, "PRODNAME");
  if (partsTable != NULL) {
    IlvPrint("Product %s parts:", prodName);
    IlInt partCount = partsTable->getRowsCount();
    for (IlInt partIdx = 0; partIdx < partCount; ++partIdx) {
        const char* partName = partsTable->at(partIdx, "PARTNAME");
        IlvPrint(" %s", partName);
    }
  }
  else
  IlvPrint("Product %s does not have parts", prodName);
}
```

The following code extract shows how a new row can be inserted in the PRODUCT table.

```
IliSQLDataSource prodDS = ...;
IliSQLTable* prodTable = prodDS->getSQLTable();

// Create the nested PARTS table.
IlInt partsColno = prodTable->getColumnIndex("PARTS");
const IliDatatype* type = prodTable->getColumnType(partsColno);
IliTable* partsTable = type->makeTable();
partsTable->lock();

IliTableBuffer* partsBuf = partsTable->getBuffer();

// Insert the first part.
partsBuf->at("PARTNO") = (IlInt)610;
partsBuf->at("PARTNAME") = "Drawer";
partsTable->appendRow(partsBuf);

// Insert the second part.
partsBuf->at("PARTNO") = (IlInt)611;
partsBuf->at("PARTNAME") = "Handle";
partsTable->appendRow(partsBuf);

partsTable->releaseBuffer(partsBuf);

// Insert the new product.
IliTableBuffer* prodBuf = prodTable->getBuffer();
prodBuf->at("PRODNO") = (IlInt)61;
prodBuf->at("PRODNAME") = "Dresser";
prodBuf->at("PRODNAME") = partsTable;
prodTable->appendRow(prodBuf);
prodTable->releaseBuffer(prodBuf);

partsTable->unLock();
```

In both examples shown above, `prodTable` designates an `IliSQLTable` whereas `partsTable` designates a nested `IliMemoryTable`. Care must be taken not to modify the nested memory table. Doing so would inevitably lead to inconsistencies between the Data Access application and the database. Instead, a PARTS_DS SQL data source based on the PRODUCTS.PARTS nested table can be created. This data source is defined as follows through the SQL Data Source inspector:

1. Set the data source name to PARTS_DS.

2. Add the PRODUCT database table by selecting Add Tables... from the Query menu of the inspector panel.

3. In the PRODUCT table, open the PARTS column by clicking on the + sign that appears to the left of the column name.

4. Drag the PARTNO and PARTNAME columns and drop them in the SELECT section.

5. To edit the definition of the PRODUCT table, double-click on the PRODUCT table. In the Table definition dialog box that appears, make sure that the following items are defined:

Parent: `PRODUCT_DS`

Alias: `PRODUCT`

The PARTS_DS data source can be used in two different settings:

- The PRODUCT_DS data source does not include the PARTS column. Instead, the PARTS_DS data source retrieves the contents of the nested PARTS table each time its `select` method is called. Note that in this case, there are no nested `IliMemoryTable` objects involved. Instead there are two `IliSQLTable` objects, one of them belongs to PRODUCT_DS and the second one belongs to PARTS_DS.
- The PRODUCT_DS data source does include the PARTS column. Consequently, there are many nested `IliMemoryTable` available, one for each row in the PRODUCT_DS data source. The PARTS_DS data source can be used in this setting to edit any nested `partsTable` `IliMemoryTable` objects contained in the PRODUCT_DS data source as shown in the following code extract:

```
partsMemoryTable->Lock();
IliSQLDataSource* prodDS = ...;
IliSQLTable* prodTable = prodDS->getSQLTable();
IlInt prodIdx = prodDS->getCurrentRow();

const IliTable* partsMemoryTable = prodTable->at(prodIdx,
                                                "PARTS").asTable();
if (partsMemoryTable == NULL) {
  IlInt partsColno = prodTable->getColumnIndex(""PARTS");
  const IliDatatype* type = prodTable->getColumnType(partsColno);
  partsMemoryTable = type->makeTable();
  prodTable->set(prodIdx, "PARTS", IliValue(partsMemoryTable);
  partsMemoryTable = prodTable->at(prodIdx, "PARTS").asTable();
}

IliSQLDataSource* partsDS = ...;
IliSQLTable* partsSQLTable = partsDS->getSQLTable();
partsSQLTable->setCache(partsMemoryTable);
partsMemoryTable->unLock();
```

Once the nested memory table has been assigned to the cache property of the PARTS_DS table, it can be edited as any SQL table through PARTS_DS.

Note that the PARTS_DS data source assumes that the nested table it is editing belongs to the current row of the PRODUCT_DS data source. As a consequence, it is necessary to adjust the value of the cache property when the PRODUCT_DS data source moves to another row.

Asynchronous Mode

Normally, calls made by a client application to a database server are blocking. This means that the client application has to wait for server replies each time it submits an SQL statement or when it attempts to fetch rows. The effect of this is that the end user may feel that the application is not sufficiently responsive.

Asynchronous mode, which is supported by some database systems, can be used to increase application responsiveness. Instead of blocking until the server responds, asynchronous calls return quickly and the caller must check whether the call has completed (that is, whether the server has responded). If not, the caller is expected to repeat the call until completion. In the meantime, the caller can proceed with other tasks (such as giving the main loop a chance to handle other events).

The advantages of asynchronous mode are that the application can become more responsive to user input and, in addition, the user may be given the opportunity to cancel a long-running task that has taken too much time. The disadvantage is that programming applications with asynchronous calls is more difficult than programming with synchronous calls.

The `IliSQLTable` class supports selecting and fetching rows asynchronously from a database server. It does not, however, support asynchronous insert, update, or delete operations.

A typical way of using asynchronous calls is to set up a timer in the application (see the `IliDbTimer` class) and to proceed as follows. Assume that an SQL data source needs to perform its select call in asynchronous mode.

◆ Put the data source session in asynchronous mode:

```
IliSQLDataSource* ds = ...;
   IliSQLTable* sqlTable = ds->getSQLTable();
   IliSQLSession session = ds->getEffectiveSQLSession();
   session->enterAsyncMode();
```

◆ Ensure that the fetch policy is “Immediate” so that a call to the `select` member function will also fetch all rows:

```
sqlTable->setFetchPolicy(IliFP_Immediate);
```

◆ Call the `select` member function:

```
sqlTable->select();
```

◆ In the timer callback, call `continueAsyncCall` as long as the call to `select` has not completed:

```
void ILVCALLBACK OnTimer(IlvGraphic*, IlAny) {
       IliSQLDataSource* ds = ...;
       IliSQLTable* sqlTable = ds->getSQLTable();
       if (!sqlTable->isAsyncCallCompleted())
          sqlTable->continueAsyncCall();
     }
```

Note that asynchronous mode is not supported by all database systems. Moreover, when it is supported by a database system, support may depend on the database server release number. You should dynamically test whether asynchronous mode is supported as shown in the following code:

```
IliSQLSession* session = sqlTable->getEffectiveSQLSession();
if (session->supportsAsyncMode()) {
...
}
```

Another important issue to consider is that there can be at most one non-completed asynchronous call among all cursors that belong to the same session. This means that you cannot execute asynchronously in parallel two SQL statements through two cursors that belong to the same session. Consequently, it is recommended that an SQL table have a private SQL session (that is, not an application-defined session) when used in asynchronous mode.

C H A P T E R

8

SQL Data Sources

This chapter provides information on SQL data sources, including how to define parameters in an SQL table and SQL data sources.

You can find information on the following topics:

- *Query Mode*
- *Parameters*
- *Working with an SQL Data Source*

Query Mode

The `IliDataSource` and `IliSQLTable` classes support query mode.

When query mode is entered, the data source substitutes a memory table for the SQL table (see `IliDataSource::switchToQueryMode`). This memory table has the same number of columns as the SQL table, but it differs in that all columns in the memory table have a String type. The user can then edit the contents of the memory table through the same set of gadgets that are used to edit the SQL table in regular (nonquery) mode.

Each column of a memory table can contain:

- A literal value (implying the = relational operator)

◆ A value containing the SQL wildcard character % or the underscore character “_” (implying the LIKE SQL operator)

◆ An SQL condition such as:

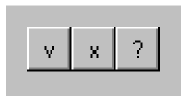
- NULL or IS NULL
- NOT NULL or IS NOT NULL
- LIKE ‘a pattern’
- NOT LIKE ‘a pattern’
- BETWEEN _literal_value AND another_literal_value
- = a_literal_value
- <> a_literal_value
- < _literal_value

A memory table can contain more than one row. All conditions that appear on the same line will be combined with an AND operator when the query is applied. Conditions that appear on different lines will be combined with an OR operator.

Then, the user can apply the query (see `IliDataSource::applyQueryMode`), which will synthesize a portion of the WHERE clause based on the contents of the memory table. This WHERE clause will then be assigned to the SQL table by calling the `IliSQLTable::setQueryConjunct` member function. The `IliTable::select` member function will be called and the data source will revert to using the SQL table instead of the memory table so that all connected gadgets show the contents of the SQL table.

Alternatively, the user can cancel the query (see `IliDataSource::cancelQueryMode`), which simply reverts to using the SQL table instead of the memory table so that all connected gadgets show the contents of the SQL table.

The `IliDbNavigator` gadget has been upgraded to optionally contain a QueryMode button.



The QueryMode button is labelled with a question mark (?). When the QueryMode button is clicked, the SQL data source switches to query mode. When in query mode, the Validate (V) and Cancel (X) buttons of the navigator behave differently than when in regular mode. Clicking on the Validate navigator button applies query mode and clicking on the Cancel navigator button cancels query mode.

Parameters

Parameters can be defined in the SQL table via the data source in IBM® ILOG® Views Studio. These parameters can be used to filter the data that is displayed in a form.

Parameters are defined in terms of a column of a data source table. This means that the content of the data source column is taken to be the value of the parameter at any particular time.

Defining a Parameter

A parameter is defined in the SQL Data Source inspector on the Parameters page. In the table on the Parameters page, you can enter as many parameters as you need.

A parameter is defined by four characteristics:

- Name
- Type
- Name of the data source that it is attached to
- Column of the data source table that defines the parameter

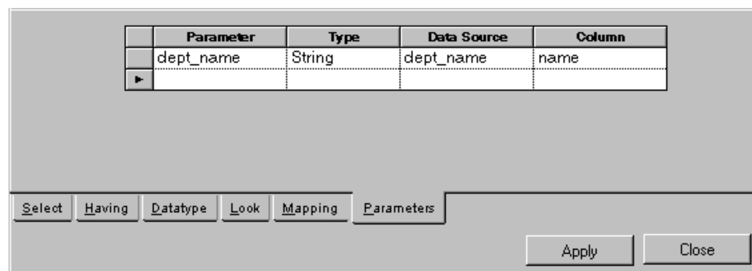


Figure 8.1 *Defining a Parameter in the SQL Data Source Inspector*

> ***Note:*** *When a parameter is defined, the parameter name is used. However, when the parameter is used, its name must be prefixed by a colon ":".*

Defining a Parameter That Accepts User Input

In the IBM ILOG Views *Data Access Getting Started Manual*, you saw how to filter data according to the values in a column of another data source table. It is possible to adapt this behavior to accept user input as a parameter. This can be done using a memory data source.

The following example shows how to filter an employee table to show only those employees that work in a particular department. The example also uses a foreign table. The tables used

in this example are the same as those used in the example described in the IBM ILOG Views *Data Access Getting Started Manual*.

You need to set up a panel containing an SQL data source, a gadget table that is connected to it, and a `DbField` gadget to control record display.

You will start with a table resembling the one shown in the following figure, that is, an SQL data source connected to a table gadget that shows the I_EMP database table:

EMPNO	NAME	STATUS	DEPTNO	SALARY
1	Rochette	2	Administration	6000.0
2	Fernandez	1	Administration	7500.0
3	Goodman	2	Documentation	6300.0
4	Thomasson	1	Marketing	8000.0
5	Williams	2	Marketing	4000.0
6	Wong	2	R & D	7700.0
7	Bornstein	1	R & D	8650.0
8	Harrison	2	Marketing	4575.0
9	Horner	1	Documentation	6375.0
10	Tanaka	1	Finance	9000.0

I_EMP

Figure 8.2 *A Table Gadget Linked to an SQL Table*

> ***Note:*** *Ensure that when you connect to the database you click on the "Keep Password" button. This avoids having to reconnect later.*

The DEPTNO column has a foreign table that is connected to it and that converts a department number to the appropriate department name (See *Foreign Tables* on page 105). The foreign table is a memory table like the one shown in the following figure:

DEPTNO	NAME
1	Administration
2	Marketing
3	R & D
4	Documentation
5	Finance

DEPT

Figure 8.3 *The Foreign Table for the DEPTNO Column*

A new `DbField` gadget and a memory data source should then be created. The `DbField` gadget will be connected to the single column of the memory data source. The memory table must be set up as a single column table that accepts an integer type.

The memory data source that acts as a "go-between" for the parameter value.

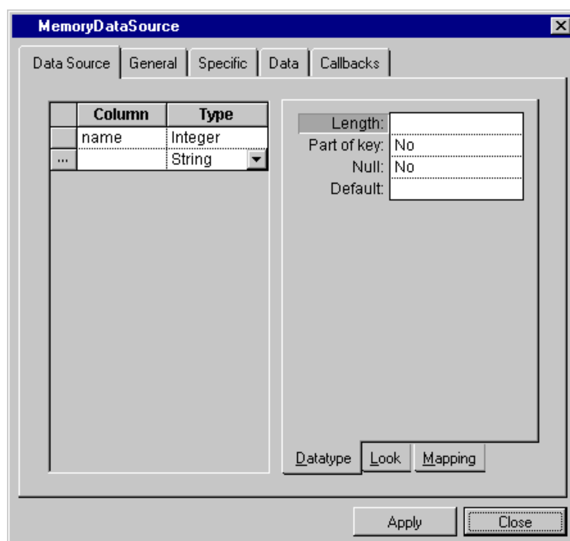


Figure 8.4 *The DEPT_NAME Memory Data Source*

Note that it is important to set up the column type as an integer even though the user will be entering a string value. This is because the foreign table will convert the string to an integer before entering the value into the memory table.

The `DbField` should be connected to this memory data source and configured in the following way:

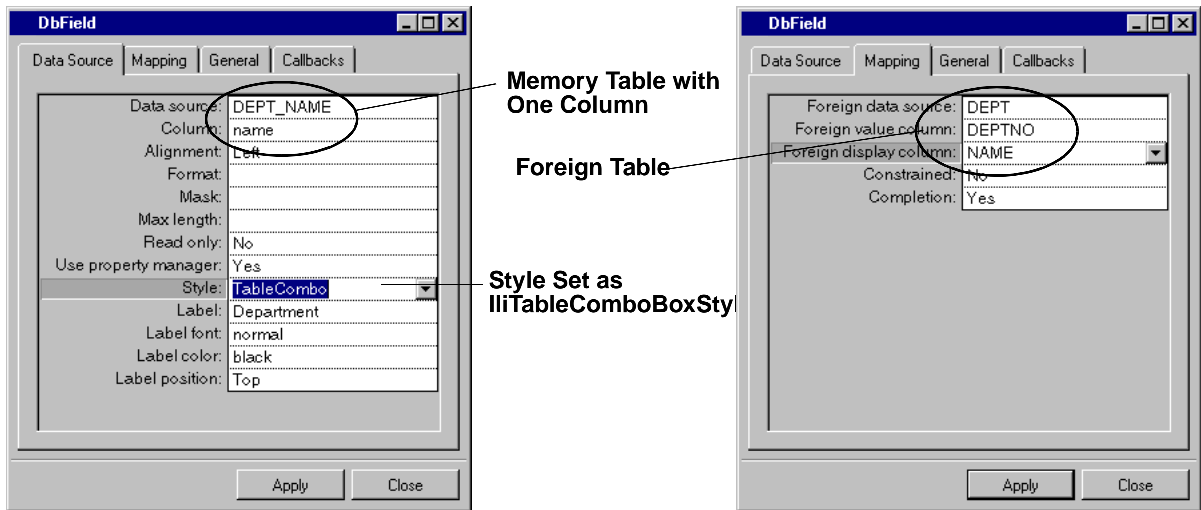


Figure 8.5 The Data Source and Mapping pages of the DbField Inspector that Accepts the Parameter Input

You will now define a parameter in the original SQL table that accepts its input from the contents of the only column in the memory data source. This column in turn accepts its input from the `DbField`. A parameter must be defined in the SQL Data Source inspector:

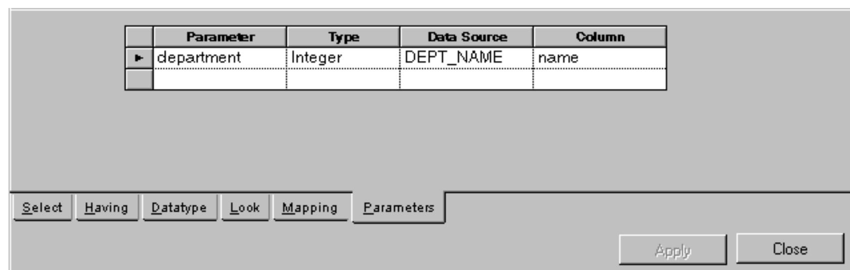


Figure 8.6 Defining a Parameter via a Memory Data Source with a Single Column

In addition to this, you must set Auto Select to `Yes` in the SQL Data Source Properties dialog box. This enables a data source that uses data from another data source table to automatically select the required value.

To specify that the data shown in the table gadget is to be filtered according to the department parameter, you need to enter the following into the Select page of the SQL Data Source inspector:

The Selection Criterion

	EMPNO	NAME	STATUS	DEPTNO	SALARY	
Select:	EMPNO	NAME	STATUS	DEPTNO	SALARY	
From:	I_EMP	I_EMP	I_EMP	I_EMP	I_EMP	
Operation:						
Order:						
Where:				= :department		
Or:						

Select Having Datatype Look Mapping Parameters

Apply Close

Figure 8.7 Specifying the Selection Criterion for a SQL Table

> ***Note:*** *When a parameter is used, its name must be prefixed by a colon (for example,* `= :department`*).*

You are now able to enter a department in the `DbField` that will be entered into the memory table column and taken as the "department" parameter value. It will then be used to make a selection in the SQL table.

One thing remains to be done. The validation of the user input in the memory table can be done by setting a predefined callback on the `DbField`.

Callback
@Validate(DEPT_NAME)

Figure 8.8 Setting a Callback on the DbField

This callback validates the user input in the memory table. Now, when you test this panel, the SQL table automatically selects according to the value that you enter in the `DbField`.

You should now have a panel that allows you to select the required department from a combo box and display only the employees that work in the department.

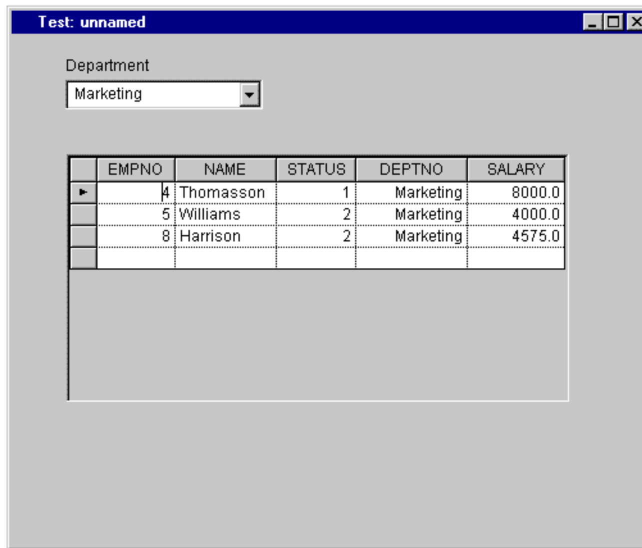


Figure 8.9 *Completed Panel Allowing Department Selection and Table Contents Filtering*

Working with an SQL Data Source

This section contains some hints for using the SQL data source.

You can find information on the following topics:

- *Defining Columns*
- *Forcing the Name of a Column*
- *The Table Primary Key*

Defining Columns

When you add a database table to a data source, you can tie the columns of the table to the columns of the data source table. This can either be done graphically or manually in the data source inspector.

The graphical method is described in the IBM ILOG Views *Data Access Getting Started Manual*. To specify the column that should be selected from a database manually, you can type directly in the Select and From cells in the appropriate column in the SQL Data Source inspector.

If you just want to select a particular column from a database, enter the name of the column in the Select field and the name of the database table in the From field. If you have entered

the column and database table name correctly, the data source table will then be updated accordingly.

It is also possible to specify a column in the data source table that is the result of a calculation carried out on data from other table columns. This is possible within the limits of the formulas that are accepted by the database. See your database documentation for more information on the formulas that can be used.

E	SALARY	SALARY/2
	SALARY	SALARY/2
	I_EMP	

E	SALARY	SALARY/2
	8000.0	4000.0
	4000.0	2000.0
	8600.0	4300.0
	4575.0	2287.5

SQL Data Source Inspector **Resulting Table Columns**

Figure 8.10 *Specifying a Computed Column*

To set up a column like this, click in the Select cell and enter the formula that you require. The From cell must remain empty. If there is a database table name in the From cell, Data Access tries to locate a column that has the name specified in the Select cell.

Forcing the Name of a Column

When a table is added to an SQL data source and the columns of the underlying table object are specified, you will notice that the name of the column is automatically set. It is automatically set to the name of the database table column. In most situations, this behavior is acceptable. However, there are certain situations when it can be useful to be able to change this name.

You may have noticed in the IBM ILOG Views *Data Access Getting Started Manual* that when an additional table is added to the data source, any columns that are not uniquely named are prefixed by the table name they originate from.

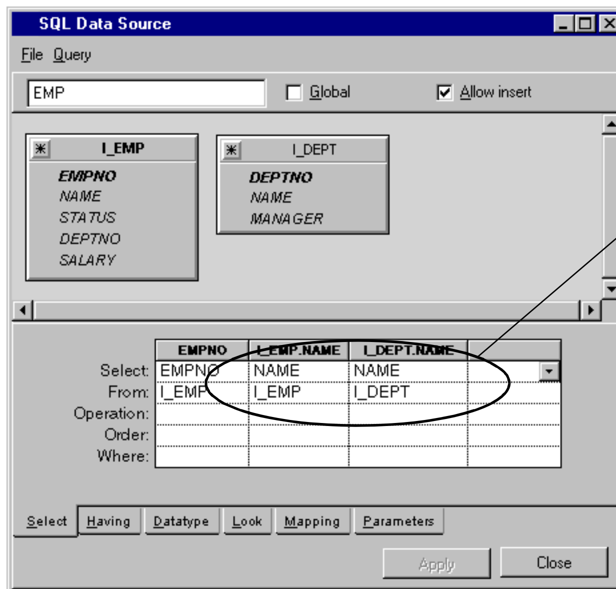


Figure 8.11 *An SQL Data Source with Two Database Tables Showing Automatic Column Naming*

If you have already set up a number of gadgets that are linked to columns and then add another table to the data source, certain table column names can change. This will have an effect on any gadgets tied to these columns. The gadget will fail to locate the column it is tied to because the column name has changed.

To work around this problem, specify a specific column name in the Datatype page of the SQL Data Source inspector. Providing you do not create any ambiguities, you can type the original column name here. Any gadgets connected to the column will then be updated to show the values in the column.

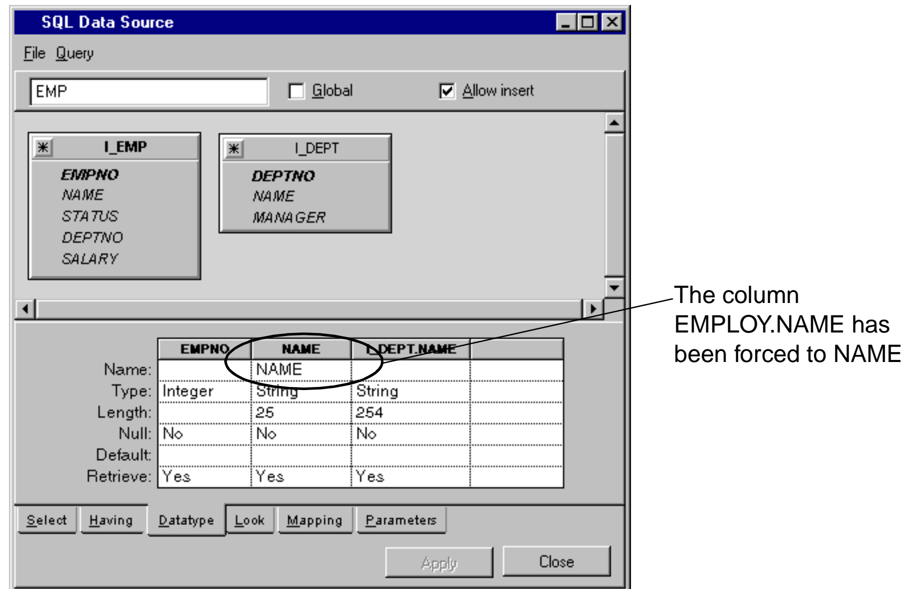


Figure 8.12 *Forcing the Name of a Column Using the Datatype Page*

Note: *The table gadget is the only gadget whose column names change dynamically with the data source.*

If you leave the Name field empty, the column name will change automatically to avoid any ambiguity whenever a new table is added.

The Table Primary Key

When you set up an SQL data source and its table object, you must include each of the primary key columns in the table object. If you do not do this, you can find that two of the rows in your table appear to be the same, even though they are uniquely identified by the primary key column(s).

If you try to update one of these rows, Data Access will not be able to identify uniquely the row in the database and the update will fail. An error message from the database will be displayed indicating that if the update were to continue, more than one row would be edited. This also applies to a deletion of a row that appears to be exactly the same as another row.

If you do not want to display the primary key columns of the table in the user interface, you should change column visibility in the SQL Data Source inspector.

C H A P T E R

9

Connecting to a Database

This chapter describes the Data Access classes and functions required for communicating with a relational database system.

You can find information on the following topics:

- *SQL Sessions and Cursor Objects*
- *Database Drivers*
- *The Connect Dialog Box*
- *Registered Sessions*

SQL Sessions and Cursor Objects

The `IliSQLSession` class establishes a communication channel with a remote database engine. An instance of this class is created whenever you connect to a relational database using the Connect panel in IBM® ILOG® Views Studio.

Each SQL data source in Data Access has an `IliSQLTable` object which in turn has an `IliSQLSession` object. These session objects are usually associated with one and only one SQL data source. However, if you create an application-wide SQL session, it can be associated with more than one SQL data source. See *Registered Sessions* on page 154.

Creating a Session

An `IliSQLSession` object is automatically created when you connect to the database using the connect panel in IBM ILOG Views Studio. An example of how the `IliSQLSession` object is created is shown in the following code:

```
IliSQLSession* session;
session = new IliSQLSession("oracle10", "scott/tiger@options");
session->lock();
...
session->unLock();
```

The first parameter in the `IliSQLSession` constructor is the name of the database driver. This can be any of the following names: "oracle","oracle9","oracle10", "oracle11", "sybase", "informix", "informix72","informix9","oledb", "mssql" or "odbc" (for the set of databases that are currently supported).

The second parameter designates the user, password, and other connection parameters required to establish the communication with the remote database engine. Its format depends on the database driver used. See `IliSQLSession::getConnectionParams` in the IBM ILOG Views *Data Access Reference Manual.*

After obtaining an `IliSQLSession` object you should lock it and keep it locked until you have finished using it. When you have finished with the session, the `IliRefCounted::unLock` member function should be called to unlock the session.

Connecting to a Database System

Once you have created a session object, you can connect to the database system using the `IliSQLSession::connect` member function. This is done in the following way:

```
if (session->connect()) {
  ...
}
else
    IlvPrint("Error: %s", session->getErrorMessage().getMessage());
```

During the session, you can check whether the session is still connected using the `IliSQLSession::isConnected` member function. An example of this is shown in the following code:

```
if (!session->isConnected())
    IlvPrint("Not connected");
```

To end a session, use the `IliSQLSession::disconnect` member function. This method rolls back any (uncommitted) work in progress and breaks the communication channel with the database system.

```
session->disconnect();
```

Cursors

Before you can do anything useful with your session object, you must obtain an `IliSQLCursor` object. The cursor allows you to submit SQL statements to a database and to retrieve any result sets produced by these statements. A cursor can be created in the following way:

```
IliSQLCursor* cursor = session->newCursor();
if (cursor != 0) {
  ...

  session->releaseCursor(cursor);
}
else
  IlvPrint("Out of cursors.");
```

Once a cursor object has been created, any SQL statements (except SELECT statements) can be submitted to the database using the `IliSQLCursor::execute` member function. An example of this is shown in the following code:

```
if (cursor->execute("UPDATE EMP SET SALARY = SALARY * 1.1")) {
                IlvPrint("Happy days!");
}
else
    IlvPrint("Error: %s", cursor->getErrorMessage().getMessage());
```

The SQL SELECT Statement and Its Result Set

To submit an SQL SELECT statement you can use the `IliSQLCursor::select` member function as follows:

```
if (cursor->select("SELECT NAME, SALARY FROM EMP")) {
  while (cursor->fetchNext() && cursor->hasTuple()) {
    IlvPrint("Employee %s : %ld",
              cursor->getStringValue(0),
              cursor->getIntegerValue(1));
  }
}
else
   IlvPrint("Error: %s", cursor->getErrorMessage().getMessage());
```

At the beginning of the inspection process just after the `select` member function has been called, the cursor is positioned before the first row. Each call to the `fetchNext` member function moves the cursor to the next row. When all rows have been seen, a call to `fetchNext` positions the cursor after the last row.

The `IliSQLCursor::hasTuple` member function can be called to determine if the cursor is positioned on a row (as opposed to being positioned before the first row or after the last row). If the cursor is positioned after the last row, the result set has been exhausted.

Once an SQL SELECT statement has been successfully executed it leaves a result set available for inspection through the cursor. A result set is an ordered collection of rows. Each of these rows conforms to the same schema.

You can retrieve the value of a column using the `IliSQLCursor::getValue` method. An example of this is shown in the following code:

```
IliValue value;
if (cursor->getValue(colno, value)) {
    ...
}
```

The columns of the result set are identified by their position, starting from 0. If you know the type of a given column in the result set, you can use one of the following methods:

```
const char* IliSQLCursor::getStringValue(Ilnt colno) const;
IlInt IliSQLCursor::getIntegerValue(IlInt colno) const;
IlFloat IliSQLCursor::getFloatValue(IlInt colno) const;
IlDouble IliSQLCursor::getDoubleValue(IlInt colno) const;
IliDate IliSQLCursor::getDateValue(IlInt colno) const;
IliBinary IliSQLCursor::getBinaryValue(IlInt colno) const;
```

The character string returned by the `getStringValue` member function and the byte array returned by the `getBinaryValue` member function (it is part of the `IliBinary` structure) belong to the cursor. Therefore, they will be overwritten the next time one of the `fetchNext`, `select`, or `execute` member functions is called.

Note that the `getStringValue` member function will return `NULL` if the column is not of type character `string`. If you want to convert a value into a string, use the `getValue` member function as shown in the following example:

```
IliValue value;
if (cursor->getValue(colno, value)) {
    IlvPrint("%s", value.getFormatted());
}
```

The member function `isNull` tests whether a given column is null. The testing of a column is shown in the following example:

```
if (cursor->isNull(colno))
    IlvPrint("NULL");
else
    IlvPrint("%s", cursor->getStringValue(colno));
```

All work done on a session object through its cursors belongs to a transaction. Because of transaction management, any work done needs to be committed or canceled (rolled back) at some point in time.

To commit or roll back the work done on a session object, use the `IliSQLSession::commit` and `IliSQLSession::rollback` member functions. An example of their usage is shown in the following code:

```
if (session->commit())
    IlvPrint("Work done");
else
    IlvPrint("Error: %s", session->getErrorMessage().getMessage());
```

If you forget to commit your work, it will eventually be canceled (rolled back) when the session is freed.

To obtain information on the structure of your result set after a successful call to the `select` member function, you can use the `IliSQLCursor::getSchema` member function.

```
const IliSchema* schema = cursor->getSchema();
```

This member function returns a schema object that belongs to the cursor. (See the `IliSchema` class in the IBM ILOG Views *Data Access Reference Manual*.) Note that this schema object can be modified the next time you call the `select` or `execute` member functions on that cursor, so it should be used as soon as possible.

When you are finished with the cursor object you should release it using the `IliSQLSession::releaseCursor` member function.

Database Drivers

The `IliSQLSession` class can be used to communicate with different database servers. To connect a session object to a given database server, the corresponding database driver must be included in the application executable file.

The database driver is included in the application at compile time. This can be done by adding the following code to the source file that contains the `main()` function:

```
#define ILDORACLE
#define ILDINFORMIX

#include <ildblink/dblink.h>
#include <ilviews/dataccess/dbms/session.h>

static IldDbms* ILVCALLBACK
CustomNewDbms(const char* dbms, const char* params) {
    return IldNewDbms(dbms, params);
}

main(int argc, char* argv[])
{
     IliSQLSession::SetNewDbmsFunction(CustomNewDbms);
}
```

This code extract specifies that the Oracle and Informix drivers are included in the application executable file.

The following list contains the macro symbols that must be defined in order to include the corresponding database driver.

- `ILDDB2`
- `ILDORACLE`

- `ILDINFORMIX`
- `ILDSYBASE`
- `ILDOLEDB`
- `ILDMSSQL`
- `ILDODBC`

Note: *The Microsoft SQL Server , OLE DB and ODBC driver are only supported on Windows platforms.*

The Connect Dialog Box

In the section *Creating a Session* on page 150, you saw how a session object is created with the connection parameters (user name, password, and so on) hard-coded in the source code. The `IliSQLSession::queryConnect` member function can be used to obtain some or all of these parameters from the end user.

The following code extract initializes a session object with a connection string from which the password is missing. A dialog box is then automatically displayed in which the user can enter his password.

```
IlvDisplay* dpy;
IlvAbstractView* view;
...
IliSQLSession* session;
session = new IliSQLSession("oracle", "scott/@options");
session->lock();

if (session->queryConnect(dpy, view, IliQueryPassword)) {
    ...
}

session->unLock();
```

Registered Sessions

When you use the IBM® ILOG® Views Studio editor to create the panels of an application, you can define application-wide sessions and then specify one or more SQL data sources that share the same session. An application-wide session is, in fact, a registered session object.

The `IliSQLSession::RegisterSession` member function registers a session object. This is shown in the following code excerpt:

```
IliSQLSession* session;
session = new IliSQLSession("oracle10", "scott/@options");
session->setSessionName("MainSession");
IliSQLSession::RegisterSession(session);
```

Alternatively, a session can be registered using the following code:

```
IliSQLSession::RegisterSession("MainSession",
                                "oracle10",
                                "scott/@options")
```

The `IliSQLSession::GetRegisteredSession` member function can be used to retrieve a registered session:

```
IliSQLSession* session =
            IliSQLSession::GetRegisteredSession("MainSession");
```

Part III

IBM ILOG Views Data Access Gadgets

This part describes how to use Data Access gadgets when connected to an SQL database. This part serves a reference to all the gadgets provided with Data Access.

The gadgets described in the following chapters can be divided into two groups:

- *Data Source Gadgets Reference*
- *Display Gadgets Reference*

C H A P T E R

10

IBM ILOG Views Studio Data Access Gadgets

This chapter introduces the Data Access gadgets found on the Palettes panel.

You can find information on the following topics:

- *The Palettes Panel*
- *Notebook Pages Common to Data Access Gadgets Inspectors*
- *Dialog Boxes Common to Data Access Gadgets Inspectors*

The Palettes Panel

The Palettes panel appears when Data Access is launched. If it has been closed and you want to open it, choose Palettes from the Tools menu in the IBM ILOG Views Studio Main window.

The Palettes panel is divided into two panes. The top pane displays a tree with various items, each corresponding to a particular gadget or graphic palette. The topmost items are for the Data Access palettes. They are:

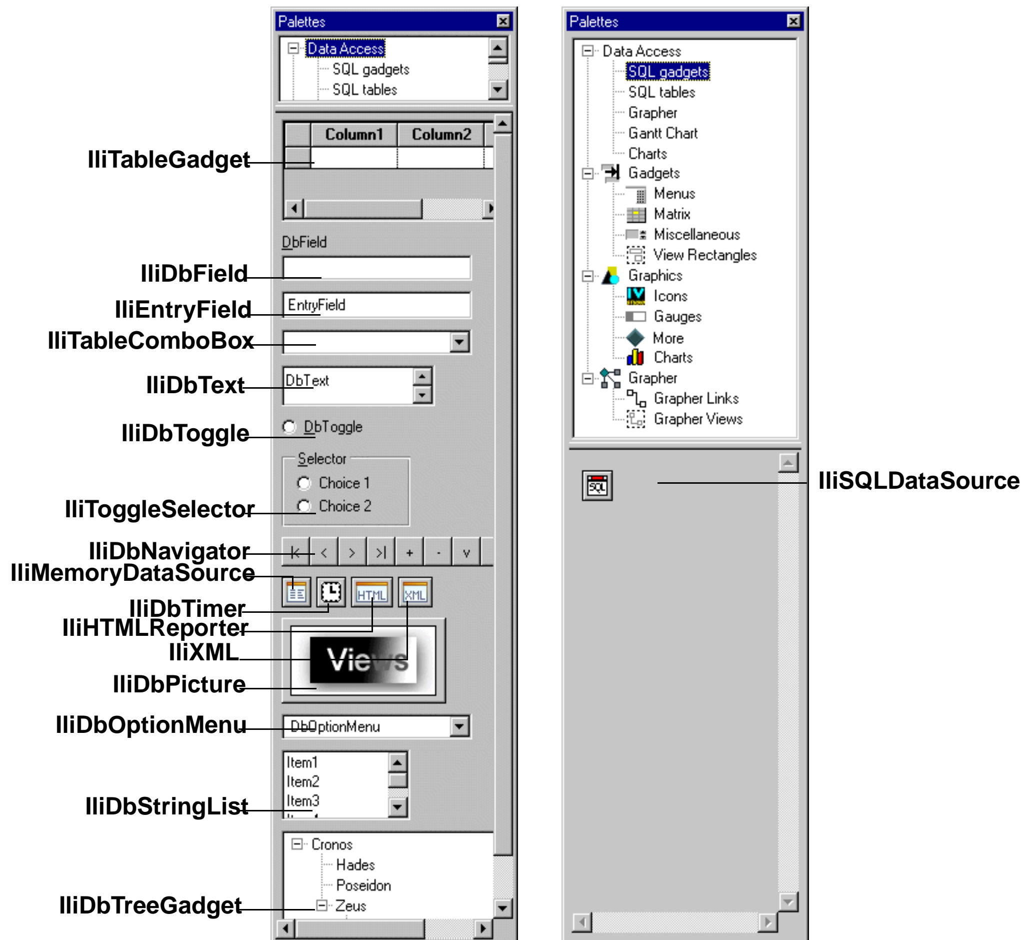
- a gadgets palette, which appears when you highlight Data Access
- SQL gadgets

- SQL tables, which is not a gadget, lists all user-created data tables and contains an SQL schema editor.
- Charts, which presents a graphic chart connected to various data sources.
- Grapher, which presents contents of a nodes and links data source in a grapher
- Gantt Chart, for defining a Gantt chart connected to various data sources

To use a gadget, drag it from the Palette and drop it in the work space in the Main window.

Data Access and SQL Gadgets

The gadgets illustrated below appear when you select Data Access or SQL Gadgets on the Palettes panel.



Use the above gadgets for the following purposes:

IliTableGadget — For editing tables.

IliDbField — For displaying data with a data source aware gadget whose appearance can be dynamically changed (to an entry field, toggle switch, and so on).

IliEntryField — For displaying text in a data source aware text field.

IliTableComboBox — For listing items in a data source aware popup menu and displaying the item chosen.

IliDbText — For displaying text in a data source aware multi-line scrollable text field.

IliDbToggle — For choosing between three states (True, False and null) with a data source aware toggle button.

IliToggleSelector — For selecting among any number of items using data source aware selector buttons.

IliDbNavigator — For creating a tool bar with buttons to navigate through rows and edit data in a data source table.

IliMemoryDataSource — For defining a local memory data source.

IliDbTimer — For calling a callback periodically

IliHTMLReporter — For generating an HTML document from a data source.

IliXML — For managing the communication between a datasource and an XML stream.

IliDbPicture — For displaying a picture in a data source aware gadget.

IliDbOptionMenu — For listing items in a data source aware popup menu and displaying the item chosen.

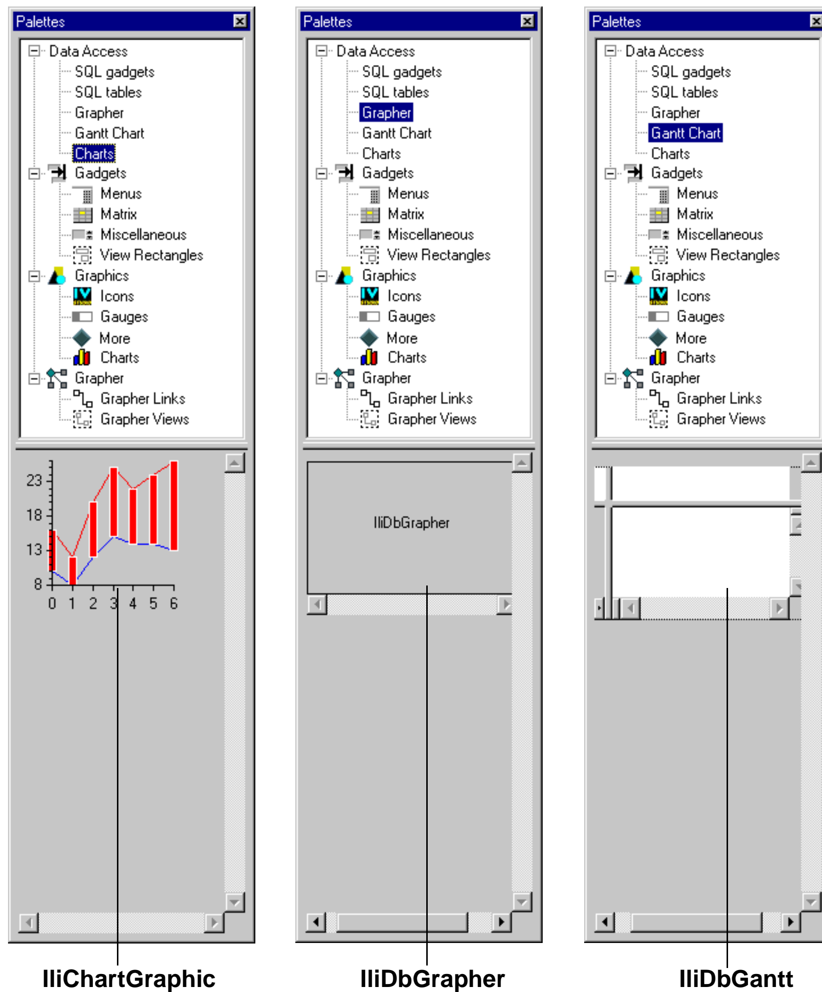
IliDbStringList — For displaying a list of labels in a data source aware multi-line string list.

IliDbTreeGadget — For displaying the contents of a data source in a tree gadget based on a parent/child relationship.

IliSQLDataSource — For providing a link to an SQL database from which a table is defined and displayed.

Charts, Grapher and Gantt Chart Gadgets

The gadgets illustrated below appear when you select Charts, Grapher and Gantt Chart under Data Access on the Palettes panel.



Use the above gadgets for the following purposes:

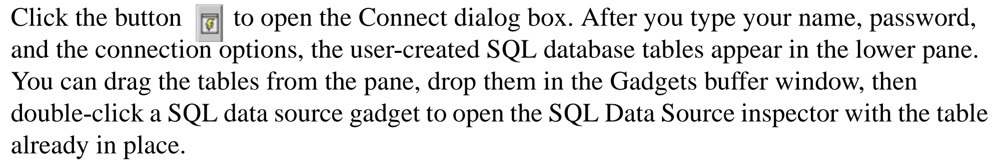
IliChartGraphic — For defining a chart graphic connected to various data sources.

IliDbGrapher — For displaying contents of a nodes and links data source in a grapher.

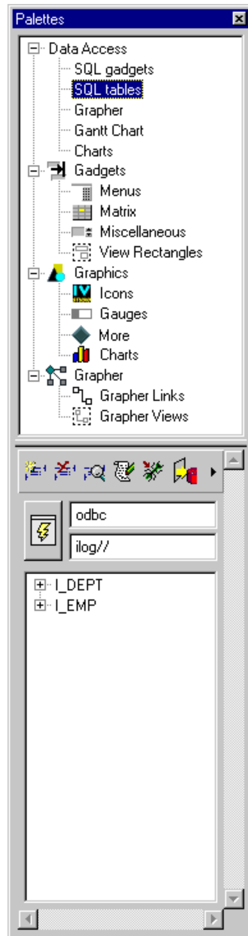
IliDbGantt — For defining a Gantt chart connected to various data sources.

SQL Tables

When you select SQL Tables under Data Access on the Palettes panel, the lower pane is empty.

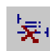


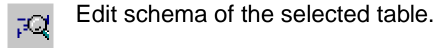
Click the button to open the Connect dialog box. After you type your name, password, and the connection options, the user-created SQL database tables appear in the lower pane. You can drag the tables from the pane, drop them in the Gadgets buffer window, then double-click a SQL data source gadget to open the SQL Data Source inspector with the table already in place.

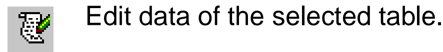


The SQL Tables palette has also an SQL Schema Editor toolbar with the following buttons:

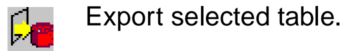
Create table.

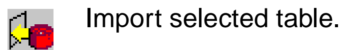
 Drop selected table.


Edit schema of the selected table.


Edit data of the selected table.

Enable / disable SQL trace.


Export selected table.

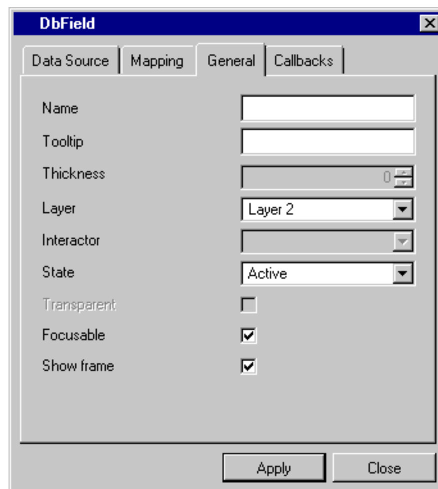

Import selected table.

Notebook Pages Common to Data Access Gadgets Inspectors

Most Data Access gadgets inspectors have a General and a Callbacks notebook page.

General Notebook Page

The General page text fields and check boxes are the same for all the inspectors, however, the availability of each selection depends on the inspector.



Label	Description
Name	**Menu:** None. **Default:** No default. **Explanation:** Name of the gadget.
Tooltip	**Menu:** None. **Default:** No default. **Explanation:** Text to appear in the tooltip.
Thickness	**Menu:** Grayed if this option is not available. **Default:** 2. **Explanation:** Increases the width of the border surrounding the gadget.
Layer	**Menu:** Layer 1, Layer 2. **Default:** Layer 2 **Explanation:** Manager layer in which the gadget will be placed.
Interactor	**Menu:** Names of the available interactors. **Default:** None. **Explanation:** Allows you to select the kind of interactor you want for this gadget.
State	**Menu:** Active, Inactive, Grayed out. **Default:** Active. **Explanation:** Specifies the gadget activity status.
Transparent	**Check box.** **Default:** Not checked. **Explanation:** When this box is checked, the gadget appears transparent.
Focusable	**Check box.** **Default:** Checked. **Explanation:** When this box is checked, the gadget can receive the mouse pointer or keyboard focus.
Show Frame	**Check box.** **Default:** Checked. **Explanation:** When this box is checked, the gadget frame is displayed.

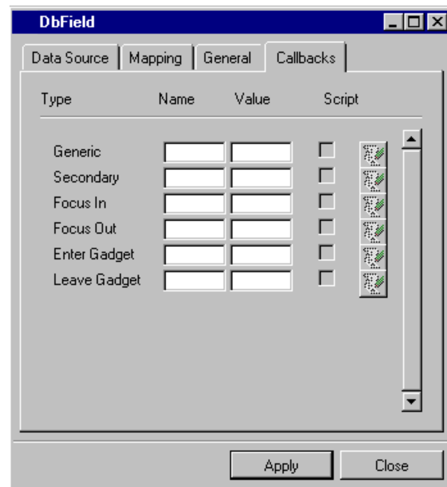
Callbacks Notebook Page

The first six callbacks are common to most inspectors. However, the `IliDbChart`, `IliDbGrapher`, and `IliDbGantt` inspectors have only the first two callbacks.

Some Data Access gadgets have additional callbacks, which are listed in this manual. Descriptions of the callbacks are located in the IBM ILOG Views *Data Access Reference Manual*.

All callbacks have the following fields:

- **Name:** Function name of the callback.
- **Value:** Type the callback value.
- **Script:** Check this box if you want to use IBM ILOG Script. The button to the right of the check box becomes active when the box is selected. Clicking the button shows you the callback source code in the Script Editor.



Callback	Description
Generic	Used to perform the main action of the gadget, for example, when a button is activated or when you double-click an item in a string list.
Secondary	Called when a change is made in the gadget, for example, when you type text in a field, highlight an item in a menu, or select a value in a list.

Callback	Description
Focus In	Called when the gadget receives the keyboard focus.
Focus Out	Called when the gadget loses the keyboard focus.
Enter Gadget	Called when the mouse pointer enters the gadget.
Leave Gadget	Called when the mouse pointer leaves the gadget.

Dialog Boxes Common to Data Access Gadgets Inspectors

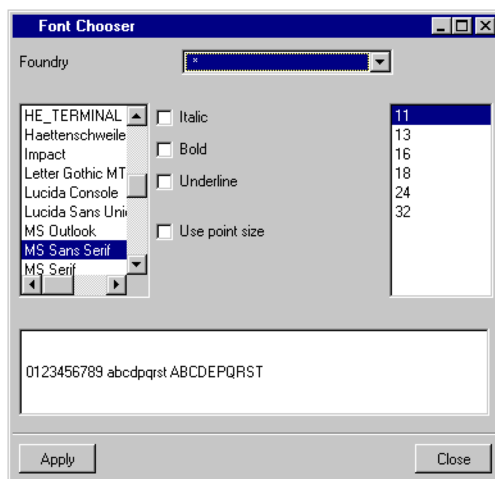
The dialog boxes that can be called from various Data Access gadgets inspectors are described in this section.

You can find information on the following topics:

- *Font Chooser Dialog Box*
- *Color Chooser Dialog Box*
- *File Chooser Dialog Box*

Font Chooser Dialog Box

The Font Chooser dialog box is used to choose the font style for text.



To use the Font Chooser dialog box, do the following:

1. Select the foundry, font, font size, and font style.

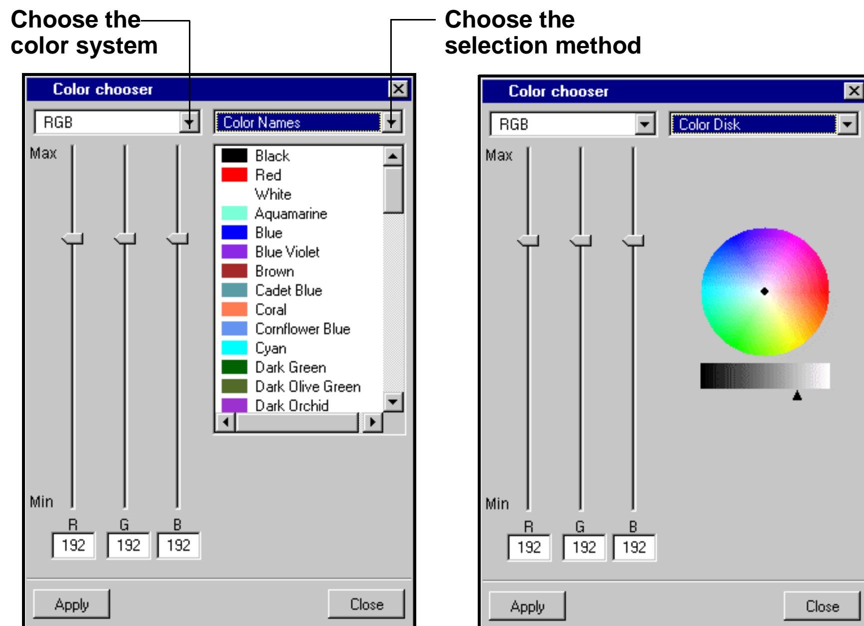
   The characters in the text field at the bottom change to match the selection.

2. Click Apply.

   The name of the font selected appears in the inspector panel field from which the Font Chooser dialog box was called.

Color Chooser Dialog Box

Use the Color Chooser dialog box for choosing the background and text colors. You can select the colors by name or by using the color disk.



To use the Color Chooser dialog box:

1. At the top of the dialog box, choose the color system and/or selection method, then select the color you want. Use the RGB/HSV values and/or the color wheel to define your own colors or use the Color Names option to use predefined colors.

   `RGB` = Red, Green, Blue

   `HSV` = Hue, Saturation, Value

   The color selected appears in the lower-right rectangle of the Color Chooser dialog box.

2. Click Apply.

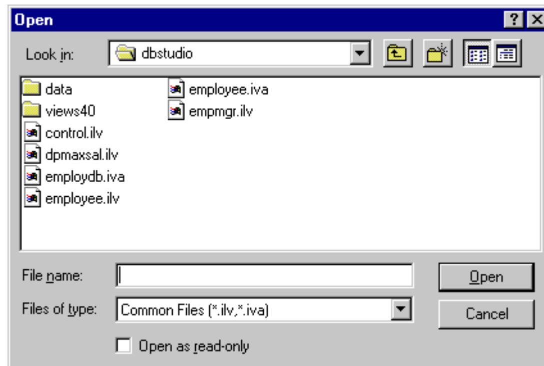
The name of the color selected appears in the inspector panel field from which the Color Chooser dialog box was called.

File Chooser Dialog Box

Data Access gadgets inspectors use a file chooser dialog box to select:

- an image file
- a text file

Note: On Unix, this dialog box is called a File Selector. In Windows, it is called Open, as illustrated below.



To use the File Chooser dialog box, do the following:

1. Select the file you want.
2. Click Open (Windows) or Apply (Unix).

   The name of the file selected appears in the inspector panel field from which the File Chooser dialog box was called.

C H A P T E R

11

Data Source Gadgets Reference

This chapter describes the two data source creation gadgets:

- `IliSQLDataSource`
- `IliMemoryDataSource`

To access `IliMemoryDataSource` gadget, click Data Access in the Palettes panel. The gadgets appear in the lower pane.

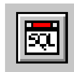
To access the `IliSQLDataSource` gadget, click SQL Gadgets in the Palettes Panel.

To use one of the above gadgets, drag and drop its gadget-icon in the Gadgets buffer window.

IliSQLDataSource

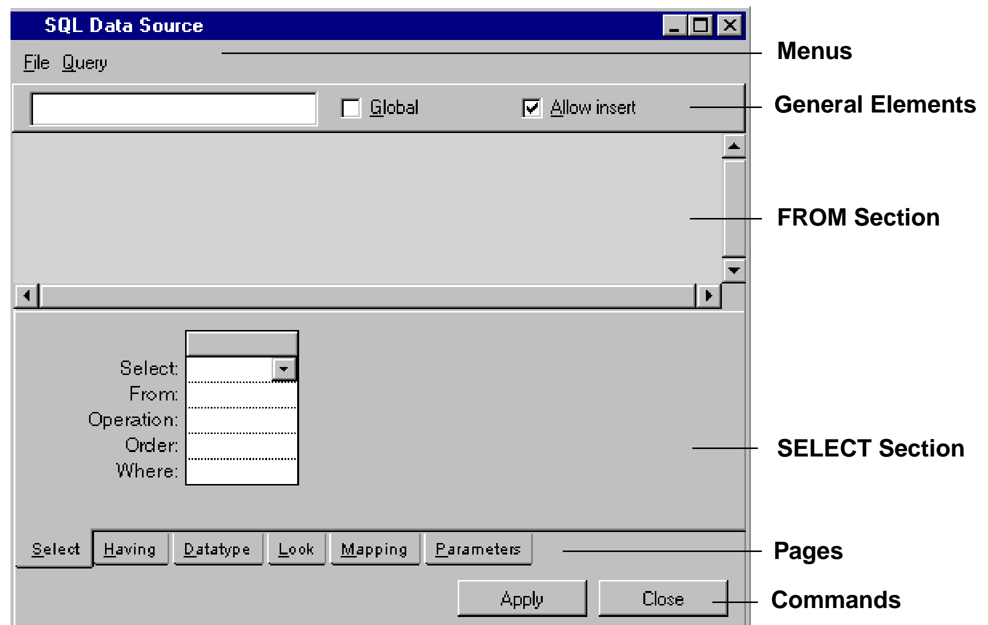
The `IliSQLDataSource` gadget is used for creating a data source by:

- connecting to a database,
- defining data source tables through the specification of selection criteria,
- defining how data is to be displayed in gadgets connected to the data source through the specification of format criteria.



IliSQLDataSource Inspector Panel

This panel appears by double-clicking the gadget-icon after having placed it in the Gadgets buffer window.



IliSQLDataSource Menus

The SQL Data Source inspector panel has two menus:

- File
- Query

File Menu

Menu Items	Description
Properties...	Displays the SQL Data Source Properties dialog box.
View Source...	Displays the Source dialog box.
Close	Closes the SQL Data Source inspector panel.

Query Menu

Menu Items	Description
Add Tables...	If not connected to a database, displays the Connect dialog box. If connected, displays the Select Tables dialog box.
Edit Table...	Displays a Table Definition dialog box that lets you change the table title and the owner name.
Remove Table...	Removes selected table in FROM section. You can also use the Delete key.
Synchronize Table with Database...	Updates the selected database table representation in the FROM section to the data source. Optionally, updates the data source columns in the SELECT section. The information updates are structural. Displays the Differences dialog box if the data source table has changed.
Synchronize All Tables With Database...	Updates all database table representations in the FROM section to the data source. Optionally, updates the data source columns in the SELECT section. The information updates are structural. Displays the Differences dialog box if the data source table has changed.
Append Column	Adds a column to the right of the existing columns.
Insert Column	Inserts a column to the left of the selected column.
Delete Column...	Removes the selected column. You can also use the Delete key.

Menu Items	Description
Edit Join...	Displays an Edit Join Table dialog box to select the type of join. A join operation only concerns lines selected in the FROM section. Inactive if no join line is selected.
Delete Join...	Displays a Question dialog box to confirm the deletion of a join operation whose line is selected in the FROM section. Inactive if no join line is selected. You can also use the Delete key.

General Elements

These elements apply to the SQL data source as a whole.

Element	Description
Name field	The SQL data source name that appears under the SQL data source gadget in the work space when Apply is clicked in the SQL Data Source inspector panel.
Global checkbox	**Default**: Not checked. When checked, allows more than one user panel to use the current SQL data source.
Allow insert checkbox	**Default**: Checked. When not checked, prevents the user from inserting a new row into the SQL data source tables, but does not prevent the user from editing existing rows.

SELECT Section Notebook Pages

The SELECT section of the SQL Data Source inspector panel has six notebook pages:

- *Select Page*
- *Having Page*
- *Datatype Page*
- *Look Page*
- *Mapping Page*
- *Parameters Page*

These pages define the criteria for selecting data from the database and for formatting the data in display gadgets.

> ***Note:*** *"Default" in the pages described below refers to what appears when a column is created in the SELECT section by dragging a line from the FROM section.*

Select Page

The Select page is used for:

- defining the data source columns in terms of the columns in the FROM section (Select and From rows),
- specifying operations to compute results (Operation row),
- defining the sort order in which the data is to be displayed (Order row),
- establishing selection criteria by which data is retrieved from the database (Where row).

> ***Note:*** *If the Operation row is used for a column, any entries in the Where row for that column must be made on the Having page.*

Label	Description
Select	**Menu:** The columns of the data source table in the From row. **Default:** No default. **Explanation:** Defines a data source column taken from the table displayed in the From row, or an SQL expression that may include one or more columns.
From	**Menu:** Tables defined for the current data source. **Default:** No default. **Explanation:** Specifies the table from which the column in the Select row is taken. Mandatory if a column exists in the Select row.
Operation	**Menu:** None, Group By, Count, Sum, Avg, Min, Max. **Default:** No default. **Explanation:** Performs operations by which rows are grouped and their aggregate values computed. If “Group By” is used in a column, all the other columns must have an operation.
Order	**Menu:** No, Asc, Desc (No=random order, Asc=ascending, Desc=descending). **Default:** No default. **Explanation:** Determines the order of the rows in the display table. If more than one column is entered, the leftmost column has priority.
Where	**Menu:** None. **Default:** No default. **Explanation:** Selection criteria applied to the column in the Select row. The selection criteria is used with a logical AND operator and added to criteria in other columns to further restrict the selection of data to be retrieved from the database. Only applies when the Operation row is empty.

Having Page

The Having page is used for establishing the selection criteria of the data extracted from the database to which the data source is connected, and when the current column has an Operation defined. Use the Select page when no Operation is defined for the column.

Label	Description
Select	**Menu:** The columns of the data source table in the From row. **Default:** No default. **Explanation:** Defines a data source column taken from the table displayed in the From row or an SQL expression that may include one or more columns.
From	**Menu:** Tables defined for the current data source. **Default:** No default. **Explanation:** Specifies the table from which the column in the Select row is taken. Mandatory if a column exists in the Select row.
Operation	**Menu:** None, Group By, Count, Sum, Avg, Min, Max. **Default:** No default. **Explanation:** Performs operations by which rows are grouped and their aggregate values computed. If "Group By" is used in a column, all the other columns must have an operation. If an operation is used without "Group By" the entire table is used to compute the value.
Having	**Menu:** None. **Default:** No default. **Explanation:** Selection criteria applied to the column in the Select row. The selection criteria is used with a logical AND operator and added to criteria in other columns to further restrict the selection of data to be retrieved from the database. Only applies when there is a value in the Operation row.

Datatype Page

The Datatype page is used for defining the type of data that can be entered in the column.

Name:
Type:
Length:
Null:
Default:
Retrieve:

Select Having Datatype Look Mapping Parameters

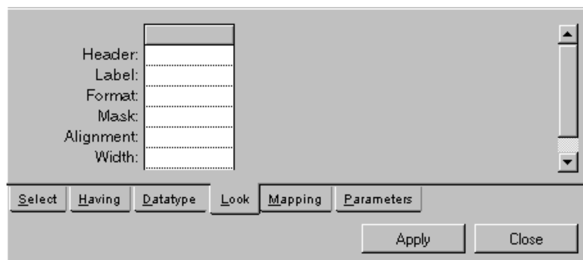
Apply Close

Label	Description
Name	**Menu:** None. **Default:** No default. **Explanation:** Each column must have a name by which it can be attached to a gadget. This name is automatically taken from the database and appears at the top of the column. This row is used to change this name. While the name given by the system can be automatically changed by adding a prefix to distinguish it from other columns having the same name in other tables, the name entered here will not change. The title at the top of the column in the SELECT section is replaced by the one entered here.
Type	**Menu:** String, Long string, Boolean, Byte, Integer, Float, Double, Decimal, Date, Time. **Default:** As defined in the database schema. **Explanation:** The type of data that can be entered in the cells of the column.
Length	**Menu:** None. **Default:** As defined in the database schema. **Explanation:** The number of characters that can be entered in the cells of the column.
Null	**Menu:** Yes, No. **Default:** As defined in the database schema. **Explanation:** Yes = The cell can remain empty. No = The cell cannot remain empty.

Label	Description
Default	**Menu:** None. **Default:** No default. **Explanation:** The data that appears in a cell when it is added to the table.
Retrieve	**Menu:** Yes, No. **Default:** No default. **Explanation:** Yes = Column is an element of the data source. No = Column is not an element of the data source and will not appear in the result. Is used only with selection criteria.

Look Page

The Look page is used to define how data entered in the column will appear.

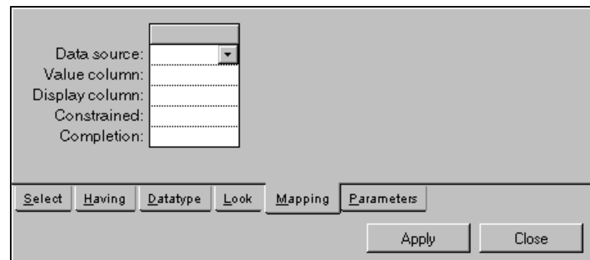


Label	Description
Header	**Menu:** None. **Default:** No default. **Explanation:** Title that will appear at the top of the column when displayed in a table gadget. If left empty, the table gadget uses the Name row on the Datatype page, or, if also empty, the name given by the system from the database schema.
Label	**Menu:** None. **Default:** No default. **Explanation:** Applies only when the Data Source Assistant is used to create a form. The caption that appears next to the form gadget containing the data for the column. If empty, the label is taken from the Name row on the Datatype page, or, if also empty, the name given by the system from the database schema. (The Header row on the Look page is not used.)
Format	**Menu:** Formats corresponding to what is entered in the Type cell on the Datatype page. **Default:** No default. **Explanation:** Predefined system and user formats from the menu or a format entered by the user by which data in the column cells will be formatted.
Mask	**Menu:** Masks corresponding to how and what data is entered in the Type cell on the Datatype page. **Default:** No default. **Explanation:** Predefined by the user for data input in the column cells. There are predefined system masks for date and time.
Alignment	**Menu:** Left, Center, Right. **Default:** Depends on the entry in the Type row on the Datatype page. **Explanation:** How data in the column cells will be aligned within the cell.
Width	**Menu:** None. **Default:** No default. **Explanation:** Display width in pixels of the column cells. Can be changed in the table gadget.

Label	Description
Read only	**Menu:** Yes, No. **Default:** No default. **Explanation:** Yes = Prevents the column cells from being edited. No = Allows the column cells to be edited.
Visible	**Menu:** Yes, No. **Default:** No default. **Explanation:** Yes = The column is visible. No = The column exists but does not appear.

Mapping Page

The Mapping page is used for displaying data in a column by referring to data in a column in another table.



Label	Description
Data source	**Menu:** Current data sources. **Default:** No default. **Explanation:** The foreign data source containing the columns to which the values for the current column are to be mapped. If a foreign data source is specified here, creates a combo box pull-down menu in the cell showing the values in the foreign data source.
Value column	**Menu:** Columns of data source selected in data source row. **Default:** No default. **Explanation:** The column containing the value to which the current column is to be mapped.

Label	Description
Display column	**Menu:** Columns of the data source selected in the data source cell. **Default:** No default. **Explanation:** The column associated with the Value column containing the data to be displayed.
Constrained	**Menu:** Yes, No. **Default:** No default. **Explanation:** Applies only when the value entered in the Value Column and Display column rows is the same. Yes = Can only enter a value that belongs to a foreign data source. No = Can enter any value.
Completion	**Menu:** Yes, No. **Default:** No default. **Explanation:** Is only in effect when constrained = Yes. Yes = Can enter a combo box list item by typing enough of its initial characters to make it unique, then leaving the cell. No = Cannot enter a combo box list item by typing its initial characters.

Parameters Page

The Parameters page is used for defining a parameter in terms of data located in a column from any other data source. This parameter can then be used as selection criteria in the Where row of the Select page.

Column	Description
Parameter	**Menu:** None. **Default:** No default. **Explanation:** The name of the parameter that represents the column from which data is to be retrieved. This name can then be used as selection criteria in a Where row on the Select page.
Type	**Menu:** String, Long string, Boolean, Byte, Integer, Float, Double, Decimal, Date, Time. **Default:** No default. **Explanation:** Type of the parameter.
Data Source	**Menu:** Current data sources. **Default:** No default. **Explanation:** Data source from which the parameter will take its value.
Column	**Menu:** Columns of data source selected in Data Source column. **Default:** No default. **Explanation:** Column from which the parameter will take its value. The column must exist in the data source shown in the Data Source column.

Callbacks

The SQL Data Source inspector has no Callbacks page. To access this gadget callbacks, open the Callbacks panel by selecting Callbacks from the Tools menu.

Buttons

The SQL Data Source inspector panel has two buttons at the bottom:

- Apply
- Close

Button	Description
Apply	Applies changes made in the SQL Data Source panel to the data source. This does not submit a query to the database, which is done by pressing the F9 key when the gadget has the focus or by pressing the "@" button in the navigation tool bar.
Close	Closes the SQL Data Source inspector panel.

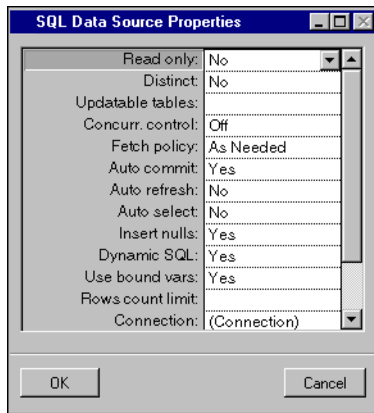
Dialog Boxes

Various elements in the SQL Data Source inspector panel can call different dialog boxes:

- *SQL Data Source Properties Dialog Box*
- *Connect Dialog Box*
- *Source Dialog Box*
- *Select Tables Dialog Box*
- *Question Dialog Box*
- *Differences Dialog Box*

SQL Data Source Properties Dialog Box

The SQL Data Source Properties dialog box is used for defining various properties of the data source. It is called by the Properties... menu item in the File menu of the SQL Data Source inspector panel.



Label	Description
Read only	**Menu:** Yes, No. **Default:** No. **Explanation:** Yes = The data source cannot be edited. No = The data source can be edited.
Distinct	**Menu:** Yes, No. **Default:** No. **Explanation:** Yes = Duplicate rows are merged. No = Duplicate rows are left intact.
Updatable tables	**Menu:** List of tables in FROM section. **Default:** First table that has been added to the data source. **Explanation:** The table that the data source updates. A data source can only update one table.
Concurr. control (concurrency control)	**Menu:** On, Off. **Default:** Off. **Explanation:** On = The data source takes extra steps to ensure that a row has not been updated by another user from the time the row was retrieved from the database and when it was resubmitted to the database. Off = The data source does not take such extra steps.
Fetch policy	**Menu:** As Needed, Immediate. **Default:** As Needed. **Explanation:** As Needed = Selected data is retrieved from the database and stored in data cache only as data is needed. For As Needed to be effective, Auto Commit (see below) must be No. When Auto Commit is Yes, Immediate is implied. Immediate = All selected data is retrieved at once and stored in data cache.

Label	Description
Auto commit	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = After each operation, a COMMIT command is automatically sent to the database. No = COMMIT command is not sent to database, and must be done by other means (for example, programming).
Auto refresh	**Menu:** Yes, No. **Default:** No. **Explanation:** Yes = Each time a row is inserted or updated, it is sent to the database and retrieved for verification in the data source. No = Row is not retrieved.
Auto select	**Menu:** Yes, No. **Default:** No. **Explanation:** Yes = Data source recomputes its data by submitting a query to the database each time a foreign data source, to which the data source is connected by parameters, changes. No = Data source does not recompute its data.
Insert nulls	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = Null columns are inserted in the database table when a row is inserted. Database schema default values are not taken into account since a null value is explicitly specified, but performance may be increased. No = Null columns are not inserted.
Dynamic SQL	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = When a row is updated in the database, only those values of columns in remote tables whose values have changed in current table are set. No = All values of columns in remote tables are set.

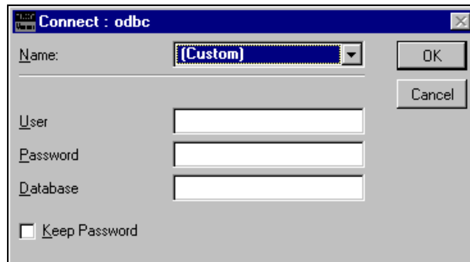
Label	Description
Use bound vars	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = When column values are sent to the database, are packaged in bound variables of the native database call interface, instead of being part of the SQL statements. When combined with Dynamic SQL = No and Insert Nulls = Yes, can greatly increase performance. No = Column values are not packaged in bound variables.
Rows count limit	**Menu:** None. **Default:** No default. **Explanation:** Maximum number of rows that can be retrieved. If empty, unlimited number of rows can be retrieved.
Connection	**Menu:** None. Click button to open the Connect dialog box. **Default:** No default. **Explanation:** Specifies the SQL session by which the data source will communicate with the database.
Query conjunct	**Menu:** None. **Default:** No default. **Explanation:** Additional SQL selection criteria that will be added to criteria in Where and/or Having rows.
Transaction manager	**Menu:** List of available transaction managers. **Default:** No default. **Explanation:** Name of the transaction manager used by this data source.
Use property manager	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = The data source uses a property manager. No = The data source does not use a property manager.

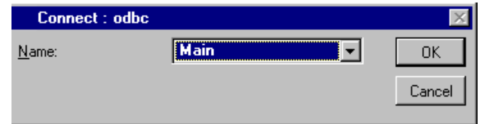
Connect Dialog Box

The Connect dialog box is used for connecting the data source to a database, that is, for establishing an SQL session. It is called by the following menu items in the SQL Data Source inspector panel when the data source is not connected to a database:

- **Properties...** —> **Connection** button in the SQL Data Source Properties panel that appears. See Properties... menu item and Connection field.
- **View Source...** menu item in the File menu. See View Source... menu item.
- **Add Tables...** menu item in the Query menu. See Add Tables... menu item.

The Connect dialog box that appears depends on the type of DBMS being used. Below is a Connect dialog box for an odbc DBMS.





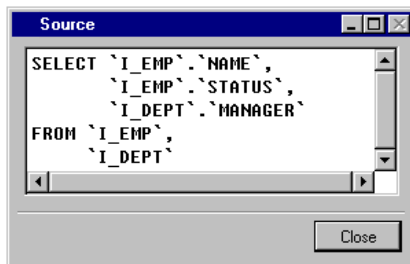
The Connect dialog box changes to this version on the right after an SQL session is chosen from the combo box menu in the Name field

Element	Description
Name field	The name of the SQL session or `(Custom)`. If `(Custom)`, must fill in other fields to create a new SQL session.
User field	The system database user name.
Password field	The user password.
Database field	The name of the database.
Options field	The data necessary to connect to the database.
Keep Password checkbox	When not checked, forces the user to enter a password each time a connection to the database is requested.
OK button	Validates the entries and creates an SQL session having the name entered in the Name field.
Cancel button	Closes the Connect dialog box without validating the entries. No SQL session is created.

Source Dialog Box

The Source dialog box is used for reading the SQL statements automatically created by the user's screen operations. The Source dialog box is called by the View Source... menu item of the File menu in the SQL Data Source inspector panel if the data source is connected to a database. If not already connected to a database, this menu item calls the Connect dialog box . After connecting to the database through the Connect dialog box, the Source dialog box then appears.

The Source dialog box shows the SQL statements corresponding to the current data source. It is read-only.



Button	Description
Close	Closes the Source dialog box.

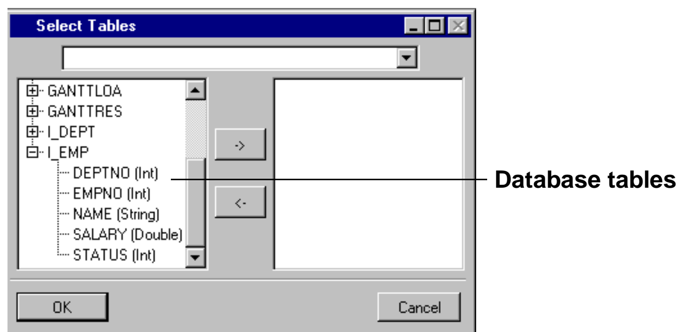
Select Tables Dialog Box

The Select Tables dialog box is used for selecting tables to be added to the data source from the currently connected database. A representation of the table is placed in the FROM section of the SQL Data Source inspector panel. See the picture of the *IliSQLDataSource Inspector Panel* on page 172.

The Select Tables dialog box is called by the Add Tables... menu item of the Query menu in the SQL Data Source inspector panel (if the data source is connected to a database). If not already connected to a database, this menu item calls the Connect dialog box. After connecting to the database through the Connect dialog box, the Select Tables dialog box then appears.

The Select Tables dialog box consists of:

- A list on the left side of the box containing all the tables owned by the selected user.
- A list on the right side of the box, initially empty, to which the end user can add tables.
- Two buttons for moving selected database tables:
  - -> Adds the table selected in the left list to the right list.
  - <- Removes the selected table from the right list.



Note: To display the columns of a table in the left list, double-click on its name or on the '+' to the left of its name.

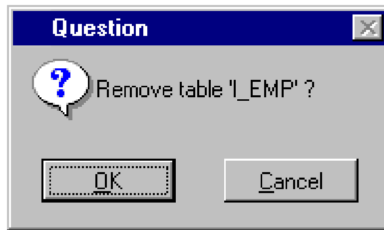
Button	Description
OK	Adds the selected database tables to the data source.
Cancel	Closes the Select Table dialog box without adding any table to the data source.

Question Dialog Box

The Question dialog box is used for confirming a command by the user. It is called by the following menu items of the Query menu in the SQL Data Source inspector panel when an item corresponding to the command is selected in the panel:

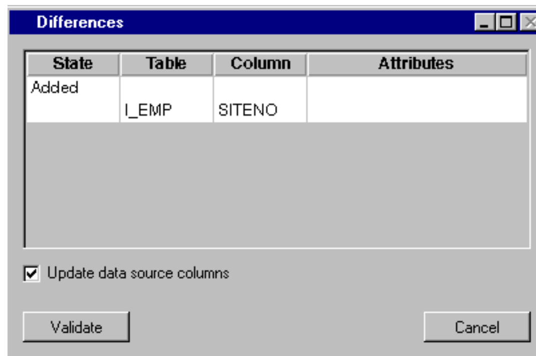
- **Remove Table...** See Remove Table... menu item.
- **Delete Column...** See Delete Column... menu item.
- **Delete Join...** See Delete Join... menu item.

The Question dialog box allows you to confirm one of the above three commands.



Button	Description
OK	Deletes the selected item.
Cancel	Closes the Question dialog box and no item is deleted.

Differences Dialog Box



This dialog box is used to show the differences between the data source tables and the database tables—in the event that a table has been modified in the database by another user. When you synchronize a data source with a table (or all tables) the structural information in the FROM data source table changes. (These operations are on the Query menu. If there are differences between the tables, the above window is displayed.) To update the table(s) in the FROM section press Validate. To update the table(s) in the SELECT section as well, check the Update data source columns box and Validate.

Column	Description
State	Indicates whether something has been added to, removed from, or changed in the database.
Table	Shows the table name being synchronized.

Column	Description
Column	Shows the column being synchronized.
Attributes	The four possible types are: Max length, PartofKey, Datatype, Nullable.

IliMemoryDataSource

The `IliMemoryDataSource` gadget is used for:

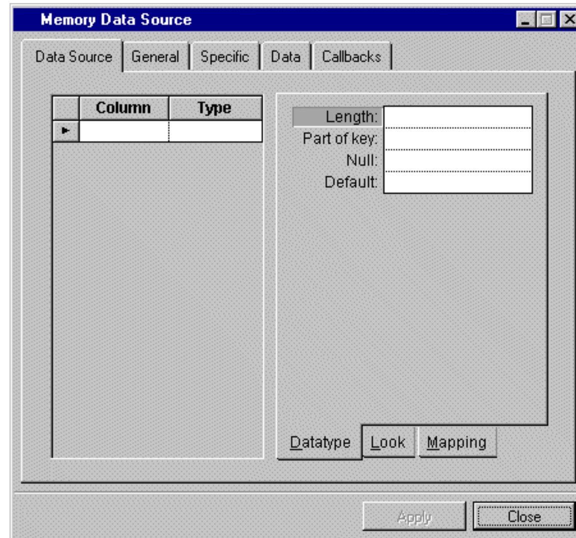
- defining temporary tables in local memory by entering data.
- defining how the data is to be displayed by specifying format criteria for display gadgets connected to the memory data source.

IliMemoryDataSource Inspector Panel

This panel appears by double-clicking its gadget-icon (seen above) in the Gadgets buffer window. The `IliMemoryDataSource` inspector has five notebook pages:

- *Data Source Page*
- *General Page*
- *Specific Page*
- *Data Page*
- *Callbacks Page*

Data Source Page



Data Source Page Table Columns

Column	Description
Column	**Menu:** None. **Default:** No default. **Explanation:** The name of the column.
Type	**Menu:** String, Boolean, Byte, Integer, Float, Double, Date **Default:** No default. **Explanation:** The type of data that can be entered in the column.

Data Source Page Notebook Pages

The Data Source page has three notebook pages. The rows on these pages define the criteria for mapping data to other data source columns and for formatting the data in display gadgets.

- Datatype Page

The Datatype page is used to define the type of data that can be entered in the column.

Length:
Part of key:
Null:
Default:

Datatype | Look | Mapping

Label	Description
Length	**Menu:** None. **Default:** No default. **Explanation:** Number of characters that can be entered in the cells of the column.
Part of key	**Menu:** Yes, No. **Default:** No **Explanation:** Yes = The column is included in the key for the table. No = The column is not included in the key.
Null	**Menu:** Yes, No. **Default:** No **Explanation:** Yes = The cell can remain empty. No = The cell cannot remain empty.
Default	**Menu:** None. **Default:** No default. **Explanation:** Data that appears in a cell when it is added to the table.

◆ Look Page

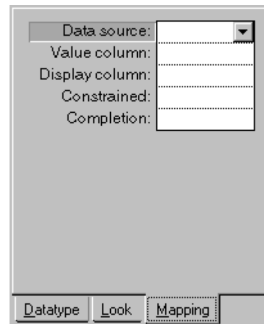
The Look page is used for defining how data entered in the column will appear.

Label	Description
Format	**Menu:** Formats corresponding to what is entered in the column Type cell in the Memory Data Source inspector panel. **Default:** No default. **Explanation:** Predefined system and user formats from the menu or a format entered by the user, by which data in the column cells will be formatted.
Mask	**Menu:** Masks corresponding to how and what data is entered in the column Type cell in the Memory Data Source inspector panel. **Default:** No default. **Explanation:** Predefined by the user for data input in the column cells. There are predefined system masks for date and time.
Alignment	**Menu:** Left, Center, Right. **Default:** Depends on the entry in the Type cell in the Memory Data Source inspector panel. **Explanation:** How data in the column cells will be aligned within the cell.
Width	**Menu:** None. **Default:** No default. **Explanation:** The display width in pixels of the column cells. Can be changed in the table gadget.
Read only	**Menu:** Yes, No. **Default:** No default. **Explanation:** Yes = Prevents the column cells from being edited. No = Allows the column cells to be edited.

Label	Description
Visible	**Menu:** Yes, No. **Default:** No default. **Explanation:** Yes = The column is visible. No = The column exists but does not appear.
Header	**Menu:** None. **Default:** No default. **Explanation:** The title that will appear at the top of the column when displayed in a table gadget. If left empty, the table gadget uses the column name.
Label	**Menu:** None. **Default:** No default. **Explanation:** Applies only when the Data Source Assistant is used to create a form. The caption that appears next to the form gadget containing the data for the column. If empty, the column name is used. (The Header row on the Look page is not used.)

◆ Mapping Page

The Mapping page is used to display data in a column by referring to data in a column in another table.

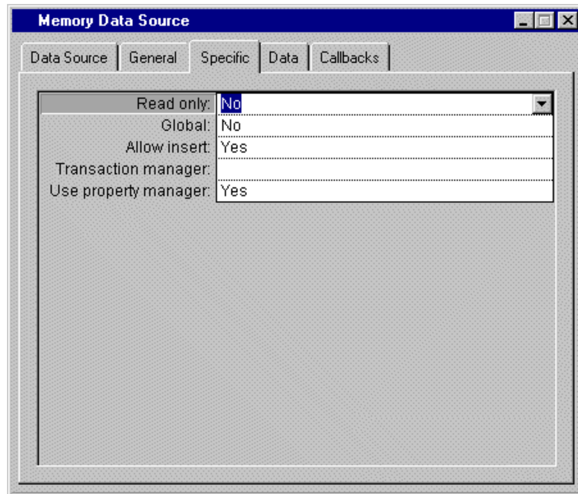


Label	Description
Data Source	**Menu:** Current data sources. **Default:** No default. **Explanation:** The foreign data source containing the columns to which the values for the current column are to be mapped. If a foreign data source is specified here, creates a combo box pull-down menu in the cell showing the values in foreign data source.
Value column	**Menu:** Columns of the data source selected in the Data source row. **Default:** No default. **Explanation:** Column containing the value to which the current column is to be mapped.
Display column	**Menu:** Columns of the data source selected in the Data source row. **Default:** No default. **Explanation:** Column associated with the Value column containing the data to be displayed.
Constrained	**Menu:** Yes, No. **Default:** No default. **Explanation:** Applies only when the value entered in the Value column and Display column rows is the same. Yes = Can only enter a value that belongs to the foreign data source. No = Can enter any value.
Completion	**Menu:** Yes, No. **Default:** No default. **Explanation:** Is only in effect when Constrained = Yes. Yes = Can enter a combo box list item by typing enough of its initial characters to make it unique, then validating it or leaving the cell. No = Cannot enter a combo box list item by typing its initial characters.

General Page

For a description of this notebook page, refer to the section *General Notebook Page* on page 165.

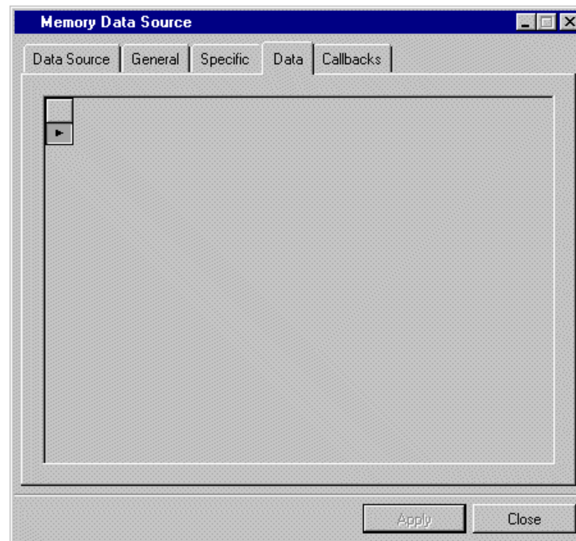
Specific Page



Column	Description
Read only	**Menu:** Yes, No. **Default:** No default. **Explanation:** Yes = Prevents the column cells from being edited. No = Allows the column cells to be edited.
Global	**Menu:** Yes, No. **Default:** No. **Explanation:** Yes = Allows more than one user panel to use the current memory data source. No = Only one user panel can use the current memory data source.
Allow insert	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = Allows a new row to be inserted into the data source table. No = Prevents the user from inserting a new row into the data source tables, but does not prevent the user from editing existing rows.

Column	Description
Transaction manager	**Menu:** List of available transaction managers. **Default:** No default. **Explanation:** Name of the transaction manager used by this data source.
Use property manager	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = The data source uses a property manager. No = The data source does not use a property manager.

Data Page



The Data page is used to edit the data source data. But the data source schema must be defined and validated.

Callbacks Page

In addition to the callbacks described in the section *Callbacks Notebook Page* on page 167, this inspector uses the callbacks listed below.

- `ValidateRow`
- `FetchRow`
- `EnterRow`
- `QuitRow`

- `EnterUpdate Mode`
- `PrepareUpdate`
- `QuitUpdateMode`
- `EnterInsertMode`
- `PrepareInsert`
- `QuitInsertMode`
- `PrepareDeleteMode`
- `CancelEdits`
- `DeleteRow`
- `EnterModifiedState`

Buttons

The Memory Data Source inspector panel has two buttons at the bottom:

- Apply
- Close

Button	Description
Apply	Applies changes made in the Memory Data Source inspector panel to the data source table(s).
Close	Closes the Memory Data Source inspector panel.

C H A P T E R

12

Display Gadgets Reference

This chapter describes the display gadgets listed below. To access these gadgets, click Data Access or Grapher or Gantt Chart in the Palettes panel. The gadgets appear in the lower pane. To use a gadget, drag-and-drop it in the Gadgets buffer window.

You can find information on the following topics:

- *IliTableGadget*
- *IliDbField*
- *IliEntryField*
- *IliTableComboBox*
- *IliDbText*
- *IliDbToggle*
- *IliToggleSelector*
- *IliDbNavigator*
- *IliDbTimer*
- *IliHTMLReporter*
- *IliXML*
- *IliDbPicture*

- *IliDbOptionMenu*
- *IliDbStringList*
- *IliDbTreeGadget*
- *IliChartGraphic*
- *IliDbGrapher*
- *IliDbGantt*

IliTableGadget

The `IliTableGadget` is used for editing and displaying tables.

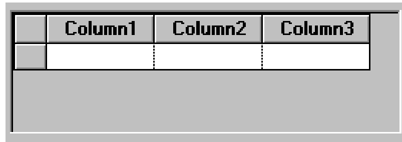
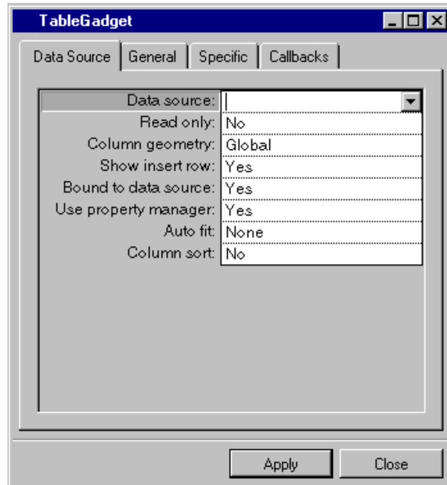


Table Gadget Inspector Panel

The `IliTableGadget` inspector has four notebook pages:

- *Data Source Page*
- *General Page*
- *Specific Page*
- *Callbacks Page*

Data Source Page



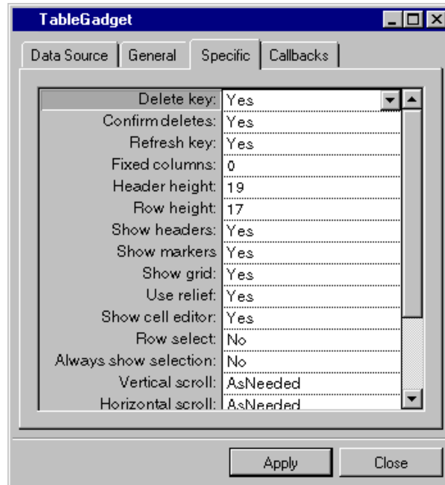
Label	Description
Data source	**Menu:** Names of current data sources. **Default:** No default. **Explanation:** Name of the source to which the table gadget is to be connected.
Read only	**Menu:** Yes, No. **Default:** No. **Explanation:** Yes = The gadget cannot be edited. No = The gadget can be edited.
Column geometry	**Menu:** Local, Global. **Default:** Global. **Explanation:** Local = The table gadget manages column widths, visibility and ordering independently of the underlying data source table. Global = The table gadget manages column widths, visibility, and ordering in conjunction with the underlying data source table. If there are many table gadgets using the same underlying data source, Local is recommended.

Label	Description
Show insert row	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = Shows insert row. No = Hides insert row.
Bound to data source	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = Current row of the table gadget is synchronized with the current row of the data source. No = Current row of the table gadget is independent of the current row of the data source.
Use property manager	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = The table gadget uses a property manager. No = The table gadget does not use a property manager.
Auto fit	**Menu:** None, Proportional, Last. **Default:** None. **Explanation:** Sets how visible column widths change as table gadget is resized. None = Column widths do not change. Proportional = Column widths change proportionally, leaving no empty space. Last = Only width of last column changes to fill all empty space.
Column sort	**Menu:** Yes, No. **Default:** No. **Explanation:** Yes = Sorts columns of the table gadget columns by alphabetical or numeric order. No = Does not sort columns of the table gadget

General Page

For a description of this notebook page, refer to the section *General Notebook Page* on page 165.

Specific Page



Label	Description
Delete key	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = The Delete key can be used to delete the current row. No = The Delete key cannot be used to delete the current row.
Confirm deletes	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = The user is prompted to confirm the deletion of a row. No = The user is not prompted to confirm the deletion of a row.
Refresh key	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = User can use the Refresh key (F9 by default) to refresh the data source table. No = Refresh key is disabled.

Label	Description
Fixed columns	**Menu:** None. **Default:** 0. **Explanation:** The number of columns on the left side of the table that do not scroll.
Header height	**Menu:** None. **Default:** Height of the font used for drawing headers. **Explanation:** Height of the header row in pixels.
Row height	**Menu:** None. **Default:** Height of the font used for drawing cells. **Explanation:** Height of the rows, other than the header, in pixels.
Show headers	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = Header row is displayed. No = Header row is not displayed.
Show markers	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = Row markers are displayed. No = Row markers are not displayed.
Show grid	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = Grid is displayed in the table gadget. No = Grid is not displayed in the table gadget.
Use relief	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = Table gadget has a relief border. No = Table gadget does not have a relief border.
Show cell editor	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = Cell editor is shown in cells when table gadget is read only. No = Cell editor is not shown when table gadget is read only.

Label	Description
Row select	**Menu:** Yes, No. **Default:** No. **Explanation:** Yes = When an attempt is made to select a cell, the entire row containing the cell is selected instead. No = A cell can be selected independently of its row.
Always show selection	**Menu:** Yes, No. **Default:** No. **Explanation:** Yes = Selection in the table gadget is always highlighted. No = Selection in the table gadget is highlighted only when table gadget has the focus.
Vertical scroll	**Menu:** Never, AsNeeded, Always. **Default:** AsNeeded. **Explanation:** Never = Vertical scroll bar is never displayed. AsNeeded = Vertical scroll bar is displayed when needed. Always = Vertical scroll bar is always displayed.
Horizontal scroll	**Menu:** Never, AsNeeded, Always. **Default:** AsNeeded. **Explanation:** Never = Horizontal scroll bar is never displayed. AsNeeded = Horizontal scroll bar is displayed when needed. Always = Horizontal scroll bar is always displayed.
Header font	**Menu:** None. Click the button to open the *Font Chooser Dialog Box*. **Default:** Font of table gadget palette. **Explanation:** Sets the font used in the header row of the table gadget.
Cell font	**Menu:** None. **Button:** Click to open the *Font Chooser Dialog Box*. **Default:** Font of table gadget palette. **Explanation:** Sets the font used for cells in the table gadget.

Label	Description
Cell background	**Menu:** None. **Button:** Click to open the *Color Chooser Dialog Box*. **Default:** Background color of table gadget palette. **Explanation:** Sets background color for cells in the table gadget.
Cell foreground	**Menu:** None. **Button:** Click to open the *Color Chooser Dialog Box*. **Default:** Foreground color of the table gadget palette. **Explanation:** Sets foreground color for cells in the table gadget.
Multi selection	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = More than one table gadget row can be selected at the same time. No = Table gadget rows cannot be selected at the same time.

Callbacks Page

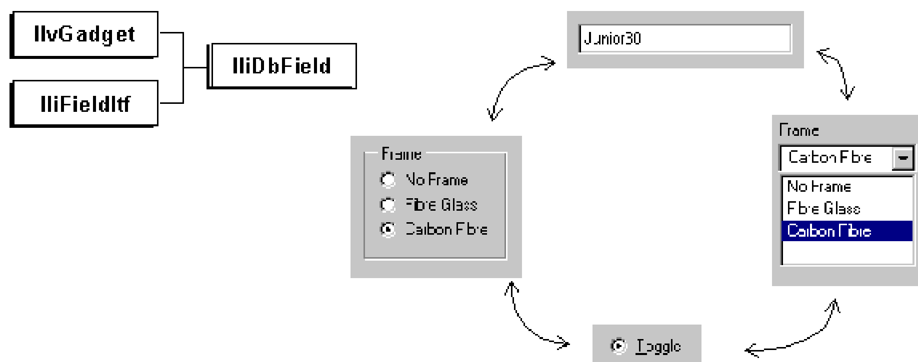
In addition to the callbacks described in the section *Callbacks Notebook Page* on page 167, this inspector uses the callbacks listed below, which are described in the `IliTableGadget` section of the IBM ILOG Views *Data Access Reference Manual*.

- `DoubleClick`
- `ValidateCell`
- `ValidateRow`
- `EnterCell`
- `QuitCell`
- `EnterRow`
- `QuitRow`
- `SelectionChange`
- `EnterUpdateMode`
- `PrepareUpdate`
- `QuitUpdateMode`
- `EnterInsertMode`
- `PrepareInsert`
- `QuitInsertMode`

- `PrepareDeleteRow`
- `CancelEdits`
- `DeleteRow`
- `FetchRow`
- `DrawCell`
- `GetCellPalette`

IliDbField

The `IliDbField` gadget is used for displaying data with a data-source-aware gadget whose appearance can be dynamically changed (see the Style field below).

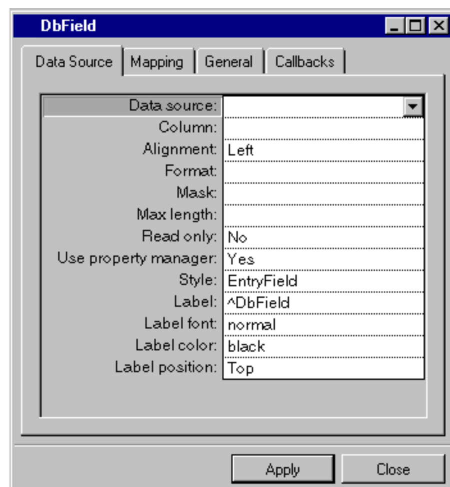


DbField Inspector Panel

The `IliDbField` inspector has four notebook pages:

- *Data Source Page*
- *Mapping Page*
- *General Page*
- *Callbacks Page*

Data Source Page



Label	Description
Data source	**Menu:** Names of current data sources. **Default:** No default. **Explanation:** Name of the data source to which the `DbField` gadget is to be connected.
Column	**Menu:** Column names of the data source selected in the Data source field. **Default:** No default. **Explanation:** Column of the data source table to which the `DbField` gadget is to be connected.
Alignment	**Menu:** Left, Center, Right. **Default:** Left. **Explanation:** Alignment of the value in the `DbField` gadget.
Format	**Menu:** List of predefined system and user formats. **Default:** No default. **Explanation:** Format to be applied to the value in the `DbField` gadget.
Mask	**Menu:** List of predefined system and user input formats. **Default:** No default. **Explanation:** Input format to be entered in the `DbField` gadget.

Label	Description
Max length	**Menu:** None. **Default:** No default. **Explanation:** Maximum number of characters that can be entered in the `DbField` gadget.
Read only	**Menu:** Yes, No. **Default:** No. **Explanation:** Yes = The field can be edited. No = The field cannot be edited.
Use property manager	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = The `DbField` gadget uses a property manager. No = The `DbField` gadget does not use a property manager.
Style	**Menu:** List of possible styles the `DbField` gadget can assume. **Default:** EntryField. **Explanation:** Sets the style for the `DbField` gadget.
Label	**Menu:** None. **Default:** DbField **Explanation:** The text for the label placed next to the gadget.
Label font	**Menu:** None. **Button:** Click to open the *Font Chooser Dialog Box*. **Default:** Font of `DbField` gadget palette. **Explanation:** Font used for the label entered in the Label field.
Label color	**Menu:** None. **Button:** Click to open the *Color Chooser Dialog Box*. **Default:** Foreground color of the `DbField` gadget palette. **Explanation:** Color used for the label entered in the Label field.
Label position	**Menu:** Top, Left. **Default:** Top. **Explanation:** The position of the label relative to the gadget.

Mapping Page

Label	Description
Foreign data source	**Menu:** Names of foreign data sources. **Default:** No default. **Explanation:** Data source containing the columns to which the values for the current column are to be mapped so as to convert the value to another value and display it.
Foreign value column	**Menu:** Column names of the data source selected in the Foreign data source field. **Default:** No default. **Explanation:** Column in the foreign data source containing the value to which the current column is to be mapped.
Foreign display column	**Menu:** Column names of the data source selected in the Foreign data source field. **Default:** No default. **Explanation:** Column in the foreign data source containing the value to be displayed when the column specified in the Foreign Value Column row is referred to.

Label	Description
Constrained	**Menu:** Yes, No. **Default:** No. **Explanation:** Applies only when the value entered in the Foreign value column and Foreign display column rows is the same. Yes = Can only enter a value that belongs to the foreign data source. No = Can enter any value.
Completion	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Is in effect only when constrained = Yes. Yes = Can enter a DbField item by typing enough of its initial characters to make it unique, then validating it or leaving the cell. No = Cannot enter a DbField item by typing its unique initial characters.

General Page

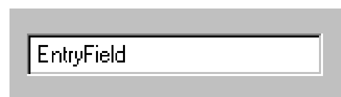
For a description of this notebook page, refer to the section *General Notebook Page* on page 165.

Callbacks Page

For a description of this notebook page, refer to the section *Callbacks Notebook Page* on page 167.

IliEntryField

The `IliEntryField` gadget is used for entering and displaying data in a data-source-aware, one-line text field.



Entry Field Inspector Panel

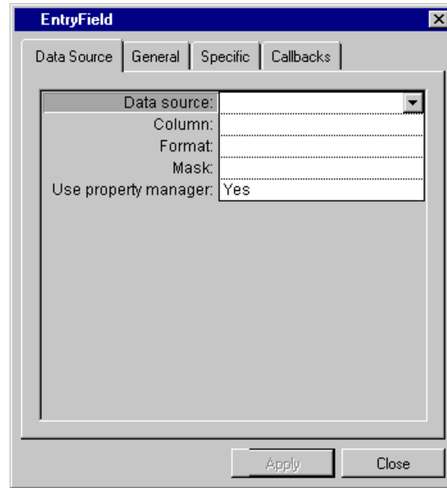
The `IliEntryField` inspector has four notebook pages:

- *Data Source Page*
- *General Page*

◆ *Specific Page*

◆ *Callbacks Page*

Data Source Page



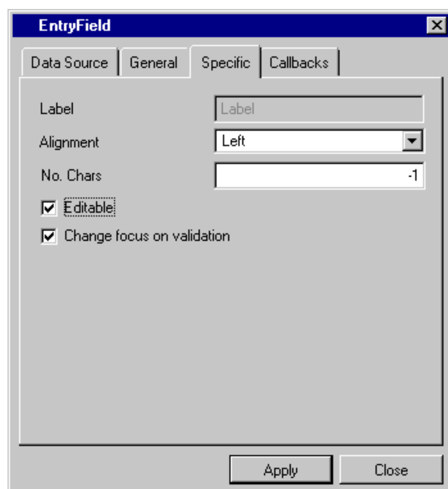
Label	Description
Data source	**Menu:** Names of current data sources. **Default:** No default. **Explanation:** Name of the source to which the entry field gadget is to be connected.
Column	**Menu:** Column names of the data source selected in the Data source field. **Default:** No default. **Explanation:** Column of the data source table to which the gadget is to be connected.
Format	**Menu:** List of predefined system and user formats. **Default:** No default. **Explanation:** Format to be applied to the value in the entry field gadget.

Label	Description
Mask	**Menu:** List of predefined system and user input formats. **Default:** No default. **Explanation:** Input format to be applied to the value in the entry field gadget.
Use property manager	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = The entry field gadget uses a property manager. No = The entry field gadget does not use a property manager.

General Page

For a description of this notebook page, refer to the section *General Notebook Page* on page 165.

Specific Page



Label	Description
Label	Not available.
Alignment	**Menu:** Left, Center, Right. **Default:** Left. **Explanation:** The alignment of the value in the Entry Field gadget.
No. Chars	**Menu:** None. **Default:** -1. **Explanation:** The maximum number of characters that can be entered in the Entry Field gadget.
Editable	**Check box.** **Default:** Checked. **Explanation:** Checked = The field in the gadget can be edited. Not checked = The field in the gadget cannot be edited.
Change focus on validation	**Check box.** **Default:** Checked. **Explanation:** Checked = Focus moves to the next gadget after validation. Not checked = Focus remains on this gadget.

Callbacks Page

For a description of this notebook page, refer to the section *Callbacks Notebook Page* on page 167.

IliTableComboBox

The `IliTableComboBox` gadget is used for displaying a list of items in a data-source-aware menu and then displaying the item selected from the list.

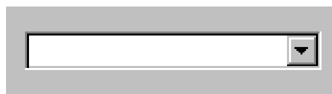
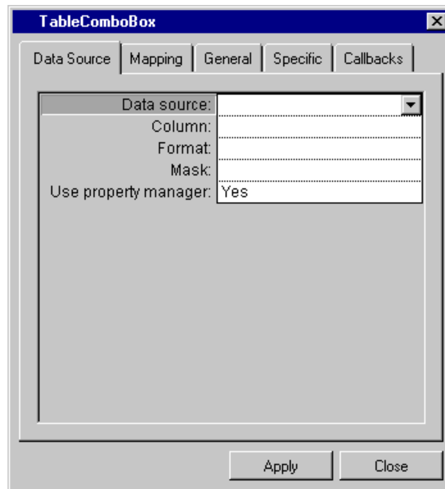


Table Combo Box Inspector Panel

The `IliTableComboBox` inspector has five notebook pages:

- *Data Source Page*
- *Mapping Page*
- *General Page*
- *Specific Page*
- *Callbacks Page*

Data Source Page



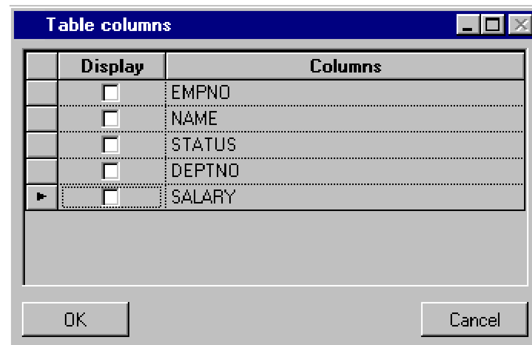
Label	Description
Data source	**Menu:** Names of current data sources. **Default:** No default. **Explanation:** Name of the source to which the table combo box gadget is to be connected.
Column	**Menu:** Column names of the data source selected in the Data source field. **Default:** No default. **Explanation:** Column of the data source table to which the gadget is to be connected.
Format	**Menu:** List of predefined system and user formats. **Default:** No default. **Explanation:** Format to be applied to the values in the table combo box gadget.
Mask	**Menu:** List of predefined system and user input formats. **Default:** No default. **Explanation:** Input format to be applied to the values in the table combo box gadget.
Use property manager	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = The table combo box uses a property manager. No = The table combo box does not use a property manager.

Mapping Page

TableComboBox
Data Source
Mapping
General
Specific
Callbacks
Foreign data source:
Foreign value column:
Foreign display column:
Constrained: No
Completion: Yes
Table columns:
Apply
Close

Label	Description
Foreign data source	**Menu:** Names of foreign data sources. **Default:** No default. **Explanation:** A data source containing the columns to which the values for the current column are to be mapped so that the value can be converted to another value and displayed. Creates a combo box pull-down menu in the table combo box field showing the values in the foreign data source.
Foreign value column	**Menu:** Column names of the data source selected in the Foreign data source field. **Default:** No default. **Explanation:** Column in the foreign data source containing the value to which the current column is to be mapped.
Foreign display column	**Menu:** Column names of the data source selected in the Foreign data source field. **Default:** No default. **Explanation:** The column in the foreign data source containing the value to be displayed when the column specified in the Foreign value column row is referred to.
Constrained	**Menu:** Yes, No. **Default:** No. **Explanation:** Applies only when the value entered in the Foreign value column and Foreign display column rows is the same. Yes = Can only enter a value that belongs to foreign data source. No = Can enter any value.
Completion	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Is only in effect when constrained = Yes. Yes = Can enter a combo box list item by typing enough of its initial characters to make it unique, then validating it or leaving the cell. No = Cannot enter a combo box list item by typing its initial characters.
Table columns	**Menu:** None. **Button:** Click to have the *Table Columns Dialog Box* appear. **Default:** No default. **Explanation:** The column(s) in the foreign data source to be displayed in the pull-down menu.

Table Columns Dialog Box



Label	Description
Display	**Check box.** **Default:** Not checked. **Explanation:** If checked, the column name appears in the pull-down menu.
Columns	Column names.

General Page

For a description of this notebook page, refer to the section *General Notebook Page* on page 165.

Specific Page

Label	Description
Label	Not available.
Alignment	**Menu:** Left, Center, Right. **Default:** Left. **Explanation:** Alignment of the values in the table combo box gadget.
No. Chars	**Menu:** None. **Default:** -1. **Explanation:** The maximum number of characters that can be entered in the Table Combo Box gadget.
Editable	**Check box.** **Default:** Checked. **Explanation:** Checked = The field in the gadget can be edited. Not checked = The field in the gadget cannot be edited.
Change focus on validation	**Check box.** **Default:** Checked. **Explanation:** Checked = Focus moves to the next gadget. Not checked = Focus remains on this gadget.

Callbacks Page

In addition to the callbacks described in the section *Callbacks Notebook Page* on page 167, this inspector uses the callbacks listed below, which are described in the `IliAbstractComboBox` section of the IBM ILOG Views *Data Access Reference Manual*.

- `Open`
- `Close`

IliDbText

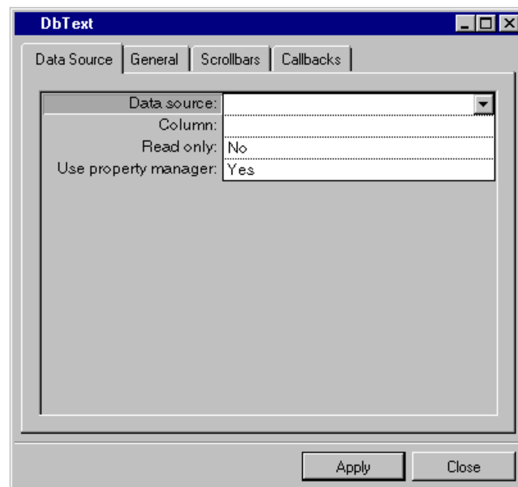
The `IliDbText` gadget is used for entering and displaying multi-line data in a scrollable data-source-aware text field.

DbText Inspector Panel

The `IliDbText` inspector has four notebook pages:

- *Data Source Page*
- *General Page*
- *Scrollbars Page*
- *Callbacks Page*

Data Source Page



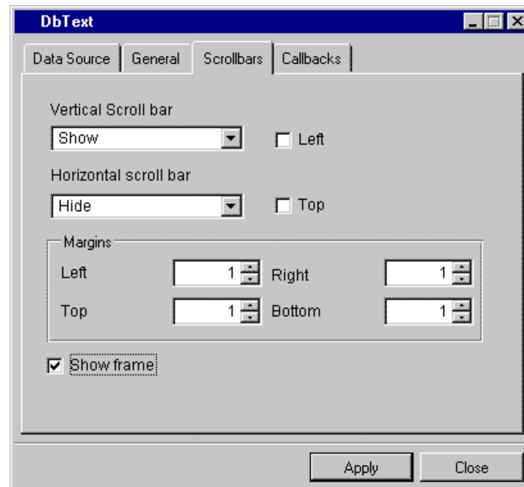
Label	Description
Data source	**Menu:** Names of current data sources. **Default:** No default. **Explanation:** Name of the data source to which the DbText gadget is to be connected.
Column	**Menu:** Column names of the data source selected in the Data source field. **Default:** No default. **Explanation:** Column of the data source table to which the gadget is to be connected.

Label	Description
Read only	**Menu:** Yes, No. **Default:** No. **Explanation:** Yes = The field can be edited. No = The field cannot be edited.
Use property manager	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = The DbText gadget uses a property manager. No = The DbText gadget does not use a property manager.

General Page

For a description of this notebook page, refer to the section *General Notebook Page* on page 165.

Scrollbars Page



Label	Description
Vertical scroll	**Menu:** Show, Hide **Default:** Show. **Explanation:** Show = The field has a vertical scroll bar. Hide = The field does not have a vertical scroll bar.
Horizontal scroll	**Menu:** Show, Hide **Default:** Hide. **Explanation:** Show = The field has a horizontal scroll bar. Hide = The field does not have a horizontal scroll bar.
Margins	**Menu:** None. **Default:** 1. **Explanation:** Allows you to type the value of the left, right, top, and bottom margins.
Show frame	**Check Box**. **Default**: Checked. **Explanation**: If the check box is checked, a frame appears around the DbText gadget.

Callbacks Page

In addition to the callbacks described in the section *Callbacks Notebook Page* on page 167, this inspector uses the callbacks listed below.

- `ScrollBar Moved`
- `ScrollBar Visibility Changed`
- `Cursor Move`
- `Selection Changed`
- `Value Changed`

IliDbToggle

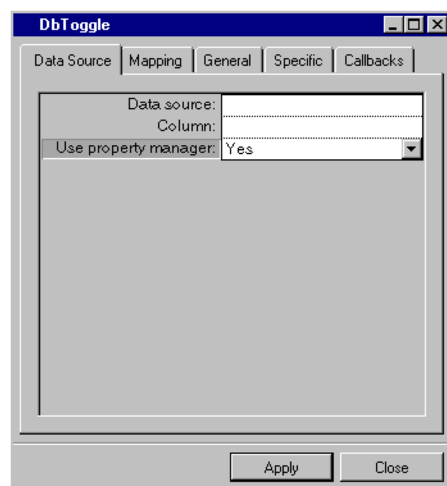
The `IliDbToggle` gadget is used for selecting between two or sometimes three states.

DbToggle Inspector Panel

The `IliDbToggle` inspector has five notebook pages:

- *Data Source Page*
- *Mapping Page*
- *General Page*
- *Specific Page*
- *Callbacks Page*

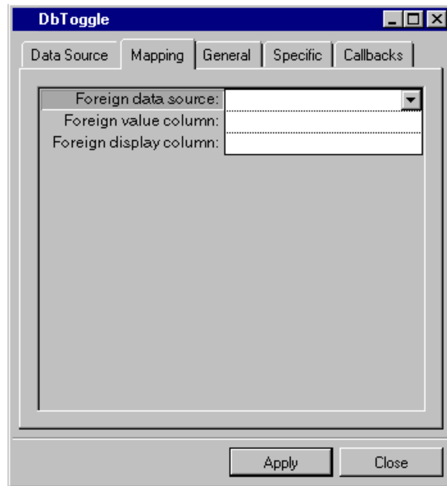
Data Source Page



Label	Description
Data source	**Menu:** Names of current data sources. **Default:** No default. **Explanation:** Name of the data source to which the DbToggle is to be connected.

Label	Description
Column	**Menu:** Column names of the data source selected in the Data source field. **Default:** No default. **Explanation:** Column of the data source table to which the gadget is to be connected.
Use property manager	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = The DbToggle gadget uses a property manager. No = The DbToggle gadget does not use a property manager.

Mapping Page

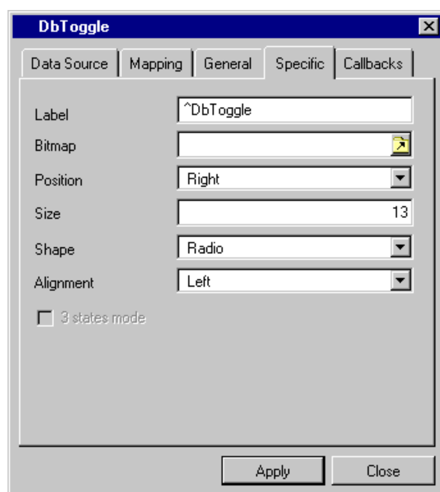


Label	Description
Foreign data source	**Menu:** Names of foreign data sources. **Default:** No default. **Explanation:** Data source containing the columns to which the values for the current column are to be mapped so as to convert the value to another value and display it.
Foreign value column	**Menu:** Column names of the data source selected in the Foreign data source field. **Default:** No default. **Explanation:** Column in the foreign data source containing the value to which the current column is to be mapped.
Foreign display column	**Menu:** Column names of the data source selected in the Foreign data source field. **Default:** No default. **Explanation:** Column in the foreign data source containing the value to be displayed when the column specified in the Foreign value column row is referred to.

General Page

For a description of this notebook page, refer to the section *General Notebook Page* on page 165.

Specific Page



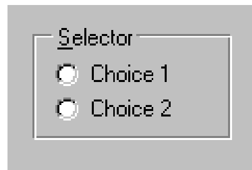
Label	Description
Label	**Menu:** None. **Default:** DbToggle. **Explanation:** The text for the label placed next to the toggle gadget.
Bitmap	**Menu:** None. Button: Click to open the Open dialog box. **Default:** No default. **Explanation:** The bitmap image to be placed as the label next to the toggle gadget.
Position	**Menu:** Left, Right. **Default:** Right. **Explanation:** Position of the label relative to the toggle gadget.
Size	**Menu:** None. **Default:** 13. **Explanation:** The width or height of the state marker.
Shape	**Menu:** Radio, CheckBox. **Default:** Radio. **Explanation:** The type of toggle gadget to be used.
Alignment	**Menu:** Left, Center, Right. **Default:** Left. **Explanation:** Alignment of the label within the bounding box of the toggle gadget.
3 States Mode	**Check box:** Available if Shape = CheckBox (see Shape above). **Default:** Unchecked. **Explanation:** The toggle can have three states (True, False, Null) when the toggle is a check box and if this field is true.

Callbacks Page

For a description of this notebook page, refer to the section *Callbacks Notebook Page* on page 167.

IliToggleSelector

The `IliToggleSelector` gadget is used to select a gadget among any number of items having data-source-aware selector buttons.

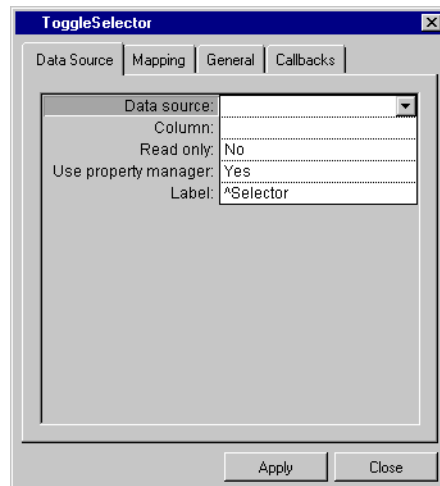


ToggleSelector Inspector Panel

The `IliToggleSelector` inspector has four notebook pages:

- *Data Source Page*
- *Mapping Page*
- *General Page*
- *Callbacks Page*

Data Source Page



Label	Description
Data source	**Menu:** Names of current data sources. **Default:** No default. **Explanation:** Name of the data source to which the toggle selector gadget is to be connected.
Column	**Menu:** Column names of the data source selected in the Data source field. **Default:** No default. **Explanation:** Column of the data source table to which the gadget is to be connected.
Read Only	**Menu:** Yes, No. **Default:** No. **Explanation:** Yes = The toggle selector gadget can be edited. No = The toggle selector gadget cannot be edited.
Use property manager	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = The toggle selector uses a property manager. No = The toggle selector does not use a property manager.
Label	**Menu:** None. **Default:** Selector. **Explanation:** The text for the label placed next to the toggle selector gadget.

Mapping Page

Label	Description
Foreign data source	**Menu:** Names of foreign data sources. **Default:** No default. **Explanation:** Data source containing the columns to which the values for the current column are to be mapped so as to convert the value to another value and display it.
Foreign value column	**Menu:** Column names of the data source selected in the Foreign data source field. **Default:** No default. **Explanation:** Column in the foreign data source containing the value to which the current column is to be mapped.
Foreign display column	**Menu:** Column names of the data source selected in the Foreign Data Source field. **Default:** No default. **Explanation:** Column in the foreign data source containing the value to be displayed when the column specified in the Foreign value column row is referred to.

General Page

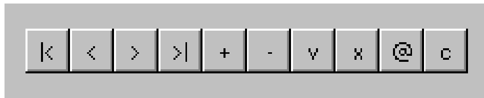
For a description of this notebook page, refer to the section *General Notebook Page* on page 165.

Callbacks Page

For a description of this notebook page, refer to the section *Callbacks Notebook Page* on page 167.

IliDbNavigator

The `IliDbNavigator` gadget is a tool bar with buttons for navigating through rows and editing data in a data source table.

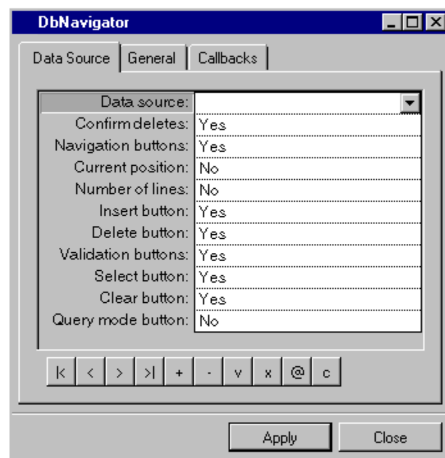


DbNavigator Inspector Panel

The `IliDbNavigator` inspector has three notebook pages:

- Data Source Page
- *General Page*
- *Callbacks Page*

Data Source Page



Label	Description
Data source	**Menu:** Names of current data sources. **Default:** No default. **Explanation:** Name of the data source to which the DbNavigator gadget is to be connected.
Confirm deletes	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = The user is prompted to confirm the deletion of a row. No = The user is not prompted to confirm the deletion of a row.
Navigation buttons	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Yes = Displays four navigation buttons (|<, <, >, >|). No = Does not display four navigation buttons (|<, <, >, >|). |< = go to first row, < = previous row, > = next row, >| = last row.
Current position	**Menu:** Yes, No. **Default:** No. **Explanation:** Yes = Displays the current row position of the query if in Query mode, of the Data source if in Normal mode. No = Does not display the current row position.
Number of lines	**Menu:** Yes, No. **Default:** No. **Explanation:** Yes = Displays the number of lines of the query if in Query mode, of the Data source if in Normal mode. No = Does not display the number of lines.
Insert button	**Menu:** Yes, No. **Default:** Yes. **Explanation:** The insert button gives the focus to (makes current) the insert row. Yes = Displays the insert button (+). No = Does not display the insert button (+).

Label	Description
Delete button	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Applies when a row is selected for deletion. It displays a confirmation dialog box before deleting the row if Confirm Deletes = Yes (see above). If Confirm Deletes = No, row is immediately deleted. Yes = Displays the delete button (-). No = Do not display the delete button (-).
Validation buttons	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Applies when an edit has been made in a row. Yes = Displays the two validation buttons (v, x). No = Do not display the two validation buttons (v, x). In Normal mode: v = validate the edit. x = cancel the edit and return to original state. In Query mode: v = apply Query mode. x = cancel Query mode.
Select button	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Clears the data source cache, queries the database, retrieves the result from the data source, and displays the result in the display gadget. Yes = Displays the select button (@). No = Do not display the select button (@).
Clear button	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Empties the data source cache, thus clearing the display gadget. Yes = Display the clear button (c). No = Do not display the clear button (c).
Query mode button	**Menu:** Yes, No. **Default:** No. **Explanation:** Puts the data source in Query mode, allowing you to use other buttons on this notebook page. Query mode remains active until one of the other validation buttons is used to return to normal mode. Yes = Query mode is active. No = Query mode is not active.

General Page

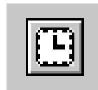
For a description of this notebook page, refer to the section *General Notebook Page* on page 165.

Callbacks Page

For a description of this notebook page, refer to the section *Callbacks Notebook Page* on page 167.

IliDbTimer

The `IliDbTimer` gadget is used for calling a callback periodically.

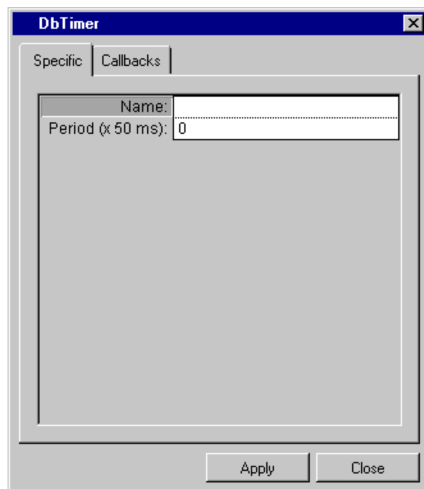


DbTimer Inspector Panel

The `IliDbTimer` inspector has two notebook pages:

- *Specific Page*
- *Callbacks Page*

Specific Page



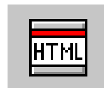
Label	Description
Name	**Menu:** None. **Default:** None. **Explanation:** The name of the gadget.
Period	**Menu:** None. **Default:** 0 **Explanation:** The period with which the callback associated with the gadget will be called. The value entered is multiplied by 0.05 to get seconds (for example, the user needs to type 40 if a period of 2 seconds is desired). If the value is 0, the callback is not called.

Callbacks Page

For a description of this notebook page, refer to the section *Callbacks Notebook Page* on page 167.

IliHTMLReporter

The `IliHTMLReporter` gadget is used to generate an HTML document from data source contents.

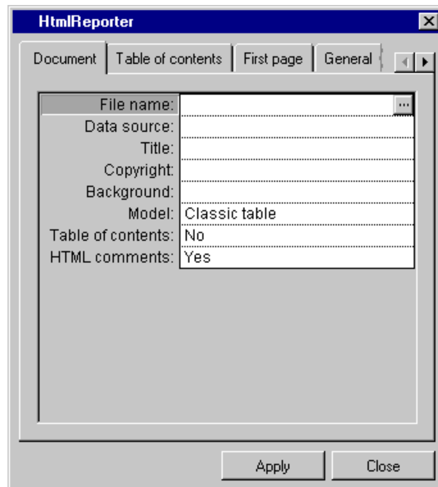


HTMLReporter Inspector Panel

The `IliHTMLReporter` inspector has five notebook pages:

- *Document Page*
- *Table of contents Page*
- *First page Page*
- *General Page*
- *Callbacks Page*

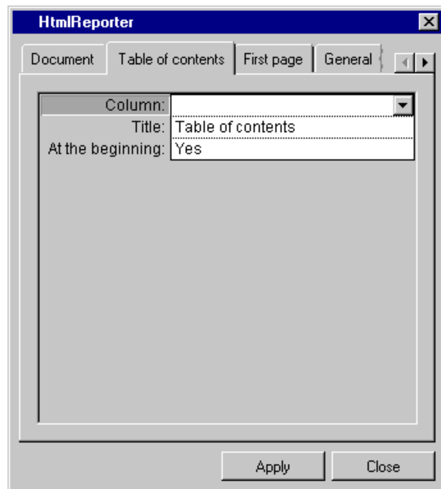
Document Page



Label	Description
File name	**Menu:** None. **Button:** Click to open the *File Chooser Dialog Box*. **Default:** None. **Explanation:** Name of the file to be generated.
Data source	**Menu:** Names of current data sources. **Default:** None. **Explanation:** Data source whose contents will be used to create the HTML file.
Title	**Menu:** None. **Default:** None. **Explanation:** Title of the HTML file.
Copyright	**Menu:** None. **Default:** None. **Explanation:** The HTML document copyright.
Background	**Menu:** None. **Button:** Click to open the *Color Chooser Dialog Box*. **Default:** None. **Explanation:** The HTML page background color.

Label	Description
Model	**Menu:** Classic table, Classic form, Table, Form, Dynamic Form. **Default:** Classic table. **Explanation:** The model that will be used to create the HTML document.
Table of contents	**Menu: Yes, No.** **Default:** No. **Explanation:** If “Yes”, a table of contents will be present in the HTML file.
HTML comments	**Menu: Yes, No.** **Default**: Yes. **Explanation**: Enable or disable comments in the HTML file.

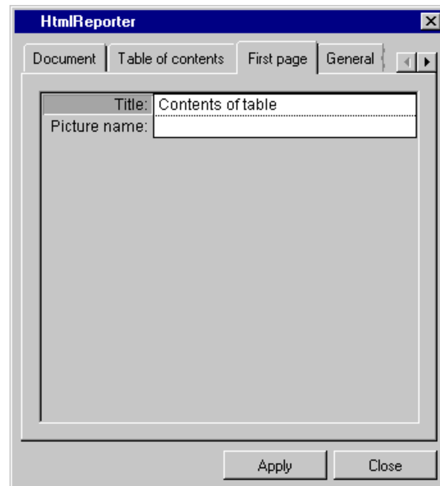
Table of contents Page



Label	Description
Column	**Menu:** Column names of the data source selected in the Document page. **Default:** None. **Explanation:** The column that will be used to create the table of contents.
Title	**Menu:** None. **Default:** Table of contents. **Explanation:** The table of contents title.
At the beginning	**Menu: Yes, No.** **Default:** Yes **Explanation:** Yes = The table of contents appears at the beginning of the document. No = The table of contents appears at the end of the document.

First page Page

This page is used only if the value of Table of contents field on the Documents page is "Yes".



Label	Description
Title	**Menu:** None. **Default:** Contents of the table. **Explanation:** The first page title.
Picture Name	**Menu:** None. Button: Click to open the File Chooser dialog box. **Default:** None. **Explanation:** The picture to appear above the first page title.

General Page

For a description of this notebook page, refer to the section *General Notebook Page* on page 165.

Callbacks Page

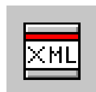
In addition to the callbacks described in the section *Callbacks Notebook Page* on page 167, this inspector uses the callbacks listed below, which are described in the `IliHTMLReporter` section of the IBM ILOG Views *Data Access Reference Manual*.

- `Generic`
- Secondary
- Focus In
- Focus Out
- Enter Gadget
- Leave Gadget
- `ReportBeginDocument`
- `ReportEndDocument`
- `ReportFirstPageHeading`
- `ReportFirstPageContents`
- `ReportFirstPageFooting`
- `ReportTableHeading`
- `ReportTableTitle`
- `ReportTableBeginEntries`
- `ReportTableEndEntries`
- `ReportTableFooting`
- `ReportHeading`

- `ReportBeginRows`
- `ReportRowContent`
- `ReportEndrows`
- `ReportFooting`
- `ReportLastPageHeading`
- `ReportLastPageContents`
- `ReportLastPageFooting`

IliXML

The `IliXML` gadget manages the communication between a datasource and an XML document. It also manages the import and export of notification and definition.

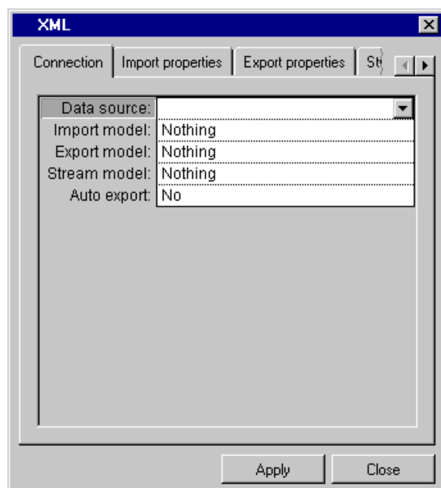


XML Inspector Panel

The `IliXML` inspector has six notebook pages:

- *Connection Page*
- *Import properties Page, Export properties Page, Stream properties Page*
- *General Page*
- *Callbacks Page*

Connection Page



Label	Description
Data source	**Menu:** Names of current data sources. **Default:** No default. **Explanation:** Name of the data source to which the XML gadget is to be connected.
Import model	**Menu:** Nothing, Dynamic, Default. **Default:** Nothing. **Explanation:** The model name that is used to import the XML document.
Export model	**Menu:** Nothing, Dynamic, Default. **Default:** Nothing. **Explanation:** The model name that is used to export the XML document.
Stream model	**Menu:** Nothing, File. **Default:** Nothing. **Explanation:** The model name that is used to connect to an XML document.
Auto export	**Menu:** Yes, No. **Default:** No. **Explanation:** Enable or disable the automatic export of notification.

Import properties Page, Export properties Page, Stream properties Page

These pages contain the properties list of the selected model.

General Page

For a description of this notebook page, refer to the section *General Notebook Page* on page 165.

Callbacks Page

In addition to the callbacks described in the section *Callbacks Notebook Page* on page 167, this inspector uses the callback listed below which is described in the `IliXML` section of the IBM ILOG Views *Data Access Reference Manual*:

- `XMLNotificationExported`

IliDbPicture

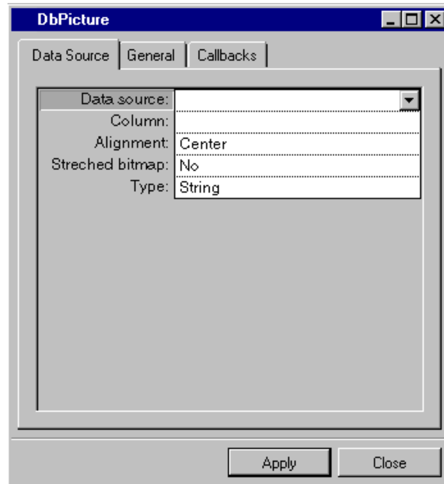
The `IliDbPicture` gadget is used for displaying a picture. The picture file name comes from a data source column.

DbPicture Inspector Panel

The `IliDbPicture` inspector has three notebook pages:

- *Data Source Page*
- *General Page*
- *Callbacks Page*

Data Source Page



Label	Description
Data source	**Menu:** Names of current data sources. **Default:** No default. **Explanation:** Name of the data source to which the DbPicture gadget is to be connected.
Column	**Menu:** Column names of the data source to which the file name of the picture is attached. **Default:** No default. **Explanation:** Column of the data source table to which the gadget is to be connected.
Alignment	**Menu:** Left, Center, Right. **Default:** Center. **Explanation:** Alignment of the picture within the picture gadget bounding box.

Label	Description
Stretched bitmap	**Menu:** Yes, No. **Default:** No. **Explanation:** Yes = The image is stretched in the gadget rectangle. No = The image is not stretched in the gadget rectangle.
Type	**Menu:** String, Bitmap (later). **Default:** String. **Explanation:** Type of data that can be entered in the column. For the time being you can only use the type string for a file name.

General Page

For a description of this notebook page, refer to the section *General Notebook Page* on page 165.

Callbacks Page

For a description of this notebook page, refer to the section *Callbacks Notebook Page* on page 167.

IliDbOptionMenu

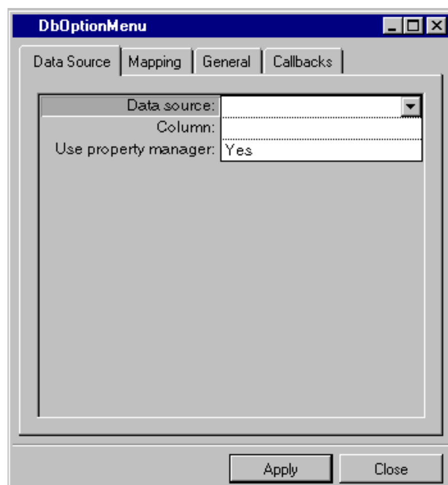
The `IliDbOptionMenu` gadget is used to display a data source list of items.

DbOptionMenu Inspector Panel

The `IliDbOptionMenu` inspector has four notebook pages:

- *Data Source Page*
- *Mapping Page*
- *General Page*
- *Callbacks Page*

Data Source Page



Label	Description
Data source	**Menu:** Name of current data sources. **Default:** No default. **Explanation:** Name of the data source to which the option menu gadget is to be connected.
Column	**Menu:** Column name of the data source selected in the Data source field. **Default:** No default. **Explanation:** Column of the data source table to which the gadget is to be connected.
Use property manager	**Menu:** Yes, No. **Default:** No. **Explanation:** Yes = The DbOptionMenu uses a property manager. No = The DbOptionMenu does not use a property manager.

Mapping Page

Label	Description
Foreign data source	**Menu:** Name of current data sources. **Default:** No default. **Explanation:** Data source containing the columns to which the values for the current column are to be mapped so as to convert the value to another value and display it.
Foreign value column	**Menu:** Column names of the data source selected in the Foreign data source field. **Default:** No default. **Explanation:** Column in the foreign data source containing the value to which the current column is to be mapped.
Foreign display column	**Menu:** Column names of the data source selected in the Foreign data source field. **Default:** No default. **Explanation:** The column in the foreign data source containing the value to be displayed when the column specified in the Foreign value column row is referred to.

General Page

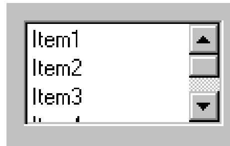
For a description of this notebook page, refer to the section *General Notebook Page* on page 165.

Callbacks Page

For a description of this notebook page, refer to the section *Callbacks Notebook Page* on page 167.

IliDbStringList

The `IliDbStringList` gadget is used to display a data source list of items.

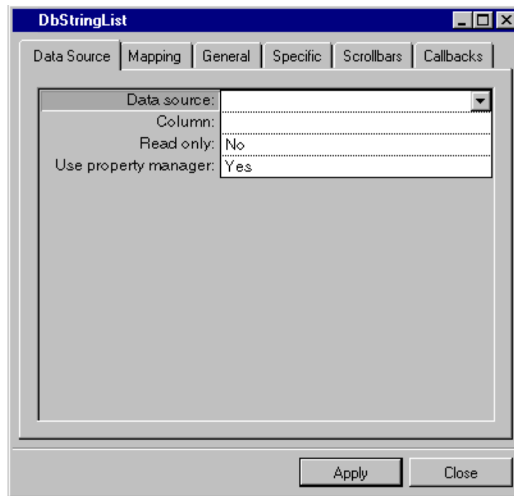


DbStringList Inspector Panel

The `IliDbStringList` inspector has six notebook pages:

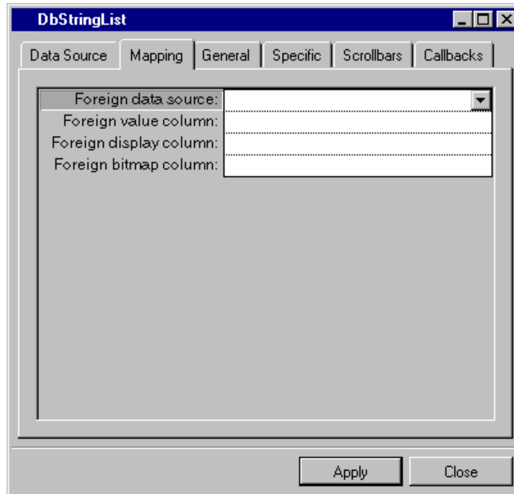
- *Data Source Page*
- *Mapping Page*
- *General Page*
- *Specific Page*
- *Scrollbars Page*
- *Callbacks Page*

Data Source Page



Label	Description
Data source	**Menu:** Names of current data sources. **Default:** No default. **Explanation:** Name of the data source to which the string list gadget is to be connected.
Column	**Menu:** Column names of the data source selected in the Data source field. **Default:** No default. **Explanation:** Column of the data source table to which the gadget is to be connected.
Read only	**Menu:** Yes, No. **Default:** No. **Explanation:** Yes = The field cannot be edited. No = The field can be edited.
Use property manager	**Menu:** Yes, No. **Default:** No. **Explanation:** Yes = The DbStringList uses a property manager. No = The DbStringList does not use a property manager.

Mapping Page



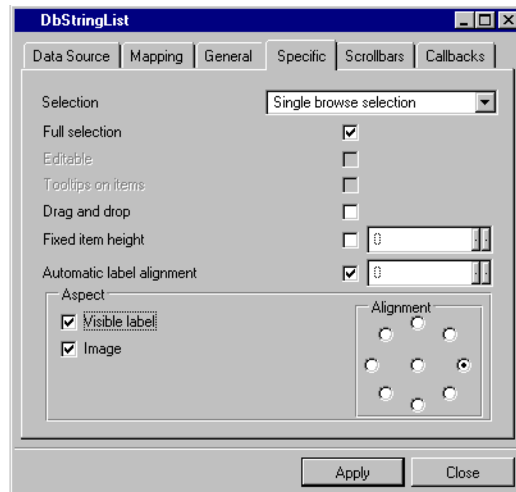
Label	Description
Foreign data source	**Menu:** Name of current data sources. **Default:** No default. **Explanation:** Data source containing the columns to which the values for the current column are to be mapped so as to convert the value to another value and display it.
Foreign value column	**Menu:** Column names of the data source selected in the Foreign data source field. **Default:** No default. **Explanation:** The column in the foreign data source containing the value to which the current column is to be mapped.

Label	Description
Foreign display column	**Menu:** Column names of the data source selected in the Foreign data source field. **Default:** No default. **Explanation:** Column in the foreign data source containing the value to be displayed when the column specified in the Foreign value column row is referred to.
Foreign bitmap column	**Menu:** Column names of the data source selected in the Foreign data source field. **Default:** No default. **Explanation:** Column in the Foreign data source containing the bitmap to be displayed.

General Page

For a description of this notebook page, refer to the section *General Notebook Page* on page 165.

Specific Page



Label	Description
Selection	**Menu:** Single selection, Single browse selection, Multiple selection, Extended selection, Browse selection. **Default:** Single browse selection. **Explanation:** Type of selection to be used by the gadget.
Full selection	**Check box.** **Default:** Checked. **Explanation:** Checked = Selects the entire line. Not checked = Selects only the item length.
Editable	Not available.
Tooltips on items	Not available.
Drag and drop	**Check box.** **Default:** Not checked. **Explanation:** Determines whether items can be dragged and dropped.
Fixed item height	**Check box.** **Default:** Not checked. **Explanation:** When checked, 20 appears as the default.
Automatic label alignment	**Check box.** **Default:** Checked. **Explanation:** When not checked, 28 appears as the default if a foreign data source is used. 0 appears as the default if a foreign data source is not used.
Visible label	**Check box.** **Default:** Checked. **Explanation:** Determines whether labels are visible in the gadget.
Image	**Check box.** **Default:** Checked. **Explanation:** Determines whether images are visible in the gadget.
Alignment	**Menu:** None. Available positions are indicated by graphic. **Default:** Right center. **Explanation:** When Image and Visible label are selected, gives the position of the label relative to the image.

Scrollbars Page

Label	Description
Vertical scroll bar	**Menu:** Show, Hide, As Needed. **Default:** As Needed. **Explanation:** Show = The field has a vertical scroll bar. Hide = The field does not have a vertical scroll bar. As Needed = Vertical scroll bar if needed.
Left	**Check box.** **Default:** Not checked. **Explanation:** Checked = The vertical scroll bar is on the left side of the gadget. Not checked = The vertical scroll bar is on the right side of the gadget.
Horizontal scrollbar	**Menu:** Show, Hide, As Needed. **Default:** As Needed. **Explanation:** Show = The field has a horizontal scroll bar. Hide = The field does not have a horizontal scroll bar. As Needed = Horizontal scroll bar if needed.

Label	Description
Top	**Check box.** **Default:** Not checked. **Explanation:** Checked = The horizontal scroll bar is above the gadget. Not checked = The horizontal scroll bar is below the gadget.
Margins	**Menu:** None. **Default:** 0. **Explanation:** Allows you to type the value of the left, right, top, and bottom margins.
Show Frame	**Check box.** **Default:** Checked. **Explanation:** Determines whether frames are visible.

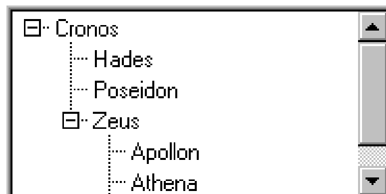
Callbacks Page

In addition to the callbacks described in the section *Callbacks Notebook Page* on page 167, this inspector uses the callbacks listed below.

- `ScrollBar Moved`
- `ScrollBar Visibility Changed`
- `Start Edit Item`
- `End Edit Item`
- `Start Drag Item`
- `Item Dragged`
- `End Dragged Item`

IliDbTreeGadget

The `IliDbTreeGadget` gadget is used to display the data source contents as a tree structure.

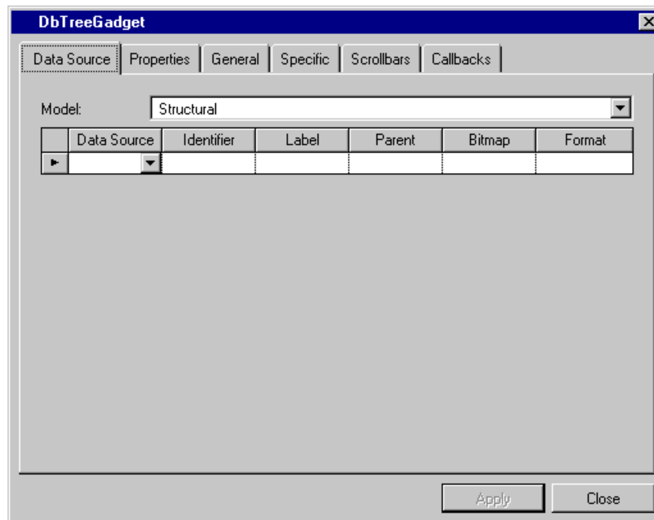


DbTreeGadget Inspector Panel

The `IliDbTreeGadget` inspector has six notebook pages:

- *Data Source Page*
- *Properties Page*
- *General Page*
- *Specific Page*
- *Scrollbars Page*
- *Callbacks Page*

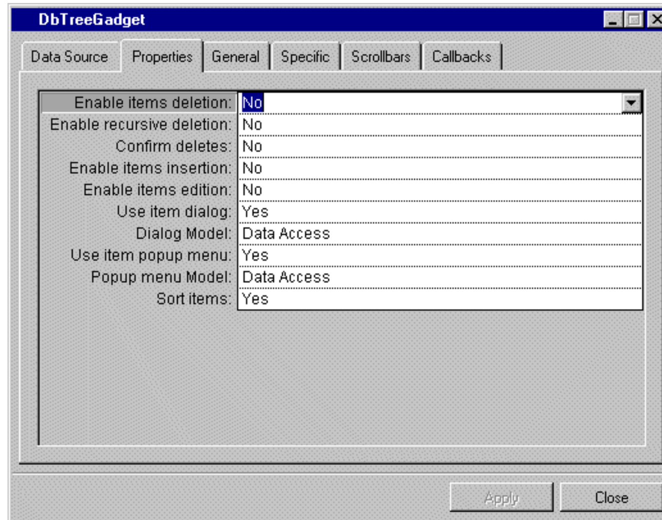
Data Source Page



Label	Description
Model	**Menu:** Structural, Recursive. **Default:** Structural. **Explanation:** Model of the tree gadget.
Data Source column	**Menu:** Names of current data sources. **Default:** No default. **Explanation:** Name of the data source to which the tree gadget is to be connected.

Label	Description
Identifier column	**Menu:** Column names of the data source selected in the DataSource field. **Default:** No default. **Explanation:** Column in the data source containing the child identifier.
Label column	**Menu:** Column names of the data source selected in the DataSource field. **Default:** No default. **Explanation:** Column in the data source containing the child label.
Parent column	**Menu:** Column names of the data source selected in the Data Source field. **Default:** No default. **Explanation:** Column in the data source containing the value for parents.
Bitmap column	**Menu:** Column names of the data source selected in the Data Source field. **Default:** No default. **Explanation:** Column containing the file name of the picture.
Format column	**Menu:** List of predefined system and user formats. **Default:** No default. **Explanation:** Format to be applied to the label in the tree gadget, if a label exists, or to the value if no label exists.

Properties Page



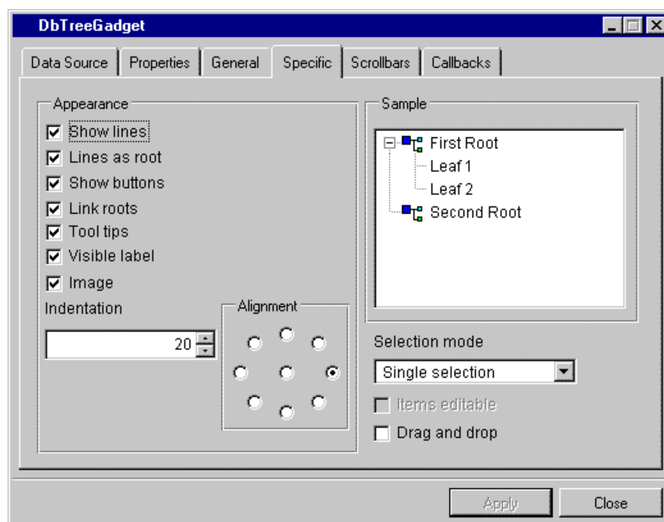
Label	Description
Enable items deletion	**Menu:** Yes, No. **Default:** No. **Explanation:** Enable or disable items deletion.
Enable recursive deletion	**Menu:** Yes, No. **Default:** No. **Explanation:** Enable or disable recursive deletion. Yes: When a parent is deleted, its child is also deleted. No: A parent will not be deleted if it has one or more children.
Confirm deletes	**Menu:** Yes, No. **Default:** No. **Explanation:** Enable or disable message to ask confirmation of a deletion.
Enable items insertion	**Menu:** Yes, No. **Default:** No. **Explanation:** Enable or disable items insertion.
Enable items edition	**Menu:** Yes, No. **Default:** No. **Explanation:** Enable or disable items edition.

Label	Description
Use item dialog	**Menu:** Yes, No. **Default:** Yes **Explanation:** Enable or disable item dialog box.
Dialog Model	**Menu:** Data Access. **Default:** Data Access. **Explanation:** Model of the dialog box.
Use item popup menu	**Menu:** Yes, No. **Default:** No. **Explanation:** Enable or disable item popup menu which appears when clicking the desired item and then the mouse right button.
Popup menu Model	**Menu:** Data Access. **Default:** Data Access. **Explanation:** Model of the popup menu.
Sort items	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Enable or disable items sorting.

General Page

For a description of this notebook page, refer to the section *General Notebook Page* on page 165.

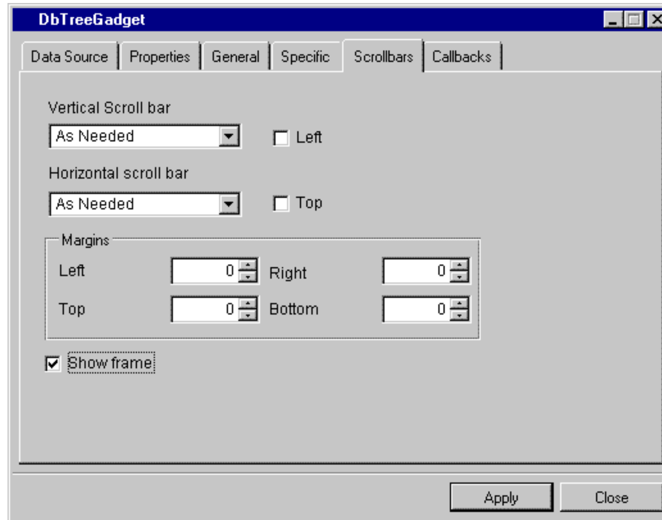
Specific Page



Label	Description
Show lines	**Check box.** **Default:** Checked. **Explanation:** Checked = Lines connecting elements in the tree are shown. Not checked = Lines are not shown.
Lines as root	**Check box.** **Default:** Checked. **Explanation:** Checked = Lines connect roots. Not checked = Lines do not connect roots.
Show button	**Check box.** **Default:** Checked. **Explanation:** Checked = Shows the buttons that indicate whether the tree is expanded. Not checked = The button is not shown.
Link roots	**Check box.** **Default:** Checked. **Explanation:** Checked = A line links the roots. Not checked = The roots are not visibly linked.
Tool tips	**Check box.** **Default:** Checked. **Explanation:** Checked = Displays tooltips if the item length is larger than the gadget width. Not checked = There are no tooltips.
Visible label	**Check box.** **Default:** Checked. **Explanation:** Checked = The labels are visible. Not checked = The labels are not visible.
Image	**Check box.** **Default:** Checked. **Explanation:** Checked = The images showing roots and nodes are visible. Not checked = The images are not visible.

Label	Description
Indentation	**Menu:** None. **Default:** 20. **Explanation:** Distance of roots and nodes from the left of the tree.
Alignment	**Menu:** None. Available positions are indicated by graphic. **Default:** Right. **Explanation:** Gives the position of the label relative to the image.
Sample	Shows how the tree looks as you change the options in the Appearance column.
Selection mode	**Menu:** Single selection, Extended selection. **Default:** Single selection. **Explanation:** Single = Only one item in the tree can be selected. Extended = More than one item can be selected.
Items editable	Not available.
Drag and drop	**Check box.** **Default:** Checked. **Explanation:** Determines whether items can be dragged and dropped.

Scrollbars Page



Label	Description
Vertical scroll bar	**Menu:** Show, Hide, As Needed. **Default:** As Needed. **Explanation:** Show = The field has a vertical scroll bar. Hide = The field does not have a vertical scroll bar. As Needed = Vertical scroll bar if needed.
Left	**Check box.** **Default:** Not checked. **Explanation:** Checked = The vertical scroll bar is on the left side of the gadget. Not checked = The vertical scroll bar is on the right side of the gadget.
Horizontal scroll bar	**Menu:** Show, Hide, As Needed. **Default:** As Needed. **Explanation:** Show = The field has a horizontal scroll bar. Hide = The field does not have a horizontal scroll bar. As Needed = Horizontal scroll bar if needed.

Label	Description
Top	**Check box.** **Default:** Not checked. **Explanation:** Checked = The horizontal scroll bar is above the gadget. Not checked = The horizontal scroll bar is below the gadget.
Margins	**Menu:** None. **Default:** 0. **Explanation:** Allows you to type the value of the left, right, top, and bottom margins.
Show Frame	**Check box.** **Default:** Checked. **Explanation:** Determines whether frames are visible.

Callbacks Page

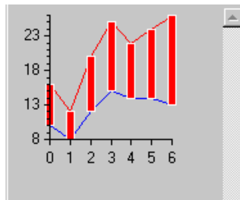
In addition to the Generic and Secondary callbacks described in the section *Callbacks Notebook Page* on page 167, this inspector uses the callbacks listed below. The callbacks `IncoherentTreeData`, `DeleteItem`, `InsertChildItem`, `InsertSiblingItem` and `EditItem` are described in the IBM ILOG Views *Data Access Reference Manual* `IliTreeGadget` section.

- `ScrollBar Moved`
- `ScrollBar Visibility Changed`
- `Item Selected`
- `Item Expanded`
- `Item Shrinked`
- `Start Edit Item`
- `Abort Edit Item`
- `End Edit Item`
- `Start Drag Item`
- `Item Dragged`
- `End Drag Item`
- `Abort Drag Item`
- `IncoherentTreeData`
- `DeleteItem`
- `InsertChildItem`

- `InsertSiblingItem`
- `EditItem`

IliChartGraphic

The `IliChartGraphic` dadget is used to display the data source as a chart graphic.



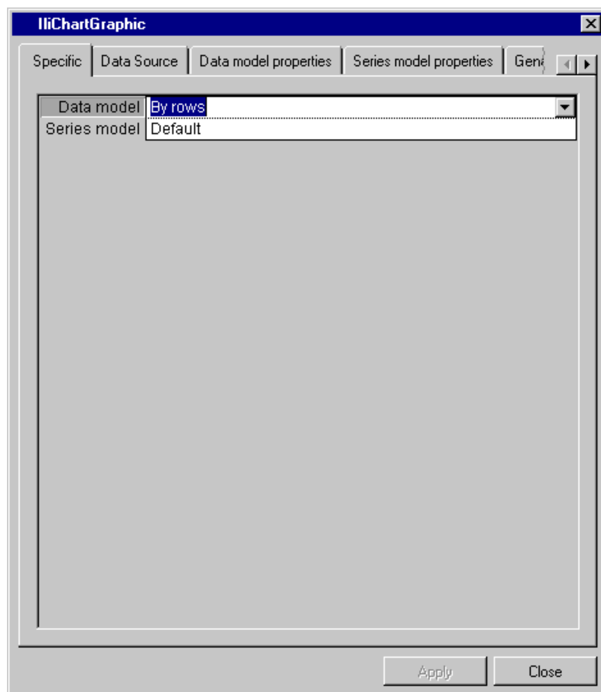
ChartGraphic Inspector Panel

The `IliChartGraphic` inspector has twelve notebook pages:

- *Specific Page*
- *Data Source Page*
- *Data Model Properties Page*
- *Series Model Properties Page*
- *General Page*
- *Callbacks Page*

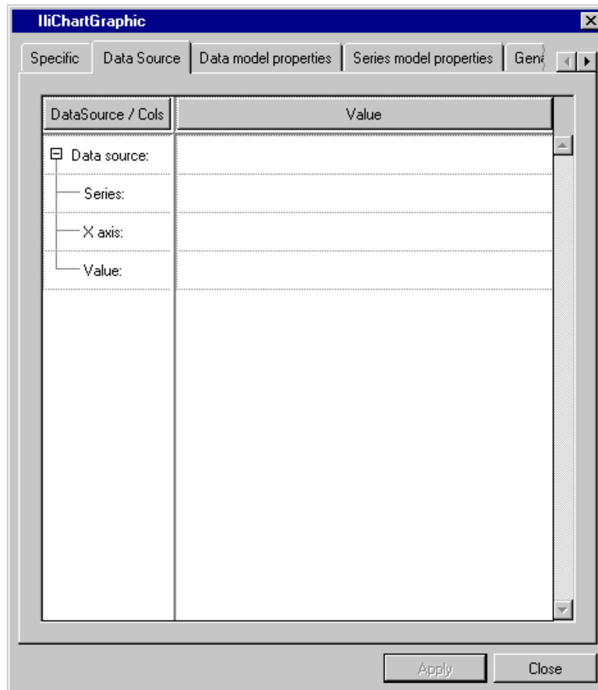
For a description of the Data Sets, Displayers, Projection, Scales, Layout, and Miscellaneous pages, refer to the section *Using the Chart Inspector in the Charts User's Manual*.

Specific Page



Label	Description
Data model	**Menu:** The current data model name. **Default:** By row. **Explanation:** Name of the data model which is used to extract the data from the data source.
Series model	**Menu:** The current series model name. **Default:** Default model. **Explanation:** Name of the series model which is used to manage the new series.

Data Source Page



This page is used to define the data sources and the columns of the data model. The contents of this page depends on the data model.

Data Model Properties Page

This page is used to edit the porperties of the current data model. If this page is empty, the model has no property.

Series Model Properties Page

This page is used to edit the properties of the current series model. if this page is empty, the model has no property.

General Page

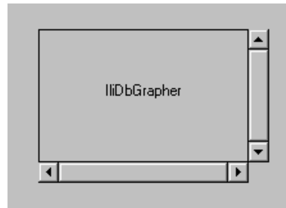
For a description of this notebook page, refer to the section *General Notebook Page* on page 165.

Callbacks Page

For a description of this notebook page, refer to the section *Callbacks Notebook Page* on page 167.

IliDbGrapher

The `IliDbGrapher` gadget is used to display the data source contents as nodes and links.

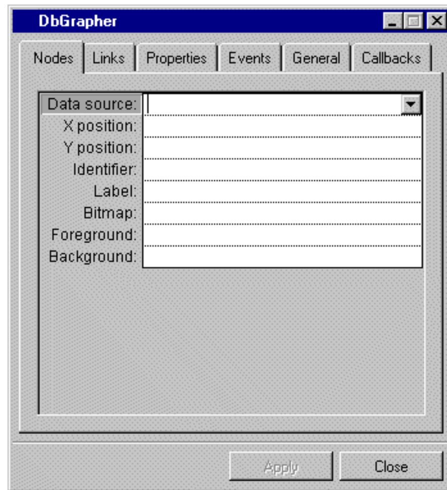


DbGrapher Inspector Panel

The `IliDbGrapher` inspector has six notebook pages:

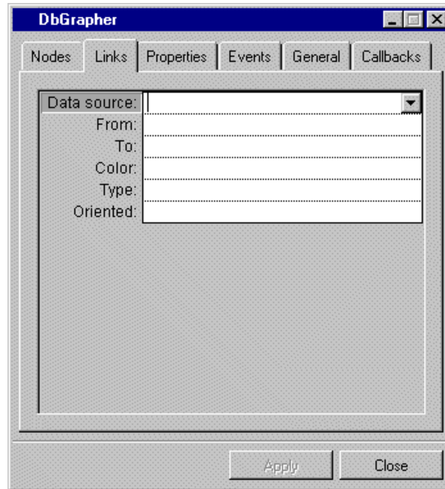
- *Nodes Page*
- *Links Page*
- *Properties Page*
- *Events Page*
- *General Page*
- *Callbacks Page*

Nodes Page



Label	Description
Data source	**Menu:** The names of current data sources. **Default:** None. **Explanation:** Name of the node data source to which the grapher is to be connected.
X position	**Menu:** Numeric column names of the selected data source. **Default:** None. **Explanation:** The column name contains the values for the x axis.
Y position	**Menu:** Numeric column names of the selected data source. **Default:** None. **Explanation:** The column name contains the values for the y axis.
Identifier	**Menu:** Column names of the selected data source. **Default:** None. **Explanation:** The column name contains the nodes identifier.
Label	**Menu:** String column names of the selected data source. **Default:** None. **Explanation:** The column name contains the nodes label.
Bitmap	**Menu:** String column names of the selected data source. **Default:** None. **Explanation:** The column name contains the nodes bitmap.
Foreground	**Menu:** String column names of the selected data source. **Default:** None. **Explanation:** The column name contains the nodes foreground color.
Background	**Menu:** String column names of the selected data source. **Default:** None. **Explanation:** The column name contains the nodes background color.

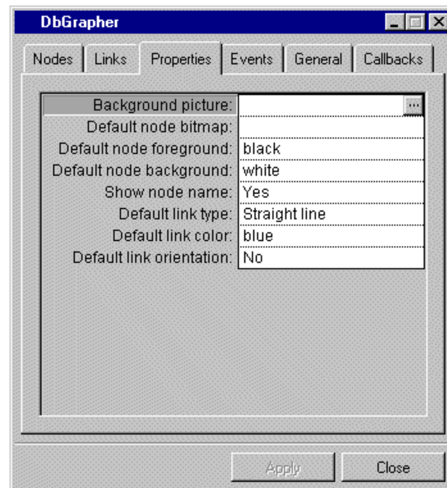
Links Page



Label	Description
Data source	**Menu:** The names of current data sources. **Default:** None. **Explanation:** Name of the data source to which the grapher is to be connected.
From	**Menu:** Column names of the selected data source. **Default:** None. **Explanation:** The column name contains the node from which the links start.
To	**Menu:** Column names of the selected data source. **Default:** None. **Explanation:** The column name contains the node to which the links go.
Color	**Menu:** String column names of the selected data source. **Default:** None. **Explanation:** The column name contains the links color.

Label	Description
Type	**Menu:** Numeric column names of the selected data source. **Default:** None. **Explanation:** The column name contains the links type (from 0 through 6).
Oriented	**Menu:** Boolean column names of the selected data source. **Default:** None. **Explanation:** The column name indicates whether the links are oriented.

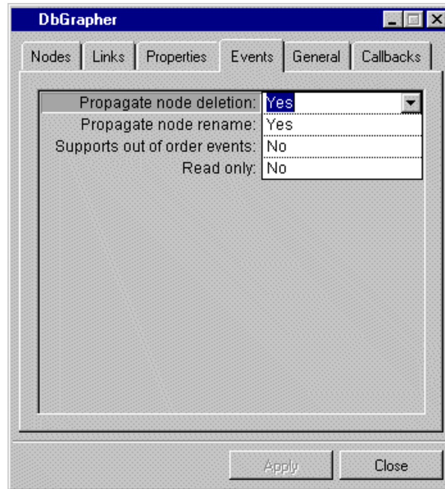
Properties Page



Label	Description
Background picture	**Menu:** None. **Button:** Click to open the *File Chooser Dialog Box*. **Default:** None. **Explanation:** The picture to be placed as the grapher background.
Default node bitmap	**Menu:** None. **Button:** Click to open the *File Chooser Dialog Box*. **Default:** None. **Explanation:** The default bitmap used for the nodes if not specified in the nodes data source.

Label	Description
Default node foreground	**Menu:** None. **Button:** Click to open the *Color Chooser Dialog Box*. **Default:** black. **Explanation:** The default foreground color used for the nodes if not specified in the nodes data source.
Default node background	**Menu:** None. **Button:** Click to open the *Color Chooser Dialog Box*. **Default:** white. **Explanation:** The default background color used for the nodes if not specified in the nodes data source.
Show node name	**Menu**: Yes, No. **Default**: Yes. **Explanation**: Enable or disable node name displaying.
Default link type	**Menu:** List of available link types. **Default:** Straight line. **Explanation:** The default link type used if not specified in the links data source.
Default link color	**Menu:** None. **Button:** Click to open the *Color Chooser Dialog Box*. **Default:** blue. **Explanation:** The default link color used if not specified in the links data source.
Default link orientation	**Menu:** Yes, No. **Default:** No. **Explanation:** Enable or disable the default link orientation. Yes: The links are oriented if not specified in the links data source. No: The links are oriented or not as specified in the links data source.

Events Page



Label	**Description**
Propagate node deletion	**Menu:** Yes, No. **Default:** Yes. **Explanation:** When a node is deleted, all its links are also deleted.
Propagate node rename	**Menu:** Yes, No. **Default:** Yes. **Explanation:** When a node is renamed, the links data source is updated with the new name.
Supports out of order events	**Menu:** Yes, No. **Default:** No. **Explanation:** If a link is created before its node(s), it will be displayed on the grapher as soon as the node(s) is (are) created.
Read only	**Menu: Yes, No.** **Default:** No. **Explanation:** No = The grapher can be edited: the nodes can be moved. Yes = The grapher cannot be edited: the nodes cannot be moved.

General Page

For a description of this notebook page, refer to the section *General Notebook Page* on page 165.

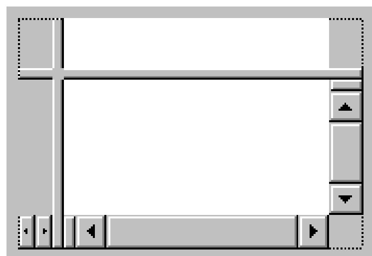
Callbacks Page

In addition to the Generic and Secondary callbacks described in the section *Callbacks Notebook Page* on page 167, this inspector uses the callbacks listed below, which are described in the `IliDbGrapher` section of the IBM ILOG Views *Data Access Reference Manual*.

- `NodeMoved`
- `NodeDoubleClicked`
- `LinkDoubleClicked`
- `PrepareDeleteObject`

IliDbGantt

The `IliDbGantt` gadget is used for defining a Gantt chart connected to various data sources.

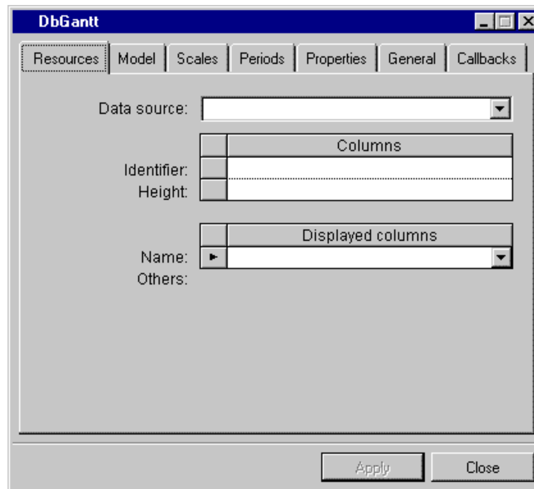


DbGantt Inspector Panel

The `IliDbGantt` inspector has eleven notebook pages:

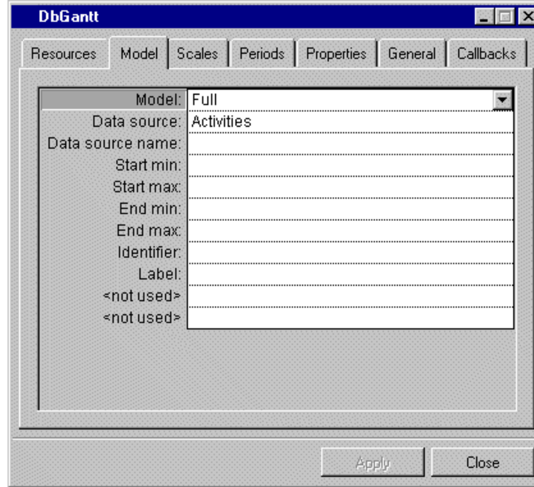
- *Resources Page*
- *Model Page*
- *Scales Page*
- *Periods Page*
- *Properties Page*
- *General Page*
- *Callbacks Page*

Resources Page



Label	Description
Data source	**Menu:** The names of current data sources. **Default:** None. **Explanation:** Name of the resources data source file to which the gantt is connected.
Identifier column	**Menu:** Column names of the selected data source. **Default:** None. **Explanation:** The column name contains the resource identifier.
Height column	**Menu:** Integer column names of the selected data source. **Default:** None. **Explanation:** The column name contains the resource height.
Name displayed column	**Menu:** Column names of the selected data source. **Default:** None. **Explanation:** The column name contains the first name to be displayed.
Others displayed column	**Menu:** Column names of the selected data source. **Default:** None. **Explanation:** Lets you specify other column names to be displayed.

Model Page



Label	Description
Model	**Menu**: Full, light. **Default**: Full. **Explanation**: The model name of the data model.
Data source	**Menu:** Activities, Constraints, Precedences, Breaks, Load. Activities is only available if the selected model is `Full`. **Default:** Activities. **Explanation:** The datasource usages.

Activities Data Source Properties

Label	Description
Data source name	**Menu**: The name of the current data sources. **Default**: None. **Explanation**: Name of the activities data source to which the gantt is to be connected.
Start min	**Menu:** Column names of the selected data source. **Default:** None. **Explanation:** The column name contains the activity start minimum value.

Label	Description
Start max	**Menu:** Column names of the selected data source. **Default:** None. **Explanation:** The column name contains the activity start maximum value.
End min	**Menu:** Column names of the selected data source. **Default:** None. **Explanation:** The column name contains the activity end minimum value.
End max	**Menu:** Column names of the selected data source. **Default:** None. **Explanation:** The column name contains the activity end maximum value.
Identifier	**Menu:** Column names of the selected data source. **Default:** None. **Explanation:** The column name contains the activity identifier.
Label	**Menu:** Column names of the selected data source. **Default:** None. **Explanation:** The column name contains the activity label.

Constraints Data Source Properties

Full Model

DbGantt

Resources | Model | Scales | Periods | Properties | General | Callbacks

Model: Full
Data source: Constraints
Data source name:
Identifier:
Resource identifier:
Activity identifier:
Capacity:
Foreground:
Background:
<not used>
<not used>

Apply | Close

Light Model

DbGantt

Resources | Model | Scales | Periods | Properties | General | Callbacks

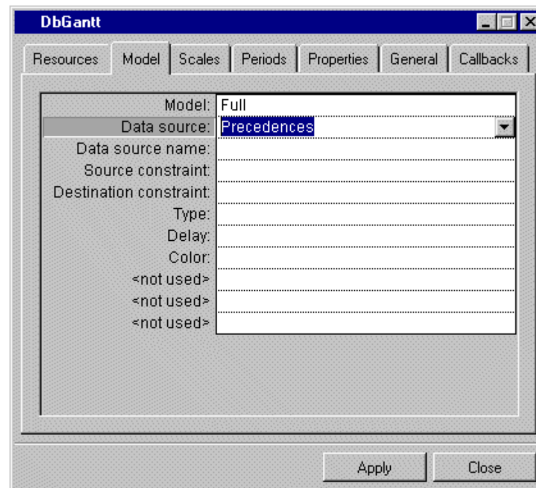
Model: Light
Data source: Constraints
Data source name:
Identifier:
Resource identifier:
Label:
Capacity:
Foreground:
Background:
Start:
End:

Apply | Close

Label	Description
Data source name	**Menu**: The name of the current data sources. **Default**: None. **Explanation**: Name of the constraints data source to which the gantt is to be connected.
Identifier	**Menu:** Column names of the selected data source. **Default**: None. **Explanation:** The column name contains the constraint.
Resource identifier	**Menu**: Column names of the selected data source. **Default**: None. **Explanation**: The column name contains the resource identifier to which the constraint is linked.
Label	Only available if the selected mode is `Light`. **Menu**: Column names of the selected data source. **Default**: None. **Explanation**: The column name contains the constraint label.
Activity identifier	Only available if the selected model is `Full`. **Menu**: Column names of the selected data source. **Default**: None. **Explanation**: The column name contains the activity identifier to which the constraint is linked.
Capacity	**Menu**: Integer column names of the selected data source. **Default**: None. **Explanation**: The column name contains the constraint capacity.
Foreground	**Menu**: String column names of the selected data source. **Default**: None. **Explanation**: The column name contains the foreground color for the costraint.
Background	**Menu**: String column names of the selected data source. **Default**: None. **Explanation**: The column name contains the background color for the costraint.

Label	Description
Start	Only available if the selected model is `Light`. **Menu**: Integer column names of the selected data source. **Default**: None. **Explanation**: The column name contains the constraint start value.
End	Only available is the selected model is `Light`. **Menu**: Integer column names of the selected data source. **Default**: None. **Explanation**: Name of the precedences data source to which the gantt is to be connected.

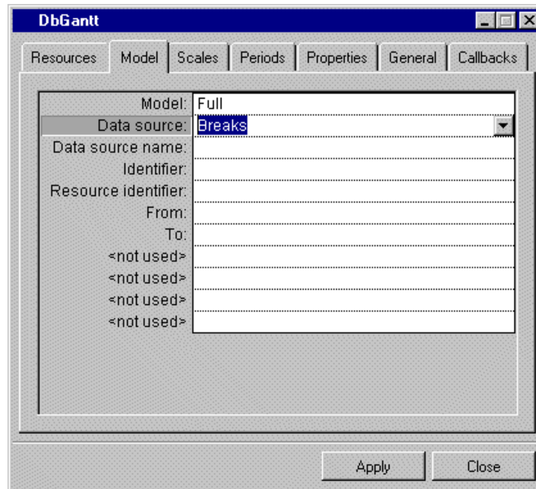
Precedences Data Source Properties



Label	Description
Data source name	**Menu**: The names of the current data sources. **Default**: None. **Explanation**: Name of the precedences data source to which the gantt is to be connected.
Source constraint	**Menu**: Column names of the selected data source. **Default**: None. **Explanation**: The column name contains the constraint identifier from which the precedences start.

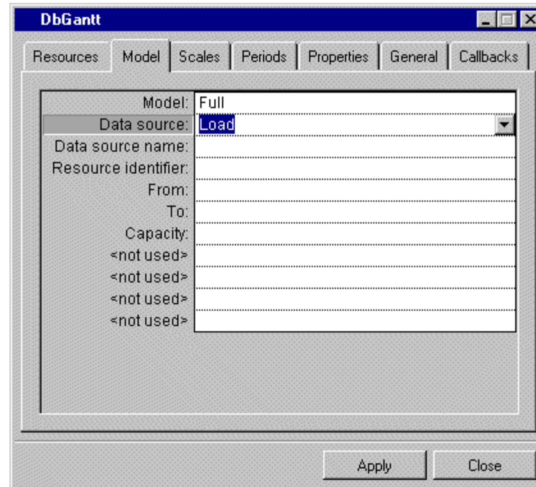
Label	Description
Destination constraint	**Menu**: Column names of the selected data source. **Default**: None. **Explanation**: The column name contains the constraint identifier to which the precedences go.
Type	**Menu**: Integer column names of the selected data source. **Default**: None **Explanation**: The column name contains the precedence type.
Delay	**Menu**: Integer column names of the selected data source. **Default**: None. **Explanation**: The column name contains the precedence delay.
Color	**Menu**: String column names of the selected data source. **Default**: None. **Explanation**: The column name contains the color of the precedence.

Breaks Data Source Properties



Label	Description
Data source name	**Menu**: The names of the current data sources. **Default**: None. **Explanation**: Name of the breaks data source to which the gantt is to be connected.
Identifier	**Menu**: Column names of the selected data source. **Default**: None. **Explanation**: The column name contains the break identifier.
Resource identifier	**Menu**: Column names of the selected data source. **Default**: None. **Explanation**: The column name contains the resource identifier to which the break is linked.
From	**Menu**: Integer column names of the selected data source. **Default**: None. **Explanation**: The column name contains the value where the break starts, in relation to the horizontal scale.
To	**Menu**: Integer column names of the selected data source. **Default**: None. **Explanation**: The column name contains the value where the break stops, in relation to the horizontal scale.

Load Data Source Properties



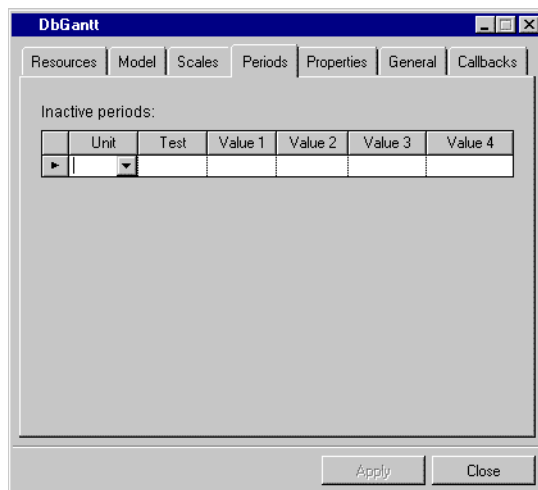
Label	Description
Data source name	**Menu**: The names of the current data sources. **Default**: None. **Explanation**: Name of the load data source to which the gantt is to be connected.
Resource identifier	**Menu**: Column names of the selected data source. **Default**: None. **Explanation**: The column name contains the resource identifier to which the work load curve is linked.
From	**Menu**: Integer column names of the selected data source. **Default**: None. **Explanation**: The column name contains the value where the work load curve starts.
To	**Menu**: Integer column names of the selected data source. **Default**: None. **Explanation**: The column name contains the value where the work load curve stops.
Capacity	**Menu**: Integer column names of the selected data source. **Default**: None. **Explanation**: The column name contains the work load curve capacity.

Scales Page

Label	Description
Reference year	**Menu:** None. **Default:** 1998. **Explanation:** Reference year for the Gantt chart.
Reference month	**Menu:** The months of the year. **Default:** January. **Explanation:** Reference month for the Gantt chart.
Display full name	**Menu: Yes, No.** **Default:** Yes. **Explanation:** Yes = Full display of the selected reference time period. No = Short display of the selected reference time period.

Label	Description
Type	**Menu:** Various date formats, for example: Date with week days Date by hour Week days by 30 minutes Hours and minutes Numeric. **Default:** Date. **Explanation:** Type of scale.
Time unit	**Menu**: Various units of time in seconds, minutes, hours, day or numeric. **Default**: Day. **Explanation**: Unit of the time scale.

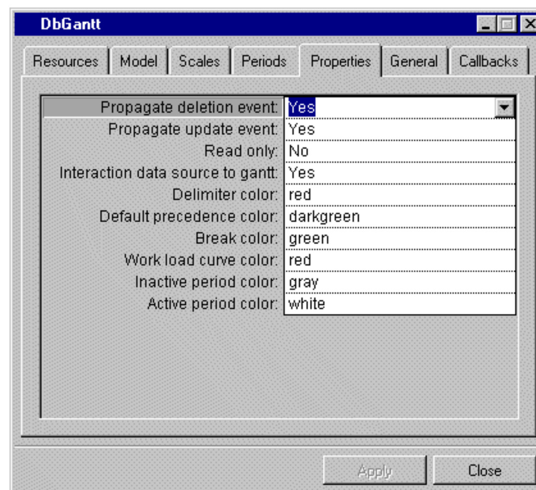
Periods Page



Label	Description
Unit column	**Menu:** Month, Day, Weekday, Hour, Minute, Second, Month and day, Hour and minute. **Default:** None. **Explanation:** Time unit for inactive period.
Test column	**Menu:** Equal, Not equal, Less, Greater, Include, Exclude. **Default:** None. **Explanation:** Test for inactive period.

Label	Description
Value 1 column	**Menu:** Depends on the selected unit. For example, if the selected unit is Day, the menu is Monday to Sunday; if the unit is Hour, the menu is 0 to 23. **Default:** None. **Explanation:** Value of the inactive period.
Value 2, Value 3, Value 4 columns	**Menu:** Depends on the selected unit and test. **Default:** None. **Explanation:** Value of the inactive period. Ignore = There is no use for the test.

Properties Page



Label	Description
Propagate deletion event	**Menu:** Yes, No. **Default:** Yes **Explanation:** A deletion event in a data source deletes corresponding events in other data sources.
Propagate update event	**Menu:** Yes, No. **Default:** Yes. **Explanation:** An update event in a data source updates corresponding events in other data sources.

Label	Description
Read only	**Menu:** Yes, No. **Default:** No. **Explanation:** Yes: the Gantt chart cannot be edited: the constraints cannot be moved. No: The Gantt chart can be edited: the constraints can be moved.
Interaction data source to gantt	**Menu:** Yes, No. **Default:** Yes. **Explanation:** Gantt chart is updated or not when the data source is updated or modified.
Delimiter color	**Menu**: None. Click the button to open the Color Selector Dialog Box. **Default**: red. **Explanation**: Delimiter color of the constraint.
Default precedence color	**Menu**: None. Click the button to open the Color Selector Dialog Box. **Default**: darkgreen. **Explanation**: Default color used for precedences if not specified in the precedences data source.
Break color	**Menu**: None. Click the button to open the Color Selector Dialog Box. **Default**: green. **Explanation**: The color used for the breaks.
Work load curve color	**Menu**: None. Click the button to open the Color Selector Dialog Box. **Default**: red. **Explanation**: The color used for the work load curves.
Inactive period color	**Menu**: None. Click the button to open the Color Selector Dialog Box. **Default**: gray. **Explanation**: Color for the inactive periods in the Gantt chart.
Active period color	**Menu**: None. Click the button to open the Color Selector Dialog Box. **Default**: white. **Explanation**: Color for the active periods in the Gantt chart.

General Page

For a description of this notebook page, refer to the section *General Notebook Page* on page 165.

Callbacks Page

In addition to the callbacks described in the section *Callbacks Notebook Page* on page 167, this inspector uses the callbacks listed below, which are described in the IBM ILOG Views *Data Access Reference Manual* `IliDbGantt` section.

- `IsActivePeriod`
- `ConstraintDoubleClicked`
- `PrecedenceDoubleClicked`
- `ScaleNumericLabel`

A P P E N D I X

A

Utility Classes

This appendix describes the following utility classes of Data Access: `IliString`, `IliDecimal`, `IliDate`, `IliFormat`, and `IliInputMask`.

You can find information on the following topics:

- *The IliString Class*
- *The IliDecimal Class*
- *The IliDate Class*
- *The IliFormat Class*
- *The IliInputMask Class*

The IliString Class

The `IliString` class defines objects that manage a character string.

```
IliString str;
// Assign a new value to the string.
str = "Hello ";

// Append to it.
str << "World !";
```

```
// Convert it to a pointer to characters.
const char* ptr = str;

// Query its length.
IlUInt len = str.length();
```

This class manages transparently the memory required to store its character string value. It is, therefore, used when buffering is required.

The following member function makes use of the `IliString` class:

```
void IliValue::format(IliString& dest,
                         const IliFormat& fmt) const;
```

The IliDecimal Class

The `IliDecimal` class is used to hold floating point numbers with up to 38 digits of precision represented internally in base 10. This contrasts with the C++ double type that represents floating point numbers in base 2 and has a machine-dependent precision.

The following code extract shows how this class is used:

```
IliDecimal dec = someField->f_getValue().asDecimal();
someField->f_setValue(dec + IliDecimal(0.5));
```

The IliDate Class

The `IliDate` class is used to hold date and time information.

```
IliDate dt;

// Initialize a date object.
dt.setYear(1998);
dt.setMonth(6);
dt.setMonthDay(10);

// Convert the date into an IliValue object.
IliValue val = dt;

// Assign it to a data source aware gadget.
someField->f_setValue(val);
```

In this example, the date is created, converted in an `IlvValue` object, assigned to a data-source-aware gadget and displayed.

The IliFormat Class

The `IliFormat` class is used to format values into character strings using specific rules. A format can be user-defined or predefined. Once an `IliFormat` has been created, it can be named and referenced whenever required. This can be done using the aliasing mechanism provided in the `IliFormat` class.

You can specify formats for numbers, dates, or character strings.

```
// Create a predefined format named "MyFormat".
IliFormat::AddAlias("MyFormat",
                      "#,##0.00 FF.",
                      IliNumberFormatType);

IliFormat fmt("MyFormat");
IliValue val = someField->f_getValue();
const char* txt = val.getFormatted(fmt);
IlvPrint("Here is the formatted value : %s", txt);
```

The `IliFormat` class controls the format specification and uses some global settings. These settings include properties such as the character that is used to represent a decimal point. Consequently, the `IliFormat` class contains a set of static member functions that can be used to query and set the global settings.

```
IliFormat::SetDecimalPoint(',');
IliFormat::SetThousandsSeparator(' ');
IliFormat::SetCurrencySymbol("F.");
...
```

For more information on the syntax used to specify a format, see Appendix B, *Format Syntax*.

It is also possible to code formats in C++ by subclassing the `IliFormatIpl` class as the following code extract shows:

```
#include <ctype.h>
#include <ilviews/dataccess/format.h>

const char* MyFormatAlias = "FancyFormat";
const char* MyFormatName = "FancyFormat";

// This format reverses the case of alphabetic
// characters: lowercase are displayed uppercase and
// vice-versa.

class FancyFormat
  : public IliFormatIpl
{
public:
  FancyFormat()
  : IliFormatIpl(MyFormatName)
  {}

  virtual IliFormatType getType() const {
    return IliStringFormatType;
  }

  virtual void formatString(IliString& dest,
                            const char* src) const {
    if (src) {
      if (isEditModeOn())
        dest << src;
      else
        while (*src) {
          if (isalpha(*src)) {
            if (isupper(*src))
            dest << (char)tolower(*src);
          else
            dest << (char)toupper(*src);
        }
        else
          dest << *src;
        ++src;
      }
    }
  }
};

int main() {
  ...
  IliFormatIpl::AddCustomFormat(new FancyFormat());
  IliFormat::AddAlias(MyFormatAlias,
                MyFormatName,
                IliStringFormatType);
  ...
}
```

The IliInputMask Class

An input mask is similar to a format except that it allows you to control how values are entered by the end user in addition to the formatting.

The following is an example of an input mask:

```
00"-"00"-"00
```

This input mask lets the end user type six digits and nothing else. It also separates each pair of digits from the next by displaying a '-' character between them. This character is only used for display and is not part of the value.

See Appendix C, *Mask Syntax* for more information on how masks can be specified.

There can be cases where the mask specification language described in Appendix C is not sufficient for a given task. Fortunately, it is possible to define input masks in C++ by subclassing class `IliInputMaskIpl`.

Data Access provides the `IliDateType` (manages date and time) and the `IliTimeType` (manages time only) data types. In some circumstances, it may be necessary to use the `IliDateType` data type when only the time part needs to be managed. This happens, for instance, when using a database system that exclusively supports a date-time type. Since such a database system does not support a time-only type, it is necessary to use `IliDateType` instead of `IliTimeType`, even thought it is expected that the date part of values will be constrained to be some constant date.

The following code samples create an input mask that allows editing of the time part of a date-time value. The date part is constrained to be some constant date.

First, a subclass of `IliInputMaskIpl` must be defined:

```
#include <wctype.h>
#include <ilviews/dataccess/inpmask.h>

const char* MyMaskName = "MyMask";

// Use "30 Dec 1899" with an Access database.
const char* DatePart = "1-1-1901 ";

class MyMask : public IliInputMaskIpl
{
public:
  MyMask()
   : IliInputMaskIpl(MyMaskName),
    _format("Time")
  {
   setMaxCharMask(8);
  }

  virtual IlvBoolean unFormat(IliString& dest,
                              const char* src) const {
    if (src && *src)
```

```
      dest << DatePart << src;
    return IlvTrue;
  }

  IlvBoolean isValidChar(IlInt pos,
                         wchar_t c,
                         IlvBoolean editMode = IlTrue) const {
    if (isFixChar(pos))
      return (c == L':');
    return iswdigit(c);
  }

  virtual IlBoolean isFixChar(IlInt pos) const {
    return (pos == 2 || pos == 5);
  }

  virtual wchar_t filterChar(IlInt pos, wchar_t c) {
    return (isFixChar(pos) ? L':' : c);
  }

  virtual const IliFormat& getValueFormat() const {
    return _format;
  }

  IliFormat _format;
};
```

Then, at initialization time, the following code should be executed:

```
// Register the mask.
IliInputMaskIpl::AddCustomMask(new MyMask);
IliInputMask::AddAlias("MyMask", "MyMask");
```

Here is how the mask could be used:

```
//Use the mask.
IliDbField* fld;
...
IliInputMask m("MyMask");
fld->f_setMask(m);
```

B

Format Syntax

This appendix contains the symbols and formats you can use to create application-wide named formats and local formats for a particular field. A format specification controls the way a value will be formatted for display. There are three types of formats:

- String formats
- Number formats
- Date formats

This appendix describes the syntax of the format specifications in Data Access. For each type of format specification, there is a set of special symbols, each having a specific meaning. You can find information on the following topics:

- *String Formats*
- *Number Formats*
- *Date Formats*
- *Literal Characters*

String Formats

String formats apply to the formatting of text.

Symbols

You can use the following symbols to specify a string format:

! Formatting must proceed from right to left.

< Characters following the symbol will be converted to lowercase.

> Characters following the symbol will be converted to uppercase.

@ Placeholder for a mandatory character.

& Placeholder for an optional character.

Normally, character string formatting proceeds by scanning the character string value and the format specification from left to right. However, if the format specification contains a "`!`" symbol, then scanning of both the character string value and the format specification proceeds from right to left.

Examples

Value	Format	Result
`forms`	`>`	`FORMS`
`FormS`	`<`	`forms`
`forms`	`<@>`	`fORMS`
`forms`	`<@@>`	`foRMS`
`forms`	`!>@<`	`FORMs`
`forms`	`!>@<@@@>`	`FoRMS`
`forms`	`&&&&&&"data"`	`formsdata`
`forms`	`@@@@@@"data"`	`forms data`

`@` symbols are replaced by spaces when there is no corresponding character in the string value.

Number Formats

Number formats apply to the formatting of numbers, including currency amounts.

Symbols

You can use the following symbols to specify a number format:

- **0** Placeholder for a mandatory digit.
- **#** Placeholder for an optional digit.
- **.** Placeholder for the decimal point.
- **,** Placeholder for the thousands separator.
- **%** Formats the value as a percentage.
- **E** Placeholder for the exponent displayed in uppercase.
- **e** Placeholder for the exponent displayed in lowercase.

The exponent symbols “E” and “e” can be followed by a “+” or “-” sign. A “-” sign means that the exponent sign must be displayed only if it is negative. A “+” sign or no sign means that the sign of the exponent is always displayed.

When a “%” symbol appears in the format specification, the numeric value to be formatted is multiplied by 100 before formatting and a “%” symbol appears in the result.

“0” symbols are replaced by zeros when there is no corresponding digit in the number value.

Examples

Value	Format	Result
1234.567	#,##0.00	1 234.57
1234.567	#,##0.0#	1 234.57
1234.5	#,##0.00	1 234.50
1234.5	#,##0.0	1 234.5
1.5	0,000.00	0 001.50
1234	0.00 E+00	1.23 E+03
1234	0.00 E-00	1.23 E03
1234	0.00 E-##	1.23 E3
0.5432	#.# %	54.3 %

The characters used to represent the decimal point and the thousands separator in the formatted result depend on application settings that can be changed. Typically, they depend on the country where the application is used. These settings do not affect the symbols used as placeholders for the decimal point and for the thousands separator in format specifications. You should thus always use the placeholders listed above for these values as only the output depends on the application settings.

Note that the maximum precision is 15 digits for double values and 7 for float values.

Date Formats

Date formats refer to the formatting of days, dates, and times.

Symbols

You can use the following symbols to specify a date format:

Symbol	Description
/	Placeholder for date separator.
:	Placeholder for time separator.
<	Characters following the symbol will be converted to lowercase.
>	Characters following the symbol will be converted to uppercase.
d	Placeholder for day of month (1-31).
dd	Placeholder for day of month (01-31).
ddd	Placeholder for day of week (Sun-Sat). Depends on the language that has been set for the application. See the `IlvDisplay` class.
dddd	Placeholder for day of week (Sunday-Saturday). Depends on the language that has been set for the application. See the `IlvDisplay` class.
ddddd	Placeholder for full date (ex: 8/3/96. Depends on the global settings of your application. See the `IliFormat` class.
dddddd	Placeholder for full date (ex: 03 August 1996). Depends on the language and the global settings that have been set for the application. See the `IlvDisplay` and `IliFormat` classes.
w	Placeholder for day of week (1-7).
ww	Placeholder for week of year (1-53).
m	Placeholder for month (1-12).
mm	Placeholder for month (01-12).
mmm	Placeholder for month (Jan-Dec). Depends on the language that has been set for the application. See the `IlvDisplay` class.
mmmm	Placeholder for month (January-December). Depends on the language that has been set for the application. See the `IlvDisplay` class.

q Placeholder for quarter (1-4).

y Placeholder for year day (1-366).

yy Placeholder for year (00-99).

yyyy Placeholder for year (1970-2099).

h Placeholder for hour (0-23).

hh Placeholder for hour (00-23).

H Placeholder for hour (0-11).

HH Placeholder for hour (00-11).

p Placeholder for AM or PM.

n Placeholder for minutes (0-59).

nn Placeholder for minutes (00-59).

s Placeholder for seconds (0-59).

ss Placeholder for seconds (00-59).

ttttt Placeholder for full time (ex: 05:32:12).
Depends on the global settings of your application.
See the `IliFormat` class.

Examples

Value	Format	Result
`12 jan 96`	`d/m/yy`	`12/1/96`
`12 jan 96`	`d mmmm yyyy`	`12 January 1996`
`12 jan 96`	`q`	`1`

The placeholders for the date and time separators are formatted according to application settings that can vary (typically, depending on the country).

Also, two of the format specifications depend on application settings that control if the date should be displayed before or after the month. For example:

Value	Format	Application Properties	Result
`12 jan 96`	`dddddd`	MDY, English language	`January 12 1996`
`12 jan 96`	`dddddd`	DMY, French language	`12 janvier 1996`

Literal Characters

In any format specification, you can include literal characters. They will be output “as is” when formatting a value.

Symbols

A literal character is specified by one of the following methods:

- **`\c`** Prefix the character with a back slash.
- **`“abc”`** Enclose a string of characters in double quotes.
- Any character that is not a special symbol or cannot be part of one, is considered as being a literal character.

Examples

Value	Format	Result
1234.5	#,##0.0# Frs	1 234.5 Frs
1234.5	#,##0.0# “Frs”	1 234.5 Frs
1234.5	#,##0.0# \F\r\s	1 234.5 Frs
12 jul 96	“Quarter” q	Quarter 3
forms	ILOG >@<	ILOG Forms

C

Mask Syntax

This appendix contains the symbols and formats that you can use to create application-wide named masks called input masks and local input masks for a particular field. Masks control user input. Format specification controls how a value is formatted for display and a mask sets a style to input values. This appendix talks about the different types of masks used to input data.

Masks are used to provide a format to input data. A mask is defined by a character string and uses two principles. First, there is no format and the mask automatically formats the data displayed. Second, missing characters are replaced by default characters. There are predefined masks for date and time.

This appendix describes the syntax of format specifications and their specific meaning.

You can find information on the following topics:

- *Placeholders*
- *Predefined Masks*

Placeholders

Used in the Edit mode, placeholders replace all missing characters by default characters. There are predefined masks for the date and time. The following symbols are the placeholders that you can use for alphanumeric formatting.

0	Placeholder for a mandatory digit.
#	Placeholder for an optional digit.
S	Placeholder for a mandatory digit or sign.
s	Placeholder for an optional digit or sign.
L	Placeholder for a mandatory letter.
l	Placeholder for an optional letter.
U	Placeholder for a mandatory uppercase letter.
u	Placeholder for an optional uppercase letter.
M	Placeholder for a mandatory lowercase letter.
m	Placeholder for an optional lowercase letter.
A	Placeholder for a mandatory digit or letter.
a	Placeholder for an optional digit or letter.
P	Placeholder for a mandatory digit or uppercase letter.
p	Placeholder for an optional digit or uppercase letter.
W	Placeholder for a mandatory digit or lowercase letter.
w	Placeholder for an optional digit or lowercase letter.
X	Placeholder for a mandatory digit or letter from a to f (or A to F)
x	Placeholder for an optional digit or letter from a to f (or A to F)
C	Placeholder for a mandatory any character.
c	Placeholder for an optional any character.
.	Placeholder for the decimal point.
,	Placeholder for the thousands separator.
E,e	Placeholder for the exponent separator.
[xy	Placeholder for a mandatory digit from x to y included.
{xxxx}	Placeholder for a mandatory character from a set of characters. The list of characters is placed between two braces { and }. If you put “a” in the list and if you enter “A”, there is an automatic conversion to “a”.
&xxxx&	Placeholder for an optional character from a set of characters. The list is placed between two “&”.

(@#	Placeholder for a mandatory letter from @ to # included. If, the case of @ and # is different so the case is ignored or else the case is active.
/	Placeholder for date separator.
:	Placeholder for time separator.
****	The next character included in the mask for display and value.
!	Formatting must proceed from right to left.

“all characters between” are only displayed (not present in the value).

If a format has an exponent, digits, and number separators with a decimal separator (and only one) then the format is a float format. The float format ignores placeholder <!>. The integer part reads from right to left while the decimal part reads from left to right. The other characters which are not between double-quotes, are displayed and included in the value.

Default Value

The default value is `space` for characters that are not mandatory characters.

Predefined Masks

Predefined masks are used to set format specifications for date and time. You can define your predefined masks in the Masks section of the Application Properties panel of IBM® ILOG® Views Studio. The following table defines and names the format showing how they are displayed. In the mask format, characters representing the decimal point depend on application settings. The first column shows the decimal formats, the second shows the value, and the final column shows the format applied to the values.

Mask	Value	Display
000.0	123.8	123.8
“(“00”)”000.0##	12345.789	(12)345.789
“(“00”)”000.0##	12345.78	(12)345.78
“(“00”)”000.0##	12345	12345
“(“00”)”000.0##	12345.9999	12345.9999

The following table provides the name of the date mask in the first column and shows how the dates will be displayed in the second column.

Name	Display
`Date, mmm/dd/yyyy`	`mmm/dd/yyyy`
`Date, mmm/dd/yy`	`mmm/dd/yy`
`Date, mm/dd/yy`	`mm/dd/yy`
`Date, mm/dd/yyyy`	`mm/dd/yyyy`
`Time (This mask works only if the data type is time.)`	`hh:mm:ss`

D

Error Messages

This appendix contains a list of error messages that Data Access generates. In the following table, you will find the code error, the names of the arguments, and their type.

Code Error	Argument 1(Type)	Argument 2 (Type)
`Ili_UndefinedError`	none	none
`Ili_UnexpectedError`	none	none
`Ili_IncorrectValueError`	none	none
`Ili_TableIsReadOnlyError`	none	none
`Ili_DuplicateRowError`	none	none
`Ili_NullColumnError`	column name (s)	none
`Ili_ColumnLengthError`	column name (s)	length max (i)
`Ili_InvalidRowNumber`	row number (i)	none
`Ili_InvalidTableBuffer`	none	none
`Ili_IncorrectTableAlias`	none	none
`Ili_InvalidParameterType`	none	none

Code Error	Argument 1(Type)	Argument 2 (Type)
`Ili_UndefinedQuery`	none	none
`Ili_RowsCountLimitExceeded`	limit (i)	none
`Ili_ColumnTypeMismatch`	column name (s)	none
`Ili_ColumnNotInQuery`	column name (s)	none
`Ili_FetchPendingError`	insert or delete (s)	none
`Ili_SQLRowNotFoundError`	none	none
`Ili_SQLRowChangedSinceFetch`	none	none
`Ili_UndefinedSQLSessionError`	none	none
`Ili_CannotAllocateSQLCursorError`	none	none
`Ili_GroupedQueryIsReadOnlyError`	none	none
`Ili_NotAllColumnsAreUpdatableError`	table name (s)	column name (s)
`Ili_DatabaseRowIsNotUniqueUpdateError`	none	none
`Ili_DatabaseRowIsNotUniqueSelectError`	none	none
`Ili_AlreadyConnectedError`	none	none
`Ili_NotConnectedError`	none	none
`Ili_TableWithoutColumnsError`	none	none
`Ili_TableWithoutNameError`	none	none
`Ili_ColumnWithoutNameError`	none	none
`Ili_ColumnWithoutTypeError`	column name (s)	none

To understand the error message contained in Arguments 1 and/or 2, the user must type `%s` ("s" for string type) or `%ld` (i for an integer type) in the message text.

Index

A

Activities page
  DbGantt inspector panel **278**
addCallback member function
  IlvGraphic class **62**
addErrorMessage member function
  IliDataSource class **63**, **67**
addErrorSink member function
  IliDataSource class **66**
  IliTable class **42**
Allow insert checkbox
  SQL Data Source inspector panel **174**
applyQueryMode member function
  IliDataSource class **138**
asString member function
  IliValue class **94**
asynchronous mode **134**

B

bindToDataSource member function
  IliTableGadget class **73**
buttons
  Memory Data Source inspector panel **200**
  SQL Data Source inspector panel **183**

C

callbacks
  overview **24**
  CancelEdits **63**, **64**
  DeleteRow **66**
  DrawCell **74**
  EnterInsertMode **63**
  EnterRow **62**
  EnterUpdateMode **62**
  getCellPalette **74**
  global **88**
  predefined **90**, **143**
  PrepareDeleteRow **66**
  PrepareInsert **64**
  PrepareUpdate **62**
  primary callback **24**
  QuitInsertMode **64**
  QuitRow **62**
  QuitUpdateMode **62**
  SQL Data Source inspector panel **183**
  Validate **62**
  ValidateRow **63**
  validation callbacks **67**
Callbacks notebook page **167**
Callbacks page
  DbField inspector panel **213**
  DbGantt inspector panel **289**
  DbGrapher inspector panel **276**
  DbNavigator inspector panel **237**
  DbOptionMenu inspector panel **250**
  DbPicture inspector panel **247**
  DbStringList inspector panel **256**

DbText inspector panel **226**
DbTimer inspector panel **238**
DbToggle inspector panel **230**
DbTreeGadget inspector panel **264**
EntryField inspector panel **216**
HTMLReporter inspector panel **242**
Memory Data Source inspector panel **199**
TableComboBox inspector panel **223**
TableGadget inspector panel **208**
ToggleSelector inspector panel **234**
`cancelQueryMode` member function
`IliDataSource` class **138**
`clearRows` member function
`IliTable` class **41**
Color Selector dialog box **169**
column properties
alignment **36**
completion **37**
constrained **37**
datatype **35**
default value **36**
display column **37**
display width **36**
foreign data source name **36**
foreign table **36**
format **36**
index **35**
label **36**
maximum length **35**
name **35**
nullable **35**
part of key **35**
read only **36**
title **36**
token **35**
value column **36**
visibility **36**
`commit` member function
`IliSQLCursor` class **152**
`connect` member function
`IliSQLSession` class **150**
containers **23**
`copyTable` member function
`IliTable` class **44**
cursors
obtaining the schema of **153**
see `IliSQLCursor` class

D

data formats **300**
dates **300**
literal characters **302**
numbers **298**
string **297**
Data Source page
DbField inspector panel **210**
DbNavigator inspector panel **234**
DbOptionMenu inspector panel **248**
DbPicture inspector panel **246**
DbStringList inspector panel **251**
DbText inspector panel **224**
DbToggle inspector panel **227**
DbTreeGadget inspector panel **257**
EntryField inspector panel **214**
Memory Data Source inspector panel **193**
TableComboBox inspector panel **217**
TableGadget inspector panel **203**
ToggleSelector inspector panel **231**
data sources **57** to **68**
overview **27**
callbacks **62**
connecting gadgets **70**
creating **58**
defining parameters **139**
error handling **66**
managing rows **60**
retrieving **67**
scope **59**
table objects **34**
data types
checking the type of an object **93**
converting IBM ILOG InForm type **93**
list of supported types **92**
structured types **96**
database drivers **150**
including at compile time **153**
macro symbols **153**
database fields **209**
database systems

connecting **150**
database tables **32**
data-source-aware gadgets **69** to **90**
connecting to a data source **70**
creating **59**
DbField **99**
DbGantt **87**
DbGrapher **85**
DbNavigator **78**, **103**
DbOptionMenu **81**
DbPicture **81**
DbStringList **82**
DbText **77**
DbTimer **80**
DbToggle **78**
DbTreeGadget **82**
description of **24**, **27**
entry field **76**
foreign table **71**
HTMLReporter **80**
interface **69**
managing values **71**
setting the look **71**
table combo box **77**
table gadget **72**
toggle selector **78**, **108**
Datatype page
Memory Data Source inspector panel **193**
SQL Data Source inspector panel **177**
DbField inspector panel **209**
Callbacks page **213**
Data Source page **210**
General page **213**
Mapping page **212**
DbField styles
`IliEntryFieldStyle` **100**
`IliOptionMenuStyle` **101**
`IliStringListStyle` **102**
`IliTableComboBoxStyle` **101**, **103**
`IliTextStyle` **101**
`IliToggleSelectorStyle` **102**
`IliToggleStyle` **101**
DbFields **99** to **104**
read-only columns **112**
style **100**
DbGantt inspector panel
Activities page **278**
Callbacks page **289**
Events page **287**
General page **288**
notebook pages **276**
Periods page **286**
Resources page **277**
Scales page **285**
DbGrapher inspector panel **270**
Callbacks page **276**
Events page **275**
General page **275**
Links page **272**
Look page **273**
Nodes page **270**
DbNavigator inspector panel **234**
Callbacks page **237**
Data Source page **234**
General page **237**
DbOptionMenu inspector panel **247**
Callbacks page **250**
Data Source page **248**
General page **249**
Mapping page **249**
DbPicture inspector panel **245**
Callbacks page **247**
Data Source page **246**
General page **247**
DbStringList inspector panel **250**
Callbacks page **256**
Data Source page **251**
General page **253**
Mapping page **252**
Scrollbars page **255**
Specific page **253**
DbText inspector panel **224**
Callbacks page **226**
Data Source page **224**
General page **225**
Scrollbars page **225**
DbTimer inspector panel **237**
Callbacks page **238**
Specific page **237**
DbToggle inspector panel **227**

Callbacks page **230**
Data Source page **227**
General page **229**
Mapping page **228**
Specific page **229**
DbTreeGadget inspector panel **257**
Callbacks page **264**
Data Source page **257**
General page **260**
Scrollbars page **263**
Specific page **260**
`deleteCurrentRow` member function
`IliDataSource` class **61**
`deleteRow` member function
`IliTable` class **38**
dialog boxes
Color Selector **169**
File Selector **170**
Font Chooser **168**
SQL Data Source inspector panel
Connect **187**
Differences **191**
Question **190**
Select Table **189**
Source **188**
SQL Data Source Properties **184**
Table columns **221**
Directory class example **51**
`disconnect` member function
`IliSQLSession` class **150**
Document page
HTMLReporter inspector panel **239**
`dontValidateRow` member function
`IliDataSource` class **63**, **67**

E

editors **18**
`enableInsert` member function
`IliDataSource` class **61**
EntryField inspector panel **213**
Callbacks page **216**
Data Source page **214**
General page **215**
Specific page **215**
error messages generated by IBM ILOG InForm **307**
errors
data sources **66**
tables **41**
Events page
DbGantt inspector panel **287**
DbGrapher inspector panel **275**
`execute` member function
`IliSQLCursor` class **151**

F

`f_getGraphic` member function
`IliFieldItf` class **70**
`f_getValue` member function
`IliFieldItf` class **71**, **92**
`f_setDataSourceName` member function
`IliFieldItf` class **70**, **71**, **72**
`f_setValue` member function
`IliFieldItf` class **71**
`fetchAll` member function
`IliTable` class **41**
`fetchCompleted` member function
`IliTable` class **41**
`fetchNext` member function
`IliSQLCursor` class **151**
`IliTable` class **41**
File menu
SQL Data Source inspector panel **173**
File Selector dialog box **170**
First page page
HTMLReporter inspector panel **241**
Font Chooser dialog box **168**
foreign tables **105** to **110**
column properties **36**
constrained **109**
DbField styles **101**, **102**
setting up **105**
table combo box **77**
toggle selector **108**
troubleshooting **112**
types **113**
`format` member function
`IliValue` class **95**
Forms Assistant

creating forms **103**
read-only columns **112**
using with foreign table **109**

G

gadgets
IliDbField **209**
IliDbGantt **276**
IliDbGrapher **269**
IliDbNavigator **234**
IliDbOptionMenu **247**
IliDbPicture **245**
IliDbStringList **250**
IliDbText **223**
IliDbTimer **237**
IliDbToggle **226**
IliDbTreeGadget **256**
IliEntryField **213**
IliHTMLReporter **238**
IliMemoryDataSource **192**
IliSQLDataSource **171**
IliTableComboBox **216**
IliTableGadget **202**
IliToggleSelector **231**
General notebook page **165**
General page
DbField inspector panel **213**
DbGantt inspector panel **288**
DbGrapher inspector panel **275**
DbNavigator inspector panel **237**
DbOptionMenu inspector panel **249**
DbPicture inspector panel **247**
DbStringList inspector panel **253**
DbText inspector panel **225**
DbToggle inspector panel **229**
DbTreeGadget inspector panel **260**
EntryField inspector panel **215**
HTMLReporter inspector panel **242**
Memory Data Source inspector panel **197**
TableComboBox inspector panel **221**
TableGadget inspector panel **204**
ToggleSelector inspector panel **233**
getBinaryValue member function
IliSQLCursor class **152**
getBuffer member function
IliTable class **39**
getDateValue member function
IliSQLCursor class **152**
getDoubleValue member function
IliSQLCursor class **152**
getFloatValue member function
IliSQLCursor class **152**
getFormatted member function
IliValue class **96**
getIntegerValue member function
IliSQLCursor class **152**
getNestedSchema member function
IliDatatype class **97**
getRealColno member function
IliTableGadget class **73**
GetRegisteredSession member function
IliSQLSession class **155**
getRowsCount member function
IliTable class **40**
getSchema member function
IliSQLCursor class **153**
getSelection member function
IliTableGadget class **73**
getStringValue member function
IliSQLCursor class **152**
getType member function
IliValue class **93**
getValue member function
IliSQLCursor class **152**
IliTable class **39**
getVisualColno member function
IliTableGadget class **73**
Global checkbox
SQL Data Source inspector panel **174**
gotoFirst member function
IliDataSource class **60**
gotoLast member function
IliDataSource class **60**
gotoNext member function
IliDataSource class **60**
gotoPrevious member function
IliDataSource class **60**
gotoRow member function
IliDataSource class **60**

H

hasGlobalScope member function
  IliDataGem class **59**
hasTuple member function
  IliSQLCursor class **151**
Having page
  SQL Data Source inspector panel **176**
HTMLReporter inspector panel **238**
  Callbacks page **242**
  Document page **239**
  First page page **241**
  General page **242**
  Table of contents page **240**

I

IBM ILOG DB Link **26**
IBM ILOG InForm
  editors **18**
IBM ILOG Views classes
  IlvApplication **22**
  IlvContainer **88**
  IlvDisplay **22**
  IlvGadget **24**, **27**, **69**
  IlvGadgetContainer **23**
  IlvGraphic **24**, **62**
  IlvTextField **27**, **70**
IliCallbackManager class **89**
IliChartGraphic class **83**
IliDataGem class **58**, **59**
IliDataSource class **27**, **57**, **137**
IliDatatype class **26**, **91**, **92**, **97**
IliDate class **292**
IliDbField class **76**, **99**, **105**
IliDbField gadget **209**
IliDbGantt class **87**
IliDbGantt gadget **276**
IliDbGrapher class **85**
IliDbGrapher gadget **269**
IliDbNavigator class **60**, **103**
IliDbNavigator class **78**
IliDbNavigator gadget **234**
IliDbOptionMenu class **81**
IliDbOptionMenu gadget **247**
IliDbPicture class **81**
IliDbPicture gadget **245**
IliDbStringList class **82**
IliDbStringList gadget **250**
IliDbText class **77**
IliDbText gadget **223**
IliDbTimer class **80**
IliDbTimer gadget **237**
IliDbToggle class **78**
IliDbToggle gadget **226**
IliDbTreeGadget class **82**
IliDbTreeGadget gadget **256**
IliDecimal class **292**
IliEntryField **76**
IliEntryField class **27**, **69**, **73**
IliEntryField gadget **213**
IliErrorList class **42**
IliErrorMessage class **41**
IliErrorReporter class **67**
IliErrorSink class **42**
IliFieldItf class **27**, **69**, **72**
IliFormat class **28**, **293**
IliGraphicToField global function **70**
IliHTMLReporter class **80**
IliHTMLReporter gadget **238**
IliInputMask class **28**, **295**
IliInputMaskIpl class **28**
IliIsAField global function **70**
IliMapTable class **32**, **46**
IliMemoryDataSource class **58**
IliMemoryDataSource gadget **192**
IliMemoryTable class **26**, **32**, **37**, **38**, **45**, **58**, **105**
IliRepository class **67**
IliSchema class
  member functions **37**
IliSQLCursor class **26**, **151**
IliSQLDataSource class **58**, **118**
IliSQLDataSource gadget **171**
IliSQLSession class **26**, **149**, **150**
IliSQLTable class
  as an object of a data source **149**
  asynchronous mode **134**
  auto-commit mode **124**
  auto-refresh mode **125**
  auto-row locking mode **127**

bound variables mode **126**
concurrency control **123**
cursor buffering **127**
defining in C++ **120**
defining interactively **119**
description of **45**, **118**
dynamic SQL mode **126**
fetch policy **125**
inserting-nulls mode **126**
instantiated by `IliSQLDataSource` **58**
managing rows **26**
query mode **137**
subclass of `IliTable` **32**
transaction managers **128**
used as a foreign table **105**
`IliString` class **291**
`IliStringsTable` class **32**, **45**
`IliTable` class
description of **26**, **32**, **57**
foreign table **105**
managing rows **38**
subclassing **46**
transaction managers **128**
used with structured types **96**
`IliTableBuffer` class **38**, **78**
`IliTableComboBox` class **73**, **77**, **105**
`IliTableComboBox` gadget **216**
`IliTableGadget` class **72**
`IliTableGadget` gadget **202**
`IliTableHook` class **43**
`IliTableSelection` class **72**
`IliToggleSelector` class **78**
foreign tables **105**
`IliToggleSelector` gadget **231**
`IliTransactionManager` class **128**
`IliValue` class **91** to **96**
checking the data type **93**
constructing **92**
converting to a C++ type **94**
formatting **95**
formatting a value **95**
null values **92**
representing values **26**
used with structured types **96**
`IlvGadgetContainer` class **23**
InForm palette
opening **159**
`insertColumn` member function
`IliSchema` class **38**
`insertRow` member function
`IliTable` class **38**
inspector panels
DbField **209**
DbGantt **276**
DbGrapher **270**
DbNavigator **234**
DbOptionMenu **247**
DbPicture **245**
DbStringList **250**
DbText **224**
DbTimer **237**
DbToggle **227**
DbTreeGadget **257**
EntryField **213**
HTMLReporter **238**
Memory Data Source **192**
SQL Data Source **172**
TableComboBox **216**
TableGadget **202**
ToggleSelector **231**
`isConnected` member function
`IliSQLSession` class **150**
`isInputModified` member function
`IliDataSource` class **61**
`isNull` member function
`IliValue` class **92**

L

libraries
IBM ILOG InForm **17**
IBM ILOG Views **17**
Links page
DbGrapher inspector panel **272**
Look page
DbGrapher inspector panel **273**
Memory Data Source inspector panel **194**
SQL Data Source inspector panel **179**

M

makeTable member function
  IliDatatype class **97**
Mapping page
  DbField inspector panel **212**
  DbOptionMenu inspector panel **249**
  DbStringList inspector panel **252**
  DbToggle inspector panel **228**
  Memory Data Source inspector panel **196**
  SQL Data Source inspector panel **181**
  TableComboBox inspector panel **218**
  ToggleSelector inspector panel **233**
masks **28**
  default value **305**
  placeholders **304**
  predefined **305**
Memory Data Source inspector panel
  buttons **200**
  Callbacks page **199**
  Data Source page **193**
  Datatype page **193**
  General page **197**
  Look page **194**
  Mapping page **196**
  notebook pages **192**
  Specific page **198**

N

Name field
  SQL Data Source inspector panel **174**
Naming Conventions **13**
navigation buttons **234**
Nodes page
  DbGrapher inspector panel **270**
notebook pages
  Callbacks page **167**
  General page **165**
number data formats **298**

O

one-tier tables
  IliMapTable class **46**
  IliMemoryTable class **45**
  IliStringsTable class **45**

P

Palettes panel
  Charts gadgets **160**, **163**
  InForm gadgets **160**
panels
  see Containers
parameters **139** to **144**
Parameters page
  SQL Data Source inspector panel **182**
Periods page
  DbGantt inspector panel **286**
pointToSelection member function
  IliTableGadget class **73**
primary keys
  table object **147**

Q

Query menu
  SQL Data Source inspector panel **173**
query mode **137**
queryConnect member function
  IliSQLSession class **154**
Question dialog box **190**

R

registerCallback member function
  IlvContainer class **88**
registered sessions
  see SQL sessions (application-wide)
RegisterSession member function
  IliSQLSession class **154**
releaseBuffer member function
  IliTable class **39**
releaseCursor member function
  IliSQLCursor class **153**
removeErrorSink member function
  IliDataSource class **66**
Repository
  retrieving a data source **67**

Resources page
DbGantt inspector panel **277**
result sets **40**, **151**
rollback member function
IliSQLCursor class **152**
rowToBuffer member function
IliTable class **39**
run-time options for SQL sessions **123**

S

Scales page
DbGantt inspector panel **285**
Scrollbars page
DbStringList inspector panel **255**
DbText inspector panel **225**
DbTreeGadget inspector panel **263**
select member function
IliSQLCursor class **151**
IliTable class **40**
Select page
SQL Data Source inspector panel **175**
SELECT section
SQL Data Source inspector panel **174**
Select Tables dialog box **189**
setCallback member function
IlvGraphic class **62**
setColumnEditor member function
IliTableGadget class **74**
setColumnGeometryLocal member function
IliTableGadget class **73**
setColumnPartOfKey member function
IliSchema class **38**
setErrorReporter member function
IliDataSource class **67**
setGlobalScope member function
IliDataGem class **59**
setLanguageSensitive member function
IliMapTable class **46**
setNull member function
IliValue class **92**
setQueryConjunct member function
IliSQLTable class **138**
setQueryFrom member function
IliSQLTable class **122**
setSelection member function
IliTableGadget class **73**
setStyle member function
IliDbField class **76**
setValue member function
IliDataSource class **61**
Source dialog box **188**
Specific page
DbStringList inspector panel **253**
DbTimer inspector panel **237**
DbToggle inspector panel **229**
DbTreeGadget inspector panel **260**
EntryField inspector panel **215**
Memory Data Source inspector panel **198**
TableComboBox inspector panel **222**
TableGadget inspector panel **205**
SQL Data Source inspector
creating a data source definition **119**
specifying columns **144**
SQL Data Source inspector panel **172**
Allow insert checkbox **174**
buttons **183**
callbacks **183**
Connect dialog box **187**
Datatype page **177**
Differences dialog box **191**
File menu **173**
FROM section **189**
Global checkbox **174**
Having page **176**
Look page **179**
Mapping page **181**
menus **172**
Name field **174**
Parameters page **182**
Query menu **173**
Question dialog box **190**
Select page **175**
SELECT section **174**
Select Table dialog box **189**
Source dialog box **188**
SQL Data Source Properties dialog box **184**
SQL data sources **144** to **147**
Auto Select property **142**
defining table columns **144**

forcing column name **145**
query mode **137**
SQL sessions
application-wide **122**, **149**, **154**
committing or rolling back **152**
Connect dialog box **154**
connection parameters **150**
creating **150**
custom session **122**
locking and unlocking **150**
retrieving **123**
SQL statements
DELETE **120**
INSERT **120**
SELECT **40**, **118**, **120**, **151**, **151** to **153**
submitting to database **151**
UPDATE **120**
SQL tables **118** to **135**
asynchronous mode **134**
defining in C++ **120**
defining SQL session **122**
joining columns **118**, **119**, **121**
structured types **131**
transaction managers **128**
using parameters **128**
`startInsert` member function
`IliDataSource` class **61**
string data formats **297**
structured types **96**
SQL table **131**
subclassing
directory table example **49**
`IliFieldItf` class **72**

T

table
create **164**
drop **164**
edit data **165**
edit schema **165**
enable/disable SQL trace **165**
export **165**
import **165**
Table columns dialog box **221**
table combo boxes **77**
table gadgets **32**, **72**
callbacks **74**
column geometry **73**
columns geometry **110**
customizing **74**
editors **73**
fixed columns **112**
read-only columns **111**
table objects **32**, **103**
copying **44**
defining an instance **38**
defining the key **38**
defining the schema **37**
error catching **41**
foreign tables **105**
inserting a row **39**
local row cache **40**
look **110**
managing rows **38**
modifying a row **39**
primary key **147**
reading from a stream **44**
read-only columns **111**
removing a row **40**
subclasses **44**
subclassing guidelines **46**
writing to a stream **44**
Table of contents page
HTMLReporter inspector panel **240**
TableComboBox inspector panel **216**
Callbacks page **223**
Data Source page **217**
General page **221**
Mapping page **218**
Specific page **222**
TableGadget inspector panel
Callbacks page **208**
Data Source page **203**
General page **204**
notebook pages **202**
Specific page **205**
tables
one-tier **33**
two-tier **33**

see also table objects
tables, synchronizing with database **191**
text fields **223**
toggles **226**
ToggleSelector inspector panel **231**
Callbacks page **234**
Data Source page **231**
General page **233**
Mapping page **233**
transaction managers **128**
transactions **152**
two-tier tables **60**
callbacks **64**
extra row management techniques **40**
`IliSQLTable` class **45**
row retrieval **40**

U

`updateRow` member function
`IliTable` class **38**, **63**

V

`validate` member function
`IliDataSource` class **61**
values
see `IliValue` class

W

`writeTable` member function
`IliTable` class **44**