IBM® Workplace Forms™ Server - API

IBM

Version 2.6

JAVA API User's Manual

S229-1527-00

Note

Before using this information and the product it supports, read the information in "Notices," on page 199.

First Edition (September 2006)

This edition applies to version 2.6 of IBM Workplace Forms Server - API (product number L-DSED-6JLR37) and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright International Business Machines Corporation 2003, 2006. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Introduction . . . 1
About This Manual . . . 1
Who Should Read This Manual . . . 1
Document Conventions . . . 2

About the API . . . 3
Where the API Fits in Your System . . . 3
Differences Between the C, Java, and COM Editions of the API . . . 4
The API Data Structures . . . 4
FormNodeP Objects . . . 4
Holder Objects . . . 4
About the API Constants . . . 5

Overview of the Form Structure . . . 7
The Node Structure . . . 7
The Node Hierarchy . . . 7
References . . . 8
Dereferencing . . . 9
Namespace in References . . . 10
Advanced Information about the Node Structure . . . 10
A Sample Hierarchy . . . 11
The Sample Tree Structure . . . 11
Node Properties . . . 12

Introduction to the Form Library . . . 15

Getting Started with the Form Library 17
Setting Up Your Application . . . 17
Initializing the Form Library . . . 19
Loading a Form . . . 19
Retrieving A Value from a Form . . . 19
Setting a Value in a Form . . . 20
Writing a Form to Disk . . . 20
Closing a Form . . . 21
Compiling Your Application . . . 21
Testing your Application . . . 21
Distributing Applications That Use the Form Library 22
Summary . . . 22

Form Library Quick Reference Guide 23
Form Library Methods . . . 23
About the Method Descriptions . . . 25
Using Signatures with the Form Library . . . 26

The Certificate Class . . . 27
getBlob . . . 27
getDataByPath . . . 28
getIssuer . . . 31

The DTK Class . . . 35
initialize . . . 35
initializeWithLocale . . . 37

The FormNodeP Class . . . 41
FormNodeP Constants . . . 41
addNamespace . . . 41
checkValidFormats . . . 43
createCell . . . 43
deleteSignature . . . 44
dereferenceEx . . . 46
destroy . . . 49
duplicate . . . 49
encloseFile . . . 51
encloseInstance . . . 52
extractFile . . . 54
extractInstance . . . 55
extractXFormsInstance . . . 57
getAttribute . . . 58
getAttributeList . . . 60
getCertificateList . . . 61
getChildren . . . 63
getFormVersion . . . 64
getInfoEx . . . 65
getLiteralByRefEx . . . 66
getLiteralEx . . . 69
getLocalName . . . 69
getNamespaceURI . . . 71
getNamespaceURIFromPrefix . . . 72
getNext . . . 73
getNodeType . . . 74
getParent . . . 75
getPrefix . . . 76
getPrefixFromNamespaceURI . . . 77
getPrevious . . . 78
getReferenceEx . . . 79
getSecurityEngineName . . . 82
getSigLockCount . . . 83
getSignature . . . 84
getSignatureVerificationStatus . . . 85
isSigned . . . 85
isValidFormat . . . 86
isXFDL . . . 87
removeAttribute . . . 88
removeEnclosure . . . 90
replaceXFormsInstance . . . 91
setActiveForComputationalSystem . . . 92
setAttribute . . . 93
setFormula . . . 95
setLiteralByRefEx . . . 96
setLiteralEx . . . 98
signForm . . . 99
updateXFormsInstance . . . 101
validateHMACWithSecret . . . 103
validateHMACWithHashedSecret . . . 105
verifyAllSignatures . . . 108
verifySignature . . . 109
writeForm . . . 111
xmlModelUpdate . . . 112

© Copyright IBM Corp. 2003, 2006

The Hash Class 115
hash 115

The IFSSingleton Class 117
getFunctionCallManager. 117
getLocalizationManager 117
getSecurityManager 118
getXFDL 119

The LocalizationManager Class . . . 121
getCurrentThreadLocale 121
getDefaultLocale 123
setCurrentThreadLocale 126
setDefaultLocale 129

The SecurityManager Class 133
lookupHashAlgorithm 133

The Signature Class 135
getDataByPath 135
getSigningCert 139

The XFDL Class 141
create 141
getEngineCertificateList 143
isDigitalSignaturesAvailable 145
readForm. 146

Introduction to the FCI Library 149
About Functions, Packages and Extensions . . . 149
About the Function Call Interface (FCI). 150
How the Form and FCI Libraries Work Together 150
The FCI Extension Architecture 151

Getting Started with the FCI Library 155
Creating Extensions with the FCI methods . . . 155
Creating Extensions with the FCI methods . . . 156
Creating the Extension class 156
Implementing the extension initialization method 156
Creating a new FunctionCall object 157
Setting up the FunctionCall Class. 157
Creating a FunctionCall class 157
Retrieving the Function Call Manager 159
Registering the FunctionCall object with the IFX Manager 159
Registering your packages of custom functions with the Function Call Manager 160
Implementing your custom functions 161
Providing help information for each of your functions 162
Building the extension 163
Testing and Distributing extensions 164
Packaging extensions as JAR Files 164
Distributing Extensions for Testing or Use. . . 165
Embedding extensions in XFDL Forms 165
About MIME Types 166
Location of Installed extensions (Security Issues) 166
Additional Security Restrictions for Functions Enclosed in XFDL Forms 167
Summary. 167

FCI Library Quick Reference Guide 169
About the Method Descriptions 169

The Extension Class 171
Imports 171
Example 171
extensionInit 171

The FunctionCall Class 173
Imports 173
Example 173
FunctionCall Class Constants 174
evaluate 175
help 178

The IFX Class 181
Imports 181
Example 181
deregisterInterface. 181
getInterfaceInstances 182
registerInterface 183

The FunctionCallManager Class . . . 185
Imports 185
Example 185
deregisterFunctionCall 185
evaluateFunctionCall 186
getDefaultListener 188
registerFunctionCall 188
getFunctionCallHelp 190
getFunctionCallList 192
getFunctionCallPackageList. 192

Appendix. JSP Support 195
System Requirements. 195
Combining JSP and XFDL 195
Sample JSP Page 196
Sample JSP Application 197

Appendix. Notices 199
Trademarks 200

Index 201

Introduction

Welcome to the Java™ Edition of the user's manual for the IBM® Workplace Forms™ Server — API. The API extends the capabilities of Workplace Forms by enabling you to:

- Manipulate XFDL forms from new or existing applications.
- Create custom-built functions that may be integrated into XFDL forms.

This section discusses the organization and format of this manual. To learn more about the API, refer to "About the API" on page 3.

About This Manual

This manual has been organized as both an instruction manual and a quick reference. It describes the functions available in the API and provides examples of their use.

This manual contains the following major sections:

Section	Page
Introduction — introduces you to the features of the API.	"Introduction"
Overview of the Form Structure — explains how XFDL forms are stored in memory.	"Overview of the Form Structure" on page 7
Getting Started with the Form Library — provides a detailed tutorial demonstrating how to create a simple application that interacts with an XFDL form.	"Introduction to the Form Library" on page 15
Form Library Quick Reference — a reference to the methods contained in the Form Library. Each method description includes sample code.	"Form Library Quick Reference Guide" on page 23
Getting Started with the FCI Library — provides a detailed tutorial demonstrating how to create a simple function that you can call from an XFDL form.	"Introduction to the FCI Library" on page 149
FCI Quick Reference Guide — a reference to the Java methods contained in the FCI API. Each method description includes sample code.	"FCI Library Quick Reference Guide" on page 169

Who Should Read This Manual

The API is designed to be easy to use for any moderately experienced programmer. However, the skill level required to develop particular functions may be quite high. This document is intended for developers who have a working knowledge of:

- Java Programming and syntax.
- Extensible Forms Description Language (XFDL) and syntax. Refer to the *Extensible Forms Description Language Specification* for more information.

© Copyright IBM Corp. 2003, 2006

Document Conventions

The following conventions appear throughout this manual:

- Sample code is presented in a monospaced font, and is indented to make the code stand out:

```
public void extensionInit(Extension theExtension) throws UWIException
   {
      FunctionCall theActualCode = new SamplePluginFCI(theExtension);
   }
```

- Text in bold italics represents information that you need to supply:

```
<label sid="firstName">
   <value>your first name here</value>
<label>
```

- The hash symbol (#) represents a number.
- Angle brackets enclose placeholders. For example, *<API Program Folder>* represents the actual folder in which you installed the API.
- Braces indicate optional items. The following example indicates that the item tag (including the period after it) is optional:

```
{itemtag.} option
```

- "*xx*" or "*xxx*" appears in place of the two or three digit version number of the API. In particular, these placeholders appear when referring to file names, folders, and directories that contain the API's version number.
- Brackets are used to indicate a sequence of choices, and the pipe symbol (|) is used to indicate "or". The following example indicates that you can use a number or a name:

```
(number|name)
```

About the API

The Workplace Forms Server — Application Programmer Interface (API) consists of a collection of programming tools to help you develop applications that can interact with XFDL forms. These tools are available for both C and Java programming environments. The API enables you to access and manipulate forms as structured data types.

The API is divided into two libraries: the Form Library and the Function Call Interface (FCI) Library. The Form Library allows you to create applications that:
- Read and write forms.
- Retrieve information from form elements.
- Add cells to certain form items.
- Insert information into form elements.

For more information about the Form Library refer to the “Form Library Quick Reference Guide” on page 23.

The Function Call Interface (FCI) Library provides additional methods that:
- Create, duplicate, or delete form elements.
- Manipulate and verify digital signatures.
- Handle attachments.
- Create custom functions for use within XFDL forms.

For more information about the FCI Library refer to page “Introduction to the FCI Library” on page 149.

Where the API Fits in Your System

IBM provides a powerful suite of forms software for creating, using and transmitting forms over the Internet. The main components of this suite are:

Workplace Forms Viewer — Use the Viewer to view XFDL forms just as you would use a web browser to view HTML pages. You can also use the Viewer to fill out forms and submit them for review.

Workplace Forms Designer — The Designer provides an easy to use WYSIWYG design environment for creating XFDL forms. Use the Designer to create forms quickly and easily.

Workplace Forms API — The API is made up of Form and FCI methods. Use the Form Library of methods to develop applications that manipulate XFDL forms. Use the FCI functions to develop customized functions that can be called from within forms.

© Copyright IBM Corp. 2003, 2006

Differences Between the C, Java, and COM Editions of the API

The various editions of the API differ in the following ways:

- The Java and COM editions offer an object-oriented interface.
- The COM edition does not support the FCI Library.
- The COM edition does not include the following Form Library functions:
  - GetInfoEx
  - GetAttributeList
- The COM edition includes the following Form library functions that the other editions do not:
  - GetType
  - GetIdentifier
  - ReadFormFromASPRequest
  - WriteFormToASPResponse

In all other respects, the different editions of the API provide the same functionality, and use the same memory model for forms.

The API Data Structures

FormNodeP Objects

The methods in the Form Library store forms in memory as a series of linked nodes. Each node, regardless of its level in the hierarchy, is represented by a **FormNodeP** object. Before you can use a **FormNodeP** object, you must import the **FormNodeP** class as follows:

```
import com.PureEdge.xfdl.FormNodeP
```

The functions in the Form Library are responsible for creating and populating these nodes, and for freeing the memory they occupy.

About Memory Use

The Form methods are responsible for creating and populating these nodes. Furthermore, once you are done working with a form, you must use the **destroy** method on the root node of the form to remove it from memory.

Comparing FormNodeP Objects

Be aware that in Java, objects cannot be compared using the == operator. The API behaves in the same way. Accordingly, when comparing **FormNodeP** objects, you should always use Java's **equals** method.

Holder Objects

Because Java does not support output parameters, methods are normally limited to returning a single value. However, there are many cases in which it is useful to return multiple values from a single method. In these cases, the API uses *Holder objects*.

Holder objects are objects that are created with a single variable. These objects can then be passed into a method that sets that variable. Once the method returns, the value of variable can be retrieved, thereby creating an artificial output parameter for the method.

Holder Constructors

Each **Holder** class provides two constructors:

- An empty constructor that creates an object and declares the an empty variable called *value*. For example, to create an empty **IntHolder** you would use the following constructor:

  ```
  IntHolder myInt = new IntHolder();
  ```

- A constructor that creates an object, declares a variable called *value*, and sets the value of the variable. For example, to create a **IntHolder** object with a value of *5*, you would use the following constructor:

  ```
  IntHolder myInt = new IntHolder(5);
  ```

Getting and Setting Holder Values

To get or set the value of a particular **Holder** object, simply dereference the holder's *value*. For example:

```
int value = myIntHolder.value;
myIntHolder.value = 2;
```

Holder Types

The following table lists the holder types available, the data type of each holder's variable, and the class you must import to use the holder:

Holder	Data Type	Class
BooleanHolder	Boolean	com.PureEdge.BooleanHolder
IFSUserDataHolder	IFSUserData	com.PureEdge.IFSUserDataHolder
IntHolder	Int	com.PureEdge.IntHolder
ShortHolder	Short	com.PureEdge.ShortHolder
ShortListHolder	Short [ ]	com.PureEdge.ShortListHolder
StringHolder	String	com.PureEdge.StringHolder
StringListHolder	String [ ]	com.PureEdge.StringListHolder
StringListHolder	String [ ]	com.PureEdge.StringListHolder

Note: While certain methods in the FCI library require an IFSUserDataHolder as a parameter, you will not need to manipulate this object.

About the API Constants

Several API methods may use or return constants. When using these constants, you must:

- Import the library that contains the constant.
- Prefix the constant with its class.

For example, the ITEM_REFERENCE constant belongs to the FormNodeP class. To use it, you would first ensure that you have imported the FormNodeP class. You could then refer to the constant as:

```
FormNodeP.ITEM_REFERENCE
```

The following table lists the constant prefixes and the classes you must import:

Prefix	Class
FormNodeP	com.PureEdge.xfdl.FormNodeP

Prefix	Class
SecurityManager	com.PureEdge.security.SecurityManager
SecurityUserStatusType	com.PureEdge.security.SecurityUserStatusType
XFDL	com.PureEdge.xfdl.XFDL

Overview of the Form Structure

This section provides an overview of an XFDL form as it is represented in memory. Developers must understand the memory structure of a form to effectively develop applications using the API.

The Node Structure

When a form is loaded into memory, it is constructed as a series of linked nodes. Each node represents an element of the form, and together these nodes create a tree that describes the form. The following diagram illustrates the general composition of a single node.

Type	Identifier
Literal	Compute

Each node within the tree has the following properties:

- **Type —** For page and item nodes, this describes the type of node, such as *button*, *line*, *field*, and so on. Page nodes are always of type *page*.
- **Literal** — The literal value of the node (for example, a literal string). If the node has a formula, the result of the formula will be stored here.
- **Identifier —** The page tag, item tag, option name, or custom name assigned to the node.
- **Compute —** The compute assigned to the node (for example, "field_1.value + field_2.value"). The result of the compute will be stored in the literal of the node.

Depending on the node type, some of these properties may be null.

The Node Hierarchy

Every node is part of an overall hierarchy that describes the complete form. This hierarchy follows a standard tree structure, with the top of the tree being the top (or root) of the hierarchy.

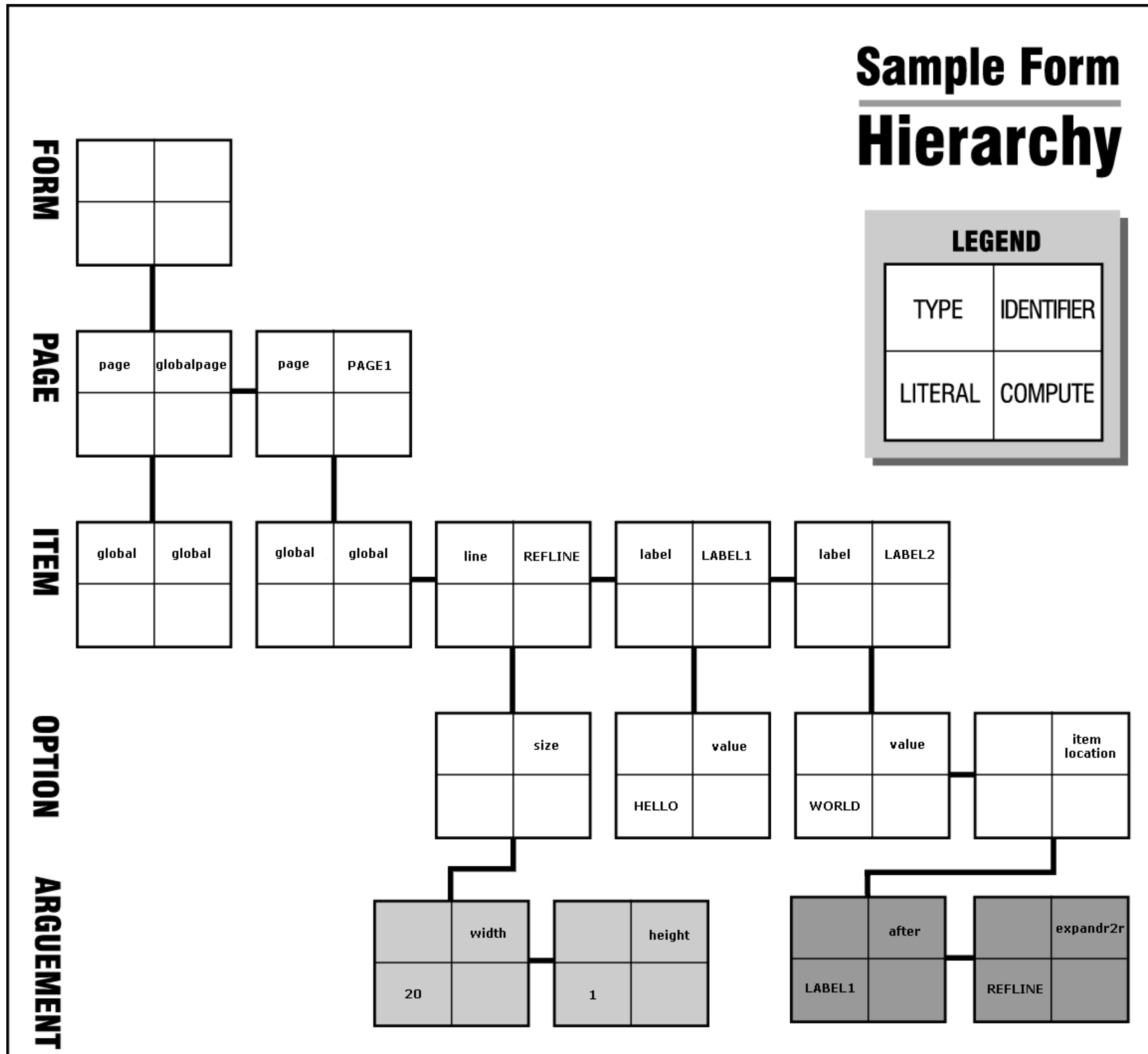
The diagram on the following page illustrates the typical tree structure for a simple form.

The elements of the hierarchy, in descending order, are:

- **Form** — Each form has one form level node. This is the root node of the tree.
- **Page** — Each form contains pages, which are represented as children of the form node. Each form has at least two page nodes - one for the globalpage, which stores the global settings, and one for the first page of the form.
- **Item** — Each page contains items, which are represented as children of the page node. An item node is created for each item, including the global item which stores page settings.
- **Option —** Each item contains options, which are represented as children of the item node. An option node is created for each option.
- **Argument** — Options often contain further settings, or arguments, which are represented as children of the option node or as children of other argument

© Copyright IBM Corp. 2003, 2006

nodes. There may be more than one level of argument node created below an option node, depending on the option's settings. The easiest way to access a particular node in the hierarchy is to use a reference. References allow you to locate a specific node without first having to locate the parent of that node.



References

References allow you to identify a specific page, item, option, or argument by providing a ″path″ to that element. This means that you can access an element directly without having to locate any of its ancestors. The syntax of a reference follows this general pattern:

```
page.item.option[argument]
```

Each element of the reference is constructed as follows:

- **Page** and **Item —** Pages and items are identified by their scope identifiers (sid). For example, *Page1* or *Field1*.

- **Options —** Options are identified by their tag name. For example, *value* or *itemlocation*.
- **Arguments —** Arguments are identified by their tag name or a zero-based numeric index. Argument references are always enclosed in brackets. For example, *[1]* or *[message]*.

  Arguments can also have any depth. For example, you might have an argument that contains arguments. You can reference additional levels of depth by adding another bracketed reference. For example, to refer to the first argument in the first argument of the *printsettings* option, you could use either *[0][0]* or the tag names in brackets, such as *[pages][filter]*.

You can create references to any level of the node hierarchy. For example, the following table illustrates a number of references starting at different levels of the form:

Start At	Ref to Page	Ref to Item	Ref to Option	Ref to Argument
Page	Page1	Page1.Field1	Page1.Field1.format	Page1.Field1.format[message]
Item	—	Field1	Field1.format	Field1.format[message]
Option	—	—	format	format[message]
Argument	—	—	—	[message]

Dereferencing

When making a reference to an item node, there may be times when you do not know which node to reference because it depends on some action from the user of the form. Consider a situation in which a user selects a cell from a list. Because you don't know beforehand which cell the user will choose, it is not possible to explicitly reference the item node for the chosen cell. In such cases you would use *dereferencing* to retrieve the node indirectly.

Essentially, dereferencing allows you to make a dynamic reference that is evaluated at runtime. This is accomplished by placing the -> symbol to the right of the dynamic reference.

For example, consider a list item called *List1* that has three cells called *Cell1*, *Cell2*, *Cell3*. If you wanted to access the item node of the cell selected by the user, we would use the following reference string:

```
List1.value->
```

At runtime, the portion of the expression that is to the left of the dereference symbol is evaluated and replaced. If the user chose the second cell, *List1.value* would be evaluated and replaced with:

```
Cell2
```

As a result, the item node for *Cell2* would be returned.

In some cases, instead of accessing the item node of the chosen cell, you may want to access one of the cell's option nodes. Again, dereferencing is used. The reference string would be:

```
List1.value->value
```

As before, the above expression is evaluated at runtime. The expression to the left of the dereference symbol is evaluated and replaced, just as before. So if the second

cell was selected, *List1.value* would be evaluated as *Cell2*. This value is then concatenated with the expression to the right of the dereference symbol. This would produce:

```
Cell2.value
```

As a result, the option node for *Cell2.value* would be returned.

Note: Do not include any spaces before or after the dereference symbol (->).

Namespace in References

References that include options or arguments in any namespace other than XFDL normally require the inclusion of the namespace prefix in the reference. For example, if you were referencing ″myOption″ in the ″custom″ namespace, you would refer to that option as ″custom:myOption″ as shown:

```
page_1.myItem.custom:myOption
```

If you are referencing named arguments, you should also use the appropriate namespace. For example:

```
page_1.myItem.custom:myOption[custom:myArgument]
```

However, if you are referencing an argument by index number you do not need to worry about namespace. All arguments, regardless of namespace, are indexed in order. For example, if ″myOption″ contained two arguments, the first in the XFDL namespace and the second in the custom namespace, you would use the following reference for the second argument:

```
page_1.myItem.custom:myOption[1]
```

Note: Page and item references never require a namespace prefix because they are uniquely identified by their sid.

The null Namespace

In some cases, forms may have no default namespace or may have a default namespace that is explicitly set to an empty string. In these cases, you can use *null* as the prefix for the empty namespace. For example, the following field declares a default namespace that is empty:

```
<page sid="Page1">
   <field sid="myField" xmlns="">
      <value>Test Value</value>
   </field>
</page>
```

In this case, to reference the value of the field, you would use the null prefix as shown:

```
Page1.null:myField.null:value
```

Advanced Information about the Node Structure

When an XFDL form is stored in memory, it exists as a series of nodes that are linked in a tree structure. As described in “The Node Hierarchy” on page 7, the tree structure follows this hierarchy: form, page, item, option, and argument.

Within a single branch of the tree, all elements of the same level are treated as siblings, each of which has a common parent, and each of which may have its own children.

The following example illustrates the node structure of a simple form, and gives a top-down description of the node structure.

A Sample Hierarchy

The following XFDL code creates the node hierarchy shown in . The result is a simple form that contains three items (a line and two labels).

```
<?xml version = "1.0"?>
<XFDL xmlns="http://www.ibm.com/xmlns/prod/XFDL/7.0"
   xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0">
   <globalpage sid="global">
      <global sid="global"></global>
   </globalpage>
   <page sid = "PAGE1">
      <global sid="global"></global>

      <line sid = "REFLINE">
         <size>
            <width>20</width>
            <height>0</height>
         </size>
      </line>
      <label sid = "LABEL1">
         <value>Hello</value>
      </label>
      <label sid = "LABEL2">
         <value>World</value>
         <itemlocation>
            <after>LABEL1</ae>
            <expandr2r>REFLINE</expandr2r>
         </itemlocation>
      </label>
   </page>
</XFDL>
```

The Sample Tree Structure

Each tree begins with the form, or root, node. This node contains no information - it simply represents the starting point of the tree structure.

Below the form node are the page nodes. In the previous example, there are two page nodes: ″global″ and ″PAGE1″. The ″global″ page node stores any global settings that apply to the form while ″PAGE1″ stores the contents of the first form page. Any additional pages would also be stored as children of the form node.

Below each page node are the item nodes. As illustrated in the previous example, the first item node for any page is always the ″global″ item. The ″global″ item stores any page settings that are applied to the items in that page. Each additional item in the page is stored as a sibling of the global item.

Note: The ″global″ page node will always have one child: the global item. This global item will always store the XFDL version number used to create the form, and is also used to store any global settings that are applied to the form.

Below each item node are the option nodes. Each option node represents an option setting for that item, such as a background color or font setting.

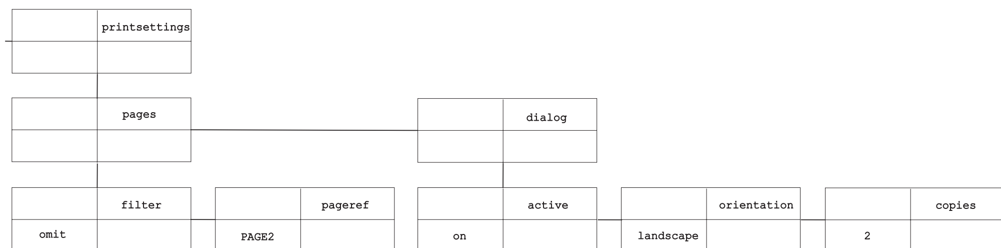
Below each option node are the argument nodes. These nodes contain the settings for the parent option. For example, the background color might be set to ″blue″.

There can be an infinite number and depth of these nodes, depending upon the number and depth of the settings for that option.

For instance, in the sample form, the *size* node for ″REFLINE″ has two argument nodes: one for the width and one for the height. In contrast, the *printsettings* option can have multiple argument nodes which themselves have argument nodes as children. The following is an example of the node structure of the *printsettings* option:

```
<printsettings>
   <pages>                                    argument node level I
      <filter>omit</filter>                   argument node level II
      <pageref>page2</pageref>                argument node level II
   </pages>
   <dialog>                                   argument node level I
      <active>on</active>                     argument node level II
      <orientation>landscape<orientation>     argument node level II
      <copies>2</copies>                      argument node level II
   </dialog>
</printsettings>
```

printsettings Node Structure



Thus, in storing the *printsettings* option, two levels of argument nodes are created. The first level describes the number of array elements in the option (two). The second level gives the arguments for each element.

Due to their potential complexity, pay careful attention to the mapping of argument nodes.

Note: In cases where an option has multiple elements in an array (for example, *printsettings*), there will be a single option node, but a separate argument node for each element in the array.

Node Properties

There are several levels of nodes in an XFDL form: form (or root), page, item, option, and argument (which can have an infinite number of levels). Each node has four properties: literal, type, identifier, and compute. A node does not necessarily contain information for every property.

For example, a page node can never have values for the compute or literal properties. And while a value for the user data property is optional, a page node must always have values for the type and identifier properties.

The following table illustrates what properties may be in use for each node level.

Level	Node Property Literal	Type	Identifier	Compute
Form	no	no	no	no
Page	no	always	always	no
Item	no	always	always	no
Option	yes	no	always	yes
Argument (at any level)	yes	no	yes	yes

yes — node can have that property
always — node always has that property
no — node cannot have that property

Introduction to the Form Library

The Form Library is a collection of methods for developing applications that manipulate XFDL forms. Using the methods in the Form Library, your applications can:

- Read and write forms.
- Retrieve information contained in a form's elements.
- Assign information to the elements of a form.
- Create new elements within a form.
- Remove elements from a form.
- Extract images or enclosures from a form.
- Verify digital signatures.

Essentially, an XFDL form may be thought of as a structured data type, with the API as the means for accessing this data structure.

© Copyright IBM Corp. 2003, 2006

Getting Started with the Form Library

This section provides a detailed tutorial to help you understand how to use the Form Library. By working through the tutorial, you will perform all of the steps involved in creating a simple application that uses the API methods, including:

- Initializing the Form Library.
- Reading a form into your application.
- Setting and retrieving form data.
- Removing a form from memory.

The sample application in this tutorial reads an input form called CalculateAge.xfd into memory. It retrieves the user's birth day, month, and year as well as the current date from the form. It then places these values into hidden fields in the form. This triggers the form to compute the user's age and display the result. When complete, the application saves the changes made to calculateAge.xfd as a new form called Output.xfd.

Note: The sample application described in this tutorial is included with the API and can be found in the folder: <*API Program Folder*>\Samples\Java\Form\Demo\Calculate_Age\

The tutorial describes the following tasks:

"Setting Up Your Application"

"Initializing the Form Library" on page 19

"Loading a Form" on page 19

"Retrieving A Value from a Form" on page 19

"Setting a Value in a Form" on page 20

"Writing a Form to Disk" on page 20

"Closing a Form" on page 21

"Compiling Your Application" on page 21

"Testing your Application" on page 21

"Distributing Applications That Use the Form Library" on page 22

Note: Before you can build applications using the Form Library, you must install the API and set up your development environment. Refer to the *IBM Workplace Forms Server - API Installation and Setup Guide* for more information.

Setting Up Your Application

As with any Java application, you must begin by importing the necessary classes and defining the program's classes.

1. Create a new Java source file called calculateAge.java.
2. Any program that calls methods from the API must import the following classes:

© Copyright IBM Corp. 2003, 2006

```
import com.PureEdge.DTK;
import com.PureEdge.xfdl.FormNodeP;
import com.PureEdge.xfdl.XFDL;
import com.PureEdge.error.UWIException;
import com.PureEdge.IFSSingleton;
```

- You must place these lines before any class or interface definitions.

3. Set up the rest of your application. This generally includes defining any classes and methods for your application as well as declaring and initializing any variables you may need. The following code sets up the Calculate Age application:
   - Create the public class **CalculateAge** and the **main** method for the class.

     ```
     public class CalculateAge
        {
     ```

   - Declare a **FormNodeP** object called **theForm** to represent the form.

     ```
           private static FormNodeP theForm;
     ```

   - Create the program's **main** method.

     ```
           public static void main(String argv[])
              {
     ```

   - Declare the program's variables.

     ```
           int birthYear;
           int birthMonth;
           int birthDay;
     ```

   - The program's **main** method essentially consists of a series of calls to other methods. The Form Library methods are called from the definition of these methods.

     ```
              try
              {
                 initialize();

                 loadForm();

                 birthYear = getBirthYear();
                 birthMonth = getBirthMonth();
                 birthDay = getBirthDay();

                 setBirthYear(birthYear);
                 setBirthMonth(birthMonth);
                 setBirthDay(birthDay);

                 saveForm();
     ```

   - Free the memory in which the form was stored. For more information see “Closing a Form” on page 21.

     ```
                 theForm.destroy();
              }
     ```

   - Finally, perform exception handling.

     ```
              catch (Exception ex)
              {
                 ex.printStackTrace();
              }

              /* Additional code removed */

           }
        }
     ```

Initializing the Form Library

All applications that use the API functions must initialize the Form Library to ensure correct error and memory handling behavior. The sample application does this in a separate method called **initialize**. In turn, **initialize** calls the Form Library method **DTK.initialize** and passes it the name of the current program.

Define the **initialize** method to call the **DTK.initialize** method. **DTK.initialize** initializes the API environment.

```
private static void initialize() throws UWIException
{
   DTK.initialize("calculateAge", "1.0.0", "2.6.0");
}
```

Note: For detailed information about the **initialize** method, including a description of its parameters, refer to "initialize" on page 35.

Loading a Form

Before your program can begin working with a form, you must load it into memory. CalculateAge does this by defining a **loadForm** method to handle these tasks.

1. Before you can load the form, declare the **XFDL** object:

   ```
   XFDL theXFDL;
   ```

2. Use **IFSSingleton.getXFDL** to assign the **XFDL** object to **theXFDL**. This allows you to access the root node of the form.

   ```
   theXFDL = IFSSingleton.getXFDL();
        if(theXFDL == null)
           throw new Exception("Could not find interface");
   ```

   - The **loadForm** method uses the Form Library method **readForm** to load the form into memory. Before you can use **readForm** you must retrieve the **XFDL** object.

3. Call the API method **readForm** to load the form into memory. The method returns a reference to the root node of the form.

   ```
   theForm = theXFDL.readForm("calculateAge.xfd", 0);
      }
   ```

   - The argument ″calculateAge.xfd″ is the name of the form to read from the local drive.

Note: For more information about the **readForm** method, refer to "readForm" on page 146.

Retrieving A Value from a Form

Once you have set up and initialized your application with the API and loaded a form into memory, your application is ready to start working with the form. The following code uses **getLiteralByRefEx** to get a specific value from the form:

1. Define the method **getBirthDay** and a string variable called temp.

   ```
   private static int getBirthDay( ) throws Exception
      {
      String temp;
   ```

2. Call **getLiteralByRefEx** to retrieve the literal information contained in the form node PAGE1.BIRTHDAY.value

```
temp = theForm.getLiteralByRefEx(null, "PAGE1.BIRTHDAY.value", 0,
   null, null);
```

- If the method returns a literal value, convert it into an integer value; otherwise, indicate that no value was entered into the field and throw an exception.

```
      if (temp.length( ) > 0)
      {
         return Integer.parseInt(temp);
      }
      else
      {
         throw new UWIException("The birth day was not entered.");
      }
   }
```

3. Define the following methods to retrieve the user's birth month and year from the input form. These methods will be exactly the same as **getBirthDay** except for the parameters passed to **getLiteralByRefEx**.
   - **getBirthMonth( ) —** retrieves the value *PAGE1.BIRTHMONTH.value* from **theForm**.
   - **getBirthYear( ) —** retrieves the value *PAGE1.BIRTHYEAR.value* from **theForm**.

Note: For detailed information about the **getLiteralByRefEx** method, including a description of its parameters, refer to "getLiteralByRefEx" on page 66.

Setting a Value in a Form

Once a form is loaded into memory, a developer can set the values associated with any of the item or option nodes located in the form by calling **setLiteralByRefEx**.

1. Define the method **setBirthDay** and an integer variable to reference the user's day of birth.

```
private static void setBirthDay(int birDay) throws Exception
{
   Integer day = new Integer(birDay);
```

2. Call the method **setLiteralByRefEx** to assign the user's day of birth to the form's hidden day field.

```
   theForm.setLiteralByRefEx(null, "PAGE1.HIDDENDAY.value", 0,
      null, null, day.toString());
}
```

3. Define the remaining methods to set the user's birth month and year in the form's hidden fields. These methods will be exactly the same as **setBirthDay** except for the parameters passed to **getLiteralByRefEx**.
   - **setBirthMonth( ) –** sets the value *PAGE1.HIDDENMONTH.value* in **theForm**.
   - **setBirthYear( ) –** sets the value *PAGE1.HIDDENYEAR.value* from **theForm**.

Note: For detailed information about **setLiteralByRefEx**, including a description of its parameters, refer to "setLiteralByRefEx" on page 96.

Writing a Form to Disk

Once you have finished making the desired changes to the form, you should save it to disk. If you want to retain the original form (calculateAge.xfd), you should save the modified form under a new name. This program saves the modified form as Output.xfd.

1. Define the method **saveForm**. This method demonstrates the use of the **FormNodeP** method **writeForm.**

   ```
   private static void saveForm( ) throws UWIException
   {
   ```

2. Call the Form method **writeForm** and pass it the new name of the form.

   ```
   theForm.writeForm("Output.xfd", null, 0);
       }
   ```

Note: For detailed information about **writeForm**, including a description of its parameters, refer to “writeForm” on page 111.

Closing a Form

Next, you must free the memory used by the form itself. This is the last operation in the main method of the program.

1. The programs main method calls the API’s **destroy** method to delete theForm object.

   ```
         theForm.destroy( );
      }
   ```

2. Display any exceptions before terminating.

   ```
      catch (Exception ex)
      {
         ex.printStackTrace( );
      }
   }
   ```

Note: For detailed information about **destroy**, including a description of its parameters, refer to “destroy” on page 49.

Compiling Your Application

Once you have generated the source files for your application, you must compile the source code.

- Use an appropriate compiler that is supported by this API to compile your files. Refer to the *IBM Workplace Forms Server - API Installation and Setup Guide* for more information about compatible development environments.
- Before building your application you should have a source file that represents your application. After compiling the file you will have a file an executable (or class file) with the same name.
- The details of compiling your source code are not included in this manual. Consult your development environment’s documentation for specific information on how to use your compiler.
- Make sure that the compiler uses the -I option when searching the directory containing the API include files.

Testing your Application

Use the sample form that accompanies the API to test the Calculate Age application.

1. Copy the file calculateAge.xfd to the folder containg your application. The file is located in the following folder:

   ```
   <API Program folder>\Samples\Java\Form\Demo\Calculate_Age\
   ```

2. Open the form in the Viewer to see the original settings.
3. Run the application that you have just created.
4. A new file will be created called Output.xfd.

Note: To view the forms provided with this API, you must have a licensed or evaluation copy of the IBM Workplace Forms Viewer installed.

Distributing Applications That Use the Form Library

32-bit applications that use methods from the Form Library will run on any computer that supports the Java Runtime Environment or the Microsoft Software Development Kit For Java v3.1 or later.

If you distribute applications that use the Form Library, you will also need to distribute a number of API files. Refer to the *IBM Workplace Forms Server - API Installation and Setup Guide* for information about distributing applications that use the Form Library.

Summary

By working through this section you have successfully built the Calculate Age application. In the process, you have learned how to initialize, compile, and test form applications using the following methods from the Form Library:

- initialize
- getXFDL
- readform
- getLiteralByRefEx
- setLiteralByRefEx
- writeForm
- destroy

The source code for the Calculate Age application is included with this API and can be found in the following folder:

```
<API Program folder>\Samples\Java\Form\Demo\Calculate_Age\
```

For a longer example using the Form Library of methods refer to the other sample application installed with the API. The source files for this application are located in the following folder:

```
<API Program folder>\Samples\Java\Form\Demo\Sample_Application\
```

To view the forms provided with the sample applications, you must have a copy of the Viewer installed.

Form Library Quick Reference Guide

This section provides detailed information about the Form Library. The available methods are divided into the following classes:

- “The Certificate Class” on page 27.
- “The DTK Class” on page 35.
- “The FormNodeP Class” on page 41.
- “The Hash Class” on page 115.
- “The IFSSingleton Class” on page 117.
- “The LocalizationManager Class” on page 121.
- “The SecurityManager Class” on page 133.
- “The Signature Class” on page 135.
- “The XFDL Class” on page 141.

Within each section, the methods are presented alphabetically.

Form Library Methods

The Form Library includes the following methods:

Class	Description	Methods
Certificate	The Certificate class includes a method for getting information about digital certificates.	getBlob getDataByPath getIssuer
DTK	The **DTK** class encapsulates methods that apply to the API as a whole.	initialize
FormNodeP	The **FormNodeP** class encapsulates methods that apply to particular form nodes.	addNamespace createCell deleteSignature dereferenceEx destroy duplicate encloseFile encloseInstance

© Copyright IBM Corp. 2003, 2006

Class	Description	Methods
FormNodeP (continued)	The **FormNodeP** class encapsulates methods that apply to particular form nodes.	extractFile extractInstance extractXFormsInstance getAttribute getAttributeList getCertificateList getChildren getFormVersion getInfoEx getLiteralEx getLiteralByRefEx getLocalName getNamespaceURI getNamespaceURIFromPrefix getNext getNodeType getParent getPrefix getPrefixFromNamespaceURI getPrevious getReferenceEx getSecurityEngineName getSigLockCount getSignature getSignatureVerificationStatus isSigned isXFDL removeAttribute removeEnclosure replaceXFormsInstance setActiveForComputationalSystem setAttribute setFormula setLiteralEx

Class	Description	Methods
FormNodeP (continued)	The **FormNodeP** class encapsulates methods that apply to particular form nodes.	setLiteralByRefEx signForm validateHMACWithSecret validateHMACWithHashedSecret verifyAllSignatures verifySignature writeForm xmlModelUpdate
Hash	The Hash class includes a method for hashing strings.	hash
IFSSingleton	The **IFSSingleton** class provides a static interface to XFDL objects.	getFunctionCallManager getLocalizationManager getSecurityManager getXFDL
LocalizationManager	The LocalizationManager class includes a method for setting the locale (language) that the API uses.	getCurrentThreadLocale getDefaultLocale setCurrentThreadLocale setDefaultLocale
SecurityManager	The SecurityManager class includes a method for obtaining a hashing algorithm.	lookupHashAlgorithm
Signature	The Signature class includes a method for getting information about signature objects.	getDataByPath getSigningCert
XFDL	The **XFDL** class encapsulates methods that relate to **FormNodeP** objects.	create getEngineCertificateList isDigitalSignaturesAvailable readForm

About the Method Descriptions

The methods in this reference guide are listed according to the class they belong to and are described using the following format:

- **Description:** Provides a general description of what the method does.
- **Method:** Lists the method's signature and type of value returned (if any).
- **Parameters:** Lists and describes each parameter in detail.
- **Returns:** Indicates what value is returned by the method.
- **Notes:** Provides additional information to help you use the method.

- **Example:** Provides sample code that uses the method in question.

Using Signatures with the Form Library

Computed options often contain their current computed value. If this value is signed, it will not change, even if something in the form changes that would normally trigger the compute.

The literal value is stored as simple character data in the computed option, as shown below:

```
<field sid="FIELD1">
   <value compute="page1.nameField.value">Jane E. Smith</value>
</field>
```

The node structure for this *value* option is:

field	FIELD1
	value
Jane E. Smith	Page1.nameField.value

The Viewer sets this literal value when a form is signed, submitted, or saved (and discards any old value if necessary). When **readForm** is invoked, the current value (cval) is set and cannot be changed. Because a digitally signed formula never fires after being signed, the current value for the option is always the same - and therefore it is possible to reference the option and get the signed literal value.

The Certificate Class

The **Certificate** class allows you to work with *Certificate* objects.

- Any application that makes calls to the **Certificate** methods must first import the following class:

  ```
  com.PureEdge.security.Certificate
  ```

- Many of the methods in the API will throw a generic exception called a **UWIException** if an error occurs. Import the following class to any .java files that call methods from the API:

  ```
  com.PureEdge.error.UWIException
  ```

getBlob

Description

This method extracts a binary long object (Blob). This Blob is a DER-encoded certificate.

Method

```
public byte [] getBlob(
   IntHolder theStatus
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theStatus	**IntHolder**	A holder that is set with the status of the operation. This will be one of the following: **SecurityUserStatusType.SUSTATUS_OK** — The operation was successful. **SecurityUserStatusType.SUSTATUS_CANCELLED** — the operation was cancelled by the user. **SecurityUserStatusType.SUSTATUS_INPUT_REQUIRED** — the operation required user input, but could not receive it (for example, it was run on a server with no user).

Returns

The Blob as a byte array.

Example

The following method extracts the Blob from a certificate, checks the status to make sure the operation was successful, then returns the Blob.

```
public byte[] extractBlob(Certificate theCert) throws UWIException
{
IntHolder theStatus;
byte[] theBlob;
```

© Copyright IBM Corp. 2003, 2006

```
   /* Get the Blob from the certificate. */

   theBlob = theCert.getBlob(theStatus);

   /* Check the status to ensure the method worked correctly. */

   if (theStatus.value != SecurityUserStatusType.SUSTATUS_OK)
   {
      throw new UWIException("getBlob exited with the wrong status.");
   }

   /* Return the Blob. */

   return(theBlob);
}
```

getDataByPath

Description

This method retrieves a piece of data from a certificate object.

Method

```
public String getDataByPath(
   String thePath,
   boolean tagData,
   BooleanHolder encoded,
   ) throws UWIException;
```

Parameters

Expression	Type	Description
thePath	**String**	The path to the data you want to retrieve. See the Notes section below for more information on data paths.
tagData	**boolean**	true if the path should be prepended to the data, or false if not. If the path is prepended, an equals sign (=) is used as a separator. For example, suppose the path is ″Signing Cert: Issuer: CN″ and the data is ″IBM″. If true, the path will be prepended, producing ″CN=IBM″. If false, the path will not be prepended, and the result will be ″IBM″.
encoded	**BooleanHolder**	true if the return data is base 64 encoded, or false if not. The function returns binary data in base 64 encoding.

Notes

About Data Paths

Data paths describe the location of information within a certificate, just like file paths describe the location of files on a disk. You describe the path with a series of

colon separated tags. Each tag represents either a piece of data, or an object that contains further pieces of data (just like directories can contain files and subdirectories).

For example, to retrieve the version of a certificate, you would use the following data path:

```
version
```

However, to retrieve the subject's common name, you first need to locate thesigning certificate, then the subject, then the common name within the subject, as follows:

```
SigningCert: Subject: CN
```

Some tags may contain more than one piece of information. For example, the issuer's organizational unit may contain a number of entries. You can either retrieve all of the entries as a comma separated list, or you can specify a specific entry by using a zero-based element number.

For example, the following path would retrieve a comma separated list:

```
Issuer: OU
```

While adding an element number of 0 would retrieve the first organizational unit in the list, as shown:

```
Issuer: OU: 0
```

Certificate Tags

The following table lists the tags available in a certificate object:

Tag	Description
Subject	The subjects distinguished name. This is an object that contains further information, as detailed in *Distinguished Name Tags*.
Issuer	The issuer's distinguished name. This is an object that contains further information, as detailed in *Distinguished Name Tags*.
IssuerCert	The issuer's certificate. This is an object that contains the complete list of certificate tags.
Engine	The security engine that generated the certificate. This is an object that contains further information, as detailed in *Security Engine Tags*.
Version	The certificate version.
BeginDate	The date on which the certificate became valid.
EndDate	The date on which the certificate expires.
Serial	The certificates serial number.
SignatureAlg	The signature algorithm used to sign the certificate.
PublicKey	The certificates public key.
FriendlyName	The certificates friendly name.

Distinguished Name Tags

The following table lists the tags available in a distinguished name object:

Tag	Description
CN	The common name.
E	The e-mail address.
T	The title.
O	The organization.
OU	The organizational unit.
C	The country.
L	The locality.
ST	The state.
All	The entire distinguished name.

Security Engine Tags

The following table lists the tags available in the security engine object:

Tag	Description
Name	The name of the security engine used by the server.
Help	The help text for the security engine.
HashAlg	A hash algorithm supported by the security engine.

Returns

A string containing the certificate data (null if no data is found), or throws a generic exception (**UWIException**) if an error occurs.

Example

The following method uses **dereferenceEx** to locate a signature button in the form. It then uses **getCertificateList** to get a list of valid certificates for that button. Next, the method cycles through the returned certificates, uses **getDataByPath** to get the common name for each certificate, and identifies the certificate with a common name of ″Workplace Forms Server″. Then the method uses **signForm** to sign the form with the server's certificate.

```
public void serverSign(FormNodeP form) throws UWIException
{
IntHolder theStatus;
FormNodeP buttonNode;
Certificate [] certList;
Signature theSignature;
String signerCommonName;
boolean encodedResult;
int certCount;
int correctCert = -1;
int i;

   if ((buttonNode = theForm.dereferenceEx(null, "PAGE1.SIGBUTTON1",
      0, FormNodeP.UFL_ITEM_REFERENCE, null)) == null)
   {
      throw new UWIException("Could not locate SIGBUTTON1 node.");
   }

   theStatus = new IntHolder();
```

```
      certList = buttonNode.getCertificateList(null, theStatus);

      if (theStatus.value == SecurityUserStatusType.SUSTATUS_INPUT_REQUIRED)
      {
         throw new UWIException("User input required to sign form.");
      }

      certCount = certList.length;

      encodedResult = new BooleanHolder;

      for (i=0; i<certCount; i++)
      {
         signerCommonName = certList[i].getDataByPath(
            "SigningCert: Subject: CN", false, encodedResult);
         if (signerCommonName.equals("Workplace Forms Server"))
         {
            correctCert = i;
            break;
         }
      }

      if (correctCert == -1)
      {
         throw new UWIException("Could not locate required certificate");
      }

      theSignature = buttonNode.signForm(certList[correctCert], null,
         theStatus);

      if (theStatus.value == SUSTATUS_INPUT_REQUIRED)
      {
         throw new UWIException("User input required to sign form.");
      }
   }
```

getIssuer

Description

This method extracts the issuer certificate from the certificate provided.

Method

```
public Certificate getIssuer(
   IntHolder theStatus
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theStatus	**IntHolder**	A holder that is set with the status of the operation. This will be one of the following: **SecurityUserStatusType.SUSTATUS_OK** — The operation was successful. **SecurityUserStatusType.SUSTATUS_ CANCELLED** — the operation was cancelled by the user. **SecurityUserStatusType.SUSTATUS_INPUT_ REQUIRED** — the operation required user input, but could not receive it (for example, it was run on a server with no user).

Returns

The issuer certificate.

Example

The following example gets the signing certificate from a signature object, then iterates through the certificate issuers until it reaches the end of the chain. During the iteration, each certificate is passed to a method that processes them.

```
public void processCertChain(Signature theSig)
{
Certificate theCert, issuerCert;
IntHolder theStatus;

   /* Get the signing certificate from the signature. */
   theCert = theSig.getSigningCert();

   /* Loop through the certificate chain, passing each certificate to the
      ProcessCert function. The loop ends when the issuer certificate is
      null. */

   while (theCert != null)
   {

      /* Pass the certificate to the processCert method.  Note that
         this is not an API method, but rather a method you would
         write to process the certificate in some way. */

      ProcessCert(theCert);

      /* Get the issuer certificate from theCert. */

      issuerCert = theCert.getIssuer(theStatus);

      /* Check to ensure the method exited with the correct status. */

      if (theStatus.value != SecurityUserStatusType.SUSTATUS_OK)
      {
         throw new UWIException("getBlob exited with the wrong status.");
      }

      /* Assign theCert to equal the issuerCert for next iteration of the
         loop. */
```

```
        theCert = issuerCert;
    }
    return(OK);
}
```

The DTK Class

The **DTK** class encapsulates a method that initializes the API.

- You must import the following class to any .java files that call this **DTK** method:

  ```
  com.PureEdge.DTK
  ```

- Many of the methods used in the API will throw a generic exception called a **UWIException** if an error occurs. Import the following class to any .java files that call Form methods:

  ```
  com.PureEdge.error.UWIException
  ```

- Before using any Form methods you must first initialize the Form Library. Use the **initialize** method to perform this initialization.

initialize

Description

This method initializes the API. The parameters specify which version of the API your application should bind with (see the Notes below for more details).

You must call this method before calling any of the other methods in the API.

Method

```
public static void initialize(
   String progName,
   String progVer,
   String apiVer
   ) throws UWIException;
```

Parameters

Expression	Type	Description
progName	**String**	The name of the application calling **initialize**. This name is used to identify the application within the .ini file. It also sets the name that is returned by the XFDL **applicationName** function.
progVer	**String**	The version number of the application calling **initialize**. If the .ini file has an entry for this version of the application, the application will bind to the version of the API listed in that entry.
apiVer	**String**	The version number of the API the application should use by default. If the .ini file does not contain an entry for the specific application, the application will bind to the API specified by this parameter.

Returns

Nothing if call is successful or throws a generic exception (**UWIException**) if an error occurs.

Notes

initialize is a static method belonging to the **DTK** class, to use this method call:

```
DTK.initialize("progName", "progVer", "apiVer")
```

About Binding Your Applications to the API

When you initialize the API, the **initialize** method determines which version of the API to use based on the parameters you pass it. This allows you to exercise a great deal of control over which version of the API is used by your applications, and prevents the problems normally associated with common DLL files (often referred to as ″DLL hell″).

initialize uses a configuration file to determine which version of the API will bind to any application. This allows multiple versions of the API to co-exist on your computer, and ensures that your applications use the correct version of the API.

The configuration file is called PureEdgeAPI.ini and is installed with the API. Refer to the *IBM Workplace Forms Server - API Installation and Setup Guide* for the exact location of the file.

Note: You should redistribute the PureEdgeAPI.ini file with any applications that use the API. See the *IBM Workplace Forms Server - API Installation and Setup Guide* for more information about redistributing applications.

The configuration file contains a section for each application that might call the API, plus a default "API" section. Each section contains a list of version numbers in the following format:

```
<version of application> = <folder containing appropriate version of API>
```

For example, the configuration file might look like this:

```
[API]
5.1.0 = 51
5.0.0 = 50
[CustomApplication]
1.1.0 = 51
1.0.0 = 50
```

In this case, the folder indicated on the right hand side of each statement is part of the relative path to the API, and assumes the API was installed in the default folder. For example, under Windows ″50″ would resolve to:

```
c:\WinNT\System32\PureEdge\50
```

You can also specify an absolute path by placing a drive letter before the path. For example, ″c:\50″ would resolve to:

```
c:\50\
```

When you initialize the API, you include three parameters in the initialization call:

- The name of your application (as it would appear in the configuration file).
- The version of your application.
- The version of the API that your application should bind to by default.

The initialization call will first check the configuration file to see if your application is listed. For example, using the configuration file above, if you make an initialization call for “CustomApplication” version “1.1.0”, then the application binds to the API in the “51” folder.

If your application is not listed in the configuration file, the initialization call uses the default version of the API. For example, using the configuration file above, if you declare “5.1.0” as the default API, then your application binds to the API in the “51” folder.

You can add your own entries to the configuration file before distributing it to your customers, or you can rely on the default API entries.

Note: **initialize** was introduced for version 4.5.0 of the API. Binding does not work in this manner for earlier versions of the API. Do not include earlier versions of the API in the configuration file.

Example

In the example below, **initialize** initializes the API for the application called **formSample.**

```
private static void initialize( ) throws UWIException
{
   DTK.initialize("formSample", "1.0.0", "5.1.0");
}
```

initializeWithLocale

Description

This method initializes the API. The parameters specify the default locale, and which version of the API your application should bind with (see the Notes below for more details).

You must call this method before calling any of the other methods in the API.

Method

```
public static void initializeWithLocale(
   String progName,
   String progVer,
   String apiVer
   String theLocale
   ) throws UWIException;
```

Parameters

Expression	Type	Description
progName	**String**	The name of the application calling **initializeWithLocale**. This name is used to identify the application within the .ini file. It also sets the name that is returned by the XFDL **applicationName** function.
progVer	**String**	The version number of the application calling **initializeWithLocale**. If the .ini file has an entry for this version of the application, the application will bind to the version of the API listed in that entry.

Expression	Type	Description
apiVer	**String**	The version number of the API the application should use by default. If the .ini file does not contain an entry for the specific application, the application will bind to the API specified by this parameter.
theLocale	**String**	The default locale of the application.

Returns

Nothing if call is successful or throws a generic exception (**UWIException**) if an error occurs.

Notes

initializeWithLocale is a static method belonging to the **DTK** class, to use this method call:

```
DTK.initializeWithLocale("progName", "progVer", "apiVer", "theLocale")
```

About Binding Your Applications to the API

When you initialize the API, the **initializeWithLocale** method determines which version of the API to use based on the parameters you pass it. This allows you to exercise a great deal of control over which version of the API is used by your applications, and prevents the problems normally associated with common DLL files (often referred to as "DLL hell").

initializeWithLocale uses a configuration file to determine which version of the API will bind to any application. This allows multiple versions of the API to co-exist on your computer, and ensures that your applications use the correct version of the API.

The configuration file is called PureEdgeAPI.ini and is installed with the API. Refer to the *IBM Workplace Forms Server - API Installation and Setup Guide* for the exact location of the file.

Note: You should redistribute the PureEdgeAPI.ini file with any applications that use the API. See the *IBM Workplace Forms Server - API Installation and Setup Guide* for more information about redistributing applications.

The configuration file contains a section for each application that might call the API, plus a default "API" section. Each section contains a list of version numbers in the following format:

```
<version of application> = <folder containing appropriate version of API>
```

For example, the configuration file might look like this:

```
[API]
2.6.1 = 70
2.6.1 = 70
[CustomApplication]
1.1.0 = 70
1.0.0 = 70
```

In this case, the folder indicated on the right hand side of each statement is part of the relative path to the API, and assumes the API was installed in the default folder. For example, under Windows "70" would resolve to:

```
c:\Program Files\IBM\Workplace Forms\Server\xx\API\redist
   \msc32\PureEdge\xx
```

You can also specify an absolute path by placing a drive letter before the path. For example, "c:\70" would resolve to:

```
c:\70\
```

When you initialize the API, you include three parameters in the initialization call:

- The name of your application (as it would appear in the configuration file).
- The version of your application.
- The version of the API that your application should bind to by default.

The initialization call will first check the configuration file to see if your application is listed. For example, using the configuration file above, if you make an initialization call for "CustomApplication" version "1.1.0", then the application binds to the API in the "70" folder.

If your application is not listed in the configuration file, the initialization call uses the default version of the API. For example, using the configuration file above, if you declare "2.6.0" as the default API, then your application binds to the API in the "70" folder.

You can add your own entries to the configuration file before distributing it to your customers, or you can rely on the default API entries.

Example

In the example below, **initializeWithLocale** initializes the API for the application called **formSample.**

```
private static void initialize( ) throws UWIException
{
   DTK.initializeWithLocale("formSample", "1.0.0", "5.1.0", "fr-FR");
}
```

The FormNodeP Class

The **FormNodeP** class applies to particular instances of a form and the items in that form.

- Each node in a form, regardless of its level in the node hierarchy, is represented by a **FormNodeP** object. For more information about the node structure of XFDL forms refer to "Overview of the Form Structure" on page 7.
- Any application that makes calls to the **FormNodeP** methods must first import the following class:

  ```
  com.PureEdge.xfdl.FormNodeP
  ```
- Many of the methods in the API will throw a generic exception called a **UWIException** if an error occurs. Import the following class to any .java files that call methods from the API.

  ```
  com.PureEdge.error.UWIException
  ```

FormNodeP Constants

The following table lists the constants that are used by the FormNodeP functions along with a short description of each constant:

Named Constants	Description
FormNodeP.UFL_DS_CERTEXPIRED	The certificate has expired.
FormNodeP.UFL_DS_CERTNOTFOUND	The certificate was not found.
FormNodeP.UFL_DS_CERTNOTTRUSTED	The certificate is no longer trusted.
FormNodeP.UFL_DS_CERTREVOKED	The certificate has been revoked.
FormNodeP.UFL_DS_F2MATCHSIGNER	The name in the form did not match the name in the signature.
FormNodeP.UFL_DS_HASHCOMPFAILED	The hash of the document did not match the hash in the signature.
FormNodeP.UFL_DS_ISSUERNOTFOUND	The issuer could not be found.
FormNodeP.UFL_DS_ISSUERSIGFAILED	The verification of the issuer's certificate failed.
FormNodeP.UFL_DS_SIGNATUREALTERED	The signature has been altered.
FormNodeP.UFL_DS_UNEXPECTED	Generic error.

addNamespace

Description

This method adds a namespace declaration to the node it is called on. Each namespace is defined in the form by a namespace declaration, as shown:

```
xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom"
```

Each namespace declaration defines both a prefix and a URI for the namespace. In this sample, the prefix for the XFDL namespace is *xfdl* and the URI is *http://www.ibm.com/xmlns/prod/XFDL/7.0*.

© Copyright IBM Corp. 2003, 2006

Tags within the form are assigned specific namespaces by using the defined prefix. For example, to declare that an option was in the custom namespace you would use the prefix *custom* as shown:

```
<field sid="testField">
   <custom:custom_option>value</custom:custom_option>
</field>
```

Method

```
public void addNamespace(
   String theURI,
   String thePrefix
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theURI	**String**	The namespace URI. For example: `http://www.ibm.com/xmlns/prod/XFDL/7.0`
thePrefix	**String**	The prefix for the namespace. For example, *xfdl*.

Returns

Nothing or throws a generic exception (**UWIException**) if an error occurs.

Example

The following method uses **addNamespace** to add a custom namespace to a form. It then locates the global item in the global page and adds a custom option to that item which marks the status of the form as "Processed".

```
private static void addStatus(FormNodeP theNode) throws Exception
{
XFDL theXFDL;

   /* Add the custom namespace to the form. */

   theNode.addNamespace("http://www.ibm.com/xmlns/prod/XFDL/Custom", "custom");

   /* Locate the global item in the global page so we can add a global
      option. */

   if (theNode = theNode.dereferenceEx(null, "global.global", 0,
      UFL_ITEM_REFERENCE, null) == null)
      throw new UWIException("Could not locate global.global node.");

   /* Get the XFDL object so we can create a new node. */

   if (theXFDL = (XFDL)IFXMan.lookupInterface(XFDL.XFDL_INTERFACE_NAME,
      XFDL.XFDL_CURRENT_VERSION, 0, null, null)) == null)
      throw new UWIException("Could not find XFDL interface.";

   /* Create a new option node as a child of the global item.  This node
      is created in the custom namespace, called "Status", and given a
      value of "Processed". */

   if (theNode = theXFDL.create(theNode, UFL_APPEND_CHILD, null,
      "Processed", null, "custom:Status") == null)
      throw new UWIException("Could not create Status node.");
}
```

checkValidFormats

Description

This method checks the format of all items in the form and returns the number of items whose format is invalid. You can also set the function to create a list of the invalid items.

This method does not support XForms nodes.

Method

```
public FormNodeP checkValidFormats()throws UWIException;
```

Parameters

There are no parameters for this method.

Returns

An array of nodes that represent items with invalid formats.

Example

The following method uses recursion to traverse the entire node structure and check whether the nodes have the correct format. This method assumes that you are passing in the root node of the form.

```
public FormNodeP void checkFormats throws Exception
{
FormNodeP theForm;
FormNodeP[] invalidItems;

   invalidItems = theForm.checkValidFormats();
   if (invalidItems == null || invalidItems.length == 0)
   {
      System.out.prinln("All items in the form have valid formats.");
   }
   else
   {
      for (int i=0; i<invalidItems.length; i++)
      {
         theReference = invalidItems[i].getReferenceEx(null, null, null, false);
         theValue = invalidItems[i].getLiteralByRefEx(getLiteralByRefEx(null,
            "value", 0, null, null);
         System.out.println("The item "+theReference+" has an invalid format.);
      }
   }
}
```

createCell

Description

Use this method to create a new cell item for a *combobox*, *list*, or *popup*. **createCell** adds one new cell to a specific *group* on a specific page in the form. Note that this method can only assign a name to the new cell; it cannot set the cell's *value*. To set the value of a cell, you must use the **setLiteralByRefEx** method.

This method is called from a page level node, and creates the new cell in that page. Note that you cannot call this method from the global page node.

Method

```
public FormNodeP createCell(
   String theCellName,
   String theGroupName
   )throws UWIException;
```

Parameters

Expression	Type	Description
theCellName	**String**	The name of the new cell being created.
theGroupName	**String**	The name of the *group* option to which the new cell will be added.

Returns

A **FormNodeP** containing the new cell or throws a generic exception (UWIException) if an error occurs.

Example

This sample code makes two calls to the **createCell**method to add two new cells to the same group:

```
private static void addColorCells (FormNodeP theForm) throws Exception
{
FormNodeP theCell;
FormNodeP thePage;

/* The FormNodeP called thePage contains the page in which the cell will
   be added. */

   thePage = (theForm.getChildren()).getNext();
   theCell = thePage.createCell("ORANGE_CELL","POPUP1_GROUP");

/* The call to setLiteralByRefEx assigns the value Orange to the new cell.*/

   theCell.setLiteralByRefEx(null, "value", 0, null, null, "Orange");
   theCell = thePage.createCell("PURPLE_CELL ", "POPUP1_GROUP");
   theCell.setLiteralByRefEx(null, "value", 0, null, null, "Purple");
}
```

deleteSignature

Description

This method deletes the specified digital signature in the form. For security reasons, the form must meet certain criteria before this is allowed. None of the following should be locked by another signature: the signature, its descendants, the associated signature button, and its signer option. If these criteria are met, then the signature's locks are removed, and the signature item is deleted. Then, and the signer of the associated signature button is set to empty ("").

Method

```
public void deleteSignature(
   FormNodeP signatureItem
   ) throws UWIException;
```

Parameters

Expression	Type	Description
signatureItem	**FormNodeP**	The signature node to delete.

Returns

Nothing if call is successful or throws a generic exception (**UWIException**) if an error occurs.

Notes

If the *signature* item contains a *layoutinfo* option, **deleteSignature** will not remove the entire signature from the form. Instead, the *signature* item and the *layoutinfo* option will remain. To completely delete the signature item, you must delete the remaining nodes manually by using **destroy** to delete the signature item.

Example

In the following example, **dereference** is used to locate the signature node. **deleteSignature** is then used to delete the signature from the form.

```
private static void deleteSignature(formNodeP theForm) throws UWIException
{
formNodeP tempNode;
boolean layoutinfo;

   /* Locate the signature node. */

   If((tempNode = theForm.dereferenceEx(null, "PAGE1.SIGITEM2", 0,
      formNodeP.UFL_ITEM_REFERENCE, null)) == null)
   {
      throw new UWIException("Could not locate SIGITEM2 node");
   }

   /* Check to see if the signature contains a layoutinfo option.  Set
      layoutinfo to true if it does or false if it does not. */

   layoutinfo = true;
   If(tempNode.dereferenceEx(null, "layoutinfo", 0,
      formNodeP.UFL_OPTION_REFERENCE, null) == null)
      layoutinfo = false;

   /* Delete the signature. */

   theForm.deleteSignature(tempNode);

   /* If the signature contained a layoutinfo option, destroy the
      remaining nodes. */

   if (layoutinfo == true)
   {
      tempNode.Destroy();
   }
}
```

dereferenceEx

Description

Use this function to locate a particular FormNodeP, to locate a cell in a particular group, or to locate a data item in a particular datagroup. The node that this method operates on is used as the starting point of the search.

Note: It is not necessary to call this method when you are using XForms. The replaceXFormsInstance and extractXFormsInstancemethods perform this task automatically.

Method

```
public FormNodeP dereferenceEx(
   String theScheme,
   String theReference,
   int theReferenceCode,
   int referenceType,
   FormNodeP theNSNode
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theScheme	**String**	Reserved. This must be null.
theReference	**String**	The reference string.
theReferenceCode	**int**	Reserved. This must be 0.
referenceType	**int**	One of the following constants: **FormNodeP.UFL_OPTION_REFERENCE** **FormNodeP.UFL_ITEM_REFERENCE** **FormNodeP.UFL_PAGE_REFERENCE** **FormNodeP.UFL_ARRAY_REFERENCE** **FormNodeP.UFL_GROUP_REFERENCE** **FormNodeP.UFL_DATAGROUP_REFERENCE** If it is an option or argument reference, bitwise **OR** (	) with one of: **FormNodeP.UFL_SEARCH** **FormNodeP.UFL_SEARCH_AND_CREATE** If it is a group or datagroup reference, bitwise **OR** (	) with one of: **FormNodeP.UFL_FIRST**
theNSNode	**FormNodeP**	A node that is used to resolve the namespaces in *theReference* parameter (see the note about namespace below). Use null if the node that this method is operating on has inherited the necessary namespaces.

Returns

The **FormNodeP** defined by the reference string or **null** if the referenced node does not exist and **UFL_SEARCH_AND_CREATE** is not specified. On error, the function throws a **UWIException** object that describes the problem.

Notes

FormNodeP

Before you decide which FormNodeP to use this method on, be sure you understand the following:

1. The FormNodeP supplied can never be more than one level in the hierarchy above the starting point of the reference string. For example, if the reference string begins with an option, then the FormNodeP can be no higher in the hierarchy than an item.
2. If the FormNodeP is at the same level or lower in the hierarchy than the starting point of the reference string, the method will attempt to locate a common ancestor. The method will locate the ancestor of the FormNodeP that is one level in the hierarchy above the starting point of the reference string. The method will then attempt to follow the reference string back down through the hierarchy. If the reference string cannot be followed from the located ancestor (for example, if the ancestor is not common to both the FormNodeP and the reference string), the function will fail. For example, given a FormNodeP that represents *field_1* and a reference of *field_2*, the function will access the *page* node above *field_1*, and will then try to locate *field_2* below that node. If the two fields were not on the same page, the function would fail.
3. dereferenceEx does not support the XForms scheme.

Creating a Reference String

For general information about creating a reference string, see "References" on page 8.

Reference strings for groups or datagroups follow this format:

```
page.group or page.datagroup
```

In both cases, the page component is optional, and is only required if you want to search a different page than the one containing your reference node.

For example, to refer to the "State" group of cells on PAGE1 of the form, you would use:

```
PAGE1.State
```

Locating Cells or Data Items

If you want to locate a cell or a data item, you must perform a bitwise OR with UFL_FIRST or UFL_NEXT. UFL_FIRST will locate the first cell or data item in the page. UFL_NEXT will locate the next cell or data item. This allows you to loop through all the cells or data item on a page until you have found the one you want.

Note that groups and datagroups are limited to a single page, and that your search will likewise be limited to a single page.

Creating a Node

For an option or argument reference, you can have the library create a node that does not exist. To do so, perform a bitwise OR of UFL_SEARCH_AND_CREATE to the *referenceType* parameter; otherwise, perform a bitwise OR of UFL_SEARCH to the *referenceType* variable and the function will return null if the node does not exist.

Determining Namespace

In some cases, you may want to use the **dereferenceEx**method to locate a node that does not have a globally defined namespace. For example, consider the following form:

```
<label sid="Label1">
   <value>Field1.processing:myValue</value>
</label>
<field sid="Field1" xmlns:processing="URI">
   <value></value>
   <processing:myValue>10<processing:myValue>
</field>
```

In this form, the *processing* namespace is declared in the *Field1* node. Any elements within *Field1* will understand that namespace; however, elements outside of the scope of *Field1* will not.

In cases like this, you will often start your search at a node that does not understand the namespace of the node you are trying to locate. For example, you might want to locate the node referenced in the value of *Label1*. In this case, you would first locate the *Label1* value node and get its literal. Then, from the *Label1* value node, you would attempt to locate the *processing:myValue* node as shown:

```
Label1Node.dereferenceEx(null, "Field1.processing:myValue", 0,
   FormNodeP.UFL_OPTION_REFERENCE, null)
```

In this example, the **dereferenceEx**method would fail. The method cannot properly resolve the *processing* namespace because this namespace is not defined for the *Label1* value node. To correct this, you must also provide a node that understands the *processing* namespace (in this case, any node in the scope of *Field1*) as the last parameter in the method:

```
Label1Node.dereferenceEx(null, "Field1.processing:myValue", 0,
   FormNodeP.UFL_OPTION_REFERENCE, Field1Node)
```

Example

The following sample code uses **dereferenceEx** to locate the node representing the field called colorfield. It then uses **getLiteralByRef** to change the value displayed by the field to Purple.

```
private static void changeColorField( ) throws Exception
{
FormNodeP tempNode;

   if ((tempNode = theForm.dereferenceEx(null, "PAGE1.COLORFIELD", 0,
      FormNodeP.UFL_ITEM_REFERENCE, null)) == null)
   {
      throw new UWIException("Could not locate COLORLABEL node.");
   }
```

```
   tempNode.setLiteralByRefEx(null, "PAGE1.COLORFIELD.VALUE", 0,
      null, null, "Purple");

   /* additional code removed */
}
```

destroy

Description

This function destroys the indicated FormNodeP. All children of the specified FormNodeP are also destroyed.

Method

```
public void destroy( ) throws UWIException;
```

Parameters

There are no parameters for this method.

Returns

Nothing if call is successful or throws a generic exception (**UWIException**) if an error occurs.

Notes

Digital Signatures

You cannot destroy a signed item, except in the case of destroying an entire signed form. Destroying a signed item breaks the digital signature, resulting in an exception.

Example

In the following example, **dereferenceEx** is used to locate a particular node. **destroy** is then used to remove that node from the structure.

```
private static void removeRadios( ) throws UWIException
{
FormNodeP tempNode;

   if ((tempNode = theForm.dereferenceEx(null, "PAGE1.MALERADIO", 0,
      FormNodeP.UFL_ITEM_REFERENCE, null)) == null)
   {
      throw new UWIException("Could not locate MALERADIO node.");
   }

   tempNode.destroy( );

   /* additional code removed */

}
```

duplicate

Description

This method makes a copy of a node. The duplicate node can be attached to any other node as either a sibling or a child, or can be stored as a separate node

structure (that is, as a separate form). The new node can also be assigned a new identifier, as indicated by the *theIdentifier* parameter. All of the properties of the original node are duplicated, including any children and any namespace settings.

Note: If you duplicate a node that is not in the XFDL namespace, the namespace is copied as part of the duplicated node, but is not set globally.

Method

```
public FormNodeP duplicate(
 FormNodeP baseNode,
 int where,
 String theIdentifier
 ) throws UWIException;
```

Parameters

Expression	Type	Description
baseNode	**FormNodeP**	The **formNodeP** to attach the new copy to. If null, then *origNode* is used as the *baseNode*.
where	**int**	A constant that describes the location in relation to the supplied '*baseNode*' in which the new node should be placed. Can be one of: **XFDL.UFL_APPEND_CHILD** — adds the new node as the last child of the '*baseNode*'. **XFDL.UFL_AFTER_SIBLING** — adds the new node as a sibling of the '*baseNode*', placing it immediately after that node in the form structure. **XFDL.UFL_BEFORE_SIBLING** — adds the new node as a sibling of the '*baseNode*', placing it immediately before that node in the form structure. **XFDL.UFL_ORPHAN** — copies the node to a new form structure, effectively creating a separate form.
theIdentifier	**String**	A new identifier for this node. If null, the same identifier that was on the original node is used.

Returns

The duplicate node or throws a generic exception (**UWIException**) if an error occurs.

Example

In the following example, **dereferenceEx** is used to locate a specific node. **duplicate** is then used to duplicate that node.

```
private static void createMailing( )throws UWIException
{
FormNodeP tempNode;
FormNodeP duplicateNode;

   if ((tempNode = theForm.dereferenceEx(null, "PAGE1.ADDRESSFIELD",
      0, FormNodeP.UFL_ITEM_REFERENCE, null)) == null)
   {
      throw new UWIException("Could not locate ADDRESSFIELD node.");
   }
```

```
    if ((duplicateNode = tempNode.duplicate(tempNode,
       XFDL.UFL_AFTER_SIBLING, "MAILINGFIELD")) == null)
    {
       throw new UWIException("Could not duplicate ADDRESSFIELD node.");
    }

 }
```

encloseFile

Description

This method encloses a file in a form. The file must be accessible on the local computer. The FormNodeP may refer to either a page node or an item node. If the FormNodeP is a page node, the method creates a data item in that page to contain the enclosure. If the FormNodeP is an item node, it must be a data item, and the method encloses the file in that node.

The file is enclosed using base64-gzip encoding.

Method

```
READING A FILE:
   public FormNodeP encloseFile(
      String theFile,
      String mimeType,
      String dataGroup,
      String identifier
      ) throws UWIException;
```

Parameters

Expression	Type	Description
theFile	**String**	The path to the file on the local drive to enclose.
mimeType	**String**	The MIME type of the file. If null, the library will attempt to find a suitable MIME type for the file.
dataGroup	**String**	The data group to which this file should belong. If the *aNode* parameter is a page node, you must provide this parameter. If *aNode* parameter is an item node, you may use null to keep the current *datagroup* option or provide a different value to overwrite the option.
theIdentifier	**String**	The identifier to assign to the new data item if one is created. If null, either the current name is used or a unique name is automatically generated for the new data item.

Returns

The **FormNodeP** of the item that contains the enclosure or throws a generic exception (**UWIException**) if an error occurs.

Example

The following example demonstrates how to use **encloseFile** to enclose a graphics file in a form. First, **dereferenceEx** is used to locate the node for the first page. Then, depending on the gender, **encloseFile** is called to enclose one of two possible

image files. Because the subject node is a page node, **encloseFile** creates a new data node in which to store the image file.

```
   private static void enclosePic(String theGender) throws Exception
   {
      FormNodeP tempNode;

      if ((tempNode = theForm.dereferenceEx(null, "PAGE1", 0,
         FormNodeP.UFL_PAGE_REFERENCE, null)) == null)
         throw new Exception("Could not find PAGE1.");

/* The following logic detemines whether the gender is "male" or "female". */

      if (theGender.equals("male"))
      {
         if ((tempNode = tempNode.encloseFile("male.jpg", "image/jpeg",
            null, "PICDATA")) == null)
            throw new Exception("Could not enclose image file.");
      }
      else
      {

   /* This call to encloseFile is similar to the previous one. The only
      difference is that it specifies a different image. */

         if ((tempNode = tempNode.encloseFile("female.jpg", "image/jpeg",
            null, "PICDATA")) == null)
            throw new Exception("Could not enclose image file.");
      }
   }
```

encloseInstance

Description

This method modifies one instance in the data model, either updating information or appending information. Note that the form must have an existing data model.

Call this method on the root node of the form or an XML instance node.

Note: Use caution when calling this method. It can be used to overwrite signed instance data.

Method

READING A FILE:

```
   public void encloseInstance(
      String theInstanceID,
      String theFile,
      int theFlags,
      String theScheme,
      String theRootReference,
      FormNodeP theNSNode,
      boolean replaceNode
      ) throws UWIException;
```

READING A STREAM:

```
   public void encloseInstance(
      String theInstanceID,
      java.io.InputStream theStream,
      int theFlags,
      String theScheme,
```

```
      String theRootReference,
      FormNodeP theNSNode,
      boolean replaceNode
      ) throws UWIException;
```

READING FROM THE READER:

```
   public void encloseInstance(
      String theInstanceID,
      java.io.Reader  theReader,
      int theFlags,
      String theScheme,
      String theRootReference,
      FormNodeP theNSNode,
      boolean replaceNode
      ) throws UWIException;
```

Parameters

Expression	Type	Description
theInstanceID	**String**	The ID of the instance node to create or replace. This is defined by the *id* attribute of that node, and is case sensitive. If *theNode* parameter is the instance node you want to replace, set this parameter to null.
theFile	**String**	The path to the file on the local drive that contains the XML instance.
theStream	**java.io.InputStream**	The input stream that will read the instance data. Note that the data must be UTF-8.
theReader	**java.io.Reader**	The Java Reader that will read the instance data.
theFlags	**int**	Reserved. Must be 0.
theScheme	**String**	Reserved. Must be null.
theRoot Reference	**String**	A reference to the node you want to replace or append children to. This reference is relative to the instance node. Use null to default to the instance node.
theNSNode	**FormNodeP**	A node that inherits the namespaces used in the reference. This node defines the namespaces for the method. Use null if the node that this method is operating on has inherited the necessary namespaces.
replaceNode	**boolean**	If true, the node specified by *theRootReference* is replaced with data. If false, the data is appended as the last child of *theRootReference* node.

Returns

Nothing if call is successful or throws a generic exception (**UWIException**) if an error occurs.

Example

The following example shows a method that takes the root node of a form and updates the XML instance called ″data″.

```
private static void updateDataInstance(FormNodeP theForm) throws Exception
{
   theForm.encloseInstance("data",
      "c:\Instance Files\Personnel\tempdata.dat", 0, null, null, null,
      true);
}
```

extractFile

Description

This method will extract an enclosure contained in a node and save it to a file on the local computer. Note that this method does not remove the enclosure from the form.

Method

```
public void extractFile(
   String theFile
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theFile	**String**	The path showing where to store the file on the local drive. Any existing file will be overwritten.

Returns

Nothing if call is successful or throws a generic exception (**UWIException**) if an error occurs.

Example

In the following example, **dereferenceEx** is used to locate a specific data item node. **extractFile** is then used to write the image data to the local drive.

```
private static void exportImage(FormNodeP theForm) throws Exception
{
FormNodeP tempNode;

   if ((tempNode = theForm.dereferenceEx(null, "PAGE1.LOGODATA", 0,
      FormNodeP.UFL_ITEM_REFERENCE, null)) == null)
   {
      throw new UWIException("Could not find LOGODATA node.");
   }

   tempNode.extractFile("logo.jpg");
}
```

extractInstance

Description

This method copies an instance from a form's XML model to a file. Note that this method does not remove the instance from the form.

Call this method on the root node of the form or an XML instance node.

Method

WRITING TO A FILE:

```
public byte[] extractInstance(
   String theInstanceID,
   FormNodeP theFilter,
   String includedNamespaces,
   String theFile,
   int theFlags,
   String theScheme,
   String theRootReference,
   FormNodeP theNSNode
   ) throws UWIException;
```

WRITING TO A STREAM:

```
public void extractInstance(
   String theInstanceID,
   FormNodeP theFilter,
   String includedNamespaces,
   java.io.OutputStream theStream,
   int theFlags,
   String theScheme,
   String theRootReference,
   FormNodeP theNSNode
   ) throws UWIException;
```

WRITING TO A WRITER:

```
public void extractInstance(
   String theInstanceID,
   FormNodeP theFilter,
   String includedNamespaces,
   java.io.Writer theWriter,
   int theFlags,
   String theScheme,
   String theRootReference,
   FormNodeP theNSNode
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theInstanceID	**String**	The ID of the instance node to extract. This is defined by the *id* attribute of that node. If *theNode* parameter is the instance node you want to extract, set this parameter to **null**.

Expression	Type	Description
theFilter	**FormNodeP**	An item in the form, such as a button or cell, that defines the filtering for the instance. Filtering of elements is controlled by the transmit filters in the item. If all of an element's bound options are filtered out, then the element is also filtered out. Use **null** for no filtering.
included Namespaces	**String**	If set to null, a definition for each inherited namespace is added to the root node of the instance when it is extracted. To filter the namespaces, list the prefixes for those namespaces you want to include in the instance, separated by spaces. For example, to include only the *xfdl* and *custom* namespaces, you would set this parameter to: `xfdl custom` Use #*default* to indicate the default namespace for the instance. Use an empty string ("") to include only those namespaces that are used by the instance. Namespaces that are used in the instance are always included, regardless of this setting.
theFile	**String**	The path to the file on the local drive that will contain the XML instance.
theStream	**java.io.OutputStream**	The output stream that will write the data instance. Note that the data must be UTF-8.
theWriter	**java.io.Writer**	The Java Writer that will write the data instance.
theFlags	**int**	Reserved. This must be 0.
theScheme	**String**	Reserved. Must be null.
theRootReference	**String**	A reference to the root node you want to extract. This reference is relative to the instance node. Use null to default to the instance node.
theNSNode	**FormNodeP**	A node that inherits the namespaces used in the reference. This node defines the namespaces for the method. Use null if the node that this method is operating on has inherited the necessary namespaces.

Returns

Nothing if call is successful or throws a generic exception (**UWIException**) if an error occurs.

Example

The following example shows a method that takes the root node of a form and extracts an XML instance.

```
private static void updateDataInstance(formNodeP theForm)
{
   theForm.extractInstance("data", null, null,
      "c:\Instance Files\Personnel\tempdata.dat", 0, null, null, null);
}
```

extractXFormsInstance

Description

This method copies an XForms instance to a file or a memory block. This method does not remove the instance from the form.

Call this method on the root node of the form or an instance node.

Note: This method automatically updates the XForms data model.

Method

WRITING TO A FILE:

```
public byte[] extractXFormsInstance(
   String theModelID,
   String theNodeRef,
   boolean writeRelevant,
   boolean ignoreFailures,
   FormNodeP theNSNode,
   String theFilename,
   ) throws UWIException;
```

WRITING TO A STREAM:

```
public void extractXFormsInstance(
   String theModelID,
   String theNodeRef,
   boolean writeRelevant,
   boolean ignorefailures,
   FormNodeP theNSNode,
   java.io.OutputStream theStream,
   ) throws UWIException;
```

WRITING TO A WRITER:

```
public void extractXFormsInstance(
   String theModelID,
   String theNodeRef,
   boolean writeRelevant,
   boolean ignorefailures,
   FormNodeP theNSNode,
   java.io.Writer theWriter,
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theModelID	**String**	The ID of the model to extract. Use null to extract the default model.
theNodeRef	**String**	An XPath reference to a node in the instance. This node and all of its children are copied. Leave blank to extract the entire instance.
writeRelevant	**boolean**	If true, writes only relevant instance data.

Expression	Type	Description
ignoreFailures	**boolean**	If true, ignores constraint or validation failures.
theNSNode	**FormNodeP**	A node that inherits the namespaces used in the reference. This node defines the namespaces for the method. Use null if the node that this method is operating on has inherited the necessary namespaces.
theFilename	**String**	The name and path of the file to write to. Use null to write to the output memory block.
theStream	**java.io.OutputStream**	The output stream that will write the XForms data instance. Note that the data must be UTF-8.
theWriter	**java.io.Writer**	The Java Writer that will write the XForms data instance.

Returns

Nothing if call is successful or throws a generic exception (**UWIException**) if an error occurs.

Example

The following example shows a method that takes the root node of a form, extracts an XForms instance, and writes it to a file called ″InstanceData.xml″.

```
private static void updateDataInstance(java.io.Writer theWriter)
{
   theForm.extractXFormsInstance("model1",
      "instance('instance1')loanrecord/user_personal_info", true, false, null,
       "c:\InstanceData.xml");
}
```

getAttribute

Description

This method returns the value of a specific attribute for a node. For example, the following XFDL represents a MIME data node:

```
<mimedata encoding="base64"></mimedata>
```

In this sample, you could use **getAttribute** to obtain the value of the encoding attribute, which would be ″base64″.

Method

```
public String getAttribute(
   String theNamespaceURI,
   String theAttribute
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theNamespaceURI	**String**	The namespace URI for the attribute. For example: `http://www.ibm.com/xmlns/prod/XFDL/7.0`
theAttribute	**String**	The local name of the attribute. For example, *encoding*.

Returns

The attribute's value or throws a generic exception (**UWIException**) if an error occurs. If the attribute is empty or does not exist, the method returns **null**.

Notes

Namespaces

If you refer to an attribute with a namespace prefix, *getAttribute* first looks for a complete match, including both prefix and attribute name. If it does not find such a match, it will look for a matching attribute name that has no prefix but whose containing element has the same namespace.

For example, assume that the *custom* namespace and the *test* namespace both resolve to the same URI. In the following case, looking for the *id* attribute would locate the second attribute (test:id), since it has an explicit namespace declaration:

```
<a xmlns:custom="ABC" xmlns:test="ABC">
   <custom:myElement id="1" test:id="2">
</a>
```

However, in the next case, the *id* attribute does not have an explicit namespace declaration. Instead, it inherits the custom namespace. However, since the inherited namespace resolves to the same URI, the *id* attribute is still located:

```
<custom:myElement id="1">
```

Special Attributes

Forms generally use three special attributes that are not in an explicitly defined namespace and which require special commands to retrieve.

The first is the default namespace attribute, which looks like this:

```
xmlns="http://www.ibm.com/xmlns/prod/XFDL/7.0"
```

To retrieve this attribute, you must use a namespace URI of null and the attribute name *xmlns*.

The second special attribute is a namespace declaration, which looks like this:

```
xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom"
```

To retrieve this sort of attribute, you must use the namespace URI *http://www.w3.org/2000/xmlns* and the appropriate attribute name, such as *custom*.

Finally, there is the language attribute, which looks like this:

```
xml:lang="en-GB"
```

To retrieve this sort of attribute, you must use the namespace URI *http://www.w3.org/XML/1998/namespace* and the attribute name *lang*.

Example

The following example shows a shortcut method that gets the value of the encoding attribute for a specific node. A node is passed to the method which then uses **getAttribute** to get the value of encoding attribute. This sample method assumes that the attribute is always in the XFDL namespace.

```
private static String getEncodingType(FormNodeP theNode) throws Exception
{
String theEncodingType;

   theEncodingType = theNode.getAttribute(
      "http://www.ibm.com/xmlns/prod/XFDL/7.0", "encoding")
   return(theEncodingType);
}
```

getAttributeList

Description

This method returns a list of attributes and a list of corresponding namespaces for a given node. For example, the following XFDL represents a *mimedata* node:

```
<mimedata encoding="base64"></mimedata>
```

In this sample, *getAttributeList* would return a list of attributes that contained *encoding* and a list of namespaces that contained *http://www.ibm.com/xmlns/prod/XFDL/7.0*.

Method

```
public void getAttributeList(
   StringListHolder theNamespaces,
   StringListHolder theAttributeList
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theNamespaces	**StringListHolder**	A pointer that contains a list of namespace URIs. For example: `http://www.ibm.com/xmlns/prod/XFDL/7.0` Each URI corresponds to the attribute in the same position in the attribute list.
theAttributeList	**StringListHolder**	A list of attributes. For example, *compute*, *encoding*, and so on. Each attribute corresponds to a URI in the same position in the namespace list.

Returns

Nothing or throws a generic exception (**UWIException**) if an error occurs.

Example

The following method uses **getAttributeList** to retrieve the list of a node's attributes. It then searches through the list looking for a compute attribute. When if locates a compute attribute, it uses **removeAttribute** to remove the compute from the node.

```
private static void stripComputes(FormNodeP theNode) throws Exception
{
int counter;
StringListHolder URIList = new StringListHolder[];
StringListHolder attributeList = new StringListHolder[];
   /* Retrieve the list of attributes for the supplied node. */
   theNode.getAttributeList(URIList, attributeList);
   /* Step through the list searching for the compute attribute. If the
      compute attributes is found, delete it. */
   for (counter = 0; counter < attributeList.value.length; counter++)
   {
      if (attributeList.value[counter].equals("compute"))
      {
         theNode.removeAttribute(URIList.value[counter],
            attributeList.value[counter]);
      }
   }
}
```

getCertificateList

Description

This method locates all available certificates that can be used by a particular signature button. The certificates are filtered according to the signature engine defined in the *signformat* option of the button, and according to the filters defined in the *signdetails* option of the button.

This method returns the valid certificates in an undetermined order. This means that you cannot rely on the certificates being listed in the same order each time you call this method.

Method

```
public Certificate [ ] getCertificateList(
   String theFilters,
   IntHolder theStatus,
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theFilters	**String**	A string that is used to filter the subject attribute of the certificate. If the subject attribute include this substring, then that certificate will be listed. For example, you might filter against a name, such as "John Doe", or an e-mail address, such as "jdoe@ibm.com". Note that this filter is in addition to the other filters defined in the *signdetails* option of the button. If null is passed, then only the filters in the *signdetails* option are used.
theStatus	**IntHolder**	This is a status flag that reports whether the operation was successful. Possible values are: **SecurityUserStatusType.SUSTATUS_OK** — the operation was successful. **SecurityUserStatusType.SUSTATUS_CANCELLED** — the operation was cancelled by the user. **SecurityUserStatusType.SUSTATUS_INPUT_REQUIRED** — the operation required user input, but could not receive it (for example, it was run on a server with no user).

Returns

An array containing the list of certificates objects.

Example

In the following example, **dereferenceEx** is used to locate a specific signature button node. **getCertificateList** is then used to get a list of valid certificates for that button. Next, **getDataByPath** is used to search the certificate list for the Workplace Forms Server certificate, which **signForm** then uses to sign the button.

```
private static void createSignature(FormNodeP theForm)throws Exception
{
FormNodeP buttonNode;
IntHolder theStatus;
Signature theSignature;
Certificate [] certList;
String signerCommonName;
boolean encodedResult;
int correctCert = -1;

   if ((buttonNode = theForm.dereferenceEx(null, "PAGE1.SIGBUTTON1",
      0, FormNodeP.UFL_ITEM_REFERENCE, null)) == null)
   {
      throw new UWIException("Could not locate SIGBUTTON1 node.");
   }

   theStatus = new IntHolder();

   certList = buttonNode.getCertificateList(null, theStatus);
```

```
      if (theStatus.value == securityUserStatusType.SUSTATUS_INPUT_REQUIRED)
      {
         throw new UWIException("User input required to sign form.");
      }

      certCount = certList.length;
      encodedResult = new BooleanHolder;

      for (i=0; i<certCount; i++)
      {
         signerCommonName = certList[i].getDataByPath(
            "SigningCert: Subject: CN", false, encodedResult);
         if (signerCommonName.equals("Workplace Forms Server"))
         {
            correctCert = i;
            break;
         }
      }

      if (correctCert == -1)
      {
         throw new UWIException("Could not locate required certificate");
      }

      theSignature = buttonNode.signForm(certList[correctCert], null,
         theStatus);

      if (theStatus.value == securityUserStatusType.SUSTATUS_INPUT_REQUIRED)
      {
         throw new UWIException("User input required to sign form.");
      }
   }
```

getChildren

Description

This method, along with **getParent**, is used to traverse vertically along the form hierarchy. **getChildren** returns the first child of the indicated node. If the node has no children, null is returned. All children of a particular FormNodeP can be traversed using an iterator, such as a while loop, in combination with **getNext**.

Method

```
public FormNodeP getChildren( ) throws UWIException;
```

Parameters

There are no parameters for this method.

Returns

The **FormNodeP** that represents the child or **null** if no such child exists. It will throw a generic exception (**UWIException**) if an error occurs.

Example

In the following example the root node of a form is represented by a **FormNodeP** called **theForm**. The method **dereferenceEx** is used to retrieve an item from the form called PAGE1.NAMELABEL.

getChildren returns the first child node of *PAGE1.NAMELABEL* that is *PAGE1.NAMELABEL.value*.

```
public class getFunctions
{
private static FormNodeP theForm;
private static FormNodeP childNode;

   /* Additional Code Removed */

   public static void main(String argv[])
   {
   FormNodeP tempNode;

      /* Additional Code Removed */

      if ((tempNode = theForm.dereferenceEx(null, "PAGE1.NAMELABEL",
         0,FormNodeP.UFL_ITEM_REFERENCE, null)) == null)
      {
         throw new UWIException("Could not locate Name label node.");
      }

      childNode = tempNode.getChildren( );
      childNode.setLiteralEx(null, "The value option is the first ");

      /* Additional Code Removed */
   }
}
```

getFormVersion

Description

This method determines the XFDL version of a form. You can call this method on any form node that is in the XFDL namespace.

Method

```
public int getFormVersion() throws UWIException;
```

Parameters

There are no parameters for this method.

Returns

An integer in the form of 0xMMmm0300, where MM is the major number and mm is the minor number. For example, a version 6.3 form would return: 0x06030300.

Example

The following method accepts a form node and returns a boolean that indicates whether the form is version 6.5 or higher.

```
private static Boolean checkVersion(formNodeP theNode) throws Exception
{
   if (theNode.getFormVersion() >= 0x06050300)
   {
      return(true);
   }
   else
```

```
    {
       return(false);
    }
}
```

getInfoEx

Description

This method retrieves information about a particular node. If you do not want information about a particular property, simply set it to **null**.

Method

```
public void getInfoEx(
   StringHolder theType,
   StringHolder theLiteral,
   StringHolder theFormula,
   StringHolder theIdentifier,
   String theCharSet)
   throws UWIException;
```

Parameters

Expression	Type	Description
theType	***StringHolder***	A StringHolder that will store the type of the **FormNodeP**. If the type is empty or does not exist, the string is set to null.
theLiteral	***StringHolder***	A StringHolder that will store the literal of the **FormNodeP**. If the literal is empty or does not exist, the string is set to null.
theFormula	***StringHolder***	A StringHolder that will store the formula of the **FormNodeP**. If the formula is empty or does not exist, the string is set to null.
theIdentifier	***StringHolder***	A StringHolder that will store the identifier of the **FormNodeP**. If the identifier is empty or does not exist, the string is set to null.
theCharSet	***String***	The character set you want to use to view the results. Use **null** or **Unicode** for Unicode. Use **Symbol** for Symbol.

Returns

Nothing if call is successful or throws a generic exception (**UWIException**) if an error occurs.

Notes

If you are getting information about a node that is not in the XFDL namespace, **getInfoEx** may return values that include namespace prefixes as follows:

- Any item node in a non-XFDL namespace will return a *Type* that includes a namespace prefix. For example, *myNamespace:Field1*.
- Any option node in a non-XFDL namespace will return an *Identifier* that includes a namespace prefix. For example, *myNamespace:value*.

Example

In the following example, **dereferenceEx** is used to locate a specific node. **getInfoEx** is then used to get the four values from that node. The four values are then printed out.

```
private static void checkAgeFieldNode(FormNodeP theForm) throws Exception
{
FormNodeP tempNode;
StringHolder theType = new StringHolder( );
StringHolder theLiteral = new StringHolder( );
StringHolder theFormula = new StringHolder( );
StringHolder theIdentifier = new StringHolder( );
      if ((tempNode =             theForm.dereferenceEx(null, "PAGE1.AGEFIELD", 0,
         FormNodeP.UFL_ITEM_REFERENCE, null)) == null)
      {
         throw new UWIException("Could not locate AGEFIELD node.");
      }
      tempNode.getInfoEx(theType, theLiteral, theFormula, theIdentifier,
         null);
      /* Print out the information. */
      System.out.println("Type: " + theType.value);
      System.out.println("Literal: " + theLiteral.value);
      System.out.println("Formula: " + theFormula.value);
      System.out.println("Identifier: " + theIdentifier.value);
}
```

getLiteralByRefEx

Description

This method finds a particular FormNodeP on the basis of a reference string. The node you call this method on is used as the starting point for the search unless you provide an absolute reference. Once the FormNodeP is found, its literal is retrieved.

Note: It is not necessary to call this method when you are using XForms. The replaceXFormsInstance and extractXFormsInstancemethods perform this task automatically.

Method

```
public String getLiteralByRefEx(
   String theScheme,
   String theReference,
   int theReferenceCode
   String theCharSet,
   FormNodeP theNSNode
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theScheme	**String**	Reserved. This must be null.
theReference	**String**	The reference string.

Expression	Type	Description
theReferenceCode	**int**	Reserved. This must be 0.
theCharSet	**String**	The character set you want to use to view the literal string. Use **null** or **Unicode** for Unicode. Use **Symbol** for Symbol.
theNSNode	**FormNodeP**	A node that is used to resolve the namespaces in *theReference* parameter (see the note about namespace below). Use null if the node that this method is operating on has inherited the necessary namespaces.

Returns

The literal string or throws a generic exception (**UWIException**) if an error occurs. If the literal is empty or does not exist, the method returns **null**.

Notes

This method is a shortcut method and is equivalent to performing the following on a **FormNodeP** object:

```
aNode.dereferenceEx(theScheme, theReference, theReferenceCode,
   UFL_OPTION_REFERENCE | UFL_SEARCH_AND_CREATE,
   theNamespaceNode).getLiteralEx(aCharSet);
```

FormNodeP

Before you decide which FormNodeP to call the method on, be sure you understand the following:

1. The FormNodeP supplied can never be more than one level in the hierarchy above the starting point of the reference string. For example, if the reference string begins with an option, then the FormNodeP can be no higher in the hierarchy than an item.
2. If the FormNodeP is at the same level or lower in the hierarchy than the starting point of the reference string, the method will attempt to locate a common ancestor. The method will locate the ancestor of the FormNodeP that is one level in the hierarchy above the starting point of the reference string. The method will then attempt to follow the reference string back down through the hierarchy. If the reference string cannot be followed from the located ancestor (for example, if the ancestor is not common to both the FormNodeP and the reference string), the method will fail.

   For example, given a FormNodeP that represents ″field_1″ and a reference of ″field_2″, the method will access the ″page″ node above ″field_1″, and will then try to locate ″field_2″ below that node. If the two fields are not on the same page, the method will fail.
3. If the FormNodeP is at the argument level, the search will not start from that point. Instead, the nearest ancestor that is at the option level will be used as the starting point for the search.

Creating a Reference String

For more information about creating a reference, see “References” on page 8.

Determining Namespace

In some cases, you may want to use the **getLiteralByRefEx** method to get the literal of a node that does not have a globally defined namespace. For example, consider the following form:

```
<label sid="Label1">
   <value>Field1.processing:myValue</value>
</label>
<field sid="Field1" xmlns:processing="URI">
   <value></value>
   <processing:myValue>10<processing:myValue>
</field>
```

In this form, the *processing* namespace is declared in the *Field1* node. Any elements within *Field1* will understand that namespace; however, elements outside of the scope of *Field1* will not.

In cases like this, you will often start your search at a node that does not understand the namespace of the node you are trying to locate. For example, you might want to locate the node referenced in the value of *Label1*. In this case, you would first locate the *Label1* value node and get its literal. Then, from the *Label1* value node, you would attempt to locate the *processing:myValue* node as shown:

```
Label1Node.getLiteralByRefEx(null, "Field1.processing:myValue", 0,
   null, null)
```

In this example, the **getLiteralByRefEx** method would fail. The method cannot properly resolve the *processing* namespace because this namespace is not defined for the *Label1* value node. To correct this, you must also provide a node that understands the *processing* namespace (in this case, any node in the scope of *Field1*) as a parameter in the method:

```
Label1Node.getLiteralByRefEx(null, "Field1.processing:myValue", 0,
   null, Field1Node)
```

Example

The following example uses **getLiteralByRefEx** to get the literal value from a specific node. That value is then converted into an integer.

```
private static int getCurrentDay( ) throws Exception
{
String temp;

   temp = theForm.getLiteralByRefEx(null, "PAGE1.CURRENTDAY.value", 0,
      null, null);

   /* If a literal value was returned, convert it into an integer value;
   otherwise, indicate that no value was entered into the field and throw
   an exception. */

   if (temp != null)
   {
      return Integer.parseInt(temp);
   }
   else
   {
      throw new UWIException("The current day was not entered.");
   }
}
```

getLiteralEx

Description

This method retrieves the literal of a node. The literal is returned in the specified character set.

Note: It is not necessary to call this method when you are using XForms. The replaceXFormsInstance and extractXFormsInstancemethods perform this task automatically.

Method

```
public String getLiteralEx(
   String theCharSet
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theCharSet	**String**	The character set you want to use to view the literal string. Use **null** or **Unicode** for Unicode. Use **Symbol** for Symbol.

Returns

A string containing the literal of the node or throws a generic exception (**UWIException**) if an error occurs. If the literal is empty or does not exist, the method returns **null**.

Example

The following example uses **dereferenceEx** to locate a specific node. **getLiteralEx** is then used to get the literal value for that node.

```
private static void getGender( ) throws UWIException
{
FormNodeP tempNode;
String temp;

   if ((tempNode = theForm.dereferenceEx(null, "PAGE1.MALERADIO.value",
      0, FormNodeP.UFL_OPTION_REFERENCE | FormNodeP.UFL_SEARCH, null)) ==
      null)
   {
      throw new UWIException("Could not locate MALERADIO value node.");
   }

   temp = tempNode.getLiteralEx(null);

   /* additional code removed */
}
```

getLocalName

Description

This method returns the *local name* of a given node. The local name is determined by the XML tag that represents that node. For example, examine the following XML fragment:

```
<page sid="PAGE1">
   <global sid="global"></global>
   <field sid="testField">
      <value>Hello</value>
      <bgcolor>
         <ae>120</ae>
         <ae>120</ae>
         <ae>120</ae>
      <bgcolor>
   </field>
</page>
```

In this sample, the name of the page node is ″page″, the name of the field node is ″field″, the name of the value node is ″value″, and the name of the bgcolor node is ″bgcolor″. The bgcolor node is also the parent of three array element nodes, all of which are named ″ae″.

Note that the local name does not include any namespace prefix that might exist. For example, you might have a custom option in a different namespace as shown:

```
<field sid="testField">
   <custom:my_option>value</custom:my_option>
</field>
```

In this case, the local name of the custom option is returned without the prefix, resulting in ″my_option″.

Method

```
public String getLocalName( ) throws UWIException;
```

Parameters

There are no parameters for this method.

Returns

The name of the node or throws a generic exception (**UWIException**) if an error occurs.

Example

The following method takes the root node of the form and uses recursion to step through each node in the form. The method uses **isXFDL** and **getLocalName** to locate all label nodes in the XFDL namespace and changes the background color of those nodes to green.

```
private static void changeLabelColor(FormNodeP theNode) throws Exception
{
FormNodeP tempNode, bgcolorNode;

   /* Use recursion to step through each node in the form. */

   tempNode = theNode.getChildren();
   while(tempNode != null)
   {
      changeLabelColor(tempNode);
      tempNode = tempNode.getNext()
   }

   /* If the node is a label in the XFDL namespace, set the bgcolor
      option to "green". */
```

```
   if ((theNode.isXFDL()) && (theNode.getLocalName.equals("label")))
      theNode.SetLiteralByRefEx(null, "bgcolor[0]", 0, null, null,
         "green")

}
```

getNamespaceURI

Description

This method returns the *namespace URI* for the node.

Each namespace is defined in the form by a namespace declaration, as shown:

```
xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom"
```

Each namespace declaration defines both a prefix and a URI for the namespace. In this sample, the prefix for the XFDL namespace is *xfdl* and the URI is *http://www.ibm.com/xmlns/prod/XFDL/7.0*.

Tags within the form are assigned specific namespaces by using the defined prefix. For example, to declare that an option was in the custom namespace you would use the prefix *custom* as shown:

```
<field sid="testField">
   <custom:custom_option>value</custom:custom_option>
</field>
```

Method

```
public String getNamespaceURI( ) throws UWIException;
```

Parameters

There are no parameters for this method.

Returns

The namespace URI or throws a generic exception (**UWIException**) if an error occurs.

Example

The following method uses recursion to traverse the entire node structure and destroys all nodes that are in the *custom* namespace identified by the following URI: *http://www.ibm.com/xmlns/prod/XFDL/Custom*. This method assumes that you are passing in the root node of the form.

```
private static void deleteCustomInfo(FormNodeP theNode) throws Exception
{
FormNodeP tempNode, tempNode2;

   /* Use recursion to step through each node of the form. */

   tempNode = theNode.getChildren();
   while(tempNode != null)
   {
      tempNode2 = tempNode.getNext();
      deleteCustomInfo(tempNode);
      tempNode = tempNode2;
   }
```

```
    /* If the node belongs to the custom namespace, delete it. */

    if (theNode.getNamespaceURI().equals
       ("http://www.ibm.com/xmlns/prod/XFDL/Custom"))
       theNode.destroy();
}
```

getNamespaceURIFromPrefix

Description

This method returns the *namespace URI* that corresponds to a specific prefix. You can call this method from any node in the form, as long as that node either declares or inherits the namespace in question.

Each namespace is defined in the form by a namespace declaration, as shown:

```
xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom"
```

Each namespace declaration defines both a prefix and a URI for the namespace. In this sample, the prefix for the XFDL namespace is *xfdl* and the URI is *http://www.ibm.com/xmlns/prod/XFDL/7.0*.

Tags within the form are assigned specific namespaces by using the defined prefix. For example, to declare that an option was in the custom namespace you would use the prefix *custom* as shown:

```
<field sid="testField">
   <custom:custom_option>value</custom:custom_option>
</field>
```

Method

```
public String getNamespaceURIFromPrefix(
   String thePrefix
   ) throws UWIException;
```

Parameters

Expression	Type	Description
thePrefix	**String**	The namespace prefix. For example, *xfdl*.

Returns

The namespace URI or throws a generic exception (**UWIException**) if an error occurs. If the namespace URI is not declared, the result is **null**.

Example

The following method copies a custom option from one form to another. The method assumes that you know the prefix for the custom namespace, but not the URI. First, the method uses **getNamespaceURIFromPrefix** to get the URI for the custom namespace in the first form. Next, it adds that namespace to the second form as a globally available namespace. It then locates the custom node in the first form and the global item node in the second form. Finally, it copies the custom node to the second form as a child of the global item node.

```
private static void copyCustomInfo(FormNodeP form1, FormNodeP form2)
   throws Exception
{
   String theURI;
   FormNodeP tempNode, duplicateNode, globalNode;

   /* Get the URI for the custom namespace in form 1. If the URI is null,
      throw an error. */

   if ((theURI = form1.getNamespaceURIFromPrefix("custom")) == null)
      throw new UWIException("Custom namespace not declared in form.");

   /* Create a custom namespace in form 2 using that URI. */

   form2.addNamespace(theURI, "custom");

   /* Locate the custom Status node in form 1. */

   if ((tempNode = form1.dereferenceEx(null,
      "global.global.custom:Status", 0, UFL_OPTION_REFERENCE |
      UFL_SEARCH, null)) == null)
      throw new UWIException("Could not find custom Status node.");

   /* Locate the global item in form 2. */

   if ((globalNode = form2.dereferenceEx(null, "global.global", 0,
      UFL_ITEM_REFERENCE | UFL_SEARCH, null)) == null)
      throw new UWIException("Could not locate global item.");

   /* Copy the custom node from form 1 and insert it as a child of
      the global item in form 2. */

   if ((duplicateNode = tempNode.duplicate(globalNode, UFL_APPEND_CHILD,
      null)) == null)
      throw new UWIException("Could not duplicate node.");
}
```

getNext

Description

This method, along with **getPrevious**, is used to traverse horizontally along the form hierarchy. **getNext** returns the next node in the tree. For instance, the page node corresponding to the first page of your form can be reached by calling **getNext** on the global page node.

Method

```
public FormNodeP getNext( ) throws UWIException;
```

Parameters

There are no parameters for this method.

Returns

The FormNodeP that represents the next node or null if no such node exists. A generic exception (**UWIException**) is thrown if an error occurs.

Example

In the following example the root node of a form is represented by a **FormNodeP** called **theForm**. The method **dereferenceEx** is used to retrieve an item from the form called NAMELABEL. Then **getNext** is used to retrieve a second item that is the next sibling node after NAMELABEL.

```
public class getFunctions
{
private static FormNodeP theForm;
private static FormNodeP tempNode;
private static FormNodeP nextNode;

/* Additional Code Removed */

   public static void main(String argv[])
   {

      /* Additional Code Removed */

      if ((tempNode = theForm.dereferenceEx(null, "PAGE1.NAMELABEL",
         0,FormNodeP.UFL_ITEM_REFERENCE, null)) == null)
      {
         throw new UWIException("Could not locate Name label node.");
      }
      nextNode = tempNode.getNext( );

      /* Additional Code Removed */

   }
}
```

getNodeType

Description

This method returns the type for a node (for example, page, item, option, and so on). This allows you to quickly determine the type of node you are working with and what depth you are at in the node hierarchy.

Method

```
public int getNodeType() throws UWIException;
```

Parameters

There are no parameters for this method.

Returns

One of the following types:

- FormNodeP.UFL_FORM — The root node of the form.
- FormNodeP.UFL_PAGE — A page level node.
- FormNodeP.UFL_ITEM — An item level node.
- FormNodeP.UFL_OPTION — An option level node.
- FormNodeP.UFL_ARRAY — An argument level node, such as an array element.

This method throws a generic exception (**UWIException**) if an error occurs.

This function throws an exception if an error occurs.

Example

The following method recieves a node below the page level and uses **getParent** to ascend the hierarchy until it reaches a page node, as detected by **getNodeType**.

```
private static FormNodeP ascendToPage(FormNodeP theNode) throws Exception
{
   while ((theNode != null) && (theNode.getNodeType() !=
      FormNodeP.UFL_PAGE))
   {
      theNode = theNode.getParent();
   }
   return(theNode);
}
```

getParent

Description

This method, along with **getChildren**, is used to traverse vertically along the form hierarchy. **getParent** returns the parent of a node. If the node has no parent, null is returned. A form's structure can be traversed up to the root node using an iterator such as a while loop.

Method

```
public FormNodeP getParent( ) throws UWIException;
```

Parameters

There are no parameters for this method.

Returns

The FormNodeP that represents the parent node or null if no such parent exists. If an error occurs, a generic (**UWIException**) is thrown.

Example

In the following example the root node of a form is represented by a **FormNodeP** called **theForm**. The method **dereferenceEx** is used to retrieve an option node from the form called *PAGE1.AGEFIELD.size*.

getParent returns the parent node of *PAGE1.AGEFIELD.size*, that is, *PAGE1.AGEFIELD*.

```
public class getFunctions
{
private static FormNodeP theForm;
private static FormNodeP tempNode;
private static FormNodeP parentNode;

   /* Additional Code Removed */

   public static void main(String argv[])
   {
      /* Additional Code Removed */
      if ((tempNode = theForm.dereferenceEx(null,
         "PAGE1.AGEFIELD.size", 0, FormNodeP.UFL_OPTION_REFERENCE |
         FormNodeP.UFL_SEARCH, null)) == null)
      {
         throw new UWIException("Could not locate AgeField size
```

```
            label node.");
      }
      parentNode = tempNode.getParent( );

      /* Additional Code Removed */

   }
}
```

getPrefix

Description

This method returns the namespace *prefix* for the node.

Each namespace is defined in the form by a namespace declaration, as shown:

```
xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom"
```

Each namespace declaration defines both a prefix and a URI for the namespace. In this sample, the prefix for the XFDL namespace is *xfdl* and the URI is *http://www.ibm.com/xmlns/prod/XFDL/7.0*.

Tags within the form are assigned specific namespaces by using the defined prefix. For example, to declare that an option was in the custom namespace you would use the prefix *custom* as shown:

```
<field sid="testField">
   <custom:custom_option>value</custom:custom_option>
</field>
```

Note: A given prefix may not always resolve to the same namespace. Different portions of the form may define the prefix differently. For example, the custom prefix may resolve to a different namespace on the first page of a form than it does on the following pages.

Method

```
public String getPrefix( ) throws UWIException;
```

Parameters

There are no parameters for this method.

Returns

The prefix for the node's namespace or throws a generic exception (**UWIException**) if an error occurs.

Example

The following method removes all nodes from the form that have a namespace prefix of "custom". The method walks through the form using **getChildren** and **getNext** in a recursive loop. While walking the form, it uses **getPrefix** to locate nodes in the custom namespace and deletes them using **destroy**. This method assumes that you are passing it the root node of the form.

```
private static void deleteCustomInfo(FormNodeP theNode) throws Exception
{
FormNodeP tempNode, tempNode2;
```

```
   /* Use recursion to step through each node of the form. */

   tempNode = theNode.getChildren();
   while(tempNode != null)
   {
      tempNode2 = tempNode.getNext();
      deleteCustomInfo(tempNode);
      tempNode = tempNode2;
   }

   /* If the node is in the custom namespace, delete it. */

   if (theNode.getPrefix().equals("custom"))
      theNode.destroy();
}
```

getPrefixFromNamespaceURI

Description

This method returns the namespace *prefix* for a specific namespace URI. You can call this method from any node in the form, as long as that node either declares or inherits the namespace in question.

Each namespace is defined in the form by a namespace declaration, as shown:

```
xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom"
```

Each namespace declaration defines both a prefix and a URI for the namespace. In this sample, the prefix for the XFDL namespace is *xfdl* and the URI is *http://www.ibm.com/xmlns/prod/XFDL/7.0*.

Tags within the form are assigned specific namespaces by using the defined prefix. For example, to declare that an option was in the custom namespace you would use the prefix *custom* as shown:

```
<field sid="testField">
   <custom:custom_option>value</custom:custom_option>
</field>
```

Method

```
public String getPrefixFromNamespaceURI(
   String theURI
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theURI	**String**	The namespace URI. For example: `http://www.ibm.com/xmlns/prod/XFDL/7.0`

Returns

The namespace prefix or throws a generic exception (**UWIException**) if an error occurs. If the namespace URI is not declared, the result is **null**.

Example

The following method adds custom information to a form and assumes that the namespace URI for the custom information is known but that the prefix used to represent that namespace in the form is not known. First, the method uses **getPrefixFromNamespaceURI** to get the prefix in use. The method then concatenates the prefix with the name for the new node, "Status". Finally, the method locates the global item in the global page and creates a new option node.

```
private static void addStatus(FormNodeP theNode) throws Exception
{
XFDL theXFDL;
String thePrefix;
String theNodeName;

   /* Retrieve the prefix for the custom namespace. If the prefix is
      null, throw an error. */

   if ((thePrefix = theNode.getPrefixFromNamespaceURI(
      "http://www.ibm.com/xmlns/prod/XFDL/Custom")) == null)
      throw new UWIException("Custom namespace not declared in form.");

   /* Create a name for a new node by concatenating the prefix with
      "Status". */

   theNodeName = thePrefix + ":Status";

   /* Locate the global item in the global page so we can add a global
      option. */

   if (theNode = theNode.dereferenceEx(null, "global.global", 0,
      UFL_ITEM_REFERENCE | UFL_SEARCH, null) == null)
      throw new UWIException("Could not locate global.global node.");

   /* Get the XFDL object so we can create a new node. */

   if ((theXFDL = IFSSingleton.getXFDL()) == null)
      throw new UWIException("Could not find XFDL interface.");

   /* Create a new node in the custom namespace and give it a value
      of "Processed". */

   if (theNode = theXFDL.create(theNode, UFL_APPEND_CHILD, null,
      "Processed", null, theNodeName) == null)
      throw new UWIException("Could not create Status node.");
}
```

getPrevious

Description

This method, along with **getNext**, is used to traverse horizontally along the form hierarchy. **getPrevious** returns the previous node in the tree. For instance, if you call **getPrevious** one the Page1 node in your form, it will return the global page node.

Method

```
public FormNodeP getPrevious( ) throws UWIException;
```

Parameters

There are no parameters for this method.

Returns

The FormNodeP that represents the previous node or null if no such node exists. A generic exception (**UWIException**) is thrown if an error occurs.

Example

In the following example the root node of a form is represented by a **FormNodeP** called **theForm**. The method **dereferenceEx** is used to retrieve an item from the form called NAMELABEL. Then **getPrevious** is used to retrieve a second item that is the sibling node before NAMELABEL.

```
public class getFunctions
{
private static FormNodeP theForm;
private static FormNodeP tempNode;
private static FormNodeP prevNode;

/* Additional Code Removed */

   public static void main(String argv[])
   {

      /* Additional Code Removed */

      if ((tempNode = theForm.dereferenceEx(null, "PAGE1.NAMELABEL",
         0,FormNodeP.UFL_ITEM_REFERENCE, null)) == null)
      {
         throw new UWIException("Could not locate Name label node.");
      }
      prevNode = tempNode.getPrevious( );

      /* Additional Code Removed */
     }
}
```

getReferenceEx

Description

This method returns the reference string that identifies the node. For example, a value node might return a reference of *Page1.Field1.value*. The reference will either begin at the page level of the form or at a level specified by the caller.

Method

```
public String getReferenceEx(
   String theScheme,
   FormNodeP theNSNode,
   FormNodeP theStartPoint,
   boolean addNamespaces
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theScheme	**String**	Reserved. This must be null.
theNSNode	**FormNodeP**	A node that defines which namespace prefixes are used when constructing the reference. Only namespace prefixes that this node inherits are used. Use null if the node that this method is operating on has inherited the necessary namespaces.

Expression	Type	Description
theStartPoint	**FormNodeP**	A node that determines the starting point of the reference. This node must be a parent of the *aNode* parameter. The reference will begin one level below the start point node. For example, if you provide a page node the reference will begin at the item level. Use null to start the reference at the page level.
addNamespaces	**boolean**	Use true to add declarations for unknown namespaces to the namespace node (*theNSNode*). Otherwise, use false.

Returns

A string containing a reference to the node, or throws a generic exception (**UWIException**) if an error occurs.

Notes

Creating a Reference String

For more information about creating a reference, see "References" on page 8.

Working with Namespace Prefixes

In some cases, you may want to use the **getReferenceEx** method to get the reference to a node that uses a different prefix for a known namespace. For example, consider the following form:

```
<label sid="Label1" xmlns:data="URI">
   <value></value>
</label>
<field sid="Field1" xmlns:processing="URI">
   <value></value>
   <processing:myValue>10<processing:myValue>
</field>
```

In this form, *processing* and *data* are prefixes for the same namespace, since they both refer to the same URI. However, both namespaces have limited scope since they are declared at the item level. This means that *Label1* node does not understand the *processing* prefix, and that the *Field1* node does not understand the *data* prefix.

This becomes a problem if you want to refer to a namespace from a location that does not understand that namespace. For example, suppose you wanted to set the value of *Label1* to be a reference to the *myValue* node in *Field1*. Normally, you would locate the *myValue* node and use **getReferenceEx** as shown:

```
myValueNode.getReferenceEx(null, null, null, false)
```

In this case, **getReferenceEx** would return the following reference: *Page1.Field1.processing:myValue*. However, because the *processing* namespace is not defined for *Label1*, a reference to the *processing* namespace is not understood. This means that you cannot set the value of *Label1* to equal this reference, since the node would not understand that content.

Instead, you must generate a reference that includes a known namespace prefix, such as the *data* namespace. You can do this by including a second node in the

getReferenceEx method. The second node must understand the appropriate namespace. For example, you could include the *Label1* node in the method, as shown:

```
myValueNode.getReferenceEx(null, Label1Node, null, false)
```

In this case, the method will substitute the *data* prefix for the *processing* prefix, since they both resolve to the same namespace. As a result, the method will return: *Page1.Field1.data:myValue*. Since the *data* prefix is defined within *Label1*, you can use this reference to set *Label1*'s value node.

Working with Unknown Namespaces

In some cases, you may want to use the **getReferenceEx** method to get the reference to a node that uses an unknown namespace. For example, consider the following form:

```
<page sid="Page1" xmlns:processing="URI1">
   <global sid="global">
      <processing:info></processing:info>
   </global>
   <field sid="Field1" xmlns:data="URI2">
      <value></value>
      <data:info>data</data:info>
   </field>
```

In this example, you might want to store a reference to the <data:info> element in the <processing:info> element. **getReferenceEx** would return the following reference for the <data:info> element: *Page1.Field1.data:info*. However, this reference includes the *data* namespace, which is not defined for the page global. This means that you could not store this reference in the <processing:info> element, because it would not understand the reference.

To solve this problem, you can use the *addNamespaces* flag in the **getReferenceEx** method. When this flag is set to true, the method will add unknown namespaces to the *theNSNode*.

For example, if you set *theNSNode* to be the global item node for *Page1*, and set the *addNamespace* flag to true, as shown:

```
dataNode.getReferenceEx(null, pageGlobalNode, null, true)
```

The method would return the reference to the <data:info> element, but would also modify the global item node to include the unknown *data* namespaces, as shown:

```
<global sid="global" xmlns:data="URI2">
```

You could then store the reference in that global item or any of its descendants, since the namespace is now properly defined.

Example

In the following example, a page node is passed to the method. The method then uses **getChildren** and **getNext** to locate the last item node in the page. **getReferenceEx** is then called to get the reference to that node, which is returned to the caller.

```
public String getLastItemReference(FormNodeP pageNode)
{
FormNodeP itemNode, tempNode;
String theReference;
```

```
      /* Get the first item node in the page. */

      itemNode = pageNode.getChildren();

      /* Cycle through to the last item node in the page. */

      while ((tempNode = itemNode.getNext()) != null)
      {
         itemNode = tempNode;
      }
      /* Get the reference to the node and return it. */
      theReference = itemNode.getReferenceEx(null, null, null, false);
      return(theReference);
   }
```

getSecurityEngineName

Description

This method returns the name of the appropriate security engine for a given button or signature node. This is useful for determining which validation call you need to make to validate the signature.

Method

```
public String getSecurityEngineName(
   int theOperation
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theOperation	**int**	The operation you want the security engine for. Possible values are: **SecurityManager.SEOPERATION_SIGN** — the engine is needed to sign the form. **SecurityManager.SEOPERATION_VERIFY** — the engine is needed to verify the signature. **SecurityManager.SEOPERATION_LISTIDENTITIES** — the engine is needed to generate a list of valid certificates for signing.

Returns

A string containing the name of the security engine on success, or throws a generic exception (**UWIException**) if an error occurs. The possible names are:

- CryptoAPI
- Netscape
- ClickWrap
- HMAC-ClickWrap
- PenOp

Example

The following example uses **getSecurityEngineName** to get the appropriate engine for a signature verification. If the engine is *HMAC-ClickWrap*, the example calls a

method that will verify an HMAC signature. Otherwise, the example calls a method that verifies other types of signatures.

```
public short validateSignature(FormNodeP sigNode)
{
String engineName;
short validation;

   engineName = sigNode.getSecurityEngineName
      (SecurityManager.SEOPERATION_VERIFY);
   if (engineName.equals("HMAC-ClickWrap"))
   {
      validation = validateAuthenticatedClickwrapSignature(sigNode);
   }
   else
   {
      validation = validateNormalSignature(sigNode);
   }
   return(validation);
}
```

getSigLockCount

Description

This method returns the signature lock count of a node. If 0 is returned, the node is not signed by any digital signature, but it may have descendants that are signed.

Method

```
public int getSigLockCount( ) throws UWIException;
```

Parameters

There are no parameters for this method.

Returns

The number of locks on the given node or throws a generic exception (**UWIException**) if an error occurs.

Example

In the following example, **dereferenceEx** is used to locate the address field node. **getSigLockCount** is then used to determine how many signatures have locked the address field.

```
private static void checkSigLocks(FormNodeP theForm) throws Exception
{
FormNodeP addressNode;
FormNodeP tempNode;

   if ((addressNode = theForm.dereferenceEx(null, "PAGE1.ADDRESSFIELD",
      0,   FormNodeP.UFL_ITEM_REFERENCE, null)) == null)
   {
      throw new UWIException("Could not locate ADDRESSFIELD node.");
   }
   if (addressNode.getSigLockCount( ) != 2)
   {
      System.out.println("ADDRESSFIELD not signed twice.");
   }
}
```

getSignature

Description

This method returns signature object for a given *button* or *signature* item.

Method

```
public Signature getSignature() throws UWIException;
```

Parameters

There are no parameters for this method.

Returns

A signature object if the call is successful, or throws a generic exception (**UWIException**) if an error occurs.

Example

The following example uses **getSignature** to get the signature object from the signature node, and uses **getDataByPath** to get the signer's identity from the signature object. It then calls **validateHMACWithSecret** to validate the signature.

```
public short checkSignature(FormNodeP theSignatureNode,
   Certificate theServerCert)
{
Signature theSignatureObject;
String theSecret;
String signerCommonName;
BooleanHolder encodedData;
IntHolder theStatus;
short validation;

   theSignatureObject = theSignatureNode.getSignature();
   encodedData = new BooleanHolder();
   if ((signerCommonName = theSignatureObject.getDataByPath(
      "SigningCert: Subject: CN", false, encodedData)) == null)
   {
      throw new UWIException("Could not determine signer's name.");
   }

   /* Include external code that matches the signer's identity to a shared
      secret, and sets theSecret to match.  This is most likely a
      database lookup. */

   theStatus = new IntHolder();

   validation = theSignatureNode.validateHMACWithSecret(theSecret,
      theServerCert, theStatus);

   /* Check the status in case the process required user input. */

   if (theStatus.value != SecurityUserStatusType.SUSTATUS_OK)
   {
      throw new UWIException("Validation required user input.");
   }
   return(validation);
}
```

getSignatureVerificationStatus

Description

When called, this method checks to see if the digital signatures in a given form are valid.

Method

```
public short getSignatureVerificationStatus( ) throws UWIException;
```

Parameters

There are no parameters for this method.

Returns

A short having one of the following values:

Code	Status
FormNodeP.UFL_SIGS_OK	The signatures are valid.
FormNodeP.UFL_SIGS_NOTOK	One or more signatures are broken.
FormNodeP.UFL_SIGS_UNVERIFIED	One or more signatures are unverifiable.

On error, the method throws a generic exception (**UWIException**).

Example

The following example reads a form into memory, and then uses **getSignatureVerificationStatus** to check if the signatures in a loaded form are valid.

```
private static void loadForm( ) throws Exception
{
   FormNodeP.readForm("Sample.xfd", 0);

   if (theForm.getSignatureVerificationStatus()!= FormNodeP.UFL_SIGS_OK)
   {
      System.out.println("At least one digital signature is not valid.");
   }
}
```

isSigned

Description

This method determines whether a node is signed.

Method

```
public boolean isSigned(
   boolean excludeSelf
   ) throws UWIException;
```

Parameters

Expression	Type	Description
excludeSelf	**boolean**	A signature node is always self-signed. To determine whether a second signature has been applied to that node, you must exclude the self-signing from this check. To exclude the self-signing from the signature check, set this to true. To include the self-signing, set this to false.

Returns

True if the node is signed, *false* if it is not.

Example

The following function locates the value node for a Date field, checks to see if it is signed, and sets the value if the node is not signed.

```
private static void setDateValue(String theDate, formNodeP theForm)
   throws Exception
{
FormNodeP tempNode;

   /* Locate the value option for the Date field */

   if ((tempNode = theForm.dereferenceEx(null, "PAGE1.Date.value", 0,
      FormNodeP.UFL_OPTION_REFERENCE, null)) == null)
   {
      throw new UWIException("Could not locate value node for Date.");
   }

   /* Check the value node to see if it is signed. If it is signed,
      return an error. Otherwise, set it to the value passed into the
      method. */

   if (tempNode.isSigned(false) == true)
   {
      throw new UWIException("Date's value is signed");
   }
   else
   {
      tempNode.setLiteralEx(null, theDate);
   }
}
```

isValidFormat

Description

This method returns the boolean result of whether a string is valid according to the setting of the node's *format* option.

This method does not support XForms nodes.

Method

```
public boolean isValidFormat(
   String theString
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theString	**String**	A string to be checked against the format. For example, to check 23.2 against a specific format, the string would be "23.2".

Returns

True if the string does match the format, *false* if it does not.

Example

The following function locates the Currency field and checks to see if "23.2" conforms to the format required by the field's *format* option.

```
public boolean checkFormat(formNodeP theForm)
   throws Exception
{
FormNodeP theItem;

   /* Locate the Currency field */

   if ((theItem = theForm.dereferenceEx(null, "PAGE1.Currency", 0,
      FormNodeP.UFL_OPTION_REFERENCE, null)) == null)
   {
      throw new UWIException("Could not locate the Currency node.");
   }

   /* Check the string to see if it is invalid*/

   (theItem.isValidFormat("23.2") == false)
   {
      throw new UWIException("The value is not correctly formatted.");
   }
   else
   {
       System.out.println("The value is formatted correctly.");
   }
}
```

isXFDL

Description

This method determines whether a node belongs to the XFDL namespace.

Each namespace is defined in the form by a namespace declaration, as shown:

```
xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom"
```

Each namespace declaration defines both a prefix and a URI for the namespace. In this sample, the prefix for the XFDL namespace is *xfdl* and the URI is *http://www.ibm.com/xmlns/prod/XFDL/7.0*.

Tags within the form are assigned specific namespaces by using the defined prefix. For example, to declare that an option was in the custom namespace you would use the prefix *custom* as shown:

```
<field sid="testField">
   <custom:custom_option>value</custom:custom_option>
</field>
```

Method

```
public boolean isXFDL() throws UWIException;
```

Parameters

There are no parameters for this method.

Returns

True if the node belongs to the XFDL namespace, *false* if it does not, or throws a generic exception (**UWIException**) if an error occurs.

Example

The following method uses recursion to traverse the entire node structure and destroys all nodes that are not in the XFDL namespace. This method assumes that you are passing in the root node of the form.

```
private static void deleteCustomInfo(FormNodeP theNode) throws Exception
{
FormNodeP tempNode, tempNode2;

   /* Use recursion to step through each node of the form. */

   tempNode = theNode.getChildren();
   while(tempNode != null)
   {
      tempNode2 = tempNode.getNext();
      deleteCustomInfo(tempNode);
      tempNode = tempNode2;
   }

   /* If the node is not in the XFDL namespace, delete it. */

   if (theNode.isXFDL() == false)
      theNode.destroy();
}
```

removeAttribute

Description

This method removes a specific attribute from a node. For example, the following XFDL represents a value node:

```
<value custom:myAtt="x"></value>
```

To remove the custom attribute from this node, you would use *removeAttribute*.

Method

```
public void removeAttribute(
   String theNamespaceURI,
   String theAttribute
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theNamespaceURI	**String**	The namespace URI for the attribute. For example: `http://www.ibm.com/xmlns/prod/XFDL/7.0`
theAttribute	**String**	The local name of the attribute. For example, *compute*, *encoding*, and so on.

Returns

Nothing or throws a generic exception (**UWIException**) if an error occurs.

Notes

Attributes and the Null Namespace

If an attribute is on a node in a non-XFDL namespace, and that attribute has no namespace prefix, then the attribute is in the *null* namespace. For example, the following node is the custom namespace, as is the first attribute, but since the second attribute does not have a namespace prefix, it is in the null namespace:

```
<custom:processing custom:stage="2" user="tjones">
```

When an attribute is the null namespace, you may either provide a null value for the namespace URI or use the namespace URI for the containing element.

For example, to indicate *user* attribute on the *processing* node, you could use the null namespace or the custom namespace URI.

Attributes and Namespace Prefixes

If you refer to an attribute with a namespace prefix, **removeAttribute** first looks for a complete match, including both prefix and attribute name. If it does not find such a match, it will look for a matching attribute name that has no prefix but whose containing element has the same namespace.

For example, assume that the *custom* namespace and the *test* namespace both resolve to the same URI. In the following case, looking for the *id* attribute would locate the second attribute (test:id), since it has an explicit namespace declaration:

```
<a xmlns:custom="ABC" xmlns:test="ABC">
   <custom:myElement id="1" test:id="2">
</a>
```

However, in the next case, the *id* attribute does not have an explicit namespace declaration. Instead, it inherits the custom namespace. However, since the inherited namespace resolves to the same URI, the *id* attribute is still located:

```
<custom:myElement id="1">
```

Example

The following method uses **getAttributeList** to retrieve the list of a node's attributes. It then searches through the list looking for a compute attribute. When if locates a compute attribute, it uses **removeAttribute** to remove the compute from the node.

```
private static void stripCustomAttributes(FormNodeP theNode) throws Exception
{
int counter;
StringListHolder URIList = new StringListHolder[];
StringListHolder attributeList = new StringListHolder[];

   /* Get the list of attributes for the node. */

   theNode.getAttributeList(URIList, attributeList);

   /* Step through each attribute and delete the compute. */

   for (counter = 0; counter < attributeList.value.length; counter++)
   {
      if (attributeList.value[counter].equals("custom:myAtt"))
      {
         theNode.removeAttribute(URIList.value[counter],
            attributeList.value[counter]);
      }
   }
}
```

removeEnclosure

Description

This method will either remove an enclosure from a specific datagroup or delete the enclosure from the form. Call this method on the **FormNodeP** that contains the enclosure you want to remove.

Method

```
public void removeEnclosure(
   String theDataGroup
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theDataGroup	**String**	The datagroup that contains the enclosed item. If null, the item will be removed from all datagroups. If an item no longer belongs to any datagroups, it is deleted from the form.

Returns

Nothing if call is successful or throws a generic exception (**UWIException**) if an error occurs.

Example

The following example uses **UFLDereferenceEx** to locate a specific data node. **UFLRemoveEnclosure** is then used to remove the node from the form.

```
private static void deleteLogo(FormNodeP theForm) throws Exception
{
FormNodeP tempNode;

   if ((tempNode = theForm.dereferenceEx(null, "PAGE1.LOGODATA", 0,
      FormNodeP.UFL_ITEM_REFERENCE, null)) == null)
   {
```

```
      throw new UWIException("Could not locate LOGODATA node.");
   }
   tempNode.removeEnclosure(null);
}
```

replaceXFormsInstance

Description

This method either inserts or replaces an XForms instance in a form's data model. The instance can come from either from either a file, an input stream, a Java Reader, or a memory block.

Call this method on the root node of the form or an instance node.

Use caution when calling this method. It can be used to overwrite signed instance data.

Note: This method automatically updates the XForms data model.

Method

READING A FILE:

```
public void replaceXFormsInstance(
   String theModelID,
   String theNodeRef,
   FormNodeP theNSNode,
   String theFilename,
   byte[] theMemoryBlock
   boolean replaceRef
   ) throws UWIException;
```

READING FROM A STREAM:

```
public void replaceXFormsInstance(
   String theModelID,
   String theNodeRef,
   FormNodeP theNSNode,
   java.io.InputStream theStream,
   boolean replaceRef
   ) throws UWIException;
```

READING FROM A READER:

```
public void replaceXFormsInstance(
   String theModelID,
   String theNodeRef,
   FormNodeP theNSNode,
   java.io.Reader theReader,
   boolean replaceRef
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theModelID	**String**	The ID of the affected model. You must use null to use the default model.
theNodeRef	**String**	An XPath reference to the instance (or portion of an instance) you want to replace. An empty string indicates the default instance of the selected model.

Expression	Type	Description
theNSNode	**FormNodeP**	A node that inherits the namespaces used in the reference. This node defines the namespaces for the method. Use null if the node that this method is operating on has inherited the necessary namespaces.
theFilename	**String**	The file to read the instance from.
theStream	**java.io.InputStream**	The input stream that provides the data that replaces the XForms instance. Note that this data must be UTF-8.
theReader	**java.io.Reader**	The Java Reader that provides the data that replaces the XForms instance.
theMemoryBlock	**byte[]**	The memory block that contains the instance if you are not reading from a file, input stream, or Reader. Use **null** if *theFilename* is used.
replaceRef	**boolean**	If true, the node specified by *theNodeRef* is replaced with data. If false, the data is appended as the last child of the instance node.

Returns

Nothing if call is successful or throws a generic exception (**UWIException**) if an error occurs.

Example

The following example shows a method that replaces an XForms instance.

```
private static void updateDataInstance(String theFileName)
{
   theForm.replaceXFormsInstance("model1",
      "instance('instance1')"loanrecord/user_personal_info",
       null, "c:\\InstanceData.xml", true);
}
```

setActiveForComputationalSystem

Description

This method sets whether the computational system is active. When active, all computes in the form are evaluated on an on-going basis. When inactive, no computes are evaluated.

Note that turning the computational system on causes all computes in the form to be re-evaluated, which can be time consuming.

Method

```
public void setActiveForComputationalSystem (
   boolean activeFlag,
   ) throws UWIException;
```

Parameters

Expression	Type	Description
activeFlag	**boolean**	Set to true to activate the compute system or false to deactivate the compute system.

Returns

Nothing or throws a generic exception (**UWIException**) if an error occurs.

Example

The following example reads a form into memory with the computational system turned off. The example then calls a processing method that adds a large amount of information to the form. Next, **setActiveForComputationalSystem** is called to turn the computational system on and evaluate all of the computes. Finally, the updated form is written to disk.

```
private static void processForm() throws Exception
{
XFDL theXFDL;
FormNodeP theForm;

   /* Get the XFDL object */

   if ((theXFDL = IFSSingleton.getXFDL()) == null)
      throw new Exception("Could not find interface");

   /* Read the form into memory with the computes turned off */

   if ((theForm = theXFDL.readForm("input.xfd",
      XFDL.UFL_AUTOCOMPUTE_OFF)) == null)
      throw new Exception("Could not load form.");

   /* Call a method that adds information to the form from a database */

   addInformation(theForm);

   /* Activate the computational system. This will re-evaluate all
      computes with the new information in the form. */

   theForm.setActiveForComputationalSystem(true);

   /* Write the updated form to disk */

   theForm.writeForm("output.xfd", null, 0);
}
```

setAttribute

Description

This method sets the value of a specific attribute for a node. For example, the following XFDL represents a value node:

```
<value custom:myAtt="x"></value>
```

To change the custom attribute, you would use **setAttribute**. If the attribute does not already exist, **setAttribute** will create it and assign the appropriate value.

Note: Do not use **setAttribute** to set the compute attribute. Instead, use **setFormula**.

Method

```
public void setAttribute (
   String theNamespaceURI,
   String theAttribute,
   String theValue
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theNamespaceURI	**String**	The namespace URI for the attribute. For example: `http://www.ibm.com/xmlns/prod/XFDL/7.0`
theAttribute	**String**	The local name of the attribute. For example, *encoding*.
theValue	**String**	The value to assign to the attribute.

Returns

Nothing or throws a generic exception (**UWIException**) if an error occurs.

Notes

Attributes and the Null Namespace

If an attribute is on a node in a non-XFDL namespace, and that attribute has no namespace prefix, then the attribute is in the *null* namespace. For example, the following node is the custom namespace, as is the first attribute, but since the second attribute does not have a namespace prefix, it is in the null namespace:

```
<custom:processing custom:stage="2" user="tjones">
```

When an attribute is the null namespace, you may either provide a null value for the namespace URI or use the namespace URI for the containing element.

For example, to indicate *user* attribute on the *processing* node, you could use the null namespace or the custom namespace URI.

Attributes and Namespace Prefixes

If you refer to an attribute with a namespace prefix, **setAttribute** first looks for a complete match, including both prefix and attribute name. If it does not find such a match, it will look for a matching attribute name that has no prefix but whose containing element has the same namespace.

For example, assume that the *custom* namespace and the *test* namespace both resolve to the same URI. In the following case, looking for the *id* attribute would locate the second attribute (test:id), since it has an explicit namespace declaration:

```
<a xmlns:custom="ABC" xmlns:test="ABC">
   <custom:myElement id="1" test:id="2">
</a>
```

However, in the next case, the *id* attribute does not have an explicit namespace declaration. Instead, it inherits the custom namespace. However, since the inherited namespace resolves to the same URI, the *id* attribute is still located:

```
<custom:myElement id="1">
```

Example

The following example shows a shortcut method that sets a custom data attribute for a specific node. A node and a string containing the contents of the attribute are passed to the method, which then uses **setAttribute** to set the attribute for the node.

```
private static void setCustomAttribute(FormNodeP theNode, String theContents)
   throws Exception
{
   theNode.setAttribute("http://www.ibm.com/xmlns/prod/XFDL/Custom", "Data",
      theContents)
}
```

setFormula

Description

This method sets the formula for a node.

Method

```
public void setFormula(
   String theFormula
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theFormula	**String**	The formula to assign to the *aNode*. If null, the formula is assigned as null.

Returns

Nothing if call is successful or throws a generic exception (**UWIException**) if an error occurs.

Example

In this example, **dereferenceEx** is used to locate an age field. **setFormula** is then used to set the appropriate formula for the field.

```
private static void setFormula(int curMonth, int curDay, int birMonth,
   int birDay) throws Exception
{
FormNodeP tempNode;

   theForm.dereferenceEx(null, "PAGE1.AGEFIELD.value", 0,
      FormNodeP.UFL_OPTION_REFERENCE | FormNodeP.UFL_SEARCH_AND_CREATE,
      null)

/* The following logic simply identifies how the computation should be
   set.  If the current date is later in the year than the birth date,
   then the age is: current year - birth year.  If the current date is
   earlier in the year than the birth date, then the age is: current year
   - birth year - 1. */
```

```
   if ((curMonth > birMonth) ||
      (curMonth == birMonth) && (curDay > birDay)))
   {
      tempNode.setFormula("PAGE1.CURRENTYEAR.value -
         PAGE1.BIRTHYEAR.value");
   }
   else
   {
      tempNode.setFormula("PAGE1.CURRENTYEAR.value -
         PAGE1.BIRTHYEAR.value - \"1\"");
   }

   /* additional code removed */
}
```

setLiteralByRefEx

Description

This method finds a particular FormNodeP as specified by a reference string. Once the FormNodeP is found, its literal will be set as specified. If the FormNodeP does not exist, this method will create it, but only if the FormNodeP would be an option or argument node.

If necessary, this method can create several nodes at once. For example, if you set the literal for the second argument of an *itemlocation*, this method will create the *itemlocation* option node and the two argument nodes and then set the literal for the second argument node.

This method cannot create a FormNodeP at the form, page, or item level; to do so, use **create**.

The node you call this method on is used as the starting point for the search.

Note: It is not necessary to call this method when you are using XForms. The replaceXFormsInstance and extractXFormsInstancemethods perform this task automatically.

Method

```
public void setLiteralByRefEx(
   String theScheme,
   String theReference,
   int theReferenceCode,
   String theCharset,
   FormNodeP theNSNode ,
   String theLiteral
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theScheme	**String**	Reserved. This must be null.
theReference	**String**	A string that contains the reference.
theReferenceCode	**int**	Reserved. Must be 0.
theCharSet	**String**	The character set in which *theLiteral* parameter is written. **Use null** or **Unicode** for Unicode. Use **Symbol** for Symbol.

Expression	Type	Description
theNSNode	**FormNodeP**	A node that is used to resolve the namespaces in *theReference* parameter (see "Determining Namespace" on page 97). Use null if the node that you are calling this method on has inherited the necessary namespaces.
theLiteral	**String**	The string that will be assigned to the literal. If null, any existing literal is removed.

Returns

Nothing if the call is successful or throws a generic exception (**UWIException**) if an error occurs.

Notes

This method is a shortcut method and is equivalent to performing the following on a **FormNodeP** object:

```
aNode.dereferenceEx(theScheme, theReference, theReferenceCode,
   UFL_OPTION_REFERENCE | UFL_SEARCH_AND_CREATE,
   theNamespaceNode).setLiteralEx(aCharSet, aLiteral);
```

FormNodeP

Before you decide which FormNodeP to use this method on, be sure you understand the following:

1. The FormNodeP you supply can never be more than one level in the hierarchy above the level at which your reference string starts. For example, if the reference string begins with an option, then the FormNodeP can be no higher in the hierarchy than an item.
2. If the FormNodeP is at the same level or lower in the hierarchy than the starting point of the reference string, the method will attempt to locate a common ancestor. The method will locate the ancestor of the FormNodeP that is one level in the hierarchy above the starting point of the reference string. The method will then attempt to follow the reference string back down through the hierarchy. If the reference string cannot be followed from the located ancestor (for example, if the ancestor is not common to both the FormNodeP and the reference string), the method will fail. For example, given a FormNodeP that represents ″field_1″ and a reference of ″field_2″, the method will access the ″page″ node above ″field_1″, and will then try to locate ″field_2″ below that node. If the two fields were not on the same page, the method would fail.

Creating a Reference String

For more information about creating a reference, see "References" on page 8.

Digital Signatures

Do not set a node that is digitally signed. Doing so will break the digital signature and produce an error.

Determining Namespace

In some cases, you may want to use the **setLiteralByRefEx** method to set the value for a node that does not have a globally defined namespace. For example, consider the following form:

```
<label sid="Label1">
   <value>Field1.processing:myValue</value>
</label>
<field sid="Field1" xmlns:processing="URI">
   <value></value>
   <processing:myValue>10<processing:myValue>
</field>
```

In this form, the *processing* namespace is declared in the *Field1* node. Any elements within *Field1* will understand that namespace; however, elements outside of the scope of *Field1* will not.

In cases like this, you will often start your search at a node that does not understand the namespace of the node you are trying to locate. For example, you might want to locate the node referenced in the value of *Label1*. In this case, you would first locate the *Label1* value node and get its literal. Then, from the *Label1* value node, you would attempt to locate the *processing:myValue* node as shown:

```
Label1Node.setLiteralByRefEx(null, "Field1.processing:myValue", 0,
   null, null, "20")
```

In this example, the **setLiteralByRefEx** method would fail. The method cannot properly resolve the *processing* namespace because this namespace is not defined for the *Label1* value node. To correct this, you must also provide a node that understands the *processing* namespace (in this case, any node in the scope of *Field1*) as a parameter in the method:

```
Label1Node.setLiteralByRefEx(null, "Field1.processing:myValue", 0,
   null, Field1Node, "20")
```

Example

In the original form, the label for the **Age** field instructs the user to leave the field blank. However, now that the field has been filled in by a formula, this label needs to be changed. In the following example **setLiteralByRefEx** is used to change this value.

```
private static void setFormula(int curMonth, int curDay, int birMonth,
   int birDay) throws Exception
{

   /* additional code removed */

   theForm.setLiteralByRefEx(null, "PAGE1.AGELABEL.value", 0, null,
      null, "Age:");
}
```

setLiteralEx

Description

This method sets the literal of a node. You should only set the literal for option or argument nodes.

Note: It is not necessary to call this method when you are using XForms. The replaceXFormsInstance and extractXFormsInstancemethods perform this task automatically.

Method

```
public void setLiteralEx(
   String theCharSet,
   String theLiteral
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theCharSet	**String**	The character set in which *theLiteral* parameter is written. Use **null** or **Unicode** for Unicode. Use **Symbol** for Symbol.
theLiteral	**String**	The literal to assign to the node. If null, any existing literal is removed.

Returns

Nothing if call is successful or throws a generic exception (**UWIException**) if an error occurs.

Notes

Digital Signatures

Do not set the literal of a node that has already been signed. Doing so will break the digital signature and produce an error.

Example

In the following example, **dereferenceEx** is used to locate a specific node. **setLiteralEx** is then used to change the literal of that node.

```
private static void changeNameLabel(FormNodeP theForm, String newName)
   throws Exception
{
FormNodeP tempNode;

   if ((tempNode = theForm.dereferenceEx(null, "PAGE1.NAMELABEL.value",
      0, FormNodeP.UFL_OPTION_REFERENCE | FormNodeP.UFL_SEARCH, null)) ==
      null)
   {
      throw new UWIException("Could not locate value node for
         NAMELABEL.");
   }
   tempNode.setLiteralEx(null, newName);
}
```

signForm

Description

This method acts on a button node and creates a digital signature for that button. The signature is created using the signature filter in the button and the private key of the signer.

Method

```
public Signature signForm(
   Certificate theSigner,
   StringDictionary theInfo,
   IntHolder theStatus,
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theSigner	**Certificate**	The certificate to use to create the signature.
theInfo	**StringDictionary**	Always use a null value.
theStatus	**IntHolder**	This is a status flag that reports whether the operation was successful. Possible values are: **SecurityUserStatusType.SUSTATUS_OK** — the operation was successful. **SecurityUserStatusType.SUSTATUS_ CANCELLED** — the operation was cancelled by the user. **SecurityUserStatusType.SUSTATUS_INPUT_ REQUIRED** — the operation required user input, but could not receive it (for example, it was run on a server with no user).

Returns

A signature object if the call is successful, or throws a generic exception (**UWIException**) if an error occurs.

Example

The following example uses **dereferenceEx** to locate a specific signature button node. **getCertificateList** is then used to get a list of valid certificates for that button. Finally, **signForm** signs the button using the first certificate in the list.

```
private void createSignature(FormNodeP theForm)throws Exception
{
FormNodeP buttonNode;
IntHolder theStatus;
Signature theSignature;
Certificate [] certList;

   if ((buttonNode = theForm.dereferenceEx(null, "PAGE1.SIGBUTTON1",
      0, FormNodeP.UFL_ITEM_REFERENCE, null)) == null)
   {
      throw new UWIException("Could not locate SIGBUTTON1 node.");
   }
   theStatus = new IntHolder();
   certList = buttonNode.getCertificateList(null, theStatus);
   if (theStatus.value == SecurityUserStatusType.SUSTATUS_INPUT_REQUIRED)
   {
      throw new UWIException("User input required to sign form.");
   }
   theSignature = buttonNode.signForm(certList[0], null, theStatus);

   if (theStatus.value == SecurityUserStatusType.SUSTATUS_INPUT_REQUIRED)
```

```
    {
       throw new UWIException("User input required to sign form.");
    }
 }
```

updateXFormsInstance

Description

This method supplants replaceXFormsInstance. It allows developers to insert data anywhere within the XForms instance data, or replace it entirely. The instance can come from either from either a file, an input stream, a Java Reader, or a memory block. The updateXFormsInstance method automatically updates the XForms data model.

Call this method on the root node of the form or an instance node.

Use caution when calling this method. It can be used to overwrite signed instance data.

Note: This method automatically updates the XForms data model.

Method

READING A FILE:

```
public void updateXFormsInstance(
   String theModelID,
   String theNodeRef,
   FormNodeP theNSNode,
   String theFilename,
   byte[] theMemoryBlock
   int updateType

   ) throws UWIException;
```

READING FROM A STREAM:

```
public void updateXFormsInstance(
   String theModelID,
   String theNodeRef,
   FormNodeP theNSNode,
   java.io.InputStream theStream,
   int updateType

   ) throws UWIException;
```

READING FROM A READER:

```
public void updateXFormsInstance(
   String theModelID,
   String theNodeRef,
   FormNodeP theNSNode,
   java.io.Reader theReader,
   int updateType

   ) throws UWIException;
```

Parameters

Expression	Type	Description
theModelID	**String**	The ID of the affected model. You must use null to use the default model.
theNodeRef	**String**	An XPath reference to the instance (or portion of an instance) you want to insert data into or replace. An empty string indicates the default instance of the selected model.
theNSNode	**FormNodeP**	A node that inherits the namespaces used in the reference. This node defines the namespaces for the method. Use null if the node that this method is operating on has inherited the necessary namespaces.
theFilename	**String**	The file to read the instance data from. Note that if both a file and a memory block are provided, the file will take precedence.
theStream	**java.io.InputStream**	The input stream that provides the data that replaces or adds to the XForms instance. Note that this data must be UTF-8.
theReader	**java.io.Reader**	The Java Reader that provides the data that replaces or adds to the XForms instance.
theMemoryBlock	**byte[]**	The memory block that contains the instance if you are not reading from a file, input stream, or Reader. Use **null** if *theFilename* is used.
updateType	**int**	Indicates which type of update to perform: **XFDL.XFORMS_UPDATE_REPLACE** — Replaces the specified data element. **XFDL.XFORMS_UPDATE_APPEND** — Adds the data to the end of the specified data instance or element as a child element. **XFDL.XFORMS_UPDATE_INSERT_BEFORE** — Adds the data as a sibling of the specified element. This sibling is placed before the specified element.

Returns

Nothing if call is successful or throws a generic exception (**UWIException**) if an error occurs.

Example

The following example shows a method that replaces an XForms instance.

```
private static void updateDataInstance(String theFileName)
{
   theForm.updateXFormsInstance("model1",
      "instance('instance1')"loanrecord/user_personal_info",
       null, "c:\\InstanceData.xml", true, XFDL.XFORMS_UPDATE_REPLACE);
}
```

validateHMACWithSecret

Description

This method determines whether an HMAC signature is valid. HMAC signatures include both Authenticated Clickwrap and Signature Pad signatures.

For Authenticated Clickwrap signatures, you must know the signer's shared secret to use this method. For Signature Pad signatures, you may use this method without the shared secret if the signature was created without one. In any case, the shared secret should be available from a corporate database or other system.

This method will also notarize (that is, digitally sign) a valid HMAC signature if you provide a digital certificate. However, notarization will not occur if the signature does not include a shared secret. Once notarized, you must use the **verifySignature** method to validate the signature.

Note: Authenticated Clickwrap is a separately licensed product. Please ensure that your company has the license to use Authenticated Clickwrap before you provide forms or functionality that rely on it.

Method

```
public short validateHMACWithSecret(
   String theSecret,
   Certificate theServerCert,
   IntHolder theStatus,
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theSecret	**String**	The shared secret that identifies the user. This should be available from a corporate database or other system. If there is more than one shared secret, you must concatenate the strings with no separating characters. For example, if the secrets were ″blue″ and ″red″, you would pass ″bluered″ to the method. If there is no shared secret pass an empty string.
theServerCert	**Certificate**	The server certificate. If the HMAC signature is valid, the method will use the private key of this certificate to digitally sign the HMAC signature. This signature is appended to the signature item, and can be verified using verifySignature. If you pass null, the method will simply validate the HMAC signature.

Expression	Type	Description
theStatus	**IntHolder**	This is a status flag that reports whether the operation was successful. Possible values are: **SecurityUserStatusType.SUSTATUS_OK** — the operation was successful. **SecurityUserStatusType.SUSTATUS_CANCELLED** — the operation was cancelled by the user. **SecurityUserStatusType.SUSTATUS_INPUT_REQUIRED** — the operation required user input, but could not receive it (for example, it was run on a server with no user).

Returns

A constant if the verification is successful, or throws a generic exception (**UWIException**) if an error occurs. The following table lists the possible return values:

Code	Numeric Value	Status
FormNodeP.UFL_DS_OK	0	The signature is verified.
FormNodeP.UFL_DS_ALGORITHM UNAVAILABLE	13590	The appropriate verification engine for the signature is not available.
FormNodeP.UFL_DS_F2MATCHSIGNER	13529	The certificate does not match the signer's name.
FormNodeP.UFL_DS_FAILED AUTHENTICATION	1272	The signature is invalid or the secret used is incorrect.
FormNodeP.UFL_DS_HASHCOMPFAILED	13527	The document has been tampered with.
FormNodeP.UFL_DS_NOSIGNATURE	13526	There is no signature.
FormNodeP.UFL_DS_NOTAUTHENTICATED	1240	The signer cannot be authenticated.
FormNodeP.UFL_DS_UNEXPECTED	13589	An unexpected error occurred.
FormNodeP.UFL_DS_UNVERIFIABLE	859	The signature cannot be verified.

Example

The following example uses **getSignature** to get the signature object, and uses **getDataByPath** to get the signer's identity from the signature object. It then calls **validateHMACWithSecret** to validate the signature.

```
public short checkSignature(FormNodeP theSignatureNode, Certificate theServerCert)
{
Signature theSignatureObject;
String theSecret;
String signerCommonName;
BooleanHolder encodedData;
IntHolder theStatus;
short validation;
```

```
   theSignatureObject = theSignatureNode.getSignature();
   encodedData = new BooleanHolder();
   if ((signerCommonName = theSignatureObject.getDataByPath(
      "SigningCert: Subject: CN", false, encodedData)) == null)
   {
      throw new UWIException("Could not determine signer's name.");
   }

   /* Include external code that matches the signer's identity to a shared
      secret, and sets theSecret to match.  This is most likely a
      database lookup. */

   theStatus = new IntHolder();

   validation = theSignatureNode.validateHMACWithSecret(theSecret,
      theServerCert, theStatus);

   /* Check the status in case the process required user input. */

   if (theStatus.value != SecurityUserStatusType.SUSTATUS_OK)
   {
      throw new UWIException("Validation required user input.");
   }
   return(validation);
}
```

validateHMACWithHashedSecret

Description

This method determines whether an HMAC signature is valid. HMAC signatures include both Authenticated Clickwrap and Signature Pad signatures.

For Authenticated Clickwrap signatures, you must know the hash of the signer's shared secret to use this method. For Signature Pad signatures, you may use this method without the shared secret if the signature was created without one. In any case, the shared secret should be available from a corporate database or other system.

This method will also notarize (that is, digitally sign) a valid HMAC signature if you provide a digital certificate. However, notarization will not occur if the signature does not include a shared secret. Once notarized, you must use the **verifySignature** method to validate the signature.

Note: Authenticated Clickwrap is a separately licensed product. Please ensure that your company has the license to use Authenticated Clickwrap before you provide forms or functionality that rely on it.

Method

```
public short validateHMACWithHashedSecret(
   byte [] hashedSecret,
   Certificate theServerCert,
   IntHolder theStatus,
   ) throws UWIException;
```

Parameters

Expression	Type	Description
hashedSecret	**byte[]**	The hash of the shared secret that identifies the user. This should be available from a corporate database or other system. If there is more than one shared secret, you must concatenate the strings with no separating characters and then hash the combined secret. For example, if the secrets were "blue" and "red", you would pass the hash of "bluered" to the method. If there is no shared secret, pass and empty string. You must encode the byte array as follows: **Authenticated Clickwrap (HMAC)** UTF-8 **Signature Pad** UTF-16LE You can use the getBytes method to do this. For example: `mySecret.getBytes("UTF-8")`
theCertificate	**Certificate**	The server certificate. If the HMAC signature is valid, the function will use the private key of this certificate to digitally sign the HMAC signature. This signature is appended to the signature item, and can be verified using UFLVerifySignature. If you pass null, the method will simply validate the HMAC signature.
theStatus	**IntHolder**	This is a status flag that reports whether the operation was successful. Possible values are: **SecurityUserStatusType.SUSTATUS_OK** — the operation was successful. **SecurityUserStatusType.SUSTATUS_CANCELLED** — the operation was cancelled by the user. **SecurityUserStatusType.SUSTATUS_INPUT_REQUIRED** — the operation required user input, but could not receive it (for example, it was run on a server with no user).

Returns

A constant if the verification is successful, or throws a generic exception (**UWIException**) if an error occurs. The following table lists the possible return values:

Code	Numeric Value	Status
FormNodeP.UFL_DS_OK	0	The signature is verified.

Code	Numeric Value	Status
FormNodeP.UFL_DS_ALGORITHM UNAVAILABLE	13590	The appropriate verification engine for the signature is not available.
FormNodeP.UFL_DS_F2MATCHSIGNER	13529	The certificate does not match the signer's name.
FormNodeP.UFL_DS_FAILED AUTHENTICATION	1272	The signature is invalid or the secret used is incorrect.
FormNodeP.UFL_DS_HASHCOMPFAILED	13527	The document has been tampered with.
FormNodeP.UFL_DS_NOSIGNATURE	13526	There is no signature.
FormNodeP.UFL_DS_NOT AUTHENTICATED	1240	The signer cannot be authenticated.
FormNodeP.UFL_DS_UNEXPECTED	13589	An unexpected error occurred.
FormNodeP.UFL_DS_UNVERIFIABLE	859	The signature cannot be verified.

Example

The following example uses **getSignature** to get a signature object, and uses **getDataByPath** to get the signer's identity from the signature object. Next, it calls **validateHMACWithHashedSecret** to validate the signature.

```
public short checkSignature(FormNodeP theSignatureNode, Certificate theServerCert)
{
Signature theSignatureObject;
byte [] hashedSecret;
String signerCommonName;
BooleanHolder encodedData;
IntHolder theStatus;
short validation;

   theSignatureObject = theSignatureNode.getSignature();
   encodedData = new BooleanHolder();
   if ((signerCommonName = theSignatureObject.getDataByPath(
      "SigningCert: Subject: CN", false, encodedData)) == null)
   {
      throw new UWIException("Could not determine signer's name.");
   }

   /* Include external code that matches the signer's identity to a hashed
      shared secret, sets hashedSecret to match.  This is most likely a
      database lookup. */

      theStatus = new IntHolder();

   validation = theSignatureNode.validateHMACWithHashedSecret(
      hashedSecret, theServerCert, theStatus);

   /* Check the status in case the process required user input. */

   if (theStatus.value != SecurityUserStatusType.SUSTATUS_OK)
   {
      throw new UWIException("Validation required user input.");
   }
   return(validation);
}
```

verifyAllSignatures

Description

This method verifies the correctness of all digital signatures in a given form whose root node is provided. It finds all items of type signature and calls **verifySignature** for each signature. Errors are logged for all invalid signatures.

This method checks the following conditions for each signature:
- The signature item contains mimedata.
- The mimedata contains a hash value and signer certificate.
- The signer certificate contains the same ID as that recorded in the signature item's *signer* option.
- The signer certificate has not expired.

Method

```
public short verifyAllSignatures(
 boolean reportAsErrorsFlag
 ) throws UWIException;
```

Parameters

Expression	Type	Description
reportAsErrorsFlag	**boolean**	Set to true if you want errors about the signatures to be reported by throwing a **UWIException**, or false if you want the error code to be only returned through the return value.

Returns

A short having one of the following values:

Code	Status
FormNodeP.UFL_SIGS_OK	The signatures are valid.
FormNodeP.UFL_SIGS_NOTOK	One or more signatures are broken.
FormNodeP.UFL_SIGS_UNVERIFIED	One or more signatures are unverifiable.

If one or more of the signatures is not valid and the *reportAsErrorsFlag* is *true*, a generic exception (**UWIException**) is thrown. On error, the method throws a generic exception (**UWIException**).

Example

In the following example, **verifyAllSignatures** determines whether or not all the signatures in the form are valid. If any one of the digital signatures is not valid, an error message is displayed.

```
private static void checkSignatures(FormNodeP theForm) throws Exception
{
   if (theForm.verifyAllSignatures(false) == FormNodeP.UFL_SIGS_OK)
   {
      System.out.println("All the digital signatures are valid.");
   }
}
```

verifySignature

Description

This method verifies the correctness of the given digital signature. You supply the root of the form that contains the signature you want to verify. This method checks the following conditions:

- The signature item contains mimedata.
- The mimedata contains a hash value and signer certificate.
- The signer certificate contains the same ID as that recorded in the signature item's *signer* option.
- The signer certificate has not expired.

A plain text representation of the form (filtered by the signature item's filter) is constructed and the result is hashed. This hash value must match the hash value stored in the signature.

Method

```
public short verifySignature(
   FormNodeP signatureItem,
   StringHolder theCertChain,
   boolean reportAsErrorsFlag
   ) throws UWIException;
```

Parameters

Expression	Type	Description
signatureItem	**FormNodeP**	The signature to verify.
theCertChain	**StringHolder**	Reserved. Must be null.
reportAsErrorsFlag	**boolean**	Set to true if you want errors about the signatures to be reported by throwing a **UWIException**or false if you want the error code to be returned through the return value.

Returns

A **short** having one of the following values, depending on the status of the signature:

Code	Status
FormNodeP.UFL_DS_OK	The signature is verified.
FormNodeP.UFL_DS_ALGORITHMUNAVAILABLE	The appropriate verification engine for the signature is not available.
FormNodeP.UFL_DS_CERTEXPIRED	The certificate has expired.
FormNodeP.UFL_DS_CERTNOTFOUND	The certificate cannot be located.
FormNodeP.UFL_DS_CERTNOTTRUSTED	The certificate is not trusted.
FormNodeP.UFL_DS_CERTREVOKED	The certificate has been revoked.
FormNodeP.UFL_DS_CRLINVALID	The certificate revocation list is invalid.

Code	Status
FormNodeP.UFL_DS_F2MATCHSIGNER	The certificate does not match the signer's name.
FormNodeP.UFL_DS_HASHCOMPFAILED	The document has been tampered with.
FormNodeP.UFL_DS_ISSUERCERTEXPIRED	The issuer's certificate has expired.
FormNodeP.UFL_DS_ISSUERINVALID	The issuer is invalid for the certificate used to sign.
FormNodeP.UFL_DS_ISSUERKEYUSAGE UNACCEPTABLE	The issuer certificate's key usage extension does not match what the key was used for.
FormNodeP.UFL_DS_ISSUERNOTCA	The certificate's issuer is not a Certificate Authority.
FormNodeP.UFL_DS_ISSUERNOTFOUND	The issuer's certificate was not located.
FormNodeP.UFL_DS_ISSUERSIGFAILED	Verification of the issuer's certificate failed.
FormNodeP.UFL_DS_KEYREVOKED	The key used to create the signature has been revoked.
FormNodeP.UFL_DS_KEYUSAGEUNACCEPTABLE	The certificate's key usage extension does not match what the key was used for.
FormNodeP.UFL_DS_KRLINVALID	The Key Revocation List is invalid.
FormNodeP.UFL_DS_NOSIGNATURE	There is no signature.
FormNodeP.UFL_DS_NOTAUTHENTICATED	The signer cannot be authenticated.
FormNodeP.UFL_DS_POLICYUNACCEPTABLE	The certificate's policy extension does not match the acceptable policies.
FormNodeP.UFL_DS_SIGNATUREALTERED	The signature has been tampered with.
FormNodeP.UFL_DS_UNEXPECTED	An unexpected error occurred.
FormNodeP.UFL_DS_UNVERIFIABLE	The signature cannot be verified.

If the signature is not valid and the *reportAsErrorsFlag* is *true*, a generic exception (**UWIException**) is thrown. On error, the method throws a generic exception (**UWIException**).

Example

In the following example, **dereferenceEx** is used to locate a signature node. **verifySignature** then determines whether the signature is valid. If thesignature is not valid, a message is printed.

```
private static void checkSignature(FormNodeP theForm) throws Exception
{
StringHolder certChain = new StringHolder( );
FormNodeP tempNode;

   if ((tempNode = theForm.dereferenceEx(null, "PAGE1.SIGNATURE1", 0,
```

```
      FormNodeP.UFL_ITEM_REFERENCE, null)) == null)
   {
      throw new UWIException("Could not locate SIGNATURE node.");
   }
   if (theForm.verifySignature(tempNode, certChain, false) == 0)
   {
      System.out.println("The first signature is valid.");
   }

/* If verifySignature returned a value that is equal to the FormNodeP
   constant UFL_DS_F2MATCHSIGNER, a message explaining the error is
   displayed. */

   if (theForm.verifySignature(tempNode, certChain, false) ==
      FormNodeP.UFL_DS_F2MATCHSIGNER)
   {
      System.out.println("The name in the form doesn't match the name in
         the signature.");
   }
}
```

writeForm

Description

This method will write a form to the specified file or stream. Call this method on the root node of the form. The version number of the form determines the format of the output file. You can specify whether to compress the output file and whether to observe the *transmit* and *save* settings in the form.

If no format is specified, the default is to write the form in the same format in which it was read. If the form in question was created dynamically by your application, **writeForm** will, by default, write it as an XFDL form in uncompressed format.

Method

WRITING TO A FILE:

```
public void writeForm(
   String theFilePath,
   FormNodeP triggerItem,
   int flags
   ) throws UWIException;
```

WRITING TO A STREAM:

```
public void writeForm(
   OutputStream theStream,
   FormNodeP triggerItem,
   int flags
   ) throws UWIException;
```

WRITING TO A WRITER:

```
public void writeForm(
   java.io.Writer theWriter,
   FormNodeP triggerItem,
   int flags
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theFilePath	**String**	This is the path to the file on the local disk to which the form will be written.
theStream	**OutputStream**	This is the stream to which you want to write the form data.
theWriter	**java.io.Writer**	The Java Writer to which you want to write the form data.
triggerItem	**FormNodeP**	This is the item that caused the form to be submitted. Set to null if the API receives the form in a manner other than transmission.
flags	**int**	The following flags are valid: **FormNodeP.UFL_TRANSMIT_ALLOW** allows the transmit options (that is, *transmitdatagroups*, *transmitgroups*, *transmititemrefs*, *transmititems*, *transmitoptionrefs*, *transmitpagerefs* and *transmitoptions*) to control which portions of the form are sent. Without this flag, the entire form will be sent regardless of the *transmit* options in the form. **FormNodeP.UFL_SAVE_ALLOW** allows the *saveformat* option to specify what format the form should be saved in. If no format is specified then the form will be saved in the same format that it is read. **Note:** Specify 0 if you do not want to enable any of the transmit options.

Returns

Returns nothing if the call is successful, or throws a generic exception (**UWIException**) if an error occurs.

Example

The following example uses **writeForm** to write the form in memory to a file on the local drive.

```
private static void saveForm( ) throws Exception
{
   theForm.writeForm("Output.xfd", null, 0);
}
```

xmlModelUpdate

Description

This method updates the XML data model in the form. This is necessary if computes have changed the structure of the data model in some way, such as changing or adding bindings. These sorts of changes do not take effect until the **xmlModelUpdate** method is called.

Method

```
public void xmlModelUpdate() throws UWIException;
```

Parameters

There are no parameters for this method.

Returns

Returns nothing if the call is successful, or throws a generic exception (**UWIException**) if an error occurs.

Example

The following example uses **setLiteralByRefEx** to change a binding in the form, so that it binds to a different option. It then calls **xmlModelUpdate** so that the data model reflects the change.

```
private static void setBinding throws Exception
{
   theForm.setLiteralByRefEx(null,
      "global.global.xmlmodel.bindings[0][boundoption]", 0, null, null,
      "PAGE1.FIELD5.value");
   theForm.xmlModelUpdate();
}
```

The Hash Class

The **Hash** class allows you to hash messages.

- Any application that makes calls to the **Hash** methods must first import the following class:

  ```
  com.PureEdge.security.Hash
  ```

- Many of the methods in the API will throw a generic exception called a **UWIException** if an error occurs. Import the following class to any .java files that call methods from the API:

  ```
  com.PureEdge.error.UWIException
  ```

hash

Description

This method hashes a message using the hashing algorithm of your choice.

Method

```
public byte [] hash(
   byte [] theMessage
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theMessage	**byte[]**	The message you want to hash.

Returns

A hashed message, or throws a generic exception (**UWIException**) if an error occurs.

Example

The following example uses **getSignature** to get the signature object from the signature node, and uses **getDataByPath** to get the signer's identity from the signature object. It then retrieves the signer's shared secret from a database, and hashes that secret using the **hash** method. Next, it calls **validateHMACWithHashedSecret** to validate the signature.

```
public short checkSignature(FormNodeP theSignatureNode, Certificate
   theServerCert, Hash theHashObject)
{
Signature theSignatureObject;
byte[] theSecret;
byte [] hashedSecret;
String signerCommonName;
BooleanHolder encodedData;
IntHolder theStatus;
short validation;

   theSignatureObject = theSignatureNode.getSignature();
   encodedData = new BooleanHolder();
   if ((signerCommonName = theSignatureObject.getDataByPath(
```

```
      "SigningCert: Subject: CN", false, encodedData)) == null)
   {
      throw new UWIException("Could not determine signer's name.");
   }

   /* Include external code that matches the signer's identity to a
      shared secret and sets theSecret to match.  This is most likely a
      database lookup. */

   hashedSecret = theHashObject.hash(theSecret);
   theStatus = new IntHolder();
   validation = theSignatureNode.validateHMACWithHashedSecret(
      hashedSecret, theServerCert, theStatus);

   /* Check the status in case the process required user input. */

   if (theStatus.value != SecurityUserStatusType.SUSTATUS_OK)
   {
      throw new UWIException("Validation required user input.");
   }
   return(validation);
}
```

The IFSSingleton Class

The **IFSSingleton** class provides a static interface to the application's XFDL object.

- You must import the following class to any .java files that need to access an XFDL object:

```
com.PureEdge.IFSSingleton
```

- Many of the methods in the API will throw a generic exception called a **UWIException** if an error occurs. Import the following class to any .java files that call methods from the API:

```
com.PureEdge.error.UWIException
```

getFunctionCallManager

Description

Use this method to retrieve the Function Call Manager. The Function Call Manager maintains a list of all of the packages and custom functions that are available. As such, you must register all function calls with the Function Call Manager.

Method

```
public static XFDL getFunctionCallManager() throws UWIException;
```

Parameters

There are no parameters for this method.

Returns

Returns the **FunctionCallManger** object or throws a generic exception (**UWIException**) if an error occurs.

Example

In the following example, the **FciFunctionCall** method calls **getFunctionCallManager** to obtain the Function Call Manager object called **theFCM**.

```
public FciFunctionCall(IFX IFXMan) throws UWIException
{
FunctionCallManager theFCM;
   if ((theFCM = IFSSingleton.getFunctionCallManager()) == null)
   throw new UWIException("Needed Function Call Manager")
}
```

getLocalizationManager

Description

Use this method to obtain the **LocalizationManager** object. The **LocalizationManager** object is an interface through which you can set the language the API uses to report errors.

Method

```
public static LocalizationManager getLocalizationManager() throws UWIException;
```

Parameters

There are no parameters for this method.

Returns

The **LocalizationManager** object or throws a generic exception (**UWIException**) if an error occurs.

Example

In the following example, the **setLanguage** method calls **getLocalizationManager** to obtain the **LocalizationManager** object called **theManager**.

```
private static void setLanguage() throws Exception
{
LocalizationManager theManager;
   theManager = IFSSingleton.getLocalizationManager();
   if(theManager == null)
      throw new Exception("Could not find interface");
   theManager.setDefaultLocale("en_US");
}
```

getSecurityManager

Description

This method retrieves the Security Manager object. Use the Security Manager object to retrieve the available hash algorithms.

To avoid a conflict with an existing class in Java (java.lang.SecurityManager), you must refer to the Security Manager by the full class name of com.PureEdge.security.SecurityManager.

Method

```
public SecurityManager getSecurityManager() throws UWIException;
```

Parameters

There are no parameters for this method.

Returns

A *Security Manager* object, or throws a generic exception (**UWIException**) if an error occurs.

Example

The following example uses **getSecurityManager** to get the *Security Manager* object. **lookupHashAlgorithm** is then called to get the *sha1* hash algorithm.

```
public Hash getHashAlgorithm();
{
com.PureEdge.security.SecurityManager theSecurityManager;
Hash tempHashObject;
```

```
      theSecurityManager = IFSSingleton.getSecurityManager();
      theHash = theSecurityManager.lookupHashAlgorithm("sha1");
      return(theHash);
   }
```

getXFDL

Description

Use this method to obtain the **XFDL** object. The **XFDL** object is an interface through which you can access the form's root node. As a result, any program that needs to load an XFDL form must first obtain the **XFDL** object.

Method

```
public static XFDL getXFDL() throws UWIException;
```

Parameters

There are no parameters for this method.

Returns

The **XFDL** object or throws a generic exception (**UWIException**) if an error occurs.

Example

In the following example, the **loadForm** method calls **getXFDL** to obtain the **XFDL** object called **theXFDL**.

```
private static void loadForm() throws Exception
{
XFDL theXFDL;
FormNodeP theForm;
   theXFDL = IFSSingleton.getXFDL();
   if(theXFDL == null)
      throw new Exception("Could not find interface");
   theForm = theXFDL.readForm("Sample.xfd", 0);
   if(theForm == null)
      throw new Exception("Could not load form.");
}
```

The LocalizationManager Class

The **LocalizationManager** class includes methods that control which language the API uses to report errors.

- Any application that makes calls to the **LocalizationManager** methods must first import the following class:

```
com.PureEdge.i18n.LocalizationManager
```

- Many of the methods in the API will throw a generic exception called a **UWIException** if an error occurs. Import the following class to any .java files that call methods from the API:

```
com.PureEdge.error.UWIException
```

getCurrentThreadLocale

Description

This method returns which *locale* is in use for the current thread. This determines what language the API uses when reporting errors. By default, the API uses the default locale.

The API supports the following locales:

Language	Locale	Locale Name
Chinese	Simplified Han, China	zh-Hans-CN
	Simplified Han, Singapore	zh-Hans-SG
	Traditional Han, Hong Kong S.A.R., China	zh-Hant-HK
	Traditional Han, Taiwan	zh-Hant-TW
Croatian	Croatia	hr-HT
Czech	Czech Republic	cs-CZ
Danish	Denmark	da-DK
Dutch	Belgium	nl-BE
	The Netherlands	nl-NL
English	Australia	en-AU
	Belgium	en-BE
	Canada	en-Ca
	Hong Kong S.A.R., China	en-HK
	India	en-IN
	Ireland	en-IE
	New Zealand	en-NZ
	Philippines	en-PH
	Singapore	en-SG
	South Africa	en-ZA
	United Kingdom	en-GB
	United States	en-US

© Copyright IBM Corp. 2003, 2006

Language	Locale	Locale Name
Finnish	Finland	fi-FI
French	Belgium	fr-BE
	Canada	fr-CA
	France	fr-FR
	Luxembourg	fr-LU
	Switzerland	fr-CH
German	Austria	de-AT
	Germany	de-DE
	Luxembourg	de-LU
	Switzerland	de-CH
Greek	Greece	el-GR
Hungarian	Hungary	hu-HU
Italian	Italy	it-IT
	Switzerland	it-CH
Japanese	Japan	ja-JP
Korean	South Korea	ko-KR
Norwegian Bokmål	Norway	nb-NO
Polish	Poland	pl-PL
Portuguese	Brazil	pt-BR
	Portugal	pt-PT
Romanian	Romania	ro-RO
Russian	Russian	ru-RU
Slovak	Slovakia	sk-SK
Slovene	Slovenia	sl_SI
Spanish	Argentina	es-AR
	Bolivia	es-BO
	Chile	es-CL
	Colombia	es-CO
	Costa Rica	es-CR
	Dominican Republic	es-DO
	Ecuador	es-EC
	El Salvador	es-SV
	Guatemala	es-GT
	Honduras	es-HN
	Mexico	es-MX
	Nicaragua	es-NI
	Panama	es-PA
	Paraguay	es-PY
	Peru	es-PE
	Puerto Rico	es-PR
	Spain	es-ES

Language	Locale	Locale Name
	United States	es-US
	Uruguay	es-UY
	Venezuela	es-VE
Swedish	Sweden	sv-SE
Turkish	Turkey	tr-TR

The locale name consists of two parts: the language code and the country code, as shown

```
<language code>_<COUNTRY CODE>
```

For example, to specify the Japanese locale, you would type:

```
jp_JP
```

If you need a more specific locale, you may add additional codes after the country code. For example, to indicate French in France with a Euro dialect, you would type:

```
fr_FR_EU
```

Method

```
public String getCurrentThreadLocale(
   ) throws UWIException;
```

Parameters

There are no parameters for this method.

Returns

Returns nothing if the call is successful, or throws a generic exception (**UWIException**) if an error occurs.

Example

The following method calls **getCurrentThreadLocale** to get the locale.

```
public String getThreadLocale() throws UWIException
{
String theLocale;
 LocalizationManager theManager;
 theManager = IFSSingleton.getLocalizationManager();
 theLocale = theManager.getCurrentThreadLocale();
 return theLocale;
```

getDefaultLocale

Description

This method returns the default *locale* the API uses when reporting errors.

The API supports the following locales:

Language	Locale	Locale Name
Chinese	Simplified Han, China	zh-Hans-CN

Language	Locale	Locale Name
	Simplified Han, Singapore	zh-Hans-SG
	Traditional Han, Hong Kong S.A.R., China	zh-Hant-HK
	Traditional Han, Taiwan	zh-Hant-TW
Croatian	Croatia	hr-HT
Czech	Czech Republic	cs-CZ
Danish	Denmark	da-DK
Dutch	Belgium	nl-BE
	The Netherlands	nl-NL
English	Australia	en-AU
	Belgium	en-BE
	Canada	en-Ca
	Hong Kong S.A.R., China	en-HK
	India	en-IN
	Ireland	en-IE
	New Zealand	en-NZ
	Philippines	en-PH
	Singapore	en-SG
	South Africa	en-ZA
	United Kingdom	en-GB
	United States	en-US
Finnish	Finland	fi-FI
French	Belgium	fr-BE
	Canada	fr-CA
	France	fr-FR
	Luxembourg	fr-LU
	Switzerland	fr-CH
German	Austria	de-AT
	Germany	de-DE
	Luxembourg	de-LU
	Switzerland	de-CH
Greek	Greece	el-GR
Hungarian	Hungary	hu-HU
Italian	Italy	it-IT
	Switzerland	it-CH
Japanese	Japan	ja-JP
Korean	South Korea	ko-KR
Norwegian Bokmål	Norway	nb-NO
Polish	Poland	pl-PL
Portuguese	Brazil	pt-BR
	Portugal	pt-PT
Romanian	Romania	ro-RO

Language	Locale	Locale Name
Russian	Russian	ru-RU
Slovak	Slovakia	sk-SK
Slovene	Slovenia	sl_SI
Spanish	Argentina	es-AR
	Bolivia	es-BO
	Chile	es-CL
	Colombia	es-CO
	Costa Rica	es-CR
	Dominican Republic	es-DO
	Ecuador	es-EC
	El Salvador	es-SV
	Guatemala	es-GT
	Honduras	es-HN
	Mexico	es-MX
	Nicaragua	es-NI
	Panama	es-PA
	Paraguay	es-PY
	Peru	es-PE
	Puerto Rico	es-PR
	Spain	es-ES
	United States	es-US
	Uruguay	es-UY
	Venezuela	es-VE
Swedish	Sweden	sv-SE
Turkish	Turkey	tr-TR

The locale name consists of two parts: the language code and the country code, as shown

```
<language code>_<COUNTRY CODE>
```

For example, to specify the Japanese locale, you would type:

```
jp_JP
```

If you need a more specific locale, you may add additional codes after the country code. For example, to indicate French in France with a Euro dialect, you would type:

```
fr_FR_EU
```

Method

```
public String getDefaultLocale(
   ) throws UWIException;
```

Parameters

There are no parameters for this method.

Returns

Returns nothing if the call is successful, or throws a generic exception (**UWIException**) if an error occurs.

Example

The following method calls **getDefaultLocale** to get the locale.

```
public String getLocale() throws UWIException
{
   String theLocale;
   LocalizationManager theManager;
   theManager = IFSSingleton.getLocalizationManager();
   theLocale = theManager.getDefaultLocale();
 return theLocale;
}
```

setCurrentThreadLocale

Description

This method sets which *locale* the API uses when reporting errors. By default, the API uses the application's default locale.

The API supports the following locales:

Language	Locale	Locale Name
Chinese	Simplified Han, China	zh-Hans-CN
	Simplified Han, Singapore	zh-Hans-SG
	Traditional Han, Hong Kong S.A.R., China	zh-Hant-HK
	Traditional Han, Taiwan	zh-Hant-TW
Croatian	Croatia	hr-HT
Czech	Czech Republic	cs-CZ
Danish	Denmark	da-DK
Dutch	Belgium	nl-BE
	The Netherlands	nl-NL
English	Australia	en-AU
	Belgium	en-BE
	Canada	en-Ca
	Hong Kong S.A.R., China	en-HK
	India	en-IN
	Ireland	en-IE
	New Zealand	en-NZ
	Philippines	en-PH
	Singapore	en-SG
	South Africa	en-ZA
	United Kingdom	en-GB
	United States	en-US
Finnish	Finland	fi-FI

Language	Locale	Locale Name
French	Belgium	fr-BE
	Canada	fr-CA
	France	fr-FR
	Luxembourg	fr-LU
	Switzerland	fr-CH
German	Austria	de-AT
	Germany	de-DE
	Luxembourg	de-LU
	Switzerland	de-CH
Greek	Greece	el-GR
Hungarian	Hungary	hu-HU
Italian	Italy	it-IT
	Switzerland	it-CH
Japanese	Japan	ja-JP
Korean	South Korea	ko-KR
Norwegian Bokmål	Norway	nb-NO
Polish	Poland	pl-PL
Portuguese	Brazil	pt-BR
	Portugal	pt-PT
Romanian	Romania	ro-RO
Russian	Russian	ru-RU
Slovak	Slovakia	sk-SK
Slovene	Slovenia	sl_SI
Spanish	Argentina	es-AR
	Bolivia	es-BO
	Chile	es-CL
	Colombia	es-CO
	Costa Rica	es-CR
	Dominican Republic	es-DO
	Ecuador	es-EC
	El Salvador	es-SV
	Guatemala	es-GT
	Honduras	es-HN
	Mexico	es-MX
	Nicaragua	es-NI
	Panama	es-PA
	Paraguay	es-PY
	Peru	es-PE
	Puerto Rico	es-PR
	Spain	es-ES
	United States	es-US

Language	Locale	Locale Name
	Uruguay	es-UY
	Venezuela	es-VE
Swedish	Sweden	sv-SE
Turkish	Turkey	tr-TR

The locale name consists of two parts: the language code and the country code, as shown

```
<language code>_<COUNTRY CODE>
```

For example, to specify the Japanese locale, you would type:

```
jp_JP
```

If you need a more specific locale, you may add additional codes after the country code. For example, to indicate French in France with a Euro dialect, you would type:

```
fr_FR_EU
```

Method

```
public void setCurrentThreadLocale(
   String theLocale
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theLocale	**String**	The name of the locale.

Returns

Returns nothing if the call is successful, or throws a generic exception (**UWIException**) if an error occurs.

Example

The following method checks the *language* string to determine which locale to use. It then calls **setCurrentThreadLocale** to set the appropriate locale.

```
public void setCurrentLanguage(String language) throws Exception
{
LocalizationManager theManager;

   theManager = IFSSingleton.GetLocalizationManager();
   if (language.equalsIgnoreCase("english"))
      theManager.setCurrentThreadLocale("en_US");
   else
      theManager.setCurrentThreadLocale("fr_CA");
}
```

setDefaultLocale

Description

This method sets the default *locale* the API uses when reporting errors, if no other locale is specified. By default, the API uses the locale specified by the operating system.

The API supports the following locales:

Language	Locale	Locale Name
Chinese	Simplified Han, China	zh-Hans-CN
	Simplified Han, Singapore	zh-Hans-SG
	Traditional Han, Hong Kong S.A.R., China	zh-Hant-HK
	Traditional Han, Taiwan	zh-Hant-TW
Croatian	Croatia	hr-HT
Czech	Czech Republic	cs-CZ
Danish	Denmark	da-DK
Dutch	Belgium	nl-BE
	The Netherlands	nl-NL
English	Australia	en-AU
	Belgium	en-BE
	Canada	en-Ca
	Hong Kong S.A.R., China	en-HK
	India	en-IN
	Ireland	en-IE
	New Zealand	en-NZ
	Philippines	en-PH
	Singapore	en-SG
	South Africa	en-ZA
	United Kingdom	en-GB
	United States	en-US
Finnish	Finland	fi-FI
French	Belgium	fr-BE
	Canada	fr-CA
	France	fr-FR
	Luxembourg	fr-LU
	Switzerland	fr-CH
German	Austria	de-AT
	Germany	de-DE
	Luxembourg	de-LU
	Switzerland	de-CH
Greek	Greece	el-GR
Hungarian	Hungary	hu-HU

Language	Locale	Locale Name
Italian	Italy	it-IT
	Switzerland	it-CH
Japanese	Japan	ja-JP
Korean	South Korea	ko-KR
Norwegian Bokmål	Norway	nb-NO
Polish	Poland	pl-PL
Portuguese	Brazil	pt-BR
	Portugal	pt-PT
Romanian	Romania	ro-RO
Russian	Russian	ru-RU
Slovak	Slovakia	sk-SK
Slovene	Slovenia	sl_SI
Spanish	Argentina	es-AR
	Bolivia	es-BO
	Chile	es-CL
	Colombia	es-CO
	Costa Rica	es-CR
	Dominican Republic	es-DO
	Ecuador	es-EC
	El Salvador	es-SV
	Guatemala	es-GT
	Honduras	es-HN
	Mexico	es-MX
	Nicaragua	es-NI
	Panama	es-PA
	Paraguay	es-PY
	Peru	es-PE
	Puerto Rico	es-PR
	Spain	es-ES
	United States	es-US
	Uruguay	es-UY
	Venezuela	es-VE
Swedish	Sweden	sv-SE
Turkish	Turkey	tr-TR

The locale name consists of two parts: the language code and the country code, as shown

```
<language code>_<COUNTRY CODE>
```

For example, to specify the Japanese locale, you would type:

```
jp_JP
```

If you need a more specific locale, you may add additional codes after the country code. For example, to indicate French in France with a Euro dialect, you would type:

```
fr_FR_EU
```

Method

```
public void setDefaultLocale(
   String theLocale
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theLocale	**String**	The name of the locale.

Returns

Returns nothing if the call is successful, or throws a generic exception (**UWIException**) if an error occurs.

Example

The following method checks the *language* string to determine which locale to use. It then calls **setDefaultLocale** to set the appropriate locale.

```
public void setDefaultLanguage(String language) throws Exception
{
LocalizationManager theManager;

   theManager = IFSSingleton.GetLocalizationManager();
   if language.equals("english")
      theManager.setDefaultLocale("en_US");
   else
      theManager.setDefaultLocale("fr_CA");
}
```

The SecurityManager Class

The **SecurityManager** class includes a method for obtaining hash algorithms.

- To avoid a conflict with an existing Java class (java.lang.SecurityManager), any application that makes calls to the **SecurityManager** methods must use the full class name:

  ```
  com.PureEdge.security.SecurityManager
  ```

- Many of the methods in the API will throw a generic exception called a **UWIException** if an error occurs. Import the following class to any .java files that call methods from the API:

  ```
  com.PureEdge.error.UWIException
  ```

lookupHashAlgorithm

Description

This method retrieves a hash object. Use the hash object to hash shared secrets for the **validateHMACWithHashedSecret** method.

Method

```
public Hash lookupHashAlgorithm(
   String algorithmName
   ) throws UWIException;
```

Parameters

Expression	Type	Description
algorithmName	**String**	The name of the hash algorithm you want to retrieve. The available hash algorithms are *sha1* and *md5*.

Returns

A hash object, or throws a generic exception (**UWIException**) if an error occurs.

Example

The following example uses **getSecurityManager** to get the *Security Manager* object. **lookupHashAlgorithm** is then called to get the *sha1* hash algorithm.

```
public Hash getHashAlgorithm();
{
com.PureEdge.security.SecurityManager theSecurityManager;
Hash tempHashObject;
   theSecurityManager = IFSSingleton.getSecurityManager();
   theHash = theSecurityManager.lookupHashAlgorithm("sha1");
   return(theHash);
}
```

© Copyright IBM Corp. 2003, 2006

The Signature Class

The **Signature** class allows you to work with signature objects.

- Any application that makes calls to the **Signature** methods must first import the following class:

```
com.PureEdge.security.Signature
```

- Many of the methods in the API will throw a generic exception called a **UWIException** if an error occurs. Import the following class to any .java files that call methods from the API:

```
com.PureEdge.error.UWIException
```

getDataByPath

Description

This method retrieves a piece of data from a signature object.

Method

```
public String getDataByPath(
   String thePath,
   boolean tagData,
   BooleanHolder encoded,
   ) throws UWIException;
```

Parameters

Expression	Type	Description
thePath	**String**	The path to the data you want to retrieve. See the Notes section below for more information on data paths.
tagData	**boolean**	true if the path should be prepended to the data, or false if not. If the path is prepended, an equals sign (=) is used as a separator. For example, suppose the path is ″Signing Cert: Issuer: CN″ and the data is ″IBM″. If true, the path will be prepended, producing ″CN=IBM″. If false, the path will not be prepended, and the result will be ″IBM″.
encoded	**BooleanHolder**	true if the return data is base 64 encoded, or false if not. The method returns binary data in base 64 encoding.

Notes

About Data Paths

Data paths describe the location of information within a signature, just like file paths describe the location of files on a disk. You describe the path with a series of colon separated tags. Each tag represents either a piece of data, or an object that contains further pieces of data (just like directories can contain files and subdirectories).

© Copyright IBM Corp. 2003, 2006

For example, to retrieve the version of a signature, you would use the following data path:

```
Demographics
```

However, to retrieve the signer's common name, you first need to locate the signing certificate, then the subject, then finally the common name within the subject, as follows:

```
SigningCert: Subject: CN
```

Some tags may contain more than one piece of information. For example, the issuer's organizational unit may contain a number of entries. You can either retrieve all of the entries as a comma separated list, or you can specify a specific entry by using a zero-indexed element number.

For example, the following path would retrieve a comma separated list:

```
SigningCert: Issuer: OU
```

Adding an element number of 0 would retrieve the first organizational unit in the list, as shown:

```
SingingCert: Issuer: OU: 0
```

Signature Tags

The following table lists the tags available in a signature object. Note that Clickwrap and HMAC Clickwrap signatures have additional tags (detailed in Clickwrap Signature Tags and HMAC Clickwrap Tags).

Tag	Description
Engine	The security engine used to create the signature. This is an object that contains further information, as detailed in *Security Engine Tags*.
SigningCert	The certificate used to create the signature. This is an object that contains further information, as detailed in *Certificate Tags*. Note that this object does not exist for Clickwrap or HMAC Clickwrap signatures.
HashAlg	The hash algorithm used to create the signature.
CreateDate	The date on which the signature was created.
Demographics	A string describing the signature.
LastVerificationStatus	A short representing the verification status of the signature. This is updated whenever the signature is verified. See "verifySignature" on page 109 for a complete list of the possible values.

Clickwrap Signature Tags

The following table lists additional tags available in both Clickwrap and HMAC Clickwrap signatures. Note that HMAC Clickwrap signatures have further tags (detailed in HMAC Clickwrap Tags).

Tag	Description
TitleText	The text for the Windows title bar of the signature dialog box.
MainPrompt	The text for the title portion of the signature dialog box.

Tag	Description
MainText	The text for the text portion of the signature dialog box.
Question1Text	The first question in the signature dialog box.
Answer1Text	The signer's answer.
Question2Text	The second question in the signature dialog box.
Answer2Text	The signer's answer.
Question3Text	The third question in the signature dialog box.
Answer3Text	The signer's answer.
Question4Text	The fourth question in the signature dialog box.
Answer4Text	The signer's answer.
Question5Text	The fifth question in the signature dialog box.
Answer5Text	The signer's answer.
EchoPrompt	Text that the signer must echo to create a signature.
EchoText	The signer's response to the echo text.
ButtonPrompt	The text that provides instructions for the Clickwrap signature buttons.
AcceptText	The text for the accept signature button.
RejectText	The text for the reject signature button.

Certificate Tags

The following table lists the tags available in a certificate object. Note that Clickwrap and HMAC Clickwrap signatures do not contain these tags.

Tag	Description
Subject	The subject's distinguished name. This is an object that contains further information, as detailed in Distinguished Name Tags.
Issuer	The issuer's distinguished name. This is an object that contains further information, as detailed in Distinguished Name Tags.
IssuerCert	The issuer's certificate. This is an object that contains the complete list of certificate tags.
Engine	The security engine that generated the certificate. This is an object that contains further information, as detailed in Security Engine Tags.
Version	The certificate version.
BeginDate	The date on which the certificate became valid.
EndDate	The date on which the certificate expires.
Serial	The certificate's serial number.
SignatureAlg	The signature algorithm used to sign the certificate.
PublicKey	The certificate's public key.
FriendlyName	The certificate's friendly name.

Distinguished Name Tags

The following table lists the tags available in a distinguished name object. Note that Clickwrap and HMAC Clickwrap signatures do not contain these tags.

Tag	Description
CN	The common name.
E	The e-mail address.
T	The title.
O	The organization.
OU	The organizational unit.
C	The country.
L	The locality.
ST	The state.
All	The entire distinguished name.

HMAC Clickwrap Tags

The following table lists the tags available in HMAC Clickwrap signature. Note that these tags are in addition to both the regular Signature Tags and the Clickwrap Signature Tags.

Tag	Description
HMACSigner	A string indicating which answers store the signer's ID.
HMACSecret	A string indicating which answers store the signer's secret.
Notarization	The notarizing signatures. This is one or more signature objects that contain further information, as detailed in Signature Tags . There can be any number of notarizing signatures. Use an element number to retrieve a specific signature. For example, to get the first notarizing signature use: `Notarization: 0` If no element number is provided, the data will be retrieved from the first valid notarizing signature found. If no valid notarizing signatures are found, the method will return null.

Security Engine Tags

The following table lists the tags available in the security engine object:

Tag	Description
Name	The name of the security engine used by the server.
Help	The help text for the security engine.
HashAlg	A hash algorithm supported by the security engine.

Returns

A string containing the certificate data (null if no data is found), or throws a generic exception (**UWIException**) if an error occurs.

Example

The following example uses **getSignature** to get the signature object from the signature node, and uses **getDataByPath** to get the signer's identity from the signature object. It then calls **validateHMACWithSecret** to validate the signature.

```
public short checkSignature(FormNodeP theSignatureNode, Certificate
   theServerCert)
{
Signature theSignatureObject;
String theSecret;
String signerCommonName;
BooleanHolder encodedData;
IntHolder theStatus;
short validation;

   theSignatureObject = theSignatureNode.getSignature();
   encodedData = new BooleanHolder();
   if ((signerCommonName = theSignatureObject.getDataByPath(
      "SigningCert: Subject: CN", false, encodedData)) == null)
   {
      throw new UWIException("Could not determine signer's name.");
   }

   /* Include external code that matches the signer's identity to a shared
      secret, and sets theSecret to match.  This is most likely a
      database lookup. */

   theStatus = new IntHolder();

   validation = theSignatureNode.validateHMACWithSecret(theSecret,
      theServerCert, theStatus);

   /* Check the status in case the process required user input. */

   if (theStatus.value != SecurityUserStatusType.SUSTATUS_OK)
   {
      throw new UWIException("Validation required user input.");
   }
   return(validation);
}
```

getSigningCert

Description

This method retrieves the signing certificate from a signature object.

Method

```
public Certificate getSigningCert() throws UWIException;
```

Parameters

There are no parameters for this method.

Returns

The signing certificate.

Example

The following example gets the signing certificate from a signature object, then iterates through the certificate issuers until it reaches the end of the chain. During the iteration, each certificate is passed to a method that processes them.

```
public void processCertChain(Signature theSig)
{
Certificate theCert, issuerCert;
IntHolder theStatus;

   /* Get the signing certificate from the signature. */

   theCert = theSig.getSigningCert();

   /* Loop through the certificate chain, passing each certificate to the
      ProcessCert function. The loop ends when the issuer certificate is
      null. */

   while (theCert != null)
   {

      /* Pass the certificate to the processCert method.  Note that
         this is not an API method, but rather a method you would
         write to process the certificate in some way. */

      ProcessCert(theCert);

      /* Get the issuer certificate from theCert. */

      issuerCert = theCert.getIssuer(theStatus);

      /* Check to ensure the method exited with the correct status. */

      if (theStatus.value != SecurityUserStatusType.SUSTATUS_OK)
      {
         throw new UWIException("getBlob exited with the wrong status.");
      }

      /* Assign theCert to equal the issuerCert for next iteration of the
         loop. */

      theCert = issuerCert;
   }
   return(OK);
}
```

The XFDL Class

The **XFDL** class includes methods that create the root nodes of forms and handle administrative tasks related to the Form Library.

- To use the **XFDL** methods in an application, import the following class to any .java files that call **XFDL** methods:

  ```
  com.PureEdge.xfdl.XFDL
  ```

- Many of the methods in the API will throw a generic exception called a **UWIException** if an error occurs. Import the following class to any .java files that call methods from the API:

  ```
  com.PureEdge.error.UWIException
  ```

create

Description

This method creates a new FormNodeP and attaches it to the form hierarchy at the indicated location. Once created, the type and identifier of a FormNodeP cannot be changed.

Note that you can also use **setLiteralByRefEx** to create a **FormNodeP** at the option level and below. Using **setLiteralByRefEx** is often easier and faster than using **create**.

Method

```
public FormNodeP create(
   FormNodeP aNode,
   int where,
   String theType,
   String theLiteral,
   String theFormula,
   String theIdentifier
   ) throws UWIException;
```

Parameters

Expression	Type	Description
aNode	**FormNodeP**	The new FormNodeP is placed in the form hierarchy in relation to this node. If null, this creates a new FormNodeP hierarchy (a new form)

© Copyright IBM Corp. 2003, 2006

Expression	Type	Description
where	int	A constant that describes the location, in relation to the parameter *aNode*, in which the new node should be placed: **XFDL.UFL_APPEND_CHILD** — adds the new node as the last child of aNode. **XFDL.UFL_AFTER_SIBLING** — adds the new node as a sibling of *aNode*, placing it immediately after that node. **XFDL.UFL_BEFORE_SIBLING —** adds the new node as a sibling of *aNode*, placing it immediately before that node. Note: If the parameter *aNode* is null, then this parameter should be set to 0.

theType	**String**	The type to assign to the FormNodeP being created. This is only necessary for page and item nodes. Use null for all other nodes. The type cannot be changed after the node has been created.
		
		If you are creating a non-XFDL node, you must also include the namespace that the node should belong to, as shown:
		`<namespace prefix>:<type>`
		
		For example:
		`custom:myItem`
		
		If you do not provide a namespace, the method will assign the default namespace for the form.

theLiteral	**String**	The literal to assign to this FormNodeP. null is valid.
theFormula	**String**	The formula to assign to this FormNodeP. null is valid.

theIdentifier	**String**	The identifier to assign to this FormNodeP. The identifier cannot be changed after the node has been created. null is valid.
		
		If you are creating an option or argument level node, this must also include the namespace the node should belong to. Use the following format:
		`<namespace prefix>:<type>`
		
		For example:
		`custom:myOption`
		
		If you do not provide a namespace, the method will assign the default namespace for the form.

Returns

The new **FormNodeP** or throws a generic exception (**UWIException**) if an error occurs.

Example

In the following example, **dereferenceEx** is used to locate a specific node. **create** is then used to create a sibling to that node and to place it directly after that node in the form structure.

```
private static void addPicLabel(FormNodeP theForm) throws Exception
{
FormNodeP tempNode;
XFDL theXFDL;

   if ((theXFDL = (XFDL)IFXMan.lookupInterface(XFDL.XFDL_INTERFACE_NAME,
      XFDL.XFDL_CURRENT_VERSION, 0, null, null)) == null)
      throw new UWIException("Could not find interface");

/* Call theForm.dereference to locate the node for the gender label item. */

   if ((tempNode = theForm.dereferenceEx(null, "PAGE1.GENDERLABEL", 0,
      FormNodeP.UFL_ITEM_REFERENCE, null)) == null)
   {
      throw new UWIException("Could not locate GENDERLABEL node.");
   }
   tempNode = theXFDL.create(tempNode, XFDL.UFL_AFTER_SIBLING,
      "label", null, null,"PICLABEL")
}
```

getEngineCertificateList

Description

This method locates all available certificates for a particular signing engine.

Method

```
public Certificate [ ] getEngineCertificateList(
   String engineName,
   IntHolder theStatus,
   ) throws UWIException;
```

Parameters

Expression	Type	Description
engineName	**String**	The name of the signing engine. Valid signing engines include: Generic RSA, CryptoAPI, Netscape, and Entrust. (Note that Generic RSA is the union of CryptoAPI and Netscape.)
theStatus	**IntHolder**	This is a status flag that reports whether the operation was successful. Possible values are: **SecurityUserStatusType.SUSTATUS_OK** — the operation was successful. **SecurityUserStatusType.SUSTATUS_CANCELLED** — the operation was cancelled by the user. **SecurityUserStatusType.SUSTATUS_INPUT_REQUIRED** — the operation required user input, but could not receive it (for example, it was run on a server with no user).

Returns

An array containing the list of certificates objects.

Example

The following method uses **getXFDL** and **getEngineCertificateList** to get a list of valid certificates for the CryptoAPI signing engine. Next, the method cycles through the returned certificates and uses **getDataByPath** to find the certificate with a common name of "Workplace Forms Server". **getDataByPath** is then used to retrieve the common name from the existing signature, which is used to retrieve the a shared secret from a database. The method then uses **validateHMACWithSecret** to validate the signature and notarize it using the server certificate.

```
public short serverNotarize(FormNodeP theSignatureNode) throws UWIException
{
XFDL theXFDL;
IntHolder theCertStatus;
IntHolder theSigStatus;
Certificate [] certList;
Signature theSignatureObject;
String theSecret;
String signerCommonName;
booleanHolder encodedData;
int certCount;
int correctCert = -1;
int i;
short validation;

   if ((theXFDL = IFSSingleton.getXFDL()) == null)
   {
      throw new Exception("Could not find interface");
   }
   theCertStatus = new IntHolder();
   if ((certList = theXFDL.getEngineCertificateList("CryptoAPI",
      theCertStatus)) == null)
   {
      throw new Exception("Could not locate any certificates.");
   }
   if (theStatus.value == SecurityUserStatusType.SUSTATUS_INPUT_REQUIRED)
   {
      throw new UWIException("User input required to sign form.");
   }

   /* Loop through the certificates to find the Workplace Forms Server
      certificate */

   certCount = certList.length;
   encodedData = new BooleanHolder();
   for (i=0; i<certCount; i++)
   {
      signerCommonName = certList[i].getDataByPath(
         "SigningCert: Subject: CN", false, encodedData);
      if (signerCommonName.equals("Workplace Forms Server"))
      {
         correctCert = i;
         break;
      }
   }
   if (correctCert == -1)
   {
      throw new UWIException("Could not locate required certificate");
```

```
   }

   /* Get the signature object. */

   theSignatureObject = theSignatureNode.getSignature();

   /* Get the signer's common name from the signature object */

   encodedData = new BooleanHolder();
   if ((signerCommonName = theSignatureObject.getDataByPath(
      "SigningCert: Subject: CN", false, encodedData)) == null)
   {
      throw new UWIException("Could not determine signer's name.");
   }

   /* Include external code that matches the signer's identity to a shared
      secret, and sets theSecret to match.  This is most likely a
      database lookup. */

   theSigStatus = new IntHolder();

   /* Validate the signature and notarize using the server certificate */

   validation = theSignatureNode.validateHMACWithSecret(theSecret,
      certList[correctCert], theSigStatus);

   /* Check the status in case the process required user input. */

   if (theStatus.value != SecurityUserStatusType.SUSTATUS_OK)
   {
      throw new UWIException("Validation required user input.");
   }
   return(validation);
}
```

isDigitalSignaturesAvailable

Description

This method determines whether digital signatures are available on the current computer.

Method

```
public boolean isDigitalSignaturesAvailable( ) throws UWIException;
```

Parameters

There are no parameters for this method.

Returns

true if digital signatures are available on this computer; otherwise, *false*. On error, the method throws a generic exception (**UWIException**).

Example

In the following example, **isDigitalSignaturesAvailable** is used to determine whether or not digital signatures are available. A message is then printed which indicates the availability of digital signatures.

```
private static void sigsAvailable() throws UWIException
{
XFDL theXFDL;
   if ((theXFDL = (XFDL)IFXMan.lookupInterface(XFDL.XFDL_INTERFACE_NAME,
      XFDL.XFDL_CURRENT_VERSION, 0, null, null)) == null)
      throw new Exception("Could not find interface");
   if (theXFDL.isDigitalSignaturesAvailable( )== true)
   {
      System.out.println("Digital signatures are available.");
   }
   else
   {
      System.out.println("Digital signatures are not available.");
   }
}
```

readForm

Description

This method will read a form into memory from a specified file Java Reader, or input stream.

Method

READING A FILE:

```
public FormNodeP readForm(
   String theFilePath,
   int flags
   ) throws UWIException;
```

READING A STREAM:

```
public FormNodeP readForm(
   InputStream theStream,
   int flags
   ) throws UWIException;
```

READING A READER:

```
public FormNodeP readForm(
   java.io.Reader theReader,
   int flags
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theFilePath	**String**	The path to the source file on the local disk.
theStream	**InputStream**	This is the stream that will read the form data. Note that this data must be UTF-8.
theReader	**java.io.Reader**	This is the reader that will read the form data.

Expression	Type	Description
flags	**int**	The following flags cause special behaviors. If using multiple flags, combine them using a bitwise OR. For example: `UFL_AUTOCOMPUTE_OFF	UFL_AUTOCREATE_FORMATS_OFF` **0** — no special behavior. **XFDL.UFL_AUTOCOMPUTE_OFF** — Reads the form into memory, but disables the compute system so that no computes are evaluated. **XFDL.UFL_AUTOCREATE_CONTROLLED_OFF** — Reads the form into memory, but disables the creation of all options that are maintained only in memory (for example, itemnext, itemprevious, pagenext, pageprevious, and so on). **XFDL.UFL_AUTOCREATE_FORMATS_OFF** — Reads the form into memory, but disables the evaluation of all format options. **XFDL.UFL_SERVER_SPEED_FLAGS** — Turns off the following features: computes, automatic formatting, duplicate sid detection, the event model, and relative page and item tags (for example, itemprevious, itemnext, and so on). This setting significantly improves server processing times. **XFDL.UFL_XFORMS_INITIALIZE_ONLY** — Turns off the following features: controlled item construction, UI connection to the XForms model, action handling set up, and the rebuild/recalculate/revalidate/refresh sequence after instance replacements.

Returns

Returns a new **FormNodeP** that is the root node of the form, or throws a generic exception (**UWIException**) if an error occurs.

Notes

Duplicate Scope IDs

If a form contains duplicate scope IDs (for example, two items on the same page with the same SID), **readForm** will fail to read the form and will return an error. This enforces correct XFDL syntax, and eliminates certain security risks that exist when duplicate scope IDs appear in signed forms.

Digital Signatures

When a form containing one or more digital signatures is read, the signatures will be verified. The result of the verification is stored in a flag that can be checked by calling **getSignatureVerificationStatus**.

Note that this flag is only set by **readForm**, and its value will not be adjusted by changes made to the form after it has been read. This means that calls such as **setLiteralEx** may actually break a signature (by changing the value of a signed item), but that this will not adjust the flag's value. To verify a signature after changes have been made to a form, it is best to use **verifyAllSignatures**.

Note that when a form is signed, all signed computes are frozen at their start value (regardless of whether the compute system is disabled).

Server-Side Processing

Using the XFDL.UFL_SERVER_SPEED_FLAGS setting significantly improves performance during server-side processing. We strongly recommend you use this flag if you do not require computes to update while processing the form.

Example

The following example demonstrates the use of **readForm** to load a form into memory, and then returns to the root node of the form.

```
   private static FormNodeP loadForm() throws Exception
   {
   XFDL theXFDL;
formNodeP theForm;
      if ((theXFDL = IFSSingleton.getXFDL()) == null)
         throw new Exception("Could not find interface");
      if ((theForm = theXFDL.readForm("formSample.xfd", 0)) == null)
         throw new Exception("Could not load form.");
      return(theForm);
   }
```

Introduction to the FCI Library

The Function Call Interface (FCI) API provides a means for creating extremely powerful form applications in a simple and elegant manner.

The FCI Library is a collection of methods for developing custom-built functions that form developers may call from XFDL forms. By creating custom functions, you can extend the capabilities of forms without requiring an upgrade to either your forms software or the form description language (XFDL). Using the methods from this library you can:

- Create packages of functions for forms.
- Set up the packages as extensions for Workplace Forms products, such as Viewer or Designer.
- Determine how and when the functions are used. For example, you can specify that a function should run when a form opens, when it closes and so on.

About Functions, Packages and Extensions

The purpose of the FCI is to make the functionality of forms extensible without requiring updates to your forms driver software. This API allows you to create self-contained modules called *extensions* that provide *packages* of *functions* for use in XFDL forms.

Note: The forms driver software is any application that initializes and calls on the API.

Functions can be used almost anywhere in an XFDL form; the appropriateness of their use depends mainly on their behavior. For instance the *XFDL Specification* contains a default package of functions called **system**. Every application built with the API version 4.4 or greater can use these functions.

Functions are grouped together to form packages. When you call a function from a form, you must include the function's package name in the call. For example, the function **beep** is part of the package called **my_funcs**. To call the beep function from a form and assign the result to the form option *do_beep* you would type the following:

```
<label sid = "do_beep">
   <value compute = "my_funcs.beep( )"></value>
</label>
```

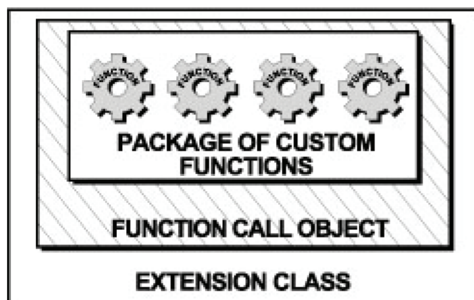
The most common use of a function is to return a value that is used to set a form option, such as the *value* of a field. For example, the **toupper** function in the **system** package, which converts a string to upper case and returns the result, might be used to set the *value* of a particular form field. This method could take as its sole argument the value of a label elsewhere on the form (or on another form) and convert it to upper case as follows:

```
<label sid = "SomeLabel">
   <value>"I am a label"</value>
</label>
<field sid = "SomeField">
   <size>
      <ae>20</ae>
```

© Copyright IBM Corp. 2003, 2006

```
      <ae>1</ae>
   </size>
   <value compute = "system.toupper(SomeLabel.value)"></value>
</field>
```

FunctionCall: To create a package of functions you must create an *extension*. The extension provides services for function calls via a *object*. The **FunctionCall** object contains your package(s) of custom-built functions.



Refer to the "The FCI Extension Architecture" on page 151 for more information. Or, for a practical guide to building your own extensions and functions refer to the section called "Getting Started with the FCI Library" on page 155.

Use the following rules to help you define your own packages and extensions:
- Each package can contain multiple functions.
- Each extension can contain multiple packages, however it is easier to define one package per extension.
- All package names must contain an underscore. IBM reserves all other package names. Refer to page "Package Naming Conventions" on page 161 for more information.
- The *XFDL Specification* contains a default package of functions called **system**. Every application built with the API version 4.4 or greater can use these functions.
- You cannot add to the system package of functions. For details on the system functions, see the *XFDL Specification*.

Once you have created your extensions you can embed them directly into XFDL forms, or you can distribute them to users as Java Archive files (JARs) or as ZIP files. Refer to "Distributing Extensions for Testing or Use" on page 165 for more information.

Note: In order to view the forms provided with this API, you must have a licensed or evaluation copy of Workplace Forms Viewer installed.

About the Function Call Interface (FCI)

The FCI is itself an extension. It is currently only available for Windows 32-bit applications. A set of Java wrapper classes, supplied as a Java Archive file (JAR file) or ZIP file, provide a Java interface to the DLL.

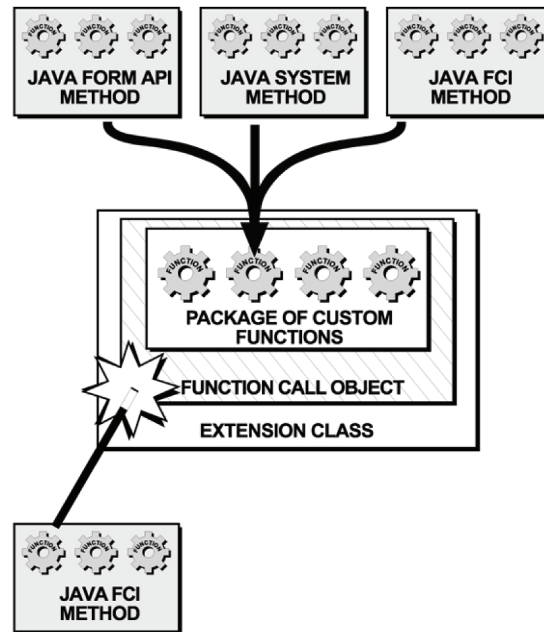
How the Form and FCI Libraries Work Together

The Form Library provides developers with tools for accessing and manipulating XFDL forms as structured data types. For instance, methods in the Form Library

will provide your applications with a means for reading and writing forms, retrieving information contained in form elements or assigning information to the elements of a form. For more information about the Form Library refer to "Introduction to the Form Library" on page 15.

The FCI Library of methods allows you to create an extension structure that contains one or more packages of your custom functions.

Once you have set up the framework for your custom functions you can use Java system methods, Form Library methods or even other FCI methods to implement the details of each function.

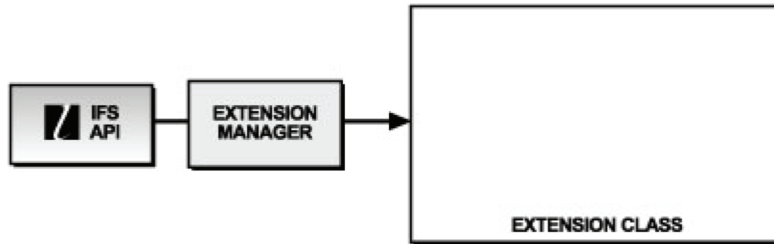


The FCI Extension Architecture

Extensions can exist in any of the following locations:

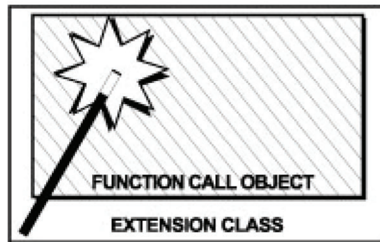
- The extensions folder of the Workplace Forms product that will use the extension (for example, the Viewer or Designer products).
- The API extensions folder, <Windows System>\PureEdge\extensions.
- The Java source folder, <Windows System>\PureEdge\java\source.
- Enclosed within XFDL forms.

When the Forms System is initialized, the API checks for extensions. If it finds any, it calls the initialization method for each extension and passes each method an object called the *IFX Manager*.



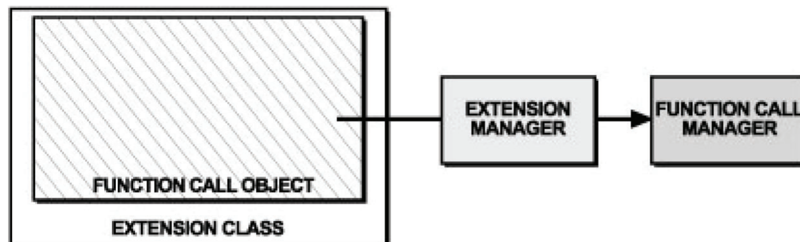
1. THE IFS API PASSES EACH METHOD AN OBJECT CALLED THE INTERNETFORMS EXTENSION MANAGER OR THE IFX MANAGER.

As part of the initialization, those extensions that provide a function call interface create one or more **FunctionCall** objects.



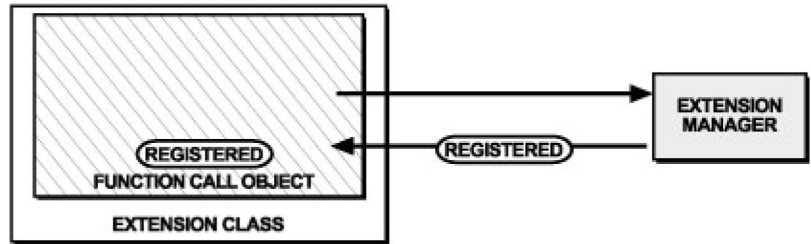
2. THE EXTENSION CLASS CREATES A FUNCTION CALL OBJECT

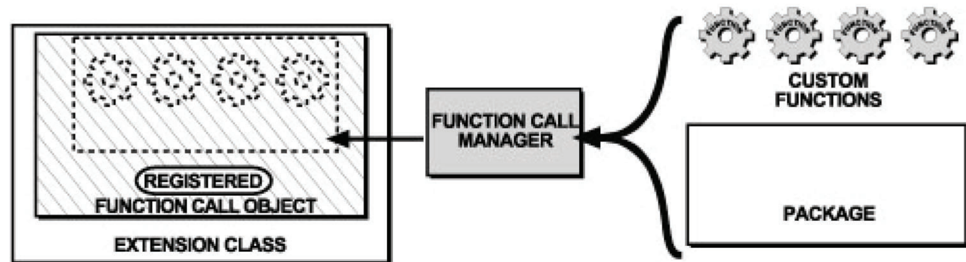
Then, each **FunctionCall** object requests a **FunctionCallManager** object from the IFX Manager.



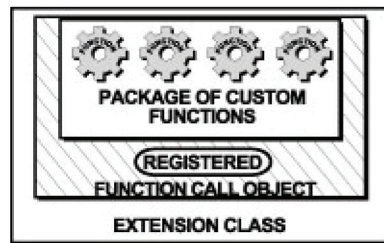
3. THE FUNCTION CALL OBJECT REQUESTS THE FUNCTION CALL MANAGER FROM THE EXTENSION MANAGER

Each **FunctionCall** object registers itself with the IFX Manager as a function call and then registers your custom-built functions and corresponding packages with the Function Call Manager.

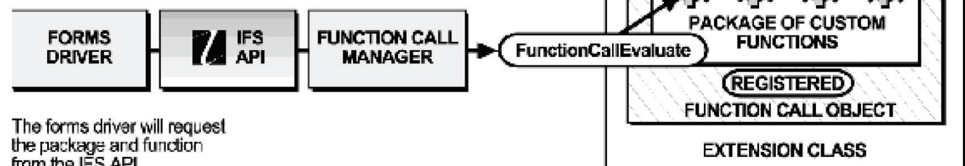




The final result is an extension containing a registered **FunctionCall** object. The registered **FunctionCall** object contains your package of custom functions.



When a function is called in a form, the forms driver requests the package and function from the API. The API will use the Function Call Manager to locate the **FunctionCall** object that contains the requested function and evaluate it.



Note: The forms driver software is any application that initializes and calls the API.

Getting Started with the FCI Library

This section acts as both a reference and a tutorial on the Function Call Interface Library. A series of practical examples is provided which you may work through to build a package of functions called **sample_package.** This section shows you how to build **sample_package** and one function called **convertDate** that converts a date to a language and format specific to another country. Try adding other functions to the package for more practice using the FCI Library of methods.

Although the FCI Library contains many methods, you only need to use a few of them to create a simple package of functions. These are:

- lookupInterface
- registerInterface
- registerFunctionCall
- evaluate
- help

The remaining FCI methods allow you to customize the behavior of your functions and extensions. For example, you can attach additional information to a particular extension, or get a list of currently registered extensions.

Refer to the "FCI Library Quick Reference Guide" on page 169 for a detailed description of the classes and methods used in this API.

Note: Before you can build extensions and functions using the FCI methods, you must set up your development environment. Refer to the *IBM Workplace Forms Server - API Installation and Setup Guide* for more information.

Creating Extensions with the FCI methods

The following table is a guide for creating extensions using the Function Call Interface. Refer to the corresponding page numbers for more details:

Procedure	Page
Install the Java Edition of the Workplace Forms Server - API and related files, as outlined in the *IBM Workplace Forms Server - API Installation and Setup Guide*.	N/A
Set up the extension.	"Creating the Extension class" on page 156
Create the **Extension** class.	"Creating the Extension class" on page 156
Create the extension initialization method.	"Implementing the extension initialization method" on page 156
Create a new **FunctionCall** object.	"Creating a new FunctionCall object" on page 157
Set up the function call.	"Creating a FunctionCall class" on page 157
Create the **FunctionCall** class.	"Creating a FunctionCall class" on page 157
Retrieve the Function Call Manager.	"Retrieving the Function Call Manager" on page 159

© Copyright IBM Corp. 2003, 2006

Procedure	Page
Register each **FunctionCall** object with the IFX Manager.	"Registering the FunctionCall object with the IFX Manager" on page 159
Register your package(s) of custom functions with the Function Call Manager.	"Registering your packages of custom functions with the Function Call Manager" on page 160
Implement your custom functions.	"Implementing your custom functions" on page 161
Provide help information for each of your functions.	"Providing help information for each of your functions" on page 162
Build the extension.	"Building the extension" on page 163
Distribute the extension for Testing or Use.	"Distributing Extensions for Testing or Use" on page 165

Creating Extensions with the FCI methods

Creating the Extension class

When the Forms System is initialized, the API checks for existing extensions and calls the initialization method (**extensionInit**) for each extension. Your first step in creating a function call is to create an **Extension** class that generates a new **FunctionCall** object. Follow the procedure below to create the **Extension** class called **FCIExtension**:

1. Create a new Java source file called **FCIExtension.java**.
2. Define the Java package. For example:
```
com.yourcompany.samples;
```
3. Import the following files and any other required files to any Java files that call FCI methods. These lines must be placed before any class or interface definitions:
```
import com.PureEdge.ifx.IFX;
import com.PureEdge.ifx.ExtensionImplBase;
import com.PureEdge.ifx.Extension;
import com.PureEdge.xfdl.FunctionCall;
import com.PureEdge.xfdl.FunctionCallManager;
import com.PureEdge.xfdl.FormNodeP;
import com.PureEdge.IFSUserDataHolder;
import com.PureEdge.error.UWIException;
```

Note: If you are using methods from the Form Library, you must import the necessary packages. For more information, refer to "Setting Up Your Application" on page 17.

Implementing the extension initialization method

The API will initialize an extension by calling the **extensionInit** method and passing the method an object known as the IFX Manager.

Implement the **extensionInit** method as part of the **Extension** class.

- **extensionInit** is the main function within the **Extension** interface. It is responsible for the registration of all the services that the extension provides.
- The following is an example of the **extensionInit** method in the **FCIExtension** class.

```
public class FCIExtension extends ExtensionImplBase implements
Extension
{
   public void extensionInit(IFX IFXMan) throws UWIException
   {
      /* Additional code removed */
   }
}
```

- The **IFXMan** object represents the IFX Manager. Through this object all other objects and services can be reached.
- **UWIException** is a generic exception.

Creating a new FunctionCall object

The **extensionInit** method creates a new **FunctionCall** object that contains your custom-built functions.

To create a new **FunctionCall** object you must define a **FunctionCall** class that contains your custom functions. Refer to ″Setting up the FunctionCall Class″ for more details.

1. Declare a new **FunctionCall** object before you create it in the **extensionInit** method.
   - The following example from the **FCIExtension** class declares a **FunctionCall** object called **theFunctionObject**.

     ```
     public class FCIExtension extends ExtensionImplBase implements
     Extension
     {
     private FunctionCall theFunctionObject;
        public void extensionInit(IFX IFXMan) throws UWIException
        {
           /* Additional code removed */
        }
     }
     ```
2. Create a new **FunctionCall** object inside the **extensionInit** method, by calling the **FunctionCall** class constructor that you will build in the next section.
   - In the following example, **extensionInit** creates a new **FunctionCall** object by calling the **FunctionCall** class constructor **FciFunctionCall** and passing it the IFX Manager.

     ```
     public class FCIExtension extends ExtensionImplBase implements
     Extension
     {
     private FunctionCall theFunctionObject;
        public void extensionInit(IFX IFXMan) throws UWIException
        {
           this.theFunctionObject = new FciFunctionCall(IFXMan);
        }
     }
     ```

Setting up the FunctionCall Class

Creating a FunctionCall class

The **FunctionCall** class contains definitions for your custom functions. It also registers the **FunctionCall** object and each of the custom functions that it supports with the Forms System so that the functions and packages that it contains will be recognized.

1. Create a new Java source file called **FciFunctionCall.java**.
2. Define the Java package. For example:
```
com.yourcompany.samples;
```
3. Import the following API packages:
```
com.PureEdge.ifx.IFX
com.PureEdge.xfdl.FormNodeP
com.PureEdge.xfdl.FunctionCallManager
com.PureEdge.xfdl.FunctionCallImplBase
com.PureEdge.xfdl.FunctionCall
com.PureEdge.error.UWIException
```
4. Import any other required files. In this case the following files are needed to implement the **convertDate** function:
```
java.util.Date
java.util.Locale
java.text.DateFormat
java.text.SimpleDateFormat
java.text.ParseException
```
5. Create a **FunctionCall** class that extends the pre-defined superclass **com.PureEdge.xfdl.FunctionCallImplBase** and implements the pre-defined interface **FunctionCall**.
   - In the following example the name of the **FunctionCall** class is **FciFunctionCall.**
```
public class FciFunctionCall extends FunctionCallImplBase
   implements FunctionCall
   {
      /* Additional code removed */
   }
```
6. Define a unique identification number for each custom function that you are going to create using the FCI.
   - In the following example, **FciFunctionCall** contains a function called **convertDate** that converts any date to the date format and language of a specific country. The **convertDate** function in **FciFunctionCall** has an ID number of 1:
```
public class FciFunctionCall extends FunctionCallImplBase
   implements FunctionCall
   {
      public static final int CONVERTDATE = 1;
      /* Additional code removed */
   }
```
7. Define a **FunctionCall** class constructor that takes as its parameter the IFX Manager.
   - In the following example, the constructor for the **FciFunctionCall** class is **FciFunctionCall.**
```
public class FciFunctionCall extends FunctionCallImplBase
   implements FunctionCall
   {
      public static final int CONVERTDATE = 1;

      public FciFunctionCall(IFX IFXMan) throws UWIException
      {
            /* Additional code removed */
      }
   }
```
   - The **IFXMan** object represents the IFX Manager. Through this object all other objects and services can be reached.
   - **UWIException** is a generic exception.

Retrieving the Function Call Manager

The Function Call Manager is used to handle services specific to function calls, such as handling requests for a particular function. The Function Call Manager is represented by a **FunctionCallManager** object.

1. Declare the Function Call Manager before requesting it from the IFX Manager.
   - In the following example, the **FciFunctionCall** constructor declares the Function Call Manager with the type **FunctionCallManager**.

     ```
     public FciFunctionCall(IFX IFXMan) throws UWIException
     {
        FunctionCallManager theFCM;
     }
     ```
2. Use the **IFSSingleton** method **getFunctionCallManager** in the function call constructor to request a **FunctionCallManager** object from the IFX Manager.
   - The **getFunctionCallManager** call requests the Function Call Manager from the IFX Manager.
   - The return value of the **getFunctionCallManager** method is a generic object, and must be typecast to the object type you have requested. In this case, the object returned from **getFunctionCallManager** is typecast to **FunctionCallManager**.
   - In the following example the **FciFunctionCall** constructor requests the Function Call Manager (**theFCM**). Notice that before the Function Call Manager is returned, it is explicitly cast to the type **FunctionCallManager**.

     ```
     public FciFunctionCall(IFX IFXMan) throws UWIException
        {
        FunctionCallManager theFCM;
           if ((theFCM = IFSSingleton.getFunctionCallManager()) == null)
              throw new UWIException("Needed Function Call Manager");
        }
     ```

Note: For detailed information about the method, including a description of its parameters, refer to “getFunctionCallManager” on page 117.

Registering the FunctionCall object with the IFX Manager

Each **FunctionCall** object registers itself with the IFX Manager as an interface that provides function call support.

In the **FunctionCall** class constructor, register the function call with the IFX Manager using the method **registerInterface**.

- In the following example the FciFunctionCall constructor uses the **registerInterface** method to register itself with the IFX Manager as a **FunctionCall** object:

  ```
  public FciFunctionCall(IFX IFXMan) throws UWIException
  {
  FunctionCallManager theFCM;
     if ((theFCM = IFSSingleton.getFunctionCallManager()) == null)
        throw new UWIException("Needed Function Call Manager");
     IFXMan.registerInterface(this,
        FunctionCall.FUNCTIONCALL_INTERFACE_NAME,
        FunctionCall.FUNCTIONCALL_CURRENT_VERSION,
        FunctionCall.FUNCTIONCALL_MIN_VERSION_SUPPORTED,
        0x01000300, 0, null, theFCM.getDefaultListener( ));
  }
  ```

Note: For detailed information about the method, including a description of its parameters, refer to "registerInterface" on page 183.

Registering your packages of custom functions with the Function Call Manager

Use the **FunctionCallManager** method **registerFunctionCall** in the function call constructor to register each of your custom functions and corresponding package(s) with the Function Call Manager.

- The FCI allows you to assign a version number to each function that you create. This allows you to provide upgrades to single functions in extensions you have already distributed to users. For more information see the next section.
- When registering your package(s) of functions with the Function Call Manager, be aware of the API package naming conventions. For more information see the next section.
- You must register each of your custom functions separately. So, if you are registering three functions with the Function Call Manager, you must call **registerFunctionCall** three times.
- In the following example, the **FciFunctionCall** constructor uses the **registerFunctionCall** method to register the **convertDate** function with the Function Call Manager:

```
public FciFunctionCall(IFX IFXMan) throws UWIException
{
FunctionCallManager theFCM;
   if ((theFCM = IFSSingleton.getFunctionCallManager()) == null)
      throw new UWIException("Needed Function Call Manager");
   IFXMan.registerInterface(this,
      FunctionCall.FUNCTIONCALL_INTERFACE_NAME,
      FunctionCall.FUNCTIONCALL_CURRENT_VERSION,
      FunctionCall.FUNCTIONCALL_MIN_VERSION_SUPPORTED,
      0x01000300, 0, null, theFCM.getDefaultListener( ));
   theFCM.registerFunctionCall(this, "sample_package",
      "convertDate", FciFunctionCall.CONVERTDATE,
      FunctionCall.FCI_FOLLOWS_STRICT_CALLING_PARAMETERS,
      "S,S", 0x01000300, "Converts a date to a different
      locale");
}
```

- Note that the ″S,S″ parameter in the **registerFunctionCall** method represents two mandatory strings that you must provide.

Note: For detailed information about the method, including a description of its parameters, refer to "registerFunctionCall" on page 188.

About Function Version Numbers

Along with registering your package(s) of custom functions with the Function Call Manager, the **registerFunctionCall** method is also used to specify a version number for each function that you create. In the previous example, the **ConvertDate** function is registered with the version number 0x01000300.

Assigning a version number to each function allows you to provide upgrades to single functions in extensions you have already distributed to users.

For example, if you distributed an extension containing a package of 50 functions for your application and then wanted to change the behavior of one of the functions, you could:

- Write a new extension containing just the upgraded function.

- Register the new function using **registerFunctionCall**, with the same package name and function name as the original function but with a higher version number.
- Distribute the new extension to users.

When the API initializes all of the extensions it would find two functions with the same package name and function name. It would deregister the one with the lower version number thereby updating your application.

Note: For more information about using version numbers, refer to "registerFunctionCall" on page 188.

Package Naming Conventions

The main purpose of package names is to distinguish the functions in a package from those in other packages that could potentially have the same names. All packages you create must contain an underscore in their names. For example, the **convertDate** function belongs to a package called **sample_package**.

- Choose a name that aptly describes the set of functions you are creating and is distinct enough to be unique within its realm of usage.
- The package name is an internal logical element of the API.
- Package names are case sensitive.
- All package names you define must contain an underscore.

Note: A group of functions is provided with the Forms System software as the package. The package is reserved for system functions that are defined in the XFDL Specification. You may not add to the system package or call your packages by the name system.

Implementing your custom functions

Implement your custom functions as part of the **FunctionCall** method **evaluate**.

- The **FunctionCall** class must implement the **evaluate** method since it is defined as part of the **FunctionCall** interface.
- **evaluate** is called whenever a particular function needs to be executed.
- In the following example, the **convertDate** function is implemented as part of **evaluate** in the **FunctionCall** class **FciFunctionCall**.

```
public class FciFunctionCall extends FunctionCallImplBase implements
   FunctionCall
   {
   /* Additional Code Removed */
      public void evaluate(String thePackageName,
         String theFunctionName, int theFunctionID,
         int theFunctionInstance, short theCommand,
         com.PureEdge.xfdl.FormNodeP theForm,
         com.PureEdge.xfdl.FormNodeP theComputeNode,
         com.PureEdge.IFSUserDataHolder theFunctionData,
         com.PureEdge.IFSUserDataHolder theFunctionInstanceData,
         com.PureEdge.xfdl.FormNodeP [] theArgList,
         com.PureEdge.xfdl.FormNodeP theResult) throws UWIException
      {
      String theDateString;
      String theLocaleString;
      String theAnswerString = null;
      Date theDate = null;
      Locale theLocale;
      DateFormat theDateFormat;

         if (theCommand == FunctionCall.FCICOMMAND_RUN)
```

```
{

/* Now we'll switch on the function ID. This makes it easy for a
   single FunctionCall object to support multiple functions. */

if (theFunctionID == FciFunctionCall.CONVERTDATE)
{

/* First, we'll grab the string values of the two arguments.
   Since    we indicated that this method has two parameters and
   that it       must have two parameters
   (FCI_FOLLOWS_STRICT_CALLING_PARAMETERS) when we registered
   it, we don't have to check to see if we actually received
   both parameters, since this code won't even be called unless
   the caller used the right number of parameters. */
   theDateString = theArgList[0].getLiteralEx(null);
   theLocaleString = theArgList[1].getLiteralEx(null);

   /* Now we perform the conversion. */
   if (theLocaleString.length( ) != 5)
      theAnswerString = "Locale must be 2 characters, " +
         "a space and    2 characters";
   else
   {
      theLocale = new Locale(theLocaleString.substring(0, 2),
         theLocaleString.substring(3));

      if ((theDateFormat = DateFormat.getDateInstance(
         DateFormat.LONG, theLocale)) == null)
         theAnswerString = "Unrecognized locale";
      else
      {
         try
         {
            if ((theDate = new SimpleDateFormat
               ("yyyyMMdd").parse(theDateString)) == null)
               theAnswerString = "Unable to parse";
         }
         catch (ParseException ex)
         {
            theAnswerString = ex.toString( );
         }

         if (theAnswerString == null)
            theAnswerString = theDateFormat.format(theDate);
      }
   }

   /* Lastly, we'll store the result in the result node */

   theResult.setLiteralEx(null, theAnswerString);
}
}
/* Additional Code Removed */
}
```

Note: For detailed information about the method, including a description of its parameters, refer to "evaluate" on page 175.

Providing help information for each of your functions

By using the method **help**, you can provide help information to form designers within a development environment (for example, Workplace Forms Designer). Use **help** to help form designers choose and use the correct functions.

Provide in-depth help information for each of the functions you create by implementing the **FunctionCall** method **help**.

- The **FunctionCall** class must implement the **help** method since it is defined as part of the **FunctionCall** interface.
- In the following example, **help** provides help information for the **convertDate** function in the class **FciFunctionCall**.

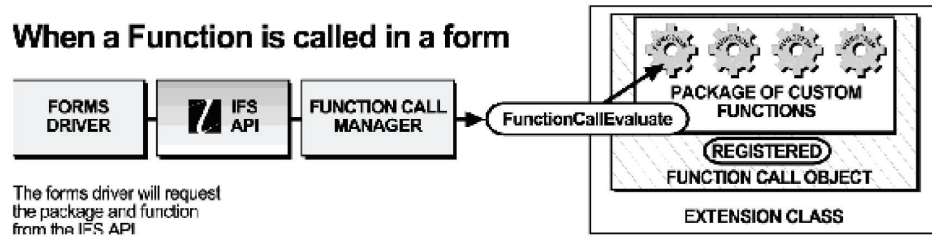
```
public class FciFunctionCall extends
com.PureEdge.xfdl.FunctionCallImplBase implements FunctionCall
{
   /* Additional Code Removed */

   public void help(String thePackageName,
      String theFunctionName, int theFunctionID,
      com.PureEdge.IFSUserDataHolder theFunctionData,
      com.PureEdge.StringHolder theQuickDesc,
      com.PureEdge.StringHolder theFunctionDesc,
      com.PureEdge.StringHolder theSampleCode,
      com.PureEdge.StringListHolder theArgsNameList,
      com.PureEdge.StringListHolder theArgsDescList,
      com.PureEdge.ShortListHolder theArgsFlagList,
      com.PureEdge.StringHolder theRetValDesc,
      com.PureEdge.ShortHolder theRetValFlag) throws
      UWIException
   {
      switch(theFunctionID)
      {
      case FciFunctionCall.CONVERTDATE:
         theQuickDesc.value = "Converts a date to a different " +
            "locale";
         theFunctionDesc.value = "This function takes a date in " +
            "the first parameter and a locale in the second " +
            "parameter and returns the date formatted for the " +
            "specified locale";
         theSampleCode.value = "\t <LABEL SID = \"LABEL1\"> \n" +
            "\t\t <VALUE COMPUTE = "sample_package.convert " +
            "(\'19980101\', \'french\')"></VALUE> \n" +
            "\t\t <SIZE CONTENT = \"ARRAY\"> \n" +
            "\t\t\t <AE>10</AE> \n" +
            "\t\t\t <AE>1</AE> \n" +
            "\t\t </SIZE> \n" +
            "\t </LABEL> \n";
         theArgsNameList.value = new String[2];
         theArgsNameList.value[0] = "theDate";
         theArgsNameList.value[1] = "theLocale";
         theArgsDescList.value = new String[2];
         theArgsDescList.value[0] = "The english date";
         theArgsDescList.value[1] = "The locale";
         theRetValDesc.value = "The formatted date";
         break;
      }
   }
}
```

Note: For detailed information about the method, including a description of its parameters, refer to “help” on page 178.

Building the extension

Once you have generated the Java source files for your **Extension** class, you must compile the source code to create the extension.

- Use a Java compiler that is supported by this API to compile your **Extension** class files. Refer to the *IBM Workplace Forms Server - API Installation and Setup Guide* for more information on compatible development environments.
- Before building your extension you should have a collection of .java files that represent your extension. After compiling the .java files you will have a set of files with the same name as the .java files but with the extension .class.



- For example, after compiling the source code for the **Extension** class FCIExtension.java and the **FunctionCall** class FciFunctionCall.java your Java compiler will create two corresponding files: FCIExtension.class and FciFunctionCall.class. These two class files make up the extension called **FCIExtension**.
- The details of compiling your source code are not included in this manual. Consult your Java documentation for specific information on how to use your Java compiler.

Testing and Distributing extensions

Once you have created your extensions you can package them for testing or distribution by using either of the following methods:

- Package the .class files into a single Java Archive (JAR) file and distribute the JAR file. Refer to “Distributing Extensions for Testing or Use” on page 165 for more details.
- Package the .class files into a single Java Archive (JAR) file and embed the JAR file directly into your XFDL forms. Refer to “Embedding extensions in XFDL Forms” on page 165 for more details.

Packaging extensions as JAR Files

Once you have created your extensions you can package the .class files into a single Java Archive (JAR) file and distribute the file. This means that you can package multiple extensions into one JAR file for distribution.

Before building the JAR file, you must create a manifest that indicates which classes in the JAR file are extensions.

1. Using your favorite text-editor, create a manifest file for the extensions you wish to package in the JAR file.
   - The manifest file has the file extension .mf. For example, the manifest file for **FCIExtension** is called:

     ```
     FCI.mf
     ```
   - The first line of the manifest must include the manifest-version number. See the following example for the correct syntax.
   - The manifest file is broken down into sections, where each section represents a particular **Extension** class and its attribute as an extension.

- The class listed in each section of the manifest file is the class that implements the Extension interface. In the following example FCIExtension.class implements **FCIExtension**.
- For example, the manifest file for **FCIExtension** will have the following syntax and format (notice that there is a space after every colon)

```
Manifest-Version: 1.0
Name: com/yourcompany/samples/FCIExtension.class
IFS-Extension: True
```

- Ensure that you add a carriage return to the last line of the manifest file. Otherwise the JAR may not work properly.

2. Create a JAR file from the .class files that make up your extension.
   - Use the following syntax to create the JAR file:

```
jar -cvfm destination.jar manifest.mf YourExtension.class
   YourFunctionCall.class
```

   - Optionally, you can replace the class names with the root folder for your package. This will include all classes that are defined in that package. For example, to create a JAR file called FormIFX for **FCIExtension**, you would type the following:

```
jar -cvfm FormIFX.jar FCI.mf com
```

Distributing Extensions for Testing or Use

Once you have packaged your extension, you can install it for testing or distribute it for general use. In either case, you place the extension in the same location.

To distribute your extensions so that they may be used by a specific Workplace Forms product:

Copy the JAR file to the Extensions folder of the Workplace Forms product that will use the extension.

- For example, in order for the **convertDate** function to work in the Viewer, you would copy the file FormIFX.jar to the following folder:

```
<Viewer program folder>\Extensions
```

To distribute your extensions so that they may be used by all Workplace Forms products:

1. Copy the JAR file to the Forms System Global Extensions folder.

- For example, in order for the **convertDate** function to work in the Viewer, you would copy the file FormIFX.jar to the following folder: C:\<*Windows System*>\PureEdge\extensions

Note: For more information about creating Java Archive files and manifest files refer to your Java documentation.

Embedding extensions in XFDL Forms

You can embed an extension in any XFDL form. The extension will run in the Viewer, but will not be processed by the API. This means that server-side applications will not run an embedded extension.

To embed an extension in a form:

1. Create a JAR file that contains your extensions.
2. Use Workplace Forms Designer to enclose the JAR file in a form.

Note: For more information about embedding extensions in XFDL forms using Designer refer to the *Designer User's Manual*.

About MIME Types

JAR files that are enclosed in forms must have the MIME type set to:

- application/uwi_jar

The corresponding XFDL code will look like this:

```
<mimetype>
   application/uwi-jar
</mimetype>
```

If you are using the Designer to enclose a JAR file into a form the Designer will set the MIME type for you.

You can also override the default Java Virtual Machine (JVM) that is used to run the enclosed JAR file. To do this, you must add a *vm* parameter to your MIME type, as shown:

```
<mimetype>
   application/uwi-jar; vm="<java virtual machine>"
</mimetype>
```

Set this to the string that the JVM uses to register itself. For example,"Sun VM <version>". The version is guaranteed to include a major and minor number, and may include further information, such as a maintenance number, build number, and so on.

Since it can be difficult to get an exact match, you can use the * wildcard in the version string. For example, you might use the following string:

```
Sun VM 1.4*
```

This will match any version beginning with 1.4, such as "1.4.2_03 JDK" or "1.4 JDK".

When there are multiple matches, the API will default to the latest version. For example, if you search for version 1.4*, and you have "Sun VM 1.4 JDK" and "Sun VM 1.4.2_03 JDK" installed, the API will use version 1.4.2_03.

Additionally, the API always chooses the JDK over the JRE. For example, if you search for version 1.4*, and you have "Sun VM 1.4.2_03 JDK" and "Sun VM 1.4.2_03 JVM" installed, the API will use the JDK.

Location of Installed extensions (Security Issues)

The location in which an extension is installed determines how much access the extension has to the user's system resources (for example, user's system files). The following table summarizes the security features that are set when an extension is installed in a particular location.

Location of Installed extension	Security Features
The extension is installed as a JAR or ZIP file in the Extensions folder of the Workplace Forms product that will use the extension.	The extension has full access to the user's system resources.

Location of Installed extension	Security Features
The extension is installed as a JAR or ZIP file in the Forms System Global Extensions folder: C:\<*Windows System*>\PureEdge\extensions	The extension has full access to the user's system resources.
The extension is installed as a set of .class files in the folder: <*Windows System*>\PureEdge\java\source.	The extension has full access to the user's system resources.
The extension is packaged as a JAR (or Zip) file and enclosed directly within a form.	The extension has access only to the form that encloses it. The extension cannot access other parts of the user's system or cause any damage. Refer to the following section regarding additional security restrictions for functions enclosed in XFDL forms.

Additional Security Restrictions for Functions Enclosed in XFDL Forms

Note that the following methods are not available to extensions that are enclosed within an XFDL form:

- com.PureEdge.xfdl.XFDL.readForm
- com.PureEdge.xfdl.FormNodeP.writeForm
- com.PureEdge.xfdl.FormNodeP.encloseFile
- com.PureEdge.xfdl.FormNodeP.extractFile
- com.PureEdge.ifx.IFX.ifxScanForExtensions

In addition extensions that are enclosed within a form **cannot** execute the following functions on a local computer:

- Create class loaders.
- Exit the Java virtual machine.
- Execute a program.
- Link to a DLL.
- Read files.
- Write files.
- Delete files.
- Print.

Summary

By working through this section you have successfully built an extension for date conversion. In the process you have learned how to set up, compile, test, and distribute extensions. You also learned how to use the following FCI methods:

- getFunctionCallManager
- registerInterface
- registerFunctionCall
- evaluate
- help

The Convert Date sample application is included with this API. you will find it in the folder <*API Program Folder*>\samples\java\fci\demo\convert_date.

In order to view the forms provided with this sample application, you must have a licensed or evaluation copy of Workplace Forms Viewer installed.

FCI Library Quick Reference Guide

The following sections provide a quick reference guide to the classes, constants, and methods used in the FCI Library:

- "Holder Objects" on page 4 describes **Holder** objects and how to use them.
- "The Extension Class" on page 171 describes the **extensionInit** method that is used to initialize an extension and provide services for function calls.
- "The FunctionCall Class" on page 173 describes the **FunctionCall** class and lists the methods associated with the class.
- "The IFX Class" on page 181 lists the methods that handle the management of extensions.
- "The FunctionCallManager Class" on page 185 lists the methods that handle the management of functions.

Note: While certain methods in the FCI library require an IFSUserDataHolder as a parameter, you will not need to manipulate this object.

About the Method Descriptions

The methods in this reference guide are listed according to the class that they belong to and are described using the following format:

- **Description:** A description of what the method does.
- **Method**: Lists the method's signature and type of value returned (if any).
- **Parameters**: An explanation of the parameters to use in the call.
- **Returns**: Indicates what value is returned by the method.
- **Notes**: Additional information to help you use the method.
- **Example**: A sample piece of code that uses the method in question.

© Copyright IBM Corp. 2003, 2006

The Extension Class

The Extension class contains the method **extensionInit** which is used to initialize an extension and provide services for function calls. When the Forms System is initialized, the API checks for existing extensions and calls **extensionInit** for each extension.

You must create an Extension class for every extension that you create. Furthermore, the Extension class must implement the method **extensionInit** since it is defined as part of the Extension interface.

The Extension class extends the pre-defined superclass com.PureEdge.ifx.ExtensionImplBase and implements the pre-defined interface Extension. It is a good idea for your Extension class to extend the super class com.PureEdge.ifx.ExtensionImplBase since the superclass takes care of many housekeeping methods that must be implemented.

The Extension class must implement the method **extensionInit** since it is defined as part of the Extension interface. Refer to the **extensionInit** method on page page extensionInit for more information.

Imports

You must import the following files as part of the **Extension** class source code:

- com.PureEdge.ifx.Extension
- com.PureEdge.ifx.IFX
- com.PureEdge.error.UWIException

Example

For an example of how to set up this class and the FunctionCall class, see "Getting Started with the FCI Library" on page 155.

extensionInit

Description

This method is responsible for the registration of all the services that the extension provides.

Method

```
public void extensionInit(
   com.PureEdge.ifx.IFX theIFX
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theIFX	**IFX**	The IFX Manager.

Returns

Nothing if call is successful or throws a generic exception (**UWIException**) if an error occurs.

Notes

Use the **extensionInit** method to create a new **FunctionCall** object that contains your custom-built functions.

Remember that in order to create a new **FunctionCall** object you must define a **FunctionCall** class that contains your custom functions. Refer to "The FunctionCall Class" on page 173 for more details.

Example

In the following example, **extensionInit** creates a **FunctionCall** object called **SimpleFunctionCall.**

```
import com.PureEdge.ifx.IFX;
import com.PureEdge.ifx.ExtensionImplBase;
import com.PureEdge.ifx.Extension;
import com.PureEdge.xfdl.FunctionCall;
import com.PureEdge.error.UWIException;
public class SimpleExtension extends ExtensionImplBase implements Extension
{
   public void extensionInit(IFX theIFX) throws UWIException
   {
      FunctionCall theFunctionObject = new SimpleFunctionCall(theIFX);
   }
}
```

The FunctionCall Class

The FunctionCall class contains definitions for your custom functions. It also registers each FunctionCall object, and custom function that the object supports with the Forms System.

The FunctionCall class extends the pre-defined superclass com.PureEdge.xfdl.FunctionCallImplBase and implements the pre-defined interface FunctionCall.

The FunctionCall class must implement the methods **evaluate** and **help** since they are defined as part of the FunctionCall interface.

- Remember that in order to make your functions available to the API you must register your FunctionCall object with the IFX Manager using the method called **registerInterface**.
- You must also retrieve the Function Call Manager from the IFX Manager using the method **getFunctionCallManager** and register each of your functions with the Function Call Manager using the method **registerFunctionCall**.
- For more information about the FunctionCallManager methods refer to "The FunctionCallManager Class" on page 185. For more information about the IFX methods mentioned above, refer to "The IFX Class" on page 181.

Imports

You must import the following files as part of the function call source code:

- com.PureEdge.ifx.Extension
- com.PureEdge.xfdl.FormNodeP
- com.PureEdge.xfdl.FunctionCallManager
- com.PureEdge.xfdl.FunctionCall
- com.PureEdge.error.UWIException
- com.PureEdge.IFSUserData
- com.PureEdge.StringHolder
- com.PureEdge.StringListHolder
- com.PureEdge.ShortHolder

Example

For an example of how to set up this class and the **Extension** class, refer to "Getting Started with the FCI Library" on page 155.

© Copyright IBM Corp. 2003, 2006

FunctionCall Class Constants

The following table lists the constants that are used within the **FunctionCall** class along with a short description of each constant:

Named Constants	Description
FunctionCall.FCI_FOLLOWS_ STRICT_CALLING_ PARAMETERS	Used in the method **registerFunctionCall** as a possible value for the parameter *theFlags***.** Indicates that the user of your custom function must provide the parameters you define in the **registerFunctionCall** parameter *theCallingParams***.**
FunctionCall.FCI_WANTS_ INSTANCE_DEREGISTER_ CALL	Used in the method **registerFunctionCall** as a possible value for the parameter *theFlags*. Indicates that the Forms System should call **evaluate** with *theCommand* set to **FCICOMMAND_ INSTANCEDEREGISTER** when an instance of the function is deregistered.
FunctionCall.FCI_WANTS_ INSTANCE_REGISTER_CALL	Used in the method **registerFunctionCall** as a possible value for the parameter *theFlags*. Indicates that the Forms System should call **evaluate** with *theCommand* set to **FCICOMMAND_INSTANCEREGISTER** when an instance of the function is registered.
FunctionCall.FCI_WANTS_ REGISTER_CALL	Used in the method **registerFunctionCall** as a possible value for the parameter *theFlags*. Indicates that the Forms System should call **evaluate** with *theCommand* set to **FCICOMMAND_REGISTER** when the function is registered.
FunctionCall.FCIARGFLAG_ OPTIONAL	Used as a possible value for the flag *theArgsFlagList* in the method **help**. This value represents an optional parameter.
FunctionCall.FCIARGFLAG_ OPTIONAL	Used as a possible value for the flag *theRetValFlag* in the method **help**. This value represents a return value that is optional.
FunctionCall.FCIARGFLAG_ REPEATING	Used as a possible value for the flag *theArgsFlagList* in the method **help**. This value represents a repeating parameter.
FunctionCall.FCIARGFLAG_ STRING	Used as a possible value for the flag *theArgsFlagList* in the method **help**. This value represents a parameter of type **String**.
FunctionCall.FCIARGFLAG_ STRING	Used as a possible value for the flag *theRetValFlag* in the method **help**. This value represents a return value of type **String**.

Named Constants	Description
FunctionCall.FCICOMMAND_ DEREGISTER	Used in the method **evaluate** as a possible value for the parameter *theCommand*. This constant indicates that **evaluate** should execute some procedure when the function has been deregistered.
FunctionCall.FCICOMMAND_ INSTANCEDEREGISTER	Used in the method **evaluate** as a possible value for the parameter *theCommand*. This constant indicates that **evaluate** should execute some procedure when an instance of the function has been deregistered.
FunctionCall.FCICOMMAND_ INSTANCEREGISTER	Used in the method **evaluate** as a possible value for the parameter *theCommand*. This constant indicates that **evaluate** should execute some procedure when an instance of the function is registered.
FunctionCall.FCICOMMAND_ REGISTER	Used in the method **evaluate** as a possible value for the parameter *theCommand*. This constant indicates that **evaluate** should execute some procedure when the function is registered.
FunctionCall.FCICOMMAND_ RUN	Used in the method **evaluate** as a possible value for the parameter *theCommand*. This constant indicates that **evaluate** should evaluate a given function.
FunctionCall.FUNCTIONCALL_ CURRENT_VERSION	Current version of the Function Call Interface. Used in the methods **registerInterface** as a value for the parameter *theInterfaceVersion*.
FunctionCall.FUNCTIONCALL_ INTERFACE_NAME	Name of the Function Call Interface. Used in the methods **registerInterface** as a value for the parameter *theInterfaceName*
FunctionCall.FUNCTIONCALL_ MIN_VERSION_SUPPORTED	The minimum version of the Function Call Interface that is supported. Used in the methods **registerInterface** as a value for the parameter *theMinInterfaceVersion*.

evaluate

Description

This method performs the necessary work for your custom-built function. You will have to insert the details of your custom functions within this method.

Method

```
public void evaluate(
   String thePackageName,
   String theFunctionName,
```

```
int theFunctionID,
int theFunctionInstance,
short theCommand,
com.PureEdge.xfdl.FormNodeP theForm,
com.PureEdge.xfdl.FormNodeP theComputeNode,
com.PureEdge.IFSUserDataHolder theFunctionData,
com.PureEdge.IFSUserDataHolder theFunctionInstanceData,
com.PureEdge.xfdl.FormNodeP[ ] theArgList,
com.PureEdge.xfdl.FormNodeP theResult
) throws UWIException;
```

Parameters

Expression	Type	Description
thePackageName	**String**	The name of the package that contains the function.
theFunctionName	**String**	The name of the function.
theFunctionID	**int**	A unique number that can be used to identify the function.
theFunctionInstance	**int**	A unique number that differentiates one instance of the function from another instance. See **Notes** for more information.
theCommand	**short**	The name of the command for this method to perform. See **Notes** for more information. Other commands can be found within the manual.
theForm	**FormNodeP**	The form that contains the function.
theComputeNode	**FormNodeP**	The node within the form that stores the function. See **Notes** for more information.
theFunctionData	**IFSUserDataHolder**	Reserved. Although this expression is not used, it must be present.
theFunctionInstanceData	**IFSUserDataHolder**	Reserved. Although this expression is not used, it must be present.
theArgList	**FormNodeP [ ]**	The list of arguments. See **Notes** for more information.
theResult	**FormNodeP**	The **FormNodeP** object in which you should store the result. Simply use **setLiteralEx** on this object to store the result.

Returns

Nothing if call is successful or throws a generic exception (**UWIException**) if an error occurs.

Notes

- *theCommand* — the value of *theCommand* represents the command that **evaluate** will perform.
  - The value of *theCommand* depends on the value of the parameter called *theFlags* in the method called **registerFunctionCall**.
  - Usually *theCommand* will be set to FCICOMMAND_RUN. This indicates that a function must be evaluated.
  - Other possible values for *theCommand* include:
  - **FCICOMMAND_INSTANCEDEREGISTER** — This constant indicates that **evaluate** should execute some procedure when an instance of the function has been deregistered.
  - **FCICOMMAND_DEREGISTER** — This constant indicates that **evaluate** should execute some procedure when the function has been deregistered.
  - **FCICOMMAND_REGISTER** — This constant indicates that **evaluate** should execute some procedure when the function is registered.
  - **FCICOMMAND_INSTANCEREGISTER** — This constant indicates that **evaluate** should execute some procedure when an instance of the function is registered.
- *theFunctionInstance* — is a unique number that differentiates one instance of the function with another instance. For example, if a form contains two calls to the function **testPackage.multiply** then two unique values for *theFunctionInstance* variable will exist.
- *theComputeNode* — is the node within the form that contains the function. For example, if you have an item such as:

```
<LABEL SID = "L1">
   <VALUE COMPUTE = "testPackage.multiply('7', '6')"></VALUE>
</LABEL>
```

Then *theComputeNode* will point to the node that represents the value option.

- *theFunctionInstanceData* - is data specific to an instance of a function. It will always be returned when the instance of the function is called. This object is only provided when the **FCI_WANTS_INSTANCE_DATA** flag is provided during the **registerFunctionCall** call.
- *theArgList* — Each argument's value is stored as a literal within a **FormNodeP** object. For example, to get the value of the first argument, type the following:

```
theArgList[0].getLiteralEx(null)
```

Note: To get the number of arguments in theArgList use: theArgList.length

Example

```
public class FciFunctionCall extends com.PureEdge.xfdl.FunctionCallImplBase
implements FunctionCall
{
public static final int FUNCTION_ID = 1;

   /* Additional Code Removed */
   public void evaluate(String thePackageName,
      String theFunctionName, int theFunctionID,
      int theFunctionInstance, short theCommand,
      com.PureEdge.xfdl.FormNodeP theForm,
      com.PureEdge.xfdl.FormNodeP theComputeNode,
      com.PureEdge.IFSUserDataHolder theFunctionData,
      com.PureEdge.IFSUserDataHolder theFunctionInstanceData,
      com.PureEdge.xfdl.FormNodeP [] theArgList,
      com.PureEdge.xfdl.FormNodeP theResult) throws UWIException
```

```
      {
   /* The first switch in this method is based on theCommand.  The only case
      that we are interested in handling is FCICOMMAND_RUN that indicates
      that we should evaluate a function. */
         switch (theCommand)
         {
         case FunctionCall.FCICOMMAND_RUN:
   /* The second switch is based on theFunctionID that you set for each
      of your custom functions. This makes it easy for a single FunctionCall
      object to support multiple functions. */
            switch(theFunctionID)
            {
            case FciFunctionCall.FUNCTION_ID:
            /* Insert the Details of your custom function here */
               break;
            }
            break;

            default:
            break;
         }
      }

      /* Additional code Removed */
   }
```

help

Description

Provides help information about each of your custom functions in the form development environment (for example, Workplace Forms Designer).

Method

```
public void help(
   String thePackageName,
   String theFunctionName,
   int theFunctionID,
   com.PureEdge.IFSUserDataHolder theFunctionData,
   com.PureEdge.StringHolder theQuickDesc,
   com.PureEdge.StringHolder theFunctionDesc,
   com.PureEdge.StringHolder theSampleCode,
   com.PureEdge.StringListHolder theArgsNameList,
   com.PureEdge.StringListHolder theArgsDescList,
   com.PureEdge.ShortListHolder theArgsFlagList,
   com.PureEdge.StringHolder theRetValDesc,
   com.PureEdge.ShortHolder theRetValFlag
   ) throws UWIException;
```

Parameters

Expression	Type	Description
thePackageName	**String**	The name of the package that contains the function.
theFunctionName	**String**	The name of the function.
theFunctionID	**int**	A unique number that can be used to identify the function.
theFunctionData	**IFSUserDataHolder**	Reserved. Although this expression is not used, it must be present.

Expression	Type	Description
theQuickDesc	**StringHolder**	The method sets a short one-line description of what the function does.
theFunctionDesc	**StringHolder**	The method will set a longer more detailed description of the function.
theSampleCode	**StringHolder**	The method will set an example of the XFDL code used to call your function, including an example of the function parameters.
theArgsNameList	**StringListHolder**	The method will set a list of arguments that your function takes. See **Notes** for more information.
theArgsDescList	**StringListHolder**	The method will set a description of each of the arguments in the theArgsNameList. See **Notes** for more information..
theArgsFlagList	**ShortListHolder**	The method will set a list of bit flags representing the type of each argument that the function takes. See **Notes** for more information.
theRetValDesc	**StringHolder**	The method will set a description of your custom function's return value.
theRetValFlag	**ShortHolder**	The method will set a bit flag representing the type of the return value. See **Notes** for more information. Simply use setLiteralEx on this object to store the result.

Returns

Nothing if call is successful or throws a generic exception (**UWIException**) if an error occurs.

Notes

- Refer to the table "FunctionCall Class Constants" on page 174 for possible values for:
  - theArgsFlagList
  - theRetValFlag
- For both *theArgsNameList* and *theArgsDescList*, if you have no arguments you must set the value to **null**. For example:

```
theArgsNameList.value = null;
```

Example

```
public void help(String thePackageName,
   String theFunctionName, int theFunctionID,
   com.PureEdge.IFSUserDataHolder theFunctionData,
   com.PureEdge.StringHolder theQuickDesc,
   com.PureEdge.StringHolder theFunctionDesc,
   com.PureEdge.StringHolder theSampleCode,
   com.PureEdge.StringListHolder theArgsNameList,
   com.PureEdge.StringListHolder theArgsDescList,
   com.PureEdge.ShortListHolder theArgsFlagList,
   com.PureEdge.StringHolder theRetValDesc,
   com.PureEdge.ShortHolder theRetValFlag) throws UWIException
{
/* Switch on theFunctionID. This makes it easy for a single FunctionCall
   object to support multiple functions. */
   switch(theFunctionID)
   {
/* Remember that you must define an ID number for each custom function
   that you create. In the example below the constant MULTIPLY represents
   the identification number for the multiply function. */
      case FciFunctionCall.MULTIPLY:
         theQuickDesc.value = "multiplies two numbers together";
         theFunctionDesc.value = "This function takes two numeric " +
            "parameters and multiplies the two numbers together and " +
            "returns the result.";
         theSampleCode.value = "\tlabel1 = new label\n" + "\t{\n" +
            "\t\tvalue = testPackage.multiply(\"10\", field2.value);\n" +
            "\t\tsize = [\"10\", \"1\"];\n" + "\t}\n"      +
            "\tfield2 = new field\n"               +
            "\t{\n" + "\t\tvalue = \"7\";\n" + "\t}\n";

/* Notice that in defining theArgsNameList below, you must create
  the list before providing a value for each element in the list. */
         theArgsNameList.value = new String[2];
         theArgsNameList.value[0] = "number1";
         theArgsNameList.value[1] = "number2";
/* Notice that in defining theArgsDescList below, you must create
   the list before providing a value for each element in the list. */
         theArgsDescList.value = new String[2];
         theArgsDescList.value[0] = "The first number";
         theArgsDescList.value[1] = "The second number";
         theRetValDesc.value = "The result";
         break;
   }
}
```

The IFX Class

The IFX class encapsulates methods that handle the management of extensions. To create a simple package of functions for calling from within XFDL forms, you need to use the **registerInteface** method in the IFX class.

For information on the FCI methods used to create a simple package, refer to “The FunctionCallManager Class” on page 185.

Imports

You must import the following files as part of the IFX class source code:
- com.PureEdge.ifx.IFX
- com.PureEdge.UWIException

Example

For an example of how to use this method, refer to “Getting Started with the FCI Library” on page 155.

deregisterInterface

Description

Deregisters a function call object that has been registered with the IFX Manager.

Method

```
public void deregisterInterface(
   com.PureEdge.GenericInterface theInterface
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theInterface	**GenericInterface**	The **FunctionCall** object that you are deregistering with the IFX Manager. See **Notes** for more information.

Returns

Nothing if call is successful or throws a generic exception (**UWIException**) if an error occurs.

Notes

Generally a **FunctionCall** object deregisters itself from the IFX Manager. So most times you can use the keyword **this** to represent the **FunctionCall** object to deregister.

© Copyright IBM Corp. 2003, 2006

Example

In the following example, **theIFX** represents the **IFX Manager**.

```
public class MyFunctionCall extends com.PureEdge.xfdl.FunctionCallImplBase
implements FunctionCall
{
   /* Additional Code Removed */
   public void shutdown(IFX theIFX) throws UWIException
   {
      theIFX.deregisterInterface(this);
   }
}
```

getInterfaceInstances

Description

Returns a list of **FunctionCall** objects that are currently registered with the IFX Manager.

Method

```
public com.PureEdge.GenericInterface[ ] getInterfaceInstances(
   String theInterfaceName,
   int theInterfaceVersion
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theInterfaceName	**String**	The name of the interface that you are looking for. In most cases, a Function Call Interface.
theInterfaceVersion	**int**	The interface version.

Returns

Returns a list of **GenericInterface** objects that must be typecast to **FunctionCall** objects or **null** if no matching interfaces are found. Throws a generic exception (**UWIException**) if an error occurs.

Example

```
public class myFunctionCall extends com.PureEdge.xfdl.FunctionCallImplBase
implements FunctionCall
{
GenericInterface theList[];
   /* Additional Code Removed */
   theList = theIFX.getInterfaceInstances
      (FunctionCall.FUNCTIONCALL_INTERFACE_NAME,
      FunctionCall.FUNCTIONCALL_CURRENT_VERSION);
   for(int i = 0 ; i < theList.length ; i++)
   {
      FunctionCall theFunctionCall;
      theFunctionCall = (FunctionCall)theList[i];
      /* Additional Code Removed */
   }
}
```

registerInterface

Description

Registers a **FunctionCall** object with the IFX Manager.

Method

```
public void registerInterface(
   com.PureEdge.GenericInterface theInterface,
   String theInterfaceName,
   int theInterfaceVersion,
   int theMinInterfaceVersion,
   int theImplementationVersion,
   int theFlags,
   String [] theCriteriaList,
   com.PureEdge.ifx.IFXCriteriaMatchingHandler theCriteriaHandler
   ) throws UWIException;
```

Parameters

Expression	Type	Description
theInterface	**GenericInterface**	The object that you are registering with the IFX Manager. Typical setting: **this** (if the object is registering itself)
theInterfaceName	**String**	The name of the Interface that you are registering. In this case a Function Call Interface. Typical setting: **FunctionCall.FUNCTIONCALL_INTERFACE_ NAME**
theInterfaceVersion	**int**	The function call interface version. Typical setting: **FunctionCall.FUNCTIONCALL_CURRENT_ VERSION**
theMinInterface Version	**int**	The minimum version that the interface will support. Typical setting: **FunctionCall.FUNCTIONCALL_ MIN_VERSION_SUPPORTED**
theImplementation Version	**int**	The version of the object you are registering. This is typically 0x01000300. If there are multiple objects with the same name available, the one with the highest version number is registered.
theFlags	**int**	Reserved. Setting: **0**
theCriteriaList	**String [ ]**	Reserved. Setting: **null**
theCriteriaHandler	**com.PureEdge.ifx.IFX Criteria-Matching Handler**	Reserved. Setting: **theFCM.getDefaultListener( )**

Returns

Nothing if call is successful or throws a generic exception (**UWIException**) if an error occurs.

Notes

- Typically, you will have to retrieve the Function Call Manager from the IFX Manager using **getFunctionCallManager** before you call **registerInterface**.
- Typically the **registerInterface** parameter called *theCriteriaHandler* is set to:

```
theFCM.getDefaultListener( )
```

Note that **theFCM** is a **FunctionCallManager** object which represents the Function Call Manager.

Example

In the following example, **theIFX** represents the IFX Manager

```
public FciFunctionCall(IFX IFXMan) throws UWIException
   {
   FunctionCallManager theFCM;
      if ((theFCM = IFSSingleton.getFunctionCallManager()) == null)
         throw new UWIException("Needed Function Call Manager");
      IFXMan.registerInterface(this,
         FunctionCall.FUNCTIONCALL_INTERFACE_NAME,
         FunctionCall.FUNCTIONCALL_CURRENT_VERSION,
         FunctionCall.FUNCTIONCALL_MIN_VERSION_SUPPORTED,
         0x01000300, 0, null, theFCM.getDefaultListener( ));
   }
```

The FunctionCallManager Class

The FunctionCallManager class encapsulates methods that handle the management of FunctionCall objects. To create a simple package of functions for calling from within XFDL forms, you need to use the following FunctionCallManager methods:

- registerFunctionCall
- evaluateFunctionCall
- getFunctionCallHelp

All of the methods described in this section act on a FunctionCallManager object. Before you can use any of these methods you must retrieve an IFSSingleton object using the method **getFunctionCallManager**. For more information on this method and other IFSSingleton methods refer to ″The IFSSingleton Class″ .

Imports

You must import the following files as part of the FunctionCallManager class source code:

- com.PureEdge.xfdl.FunctionCallManager
- com.PureEdge.xfdl.FunctionCall
- com.PureEdge.xfdl.FormNodeP
- com.PureEdge.IntHolder
- com.PureEdge.StringHolder
- com.PureEdge.StringListHolder
- com.PureEdge.ShortHolder
- com.PureEdge.UWIException

Example

For an example of how to use these methods to create a package of functions, refer to “Getting Started with the FCI Library” on page 155.

deregisterFunctionCall

Description

Deregisters a particular function from the Function Call Manager.

Method

```
public void deregisterFunctionCall(
   com.PureEdge.xfdl.FunctionCall theInterface,
   String thePackageName,
   String theFunctionName
   ) throws UWIException;
```

© Copyright IBM Corp. 2003, 2006

Parameters

Expression	Type	Description
theInterface	**FunctionCall**	The **FunctionCall** object that contains the function that you are deregistering with the IFX Manager. See **Notes** for more information.
thePackageName	**String**	The name of the package that contains the function. See **Notes** for more information.
theFunctionName	**String**	The name of the function to be deregistered. See **Notes** for more information.

Returns

Nothing if call is successful or throws a generic exception (**UWIException**) if an error occurs.

Notes

- *theInterface* — Generally a **FunctionCall** object will deregister its own functions, in this case use the keyword **this** to represent the **FunctionCall** object.
- *thePackageName* — Use the package name that you created when you registered the function using **registerFunctionCall**.
- *theFunctionName* — Use the function name that you created when you registered the function using **registerFunctionCall**.

Example

```
public FciFunctionCall(IFX theIFX) throws UWIException
{
   /* Additional code removed */
   theFCM.deregisterFunctionCall(this,"sample_package","multiply");
}
```

evaluateFunctionCall

Description

Use this method to call a package function. Since package functions assume that they are called by the API, this method establishes any additional parameters that the function may be expecting. Generally, function calls in a package should not be called directly since there are more parameters that must be passed.

Method

```
public void evaluateFunctionCall(
   String thePackageName,
   String theFunctionName,
   int theFunctionInstance,
   short theCommand,
   com.PureEdge.xfdl.FormNodeP theForm,
   com.PureEdge.xfdl.FormNodeP theComputeNode,
   com.PureEdge.xfdl.FormNodeP [] theArgList,
   com.PureEdge.xfdl.FormNodeP theResult
   ) throws UWIException;
```

Parameters

Expression	Type	Description
thePackageName	**String**	The name of the package that contains the function.
theFunctionName	**String**	The name of the function.
theFunctionInstance	**int**	A unique number that differentiates one instance of the function from another instance. See **Notes** for more information.
theCommand	**short**	The name of the command for this method to perform. Setting: Typically **FCICOMMAND_ RUN.** See **Notes** for more information. Other commands can be found within the manual.
theForm	**FormNodeP**	The form that contains the function.
theComputeNode	**FormNodeP**	The node within the form that contains the function. See **Notes** for more information.
theArgList	**FormNodeP [ ]**	The list of arguments. See **Notes** for more information.
theResult	**FormNodeP**	The **FormNodeP** object in which the result will be stored.

Returns

Nothing, or throws a generic exception (**UWIException**) if an error occurs.

Notes

- Use this method when you are calling another function from within your source code.
- *theFunctionInstance* — is a unique number that differentiates one instance of the function with another instance. For example, if a form contains two calls to the function **sample_package.multiply** then two unique values for *theFunctionInstance* variable will exist.
- *theComputeNode* — specifies which node in the form stores the function. For example, if you have an item such as:

```
<label sid = "LABEL1">
   <value>sample_package.multiply("7", "6")</value>
</label>
```

Then *theComputeNode* will point to the node that represents the *value* option.

Example

In the following example, the function **my_package.multiply** uses the **evaluateFunctionCall** method to call the **sample_package.multiply** function.

```
public void evaluate(String thePackageName,
   String theFunctionName, int theFunctionID, int theFunctionInstance,
   short    theCommand, com.PureEdge.xfdl.FormNodeP theForm,
   com.PureEdge.xfdl.FormNodeP theComputeNode,
   com.PureEdge.IFSUserDataHolder theFunctionData,
   com.PureEdge.IFSUserDataHolder theFunctionInstanceData,
   com.PureEdge.xfdl.FormNodeP [] theArgList,
```

```
   com.PureEdge.xfdl.FormNodeP theResult) throws UWIException
{
   switch (theCommand)
   {
   case FunctionCall.FCICOMMAND_RUN:
      switch(theFunctionID)
      {
      case myFC.MULTIPLY:
/* The evaluateFunctionCall method is used here to call the
   sample_package.multiply function.  The multiply function calculates
   the result and stores it in theResult. Note that the Function Call
   Manager must be retrieved. */
         theFCM.evaluateFunctionCall("sample_Package","multiply",
            theFunctionInstance, theCommand, theForm, theComputeNode,
            theArgList,theResult);

            break;
      }
      break;
      default:
      break;
   }
}
```

getDefaultListener

Description

Helps the IFX Manager determine which **FunctionCall** object implements a specific function.

Method

```
public com.PureEdge.ifx.IFXCriteriaMatchingHandler getDefaultListener( )
   throws UWIException
```

Parameters

There are no parameters for this method.

Returns

An object that can be used to locate the **FunctionCall** object that contains the specific function. Throws a generic exception (**UWIException**) if an error occurs.

Notes

Typically, this method is used when calling the **IFX** method **registerInterface**. Refer to the method description for **registerInterface** for more information.

registerFunctionCall

Description

Registers your custom function with the Function Call Manager.

Method

```
public void registerFunctionCall(
   com.PureEdge.xfdl.FunctionCall theFCIInterface,
   String thePackageName,
   String theFunctionName,
```

```
int theFunctionID,
int theFlags,
String theCallingParams,
int theVersion,
String theQuickDesc
) throws UWIException;
```

Parameters

Expression	Type	Description
theFCIInterface	**FunctionCall**	The **FunctionCall** object that will handle requests for the function. Setting: The **FunctionCall** object that is registering the function. See **Notes** for more information.
thePackageName	**String**	The name of the package that will contain the function.
theFunctionName	**String**	The name of the function.
theFunctionID	**int**	A unique number that can be used to identify the function. See **Notes** for more information.
theFlags	**int**	A set of flags which indicate how the custom function will be evaluated. Setting: Typically **FCI_FOLLOWS_STRICT_CALLING_PARAMETERS** or **0.** See **Notes** for more information.
theCallingParams	**String**	The list of parameters that this function takes. Setting: **S**, **O**, or **R**. See **Notes** for more information.
theVersion	**int**	The version number of the function. Setting: **Function Version Number**. See "Defining a Version Number" on page "Defining a Version Number" on page 190 for more information.
theQuickDesc	**String**	A short, one-line description of what the function does.

Returns

Nothing if call is successful or throws a generic exception (**UWIException**) if an error occurs.

Notes

- *theFCIInterface* — Typically a **FunctionCall** object will register its own function with the Function Call Manager. Use the keyword **this** to represent the current object.
- *theFunctionID* — Each function that you create as part of a particular package must have a unique identification number. Define each function's ID number as a constant at the beginning of the class. For example, the multiply function has an ID number of 1:

  ```
  public static final int MULTIPLY_ID = 1;
  ```

  – *theFlags* — Refer to the table of "FunctionCall Class Constants" on page 174 for a list of possible values for *theFlags*.
- *theCallingParams* — List the type of each parameter that the function will take and separate each value with a comma.
  – Use **S** to indicate a string parameter.
  – If the parameter is optional, then an **O** is added after the **S**.
  – If the parameter can repeat, then an **R** is added after the **S**.

– For example, if you were to register a function that had to have one parameter and optionally a second parameter then the *theCallingParams* would look like the following:

```
"S,SO"
```

– If there are no parameters, use an empty string ("").

Defining a Version Number

- If multiple **FunctionCall** objects register the same function for the same package, then the function with the highest version number is used.
- Version numbers are defined in hexadecimal format as follows, where the 0300 is a constant and must be present :
- 0x<major><minor><0300>
- For example, a function that is version 2.1 would be represented as
- 0x02010300
- Define a function's version number in the parameter *theVersion***.**

Note: For more information about using version numbers refer to page.

Example

```
public class FciFunctionCall extends com.PureEdge.xfdl.FunctionCallImplBase
implements FunctionCall
{
public static final int FUNCTION_ID = 1;

   /* Additional Code Removed */
    theFCM.registerFunctionCall(this,"sample_package",
      "multiply",FciFunctionCall.FUNCTION_ID,
      FCI.FCI_FOLLOWS_STRICT_CALLING_PARAMETERS, "S,S", 0x01000300,
      "Multiplies two numbers");
}
```

getFunctionCallHelp

Description

This method is used by the API to call the **FunctionCall** method **help**.

Method

```
public void getFunctionCallHelp(
   String thePackageName,
   String theFunctionName,
   int theFlagsPtr,
   com.PureEdge.IntHolder theVersion,
   com.PureEdge.StringHolder theQuickDesc,
   com.PureEdge.StringHolder theFunctionDesc,
   com.PureEdge.StringHolder theSampleCode,
   com.PureEdge.StringListHolder theArgsNameList,
   com.PureEdge.StringListHolder theArgsDescList,
   com.PureEdge.ShortListHolder theArgsFlagList,
   com.PureEdge.StringHolder theRetValDesc,
   com.PureEdge.ShortHolder theRetValFlag
   ) throws UWIException;
```

Parameters

Expression	Type	Description
thePackageName	**String**	The name of the package that contains the function.
theFunctionName	**String**	The name of the function.
theFlagsPtr	**int**	Returns the flags that were set when the function was registered with **registerFunctionCall**.
theVersion	**IntHolder**	The version number of the function.
theQuickDesc	**StringHolder**	A short, one-line description of what the function does.
theFunctionDesc	**StringHolder**	A longer more detailed description of the function.
theSampleCode	**StringHolder**	Provides an example of the XFDL code used to call your function, including an example of the function parameters.
theArgsNameList	**StringListHolder**	A list of arguments that your function takes.
theArgsDescList	**StringListHolder**	A description of each of the arguments in the **theArgsNameList**.
theArgsFlagList	**ShortListHolder**	A list of bit flags representing the type of each argument that the function takes. See **Notes** for more information.
theRetValDesc	**StringHolder**	A description of your custom function's return value.
theRetValFlag	**ShortHolder**	A bit flag representing the type of the return value. See **Notes** for more information. Simply use setLiteralEx on this object to store the result.

Returns

Nothing if call is successful or throws a generic exception (**UWIException**) if an error occurs.

Notes

Refer to the table of "FunctionCall Class Constants" on page 174 for possible values for:

- theArgsFlagList
- theRetValFlag

Example

In the example below the function **my_package.multiply** uses the **getFunctionCallHelp** method to call the function help that was defined for the **sample_package.multiply** function.

```
public void help(String thePackageName,
   String theFunctionName, int theFunctionID,
   com.PureEdge.IFSUserDataHolder theFunctionData,
   com.PureEdge.StringHolder theQuickDesc,
   com.PureEdge.StringHolder theFunctionDesc,
```

```
      com.PureEdge.StringHolder theSampleCode,
      com.PureEdge.StringListHolder theArgsNameList,
      com.PureEdge.StringListHolder theArgsDescList,
      com.PureEdge.ShortListHolder theArgsFlagList,
      com.PureEdge.StringHolder theRetValDesc,
      com.PureEdge.ShortHolder theRetValFlag) throws UWIException
   {
      /* Additional Code Removed */
      switch(theFunctionID)
      {
      case myFC.MULTIPLY:
         IntHolder theVersion = new IntHolder( );
         theFCM.getFunctionCallHelp("sample_package", "multiply",
            theVersion, theQuickDesc, theFunctionDesc, theSampleCode,
            theArgsNameList, theArgsDescList, theArgsFlagList,
            theRetValDesc, theRetValFlag);
         break;
      }
   }
```

getFunctionCallList

Description

Lists the functions that belong to a particular package.

Method

```
public String [ ] getFunctionCallList(
   String thePackageName
   ) throws UWIException;
```

Parameters

Expression	Type	Description
thePackageName	**String**	The package name.

Returns

Returns a list of functions in the package or throws a generic exception (**UWIException**) if an error occurs.

Example

```
   public class FciFunctionCall extends com.PureEdge.xfdl.FunctionCallImplBase
      implements FunctionCall
   {
      /* Additional code removed */
      public FciFunctionCall(IFX theIFX) throws UWIException
      {
         /* Additional code removed */
         String[]  functionList;
         functionList = theFCM.getFunctionCallList("sample_package");
      }
   }
```

getFunctionCallPackageList

Description

Lists the packages that are currently registered with the Function Call Manager.

Method

public String [] getFunctionCallPackageList( ) throws UWIException;

Parameters

There are no parameters for this method.

Returns

Returns a list of package names or throws a generic exception (**UWIException**) if an error occurs.

Example

```
public class FciFunctionCall extends com.PureEdge.xfdl.FunctionCallImplBase
  implements FunctionCall
{
   /* Additional code removed */
   public FciFunctionCall(IFX theIFX) throws UWIException
   {
      /* Additional code removed */
      String[] packageList;
      packageList = theFCM.getFunctionCallPackageList( );
   }
}
```

Appendix. JSP Support

Java Server Pages (JSP) is a platform independent technology designed to make it easier to add dynamic content to web pages. A web application server compiles JSP scripts into Java servlets that are executed by the server's Java virtual machine. The resulting dynamic content is inserted into the web document that is then displayed in the end user's web browser. Because JSP technology integrates with both HTML and XML documents, you can use JSP to extend the capabilities of XFDL forms.

This section assumes that you are familiar with both JSP and XFDL. The information in this section is not intended to show you how to write JSP scripts. Rather, its purpose is to explain how to integrate XFDL with JSP. For information on JSP refer to http://java.sun.com/products/jsp/. For more information on XFDL, refer to the *XFDL Specification*.

System Requirements

To process JSP pages that contain XFDL, your web server must be running the following software:

- A J2EE compliant web application server such as Tomcat 3.0 or JRun 3.0.
- Java Runtime Environment (JRE) 1.2 or greater.
- IBM Workplace Forms Server - API (if you want to call API methods from your JSP code. If you require support for streaming, you must install version 4.5.0 or greater of the PureEdge branded API, or version 2.5 or later of the IBM branded API.)

For information on how to install and configure these components, refer to the documentation that is distributed with these products.

The XFDL document that the web server produces in response to a JSP request is identical to any other XFDL form. As a result, end users must have either a PureEdge branded Viewer 3.0 or greater or version 2.5 or later of the IBM branded Viewer installed on their computer, in addition to a compatible web browser.

Combining JSP and XFDL

Creating a JSP page involves integrating JSP elements with the source code of the original HTML or XML web document. In the case of XFDL forms, this means adding appropriate JSP elements to the form's XFDL code using a text editor. Once complete, the resulting document is a JSP page and should have a .jsp extension.

Note: Once you add JSP code to an XFDL form, you will no longer be able to open the file with Workplace Forms Designer.

While the specific content of each JSP page depends on the logic of the application and the design of the form, certain fixed elements must be present in every JSP page that contains XFDL. The following elements must appear exactly as shown in every JSP that includes XFDL:

© Copyright IBM Corp. 2003, 2006

Element	Description
<?xml version="1.0"?>	This line is the standard XML file identifier. It must appear as the first line in the JSP page. There can be no blank lines or spaces ahead of this text.
<% response.setContentType("application/x-xfdl"); %>	This line sets the mime type of the http response object. In this case, it identifies the object as an XFDL document. As a result, the user's browser will display the document using the Viewer. With the exception of any optional comments or whitespace, this line should appear immediately after the XML file identifier.

Once you have included these standard elements, you can add the rest of your custom JSP code such as directives, declarations, or scriptlets.

Note: The XFDL portion of the JSP page must not be compressed but it is valid to use the XFDL <transmitformat> option to specify either ascii or binary compression for the transmission of the page.

Sample JSP Page

The following source code creates a simple JSP page containing an XFDL form. In this example, the JSP scriptlet obtains the current date and converts it into a string. The form contains one *label* item that displays the date value provided by the JSP scriptlet.

```
<?xml version="1.0"?>          – XML file identifier
   <!-- Set the content-type so that the webserver uses
      the Viewer to interpret the XFDL form. -->
   <% response.setContentType("application/x-xfdl"); %>   – Sets the mime type to XFDL
   <%@ page import="java.util.Date" %>
   <!-- This is the JSP scriptlet -->
   <%
          Date   theDate = new Date();
          theDateString = theDate.toString();
   %>
   <!-- The following XFDL code defines the form -->         – Start of XFDL code
               <XFDL xmlns="http://www.ibm.com/xmlns/prod/XFDL/7.0"
                  xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0">
                  <globalpage sid="global">
                     <global sid="global">
                        <formid>
                           <serialnumber>79F255F9-86E0-4163-B243-855EE603DF17
                           </serialnumber>
                           <version>1.3.1</version>
                        </formid>
                        <vfd_date>12/7/2001</vfd_date>
                     </global>
                  </globalpage>
                  <page sid="PAGE1">
                     <global sid="global"></global>
                     <pageid>
                        <serialnumber>05083920-872C-4C0A-8645-5096D4135D78
                        </serialnumber>
                     </pageid>
                     <vfd_pagesize>letter</vfd_pagesize>
                     <vfd_pagedpi>120</vfd_pagedpi>
                     <vfd_printsize>5;5</vfd_printsize>
                     <label>PAGE1</label>
                     <vfd_customsize>5;5;Inches</vfd_customsize>
                     <label sid="DATE_LABEL1">
                       <value> <%= theDateString%> </value>   – JSP expression
                     </label>
                  </page>
               </XFDL>
```

Sample JSP Application

The API includes a sample web application demonstrating how to use JSP pages to extend the functionality of XFDL forms. This simple application consists of three files, located in the folder <*API Program Folder*>\samples\java\jsp\demo. The following table describes the functions of each file:

File	Description
jspget.jsp	This JSP page is an example of a standard HTTP GET operation. The first JSP scriptlet obtains the current date and calculates the number of days until Christmas. The XFDL form specifies two labels to display this information. The second scriptlet creates the URL that the form's submit button uses to call jsppostt2.jsp. Within the XFDL portion of the page, a JSP *include* directive obtains a *label* item from getlabel.txt.
getlabel.txt	Although this file does not contain a complete form, it does contain XFDL code defining a *label* item. jspget.jsp accesses this code using an *include* directive.

File	Description
jsppost.jsp	This page is an example of a standard HTTP POST operation. The scriptlet uses the API's streaming method *readForm* to obtain the number of days until Christmas and the signature from jspget.jsp. It then uses *getSignatureVerificationStatus* to validate the signature. Finally, the XFDL portion of the page specifies two labels to display the results to the end user.

To run this sample in WebSphere Application Server (WAS), you must first package the above files into a WAR file. You can then deploy the WAR file using the WAS Administrator Console. For other web servers, you can package them as WARs or simply place the JSP files into your web server JSP folder. For example, if you are using Tomcat 3.0 you would copy the files to <*Tomcat*>\webapps\examples\jsp. If you are using JRun 3.0 you would copy the files to <*JRun*>\servers\default\demo-app\jsp. Refer to your web servers documentation for more information.

To run the application, users either follow a link or enter the appropriate URL in their browsers address box. For example, if the application is running on WAS, the URL would be: http://<*server_name*>:<*port*><*example_dir*>/jsp/jspget.jsp. On a Tomcat web server, the URL would be: http://<*server_name*>:<*port*>/examples/jsp/jspget.jsp. If the application is running on a JRun server, the URL would be: http://<*server_name*>:<*port*>/demo/jsp/jspget.jsp.

Appendix. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright IBM Corp. 2003, 2006

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Office 4360
One Rogers Street
Cambridge, MA 02142
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX
IBM
Workplace
Workplace Forms

Intel, Intel Inside (logos), MMX, and Pentium are trademarks of Intel Corporation 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 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.

Index

Special characters

-> symbol 9
== operator 4

A

about
  the API 3
addNamespace method 41
algorithm, looking up a hash algorithm 133
API
  about the API 3
  differences between Java, C, and COM 4
  list of Form Library functions 23
  where the API fits into your system 3
applications
  compiling your application, tutorial 21
  testing your application, tutorial 21
architecture for FCI extensions 151
argument nodes 7, 11
attachments
  attaching files to a form 51
  extracting attachments from a form 54
  removing an attachment 90
attributes
  getting a list of attributes and values 60
  getting the value of an attribute 58
  removing an attribute 88
  setting an attribute 93
Authenticated Clickwrap
  validating Authenticated Clickwrap signatures 103, 105

B

BooleanHolder 5

C

C API, differences from Java and COM 4
calling a function in a package 186
cell item
  locating a cell in a particular group 46
cells, creating 43
Certificate class
  about 27
certificates
  getting a Blob of the certificate 27
  getting a list of available certificates 61, 143
  getting specific certificate data 28
  getting the issuer certificate 31
  getting the signing certificate from a signature 139
checkValidFormats method 43
child nodes, locating 63
class files, creating 163
classes
  Certificate class 27
  DTK class 35
  Extension class 156, 171
  FormNodeP class 41
classes *(continued)*
  FunctionCall class 157, 173
  FunctionCallManager class 185
  Hash class 115
  IFSSingleton class 117
  IFX class 181
  LocalizationManager class 121
  Security Manager class 133
  Signature class 135
  XFDL 141
closing a form 49
  tutorial 21
COM API, differences from C and Java 4
compiling
  compiling your application, tutorial 21
compute node property 7
computes
  deactivating the compute system 92
  setting a compute 95
constants
  FCI_FOLLOWS_STRICT_CALLING_PARAMETERS 174
  FCI_WANTS_INSTANCE_DATA 177
  FCI_WANTS_INSTANCE_DEREGISTER_CALL 174
  FCI_WANTS_INSTANCE_REGISTER_CALL 174
  FCI_WANTS_REGISTER_CALL 174
  FCIARGFLAG_OPTIONAL 174
  FCIARGFLAG_REPEATING 174
  FCIARGFLAG_STRING 174
  FCICOMMAND_DEREGISTER 175, 177
  FCICOMMAND_INSTANCEDEREGISTER 175, 177
  FCICOMMAND_INSTANCEREGISTER 175, 177
  FCICOMMAND_REGISTER 175, 177
  FCICOMMAND_RUN 175
  FormNodeP constants 41
  FunctionCall class 174
  FUNCTIONCALL_CURRENT_VERSION 175
  FUNCTIONCALL_INTERFACE_NAME 175
  FUNCTIONCALL_MIN_VERSION_SUPPORTED 175
conventions
  for method descriptions 25, 169
  package naming conventions 161
conventions, document 2
copying a node 49
create method 141
createCell method 43
creating
  a form or node 141
  creating cells in a form 43
  creating forms 48
  creating nodes 48
current value
  about signed computes 26
custom function, convertDate 155
custom functions. See functions 149

D

data
  retrieving a value from a form, tutorial 19
  setting a value in a form, tutorial 20

© Copyright IBM Corp. 2003, 2006

data item
locating a particular data item in a datagroup 46
data model, updating the XML data model 112
data structures 4
datagroup
locating a particular data item in a datagroup 46
defining, version numbers 190
deleteSignature method 44
deleting
a form 21
deleting a form from memory 49
removing an enclosure from a form 90
dereferenceEx method 46
dereferencing 9
special notes on 97
deregisterFunctionCall method 185
deregistering a function call 185
deregisterInterface method 181
Designer 3
designing XFDL forms 3
destroy method 49
tutorial 21
digital certificates, getting a list of available cetificates 61, 143
digital signatures
determining if signatures are available 145
distributing
extensions 164
JAR files 164
distributing applications tutorial 22
document conventions 2
DTK class 35
duplicate method 49
duplicating a node 49

E

encloseFile method 51
encloseInstance method 52
enclosures
enclosing files in a form 51
extracting enclosures from a form 54
removing an enclosure 90
equals method 4
error message, getting the language for 121, 123
error message, setting the language for 126, 129
evaluate method 161, 175
evaluateFunctionCall method 186
evaluating a function 175
Extensible Forms Description Language. See XFDL 1
Extension class 171
creating extensions 156
extensionInit method 156, 169, 171
extensions
about extensions 149
about the IFX Manager 151
building an extension 163
distributing extensions 164
embedding extensions in forms 165
how FCI extensions work 151
implementing an extension 156
installing extensions 166
intializing extensions 171
testing 164
tutorial 155
extractFile method 54
extracting enclosures 54
extractInstance method 55
extractXFormsInstance method 57

F

FCI Library
about 149, 150
about functions, packages and extensions 149
about the extension architecture 151
how FCI extensions work 151
how the FCI Library works with the Form Library 150
quick reference 169
tutorial 155
FCI_FOLLOWS_STRICT_CALLING_PARAMETERS constant 174
FCI_WANTS_INSTANCE_DATA constant 177
FCI_WANTS_INSTANCE_DEREGISTER_CALL constant 174
FCI_WANTS_INSTANCE_REGISTER_CALL constant 174
FCI_WANTS_REGISTER_CALL constant 174
FCIARGFLAG_OPTIONAL constant 174
FCIARGFLAG_REPEATING constant 174
FCIARGFLAG_STRING constant 174
FCICOMMAND_DEREGISTER constant 175, 177
FCICOMMAND_INSTANCEDEREGISTER constant 175, 177
FCICOMMAND_INSTANCEREGISTER constant 175, 177
FCICOMMAND_REGISTER constant 175, 177
FCICOMMAND_RUN constant 175
form
determining the version of a form 64
Form Library
getting started 15, 17
how the Form Library works with the FCI Library 150
initializing the Form Library 19
list of Form Library functions 23
form nodes 7, 11
formatting
determining whether a node is correctly formatted 86
determining whether all the nodes in a form ares correctly formatted 43
formNodeP
creating a formNodeP 48
formNodeP structure 7
FormNodeP class
about 41
formNodeP constants
UFL_AFTER_SIBLING 50
UFL_APPEND_CHILD 50, 142
UFL_BEFORE_SIBLING 50, 142
UFL_ORPHAN 50
FormNodeP constants 41
UFL_NEXT 46
UFL_SAVE_ALLOW 112
FormNodeP objects
about 4, 41
comparing FormNodeP objects 4
creating a FormNodeP object 141
freeing FormNodeP objects from memory 4
forms
embedding extensions in forms 165
writing a form to disk 111
formula node property 7
formulas
about signed formulas 26
setting a formula 95
freeing memory 49
tutorial 21
Function Call
creating a function call 157

Function Call *(continued)*
  current version 175
  minimum version 175
  name of 175
Function Call Interface. See FCI Library 149
Function Call Manager
  accessing the Function Call Manager 117
  deregistering a function call 185
  listing the registered packages 192
  registering a function 188
  registering your package 160
  retrieving the manager 159
FunctionCall class 173
  constants 174
  creating a FunctionCall class 157
FunctionCall objects
  getting a list of 182
  registering with the IFX Manager 159
FUNCTIONCALL_CURRENT_VERSION constant 175
FUNCTIONCALL_INTERFACE_NAME constant 175
FUNCTIONCALL_MIN_VERSION_SUPPORTED constant 175
FunctionCallManager class 185
FunctionCallManager object 159
functions
  about function version numbers 160
  about functions 149
  calling a function in a package 186
  deregistering a function 181
  deregistering a function call 185
  evaluating a function 175
  implementing your functions 161
  list of Form Library functions 23
  providing help with your functions 162
  registering a function 183, 188
  registering your package with the Function Call Manager 160

G

getAttribute method 58
getAttributeList method 60
getBlob method 27
getCertificateList method 61
getChildren method 63
getCurrentThreadLocale method 121
getDataByPath method 28, 135
getDefaultListener method 188
getDefaultLocale method 123
getEngineCertificateList method 143
getFormVersion method 64
getFunctionCallHelp method 190
getFunctionCallList method 192
getFunctionCallManager method 117
getFunctionCallPackageList method 192
getInfoEx method 65
getInterfaceInstances method 182
getIssuer method 31
getLiteralByRefEx method 66
getLiteralEx method 69
getLocalizationManager method 117
getLocalName method 69
getNamespaceURI method 71
getNamespaceURIFromPrefix method 72
getNext method 73
getNodeType method 74
getParent method 75
getPrefix method 76
getPrefixFromNamespaceURI method 77
getPrevious method 78
getRererenceEx method 79
getSecurityEngineName method 82
getSecurityManager method 118
getSigLockCount method 83
getSignature method 84
getSignatureVerificationStatus method 85
getSigningCert method 139
getting started
  with the Form Library 17
getting started with the Form Library 15
getXFDL method 119
global
  item 11
  page node 11
group
  locating a cell in a particular group 46

H

hash algorithm, looking up an algorithm 133
Hash class 115
hash method 115
hashes
  creating a hash 115
help
  providing help for your function calls 178
  providing help with your functions 162
help method 162, 178
hierarchy
  about the node hierarchy 7
HMAC signatures
  validating HMAC signatures 103, 105
  validating Signature Pad signatures 103, 105
holder objects 4
  BooleanHolder 5
  IFSUserDataHolder 5
  IntHolder 5
  ShortHolder 5
  ShortListHolder 5
  StringHolder 5
  StringListHolder 5

I

identifier node property 7
IFSSingleton class 117
IFSUserDataHolder 5
IFX class 181
IFX extensions
  about 149
  about the IFX Manager 151
  architecture of 151
  building an extension 163
  defining your own 150
  distributing extensions 164
  embedding extensions in forms 165
  initialization (ExtensionInit) 169
  initializing 171
  installing extensions 166
  location in file system 151
IFX Manager
  about 151
  deregistering a function 181

IFX Manager *(continued)*
getting a list of registered function calls 182
registering an object with 183
registering the FunctionCall object 159
initialize method 35
initializeWithLocale method 37
initializing
IFX extensions 171
initializing IFX extensions 169
initializing the API 35
initializing the API with locale 37
initializing the Form Library 19
instances, XForms
adding to an instance 101
extracting an instance 57
replacing an instance 91, 101
updating an instance 101
instances, XML
enclosing an instance 52
extracting an instance 55
Interlink signatures
validating 103, 105
IntHolder 5
introduction to this manual 1
isDigitalSignaturesAvailable method 145
isSigned method 85
isValidFormat method 86
isXFDL method 87
item node 7, 11
item, global 11

J

JAR files
about MIME types 166
distributing extensions as JAR files 164
using JAR files with Workplace Forms products 165
Java API, differences from C and COM 4
Java Archive Files. See JAR files 164

L

language, getting the current language 121
language, getting the default language 123
language, setting the current language 126
language, setting the default language 129
literal property
about 7
getting the value of 66, 69
setting the value 96
setting the value of the literal property 98
loading a form
tutorial 19
loading forms
loading forms into memory 146
local names
getting the local name of a node 69
locale
initializing the API with locale 37
locale, getting the current locale 121
locale, getting the default locale 123
locale, setting the current locale 126
locale, setting the default locale 129
Localization Manager, accessing 117
LocalizationManager class
about 121
LocalizationManager objects 121
locating a node 46
lock count, getting for a node 83
lookupHashAlgorithm method 133
lookupInterface method 159

M

manifest files, creating 164
memory, freeing 21, 49
memory, freeing FormNodeP objects 4
method descriptions, about 25, 169
methods
addNamespace 41
checkValidFormats 43
create 141
createCell 43
deleteSignature 44
dereferenceEx 46
deregisterFunctionCall 185
deregisterInterface 181
destroy 49
duplicate 49
encloseFile 51
encloseInstance 52
evaluate 161, 175
evaluateFunctionCall 186
extensionInit 156, 171
extractFile 54
extractInstance 55
extractXFormsInstance 57
getAttribute 58
getAttributeList 60
getBlob 27
getCertificateList 61
getChildren 63
getCurrentThreadLocale 121
getDataByPath 28, 135
getDefaultListener 188
getDefaultLocale 123
getEngineCertificateList 143
getFormVersion 64
getFunctionCallHelp 190
getFunctionCallList 192
getFunctionCallManager 117
getFunctionCallPackageList 192
getInfoEx 65
getInterfaceInstances 182
getIssuer 31
getLiteralByRefEx 66
getLiteralEx 69
getLocalizationManager 117
getLocalName 69
getNamespaceURI 71
getNamespaceURIFromPrefix 72
getNext 73
getNodeType 74
getParent 75
getPrefix 76
getPrefixFromNamespaceURI 77
getPrevious 78
getReferenceEx 79
getSecurityEngineName 82
getSecurityManager 118
getSigLockCount 83
getSignature 84
getSignatureVerificationStatus 85

methods *(continued)*
getSigningCert 139
getXFDL 119
hash 115
help 162, 178
initialize 35
initializewithLocale 37
isDigitalSignaturesAvailable 145
isSigned 85
isValidFormat 86
isXFDL 87
lookupHashAlgorithm 133
lookupInterface 159
readForm 146
registerFunctionCall 160, 188
registerInterface 159, 183
remove Enclosure 90
removeAttribute 88
replaceXFormsInstance 91
setActiveForComputationalSystem 92
setAttribute 93
setCurrentThreadLocale 126
setDefaultLocale 129
setFormula 95
setLiteralByRefEx 96
setLiteralEx 98
signForm 99
updateXFormsInstance 101
validateHMACWithHashedSecret 105
validateHMACWithSecret 103
verifyAllSignatures 108
verifySignature 109
writeForm 111
xmlModelUpdate 112
MIME types, about 166

N

names, getting the security engine name 82
namespace
adding a namespace to a form 41
determining if a node is in the XFDL namespace 87
getting the local name of a node 69
getting the namespace prefix for a namespace URI 77
getting the namespace prefix for a node 76
getting the namespace URI for a node 71
getting the namespace URI from a prefix 72
null namespace 10
using namespace in references 10
node properties
compute property 7
formula property 7
identifier property 7
literal property 7
table of properties 12
type 7
node structure
advanced information 10
tree structure 11
nodes
about the node hierarchy 7
adding as child 50, 142
adding as new form 50
adding as sibling 50, 142
argument 11
argument nodes 7
comparing nodes 4
nodes *(continued)*
compute property 7
creating form nodes 141
creating nodes 48
determining how many times a node has been signed 83
duplicating a node 49
form nodes 7, 11
forumula property 7
getting a node's properties 65
getting the literal value 66
getting the literal value of a node 69
global page nodes 11
identifier property 7
item 11
item nodes 7
literal property 7
locating a child node 63
locating a node 46
locating the parent node 75
node properties 12
node tree structure 11
option 11
option nodes 7
page 7
page nodes 11
reference, getting for a particular node 79
root nodes 11
See also attributes 58
See also local names 69
See also namespace 69
setting the literal value 96
setting the literal value of a node 98
setting the value of signed nodes 99
table of node properties 12
traversing nodes 73, 78
type property 7
type, determining the node type 74

O

objects
accessing LocalizationManager objects 117
accessing XFDL objects 117, 119
Certificate objects 27
determining which object implements a function 188
Extension objects 171
FormNodeP objects 4, 41
FunctionCall objects 159, 173
FunctionCall objects, creating 157
FunctionCallManager 159
getting a list of FunctionCall objects 182
getting a signature object 84
Hash objects 115, 133
holder objects 4
LocalizationManager objects 121
registering an object with the IFX Manager 183
Security Manager object 118
See also holder objects 5
Signature objects 135
XFDL objects 141
operator, == 4
option nodes 7, 11
output parameters 4

P

packages
about 149
calling a function in a package 186
defining your own packages 150
listing the registered packages 192
package naming conventions 161
registering your package with the Function Call Manager 160
the sample_package 155
page node 7, 11
global page node 11
parameters, output 4
parent nodes, traversing parent nodes 75
prefix, namespace See namespace 72
properties
getting a node's properties 65
table of node properties 12

Q

quick reference
FCI library 169

R

readForm method 146
setting the current value of items 26
reading
reading forms into memory 146
references
getting a reference to a particular node 79
syntax of a reference 8
using namespace in references 10
using the null namespace in references 10
registerFunctionCall method 160, 188
registering
registering a function with the Function Call Manager 188
registering an object with the IFX Manager 183
registering extensions 171
registering services 169
registerInterface method 159, 183
removeAttribute method 88
removeEnclosure method 90
removing
removing a form from memory 49
removing enclosures 90
replaceXFormsInstance method 91
root nodes 11

S

saving a form to disk 111
tutorial 20
saving enclosures to disk 54
secret, hashing a secret 115
security
when installing extensions 166
security engines, getting the name 82
Security Manager, getting the Security Manager 118
SecurityManager class
about 133
services, registering 171
setActiveForComputationalSystem method 92
setAttribute method 93
setCurrentThreadLocale method 126
setDefaultLocale method 129
setFormula method 95
setLiteralByRefEx method 96
setLiteralEx method 98
shared secret, hashing a shared secret 115
ShortHolder 5
ShortListHolder 5
Signature class
about 135
Signature Pad signatures, validating
signatures
validtating Signature Pad signatures 103, 105
signatures
creating signatures 99
deleting signatures 44
destroying signatures 49
determining how many times a node has been signed 83
determining if a signature is valid 85
determining whether a node is signed 85
getting a signature object 84
getting specific signature data 135
getting the signing certificate from a signature 139
setting the value of nodes that are already signed 99
validating HMAC signatures 103, 105
validating Interlink signatures 103, 105
validating Topaz signatures 103, 105
validating WinTab signatures 103, 105
verifying 109
verifying signatures 108
signForm method 99
signing
signing a formula 26
singletons
Security Manager object 118
StringHolder 5
StringListHolder 5
strings
hashing a string 115
structures
formNodeP structure 7
system, where the API fits 3

T

testing extensions 164
Topaz signatures, validating 103, 105
traversing nodes 63, 73, 78
traversing child nodes to particular count 63
traversing parent nodes 75
tree structure
sample 11
XFDL 10
tutorials
closing a form 21
compiling your application 21
distributing applications 22
freeing memory 21
getting started
with the Form Library 15
loading a form 19
retrieving a value from a form 19
setting a value in a form 20
testing your application 21
writing a form to disk 20
type
determining the node type 74

type *(continued)*
  node property 7

U

UFL_AFTER_SIBLING constant 50
UFL_APPEND_CHILD constant 50, 142
UFL_BEFORE_SIBLING constant 50, 142
UFL_NEXT constant 46
UFL_ORPHAN constant 50
UFL_SAVE_ALLOW constant 112
updateXFormsInstance method 101
updating the XForms data model 57, 91, 101

V

validateHMACWithHashedSecret method 105
validateHMACWithSecret method 103
validating signatures 85
values, setting a value in a form, tutorial 20
verifyAllSignatures method 108
verifying signatures 108, 109
verifySignature method 109
version
  determining the version of a form 64
version numbers
  about function version numbers 160, 190
  defining version numbers 190
  example of function version numbers 190
Viewer 3
viewing XFDL forms 3

W

WinTab signatures, validating 103, 105
Workplace Forms Designer 3
Workplace Forms Viewer 3
writeForm method 111
  tutorial 20
writing a form to disk 111
  tutorial 20

X

XFDL
  about 1
  relation to FCI 1
XFDL class 141
XFDL object, accessing 119
XFDL objects 141
  accessing XFDL objects 117
XFDL tree structure 10
XForms data model, updating 57, 91, 101
XForms instances
  adding to an instance 101
  extracting an instance 57
  replacing an instance 91, 101
  updating an instance 101
XForms Model update 57, 91, 101
XML data model, updating 112
XML instances
  enclosing an instance 52
  extracting an instance 55
xmlModelUpdate method 112

Program Number: 5724-N08

Printed in USA

S229-1527-00