



IDL Connectivity Bridges

IDL Version 7.0
November 2007 Edition
Copyright © ITT Visual Information Solutions
All Rights Reserved

Restricted Rights Notice

The IDL®, IDL Analyst™, ENVI®, and ENVI Zoom™ software programs and the accompanying procedures, functions, and documentation described herein are sold under license agreement. Their use, duplication, and disclosure are subject to the restrictions stated in the license agreement. ITT Visual Information Solutions reserves the right to make changes to this document at any time and without notice.

Limitation of Warranty

ITT Visual Information Solutions makes no warranties, either express or implied, as to any matter not expressly set forth in the license agreement, including without limitation the condition of the software, merchantability, or fitness for any particular purpose.

ITT Visual Information Solutions shall not be liable for any direct, consequential, or other damages suffered by the Licensee or any others resulting from use of the software packages or their documentation.

Permission to Reproduce this Manual

If you are a licensed user of these products, ITT Visual Information Solutions grants you a limited, nontransferable license to reproduce this particular document provided such copies are for your use only and are not sold or distributed to third parties. All such copies must contain the title page and this notice page in their entirety.

Export Control Information

This software and its associated documentation are subject to the controls of the Export Administration Regulations (EAR). It has been determined that this software is classified as EAR99 under U.S. Export Control laws and regulations, and may not be re-transferred to any destination expressly prohibited by U.S. laws and regulations. The recipient is responsible for ensuring compliance to all applicable U.S. Export Control laws and regulations.

Acknowledgments

ENVI® and IDL® are registered trademarks of ITT Corporation, registered in the United States Patent and Trademark Office. ION™, ION Script™, ION Java™, and ENVI Zoom™ are trademarks of ITT Visual Information Solutions.

Numerical Recipes™ is a trademark of Numerical Recipes Software. Numerical Recipes routines are used by permission.

GRG2™ is a trademark of Windward Technologies, Inc. The GRG2 software for nonlinear optimization is used by permission.

NCSA Hierarchical Data Format (HDF) Software Library and Utilities. Copyright © 1988-2001, The Board of Trustees of the University of Illinois. All rights reserved.

NCSA HDF5 (Hierarchical Data Format 5) Software Library and Utilities. Copyright © 1998-2002, by the Board of Trustees of the University of Illinois. All rights reserved.

CDF Library. Copyright © 2002, National Space Science Data Center, NASA/Goddard Space Flight Center.

NetCDF Library. Copyright © 1993-1999, University Corporation for Atmospheric Research/Unidata.

HDF EOS Library. Copyright © 1996, Hughes and Applied Research Corporation.

SMACC. Copyright © 2000-2004, Spectral Sciences, Inc. and ITT Visual Information Solutions. All rights reserved.

This software is based in part on the work of the Independent JPEG Group.

Portions of this software are copyrighted by DataDirect Technologies, © 1991-2003.

BandMax®. Copyright © 2003, The Galileo Group Inc.

Portions of this computer program are copyright © 1995-1999, LizardTech, Inc. All rights reserved. MrSID is protected by U.S. Patent No. 5,710,835. Foreign Patents Pending.

Portions of this software were developed using Unisearch's Kakadu software, for which ITT has a commercial license. Kakadu Software. Copyright © 2001. The University of New South Wales, UNSW, Sydney NSW 2052, Australia, and Unisearch Ltd, Australia.

This product includes software developed by the Apache Software Foundation (www.apache.org/).

MODTRAN is licensed from the United States of America under U.S. Patent No. 5,315,513 and U.S. Patent No. 5,884,226.

FLAASH is licensed from Spectral Sciences, Inc. under a U.S. Patent Pending.

Portions of this software are copyrighted by Merge Technologies Incorporated.

Support Vector Machine (SVM) is based on the LIBSVM library written by Chih-Chung Chang and Chih-Jen Lin (www.csie.ntu.edu.tw/~cjlin/libsvm/), adapted by ITT Visual Information Solutions for remote sensing image supervised classification purposes.

IDL Wavelet Toolkit Copyright © 2002, Christopher Torrence.

IMSL is a trademark of Visual Numerics, Inc. Copyright © 1970-2006 by Visual Numerics, Inc. All Rights Reserved.

Other trademarks and registered trademarks are the property of the respective trademark holders.



Contents

Chapter 1	
About the IDL Bridges	9
What Is a Bridge?	10
IDL Import Bridge	11
IDL Export Bridge	12

Part I: Importing into IDL

Chapter 2	
Overview: COM and ActiveX in IDL	15
COM Objects and IDL	16
Using COM Objects with IDL	18
Skills Required to Use COM Objects	19

Chapter 3

Using COM Objects in IDL

in IDL	21
About Using COM Objects in IDL	22
IDLcomIDDispatch Object Naming Scheme	24
Creating IDLcomIDDispatch Objects	28
Method Calls on IDLcomIDDispatch Objects	29
Managing COM Object Properties	37
Passing Parameter Arrays by Reference	40
References to Other COM Objects	42
Destroying IDLcomIDDispatch Objects	43
COM-IDL Data Type Mapping	44
Example: RSIDemoComponent	46

Chapter 4

Using ActiveX Controls in IDL

Using ActiveX Controls in IDL	49
About Using ActiveX Controls in IDL	50
ActiveX Control Naming Scheme	52
Creating ActiveX Controls	53
Method Calls on ActiveX Controls	55
Managing ActiveX Control Properties	56
ActiveX Widget Events	57
Destroying ActiveX Controls	60
Example: Calendar Control	61
Example: Spreadsheet Control	65

Chapter 5

Using Java Objects in IDL

Using Java Objects in IDL	71
Overview of Using Java Objects	72
Initializing the IDL-Java Bridge	75
IDL-Java Bridge Data Type Mapping	78
Creating IDL-Java Objects	84
Method Calls on IDL-Java Objects	87
Managing IDL-Java Object Properties	89
Destroying IDL-Java Objects	91
Showing IDL-Java Output in IDL	92
The IDLJavaBridgeSession Object	94

Java Exceptions	96
IDL-Java Bridge Examples	99
Troubleshooting Your Bridge Session	118

Part II: Exporting from IDL

Chapter 6 **Exporting IDL Objects 125**

Overview of Exporting IDL Objects	126
Wrapper Objects	127
Object Lifecycle	130
IDL Access	132
Parameter Passing and Type Conversion	136
Event Handling	139
Supported Platforms and IDL Modes	140
Configuring Build and Client Machines	142

Chapter 7 **Using the Export Bridge Assistant 147**

Export Bridge Assistant Overview	148
Running the Assistant	149
Using the Assistant	150
Working with a Project	157
Building an Object	161
Exporting an Object	162
Specifying Information for Exporting	164
Information Skipped During Export	178
Exporting a Source Object's Superclasses	180
Modifying a Source Object After Export	181
Wrapper Generation Example	182

Chapter 8 **Using Exported COM Objects 189**

Overview of COM Export Objects	190
COM Wrapper Objects	191
Event Handling	208
Error Handling	211
Debugging	213

Chapter 9	
Using Exported Java Objects	215
Overview of Java Export Objects	216
Java Wrapper Objects	217
Event Handling	232
Error Handling	242
Debugging	244
Chapter 10	
Using the Connector Object	245
About the IDL Connector Object	246
Preparing to Use the IDL Connector Object	247
Connector Object COM Examples	249
Connector Object Java Examples	253
Chapter 11	
Writing IDL Objects for Exporting	261
Overview	262
Programming Limitations	263
Exporting Drawable Objects	264
Drawable Object Canvas Examples	266
Chapter 12	
Creating Custom COM Export Objects	269
About COM Export Object Examples	270
Nondrawable COM Export Example	272
Drawable COM Export Examples	276
Chapter 13	
Creating Custom Java Export Objects	291
About Java Export Object Examples	292
Nondrawable Java Export Example	294
Drawable Java Export Examples	298
Part III: Appendices	
Appendix A	
IDL Java Object API	311
Package Summary	312

Appendix B	
COM Object Creation	471
Sample IDL Object	472
Visual Basic .NET Code Sample	475
C++ Client Code Sample	477
C# Code Sample	479
Visual Basic 6 Code Sample	481
Appendix C	
Java Object Creation	483
Sample IDL Object	484
Java Object Initiation Without Parameters	487
Java Object Initiation with Parameters	489
Appendix D	
Multidimensional Array Storage and Access	493
Overview	494
Why Storage and Access Matter	495
Storage and Access in COM and IDL	496
2D Array Examples	498
Index	503



Chapter 1

About the IDL Bridges

This chapter discusses the following topics.

What Is a Bridge?	10	IDL Export Bridge	12
IDL Import Bridge	11		

What Is a Bridge?

A *bridge* is a technology path that lets applications in different programming languages or environments share information: for example, between IDL and Java. With bridge technology, you can use an application that manipulates data in its native language (e.g., Java) by calling on objects and processes from another language (e.g., IDL). In this way, you can take advantage of both environments to solve a problem that might be otherwise difficult for either environment separately: for example, embedding an IDL data object in a Java GUI to display a complex data transformation.

Note

Startup files are not executed when running bridge applications because an IDL command line is not present. See [“Understanding When Startup Files are Not Executed”](#) (Chapter 1, *Using IDL*) for details.

IDL supports import and export bridge technology. The Import Bridge lets you import the functionality of a COM or Java object to an IDL application. The Export Bridge lets you export the functionality of an IDL object to COM or Java application. See the following for more information:

- [“IDL Import Bridge”](#) on page 11
- [“IDL Export Bridge”](#) on page 12

IDL Import Bridge

The IDL Import Bridge technology lets you use COM and Java objects in IDL applications. For a general overview of this technology, see [“Overview: COM and ActiveX in IDL”](#) on page 15.

COM and ActiveX

You have two options for incorporating a COM object into IDL:

- If the COM object does not have its own interface, you can use the IDLcomIDispatch object class to communicate with the underlying COM object through the COM IDispatch interface (see [“Using COM Objects in IDL”](#) on page 21 for details)
- If the COM object does have its own interface (i.e., it is an ActiveX control), you can use IDL’s WIDGET_ACTIVEX routine to place the control in an IDL widget hierarchy (see [“Using ActiveX Controls in IDL”](#) on page 49 for details)

Java

The IDL-Java bridge lets you access Java objects within IDL code. Java objects imported into IDL behave like normal IDL objects. The bridge also provides IDL with access to exceptions created by the underlying Java object. For more information, see [“Using Java Objects in IDL”](#) on page 71.

IDL Export Bridge

The IDL Export Bridge technology lets you use IDL objects in COM and Java applications. For a general overview of this technology, see [“Exporting IDL Objects”](#) on page 125.

Note

The Export Bridge technology is installed as part of IDL. For the licensing and environment requirements of this technology, see [“Running the Assistant”](#) on page 149.

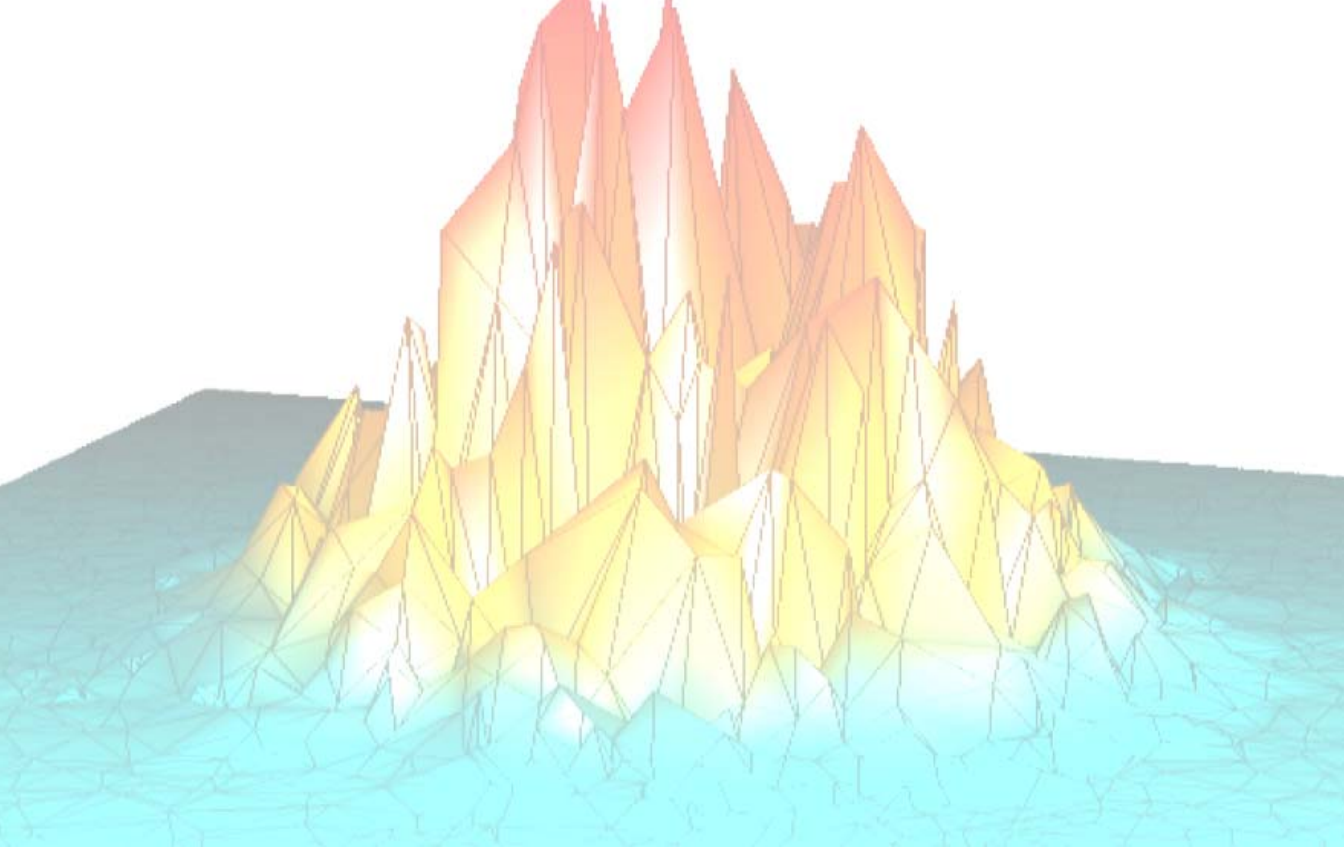
With the Export Bridge, interaction with IDL is through native Java and COM wrapper objects that are generated for each IDL object with which client applications want to interact. The wrapper objects manage all aspects of IDL loading, initialization, process management, and cleanup, so users need only be familiar with the client language (for embedding the wrapper in the client application) and the basics of IDL (for accessing and manipulating IDL data and processes).

Export Bridge Assistant

The key to creating your own exported IDL objects is the Export Bridge Assistant, which generates these native wrapper objects from IDL objects. The Assistant is an interactive dialogue in IDL that lets you customize the wrapper object you want to create from the underlying IDL object. You can select the methods, parameters, and properties that you want to export, as well as other information about the IDL object (e.g., whether to convert array majority for parameters). See [“Using the Export Bridge Assistant”](#) on page 147 for details.

Connector Object

Instead of exporting a custom IDL source object using the Assistant, you can also access IDL functionality using the prebuilt connector object that is shipped with the IDL distribution. This IDL connector object lets you quickly incorporate the processing power of IDL into an application developed in an external, object-oriented environment such as COM or Java. The connector object provides a basic, nondrawable wrapper that includes the ability to get and set IDL variables and execute command statements in the IDL process associated with the connector object. For more information, see [“Using the Connector Object”](#) on page 245.



Part I: Importing into IDL



Chapter 2

Overview: COM and ActiveX in IDL

This chapter discusses the following topics:

COM Objects and IDL	16	Skills Required to Use COM Objects	19
Using COM Objects with IDL	18		

COM Objects and IDL

Microsoft's Component Object Model, or COM, is a specification for developing modular software components. COM is not a programming language or an API, but an implementation of a *component architecture*. A component architecture is a method of designing software components so that they can be easily connected together, reused, or replaced without re-compiling the application that uses them. Other examples of this methodology include the Object Management Group's Common Object Request Broker Architecture (CORBA) and Sun's JavaBeans technologies.

ActiveX controls are a special class of COM object that follow a set of Microsoft interface specifications; they are normally designed to present a user interface.

IDL for Windows supports three methods for using COM-based software components in your applications:

- Exposing a COM object as an IDL object
- Including an ActiveX control in an IDL widget hierarchy

Note

While COM components can be developed for numerous platforms, most COM-based software is written for Microsoft Windows platforms. IDL for Windows supports the inclusion of COM technologies, but IDL for UNIX does not. The chapters in this section will discuss COM in the context of Microsoft Windows exclusively.

What Are COM Objects?

A COM object, or component, is a piece of software that:

- Is a library, rather than a stand-alone application (that is, it runs inside some sort of client application such as IDL, a Visual Basic application, or a Web browser)
- Is distributed in a compiled, executable form
- Exposes a group of methods and properties to its client application

In addition to these criteria, a component may also supply a user interface that can be manipulated by the user. COM objects that supply a user interface and send *events* to the programs that use them are generally packaged as ActiveX controls, although it is not a requirement that an ActiveX control provide a user interface.

COM objects and ActiveX controls are nearly always packaged as Windows executable (.exe), dynamic link library (.dll), or object linking and embedding (.ocx) files.

Why Use COM Objects with IDL?

There are several reasons to use COM technologies alongside IDL:

- COM objects can be designed to use the facilities of the underlying Windows operating system. If you need access to Windows features not exposed within IDL, incorporating a COM object into your IDL program may provide the functionality you need.
- COM objects have been written to provide custom user interface elements or accomplish specific tasks. Many of these components are available to you free or at minimal cost. If you work exclusively in a Windows environment, incorporating a pre-written component in your IDL program may be faster than coding the same functionality in IDL.

Using COM Objects with IDL

The methods for using COM objects with IDL are:

- [“Exposing a COM Object as an IDL Object”](#) on page 18
- [“Including an ActiveX Control in an IDL Widget Hierarchy”](#) on page 18

Exposing a COM Object as an IDL Object

IDL’s `IDLcomIDDispatch` object class creates an IDL object that communicates with an underlying COM object using the COM object’s `IDispatch` interface. When you create an `IDLcomIDDispatch` object, you provide the identifier for the COM object you wish to use, and IDL handles instantiation of and communication with the object. You can call the COM object’s methods and get and set its properties using standard IDL object conventions and syntax.

Note

The `IDLcomIDDispatch` object is useful when you want to incorporate a generic COM object into your IDL application. If the COM object you want to use is an ActiveX control, use the `WIDGET_ACTIVEX` routine, discussed below.

For details on using the `IDLcomIDDispatch` object class to incorporate COM objects into your IDL applications, see [Chapter 3, “Using COM Objects in IDL”](#).

Including an ActiveX Control in an IDL Widget Hierarchy

IDL’s `WIDGET_ACTIVEX` routine incorporates an ActiveX control directly into an IDL widget hierarchy. This allows you to place the ActiveX control in an IDL widget interface, and to receive widget events directly from the control for handling by a standard IDL widget event handler.

Internally, IDL uses the same mechanisms it uses when creating `IDLcomIDDispatch` objects when it instantiates an ActiveX control as part of an IDL widget hierarchy. After the widget hierarchy has been realized, an object reference to the IDL object that encapsulates the ActiveX control can be retrieved and used as an interface with the ActiveX control. This allows you to call the ActiveX control’s methods and get and set its properties using standard IDL object conventions and syntax.

For details on using the `WIDGET_ACTIVEX` routine to incorporate ActiveX controls into your IDL applications, see [Chapter 4, “Using ActiveX Controls in IDL”](#).

Skills Required to Use COM Objects

Although IDL provides an abstracted interface to COM functionality, you must be familiar with some aspects of COM to intertwine COM and IDL successfully.

If You Are Using COM Objects

If you are using a COM object directly, via the `IDLcomIDispatch` object, you will need a thorough understanding of the COM object you are using, including its methods and properties. An understanding of the Windows tools used to discover information about COM objects is useful.

If You Are Using ActiveX Controls

If you are incorporating an ActiveX control into an IDL widget hierarchy using `WIDGET_ACTIVEX`, you will need a thorough understanding of the ActiveX control you are using, including its methods, properties, and the information returned when an event is generated. An understanding of the Windows tools used to discover information about ActiveX controls is useful.

If You Are Creating Your Own COM Object

If you are creating your own COM object to be included in IDL, you will need a thorough understanding both of your development environment and of COM itself. It is beyond the scope of this manual to discuss creation of COM objects, but you should be able to incorporate any component created by following the COM specification into IDL by following the procedures outlined here.



Chapter 3

Using COM Objects in IDL

This chapter discusses the following topics:

About Using COM Objects in IDL	22	Passing Parameter Arrays by Reference . .	40
IDLcomIDDispatch Object Naming Scheme .	24	References to Other COM Objects	42
Creating IDLcomIDDispatch Objects	28	Destroying IDLcomIDDispatch Objects . . .	43
Method Calls on IDLcomIDDispatch Objects	29	COM-IDL Data Type Mapping	44
Managing COM Object Properties	37	Example: RSIDemoComponent	46

About Using COM Objects in IDL

If you want to incorporate a COM object that does not present its own user interface into your IDL application, use IDL's `IDLcomIDDispatch` object class.

IDL's `IDLcomIDDispatch` object class creates an IDL object that uses the COM `IDDispatch` interface to communicate with an underlying COM object. When you create an `IDLcomIDDispatch` object, you provide information about the COM object you wish to use, and IDL handles instantiation of and communication with the object. You can call the COM object's methods and get and set its properties using standard IDL object conventions and syntax.

Note

If the COM object you want to use in your IDL application is an ActiveX control, use the `WIDGET_ACTIVEX` routine, discussed in [Chapter 4, “Using ActiveX Controls in IDL”](#).

Array Data Storage Format

COM, like C, stores array data in row-major format. IDL stores array data in column-major format. See [Appendix D, “Multidimensional Array Storage and Access”](#) for a detailed discussion of this issue and its implications for IDL application design.

Object Creation

To create an IDL object that encapsulates a COM object, use the `OBJ_NEW` function as described in [“Creating IDLcomIDDispatch Objects”](#) on page 28. IDL creates a dynamic subclass of the `IDLcomIDDispatch` object class, based on information you specify for the COM object.

Method Calls and Property Management

Once you have created your `IDLcomIDDispatch` object within IDL, use normal IDL object method calls to interact with the object. (See [Chapter 1, “The Basics of Using Objects in IDL”](#) (*Object Programming*) for a discussion of IDL objects.) COM object properties can be set and retrieved using the `GetProperty` and `SetProperty` methods implemented for the `IDLcomIDDispatch` class. See [“Method Calls on IDLcomIDDispatch Objects”](#) on page 29 and [“Managing COM Object Properties”](#) on page 37 for details.

Object Destruction

Destroy IDLcomIDDispatch objects using the OBJ_DESTROY procedure. See [“Destroying IDLcomIDDispatch Objects”](#) on page 43 for details.

Registering COM Components on a Windows Machine

Before a COM object or ActiveX control can be used by a client program, it must be *registered* on the Windows machine. In most cases, components are registered by the program that installs them on the machine. If you are using a component that is not installed by an installation program that handles the registration, you can register the component manually.

To register a component (.dll or .exe) or a control (.ocx), use the Windows command line program regsvr32, supplying it with name of the component or control to register. For example, the IDL distribution includes a COM component named RSIDemoComponent, contained in a file named RSIDemoComponent.dll located in the examples\doc\bridges\COM subdirectory of the IDL distribution. To register this component, do the following:

1. Open a Windows command prompt.
2. Change directories to the examples\doc\bridges\COM subdirectory of the IDL distribution.
3. Enter the following command:

```
regsvr32 RSIDemoComponent.dll
```

Windows will display a pop-up dialog informing you that the component has been registered. (You can specify the “/s” parameter to regsvr32 to prevent the dialog from being displayed.)

Note

You only need to register a component once on a given machine. It is not necessary to register a component before each use.

IDLcomIDispatch Object Naming Scheme

When you create an IDLcomIDispatch object, IDL automatically creates a *dynamic subclass* of the IDLcomIDispatch class to contain the COM object. IDL determines which COM object to instantiate by parsing the class name you provide to the OBJ_NEW function. You specify the COM object to use by creating a class name that combines the name of the base class (IDLcomIDispatch) with either the COM class identifier or the COM program identifier for the object. The resulting class name looks like

```
IDLcomIDispatch$ID_type$ID
```

where *ID_type* is one of the following:

- CLSID if the object is identified by its COM class ID
- PROGID if the object is identified by its COM program ID

and *ID* is the COM object's actual class or program identifier string.

Note

While COM objects incorporated into IDL are instances of the dynamic subclass created when the COM object is instantiated, they still expose the functionality of the class IDLcomIDispatch, which is the direct superclass of the dynamic subclass. All IDLcomIDispatch methods are available to the dynamic subclass.

Class Identifiers

A COM object's class identifier (generally referred to as the CLSID) is a 128-bit identifying string that is guaranteed to be unique for each object class. The strings used by COM as class IDs are also referred to as *Globally Unique Identifiers* (GUIDs) or *Universally Unique Identifiers* (UUIDs). It is beyond the scope of this chapter to discuss how class IDs are generated, but it is certain that every COM object has a unique CLSID.

COM class IDs are 32-character strings of alphanumeric characters and numerals that look like this:

```
{A77BC2B2-88EC-4D2A-B2B3-F556ACB52E52}
```

The above class identifier identifies the RSIDemoComponent class included with IDL.

When you create an `IDLcomIDispatch` object using a CLSID, you must modify the standard CLSID string in two ways:

1. You must omit the opening and closing braces ({ }).
2. You must replace the dash characters (-) in the CLSID string with underscores (_).

See [“Creating IDLcomIDispatch Objects”](#) on page 28 for example class names supplied to the `OBJ_NEW` function.

Note

If you do not know the class ID of the COM object you wish to expose as an IDL object, you may be able to determine it using an application provided by Microsoft. See [“Finding COM Class and Program IDs”](#) on page 26 for details.

Program Identifiers

A COM object’s program identifier (generally referred to as the PROGID) is a mapping of the class identifier to a more human-friendly string. Unlike class IDs, program IDs are not guaranteed to be unique, so namespace conflicts are possible. Program IDs are, however, easier to work with; if you are not worried about name conflicts, use the identifier you are most comfortable with.

Program IDs are alphanumeric strings that can take virtually any form, although by convention they look like this:

```
PROGRAM.Component.version
```

For example, the `RSIDemoComponent` class included with IDL has the following program ID:

```
RSIDemoComponent.RSIDemoObj1.1
```

When you create an `IDLcomIDispatch` object using a PROGID, you must modify the standard PROGID string by replacing the dot characters (.) with underscores (_).

See [“Creating IDLcomIDispatch Objects”](#) on page 28 for example class names supplied to the `OBJ_NEW` function.

Note

If you do not know the program ID of the COM object you wish to expose as an IDL object, you may be able to determine it using an application provided by Microsoft; see [“Finding COM Class and Program IDs”](#) on page 26 for details.

Finding COM Class and Program IDs

In general, if you wish to incorporate a COM object into an IDL program, you will know the COM class or program ID — either because you created the COM object yourself, or because the developer of the object provided you with the information.

If you do not know the class or program ID for the COM object you want to use, you may be able to determine them using the *OLE/COM Object Viewer* application provided by Microsoft. You can download the OLE/COM Object Viewer at no charge directly from Microsoft. As of this writing, you can locate the tool by pointing your Web browser to the following URL:

<http://www.microsoft.com/com>

and then selecting **Downloads** from the **Resources** menu.

The OLE/COM Object Viewer displays all of the COM objects installed on a computer, and allows you to view information about the objects and their interfaces.

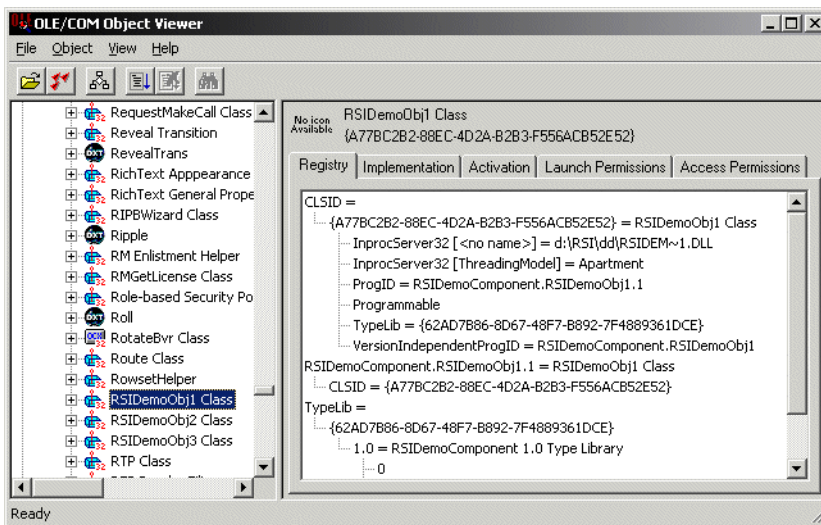


Figure 3-1: Microsoft's OLE/COM Object Viewer Application

Note

You can copy an object's class ID to the clipboard by selecting the object in the leftmost panel of the object viewer, clicking the right mouse button, and selecting "Copy CLSID to Clipboard" from the context menu.

If you have an IDL program that instantiates a COM object running on your computer, you can determine either the class ID or the program ID by using the `HELP` command with the `OBJECTS` keyword. IDL displays the full dynamic subclass name, including the class ID or program ID that was used when the object was created.

Creating IDLcomIDDispatch Objects

To expose a COM object as an IDL object, use the [OBJ_NEW](#) function to create a dynamic subclass of the [IDLcomIDDispatch](#) object class. The name of the subclass must be constructed as described in “[IDLcomIDDispatch Object Naming Scheme](#)” on page 24, and identifies the COM object to be instantiated.

Note

If the COM object you want to use within IDL is an ActiveX control, use the `WIDGET_ACTIVEX` routine as described in [Chapter 4, “Using ActiveX Controls in IDL”](#). Instantiating the ActiveX control as part of an IDL widget hierarchy allows you to respond to events generated by the control, whereas COM objects that are instantiated using the `OBJ_NEW` do not generate events that IDL is aware of.

For example, suppose you wish to include a COM component with the class ID

```
{A77BC2B2-88EC-4D2A-B2B3-F556ACB52E52}
```

and the program ID

```
RSIDemoComponent.RSIDemoObj1.1
```

in an IDL program. Use either of the following calls to the `OBJ_NEW` function:

```
ObjRef = OBJ_NEW( $
    'IDLcomIDDispatch$CLSID$A77BC2B2_88EC_4D2A_B2B3_F556ACB52E52' )
```

or

```
ObjRef = OBJ_NEW( $
    'IDLcomIDDispatch$PROGID$RSIDemoComponent_RSIDemoObj1_1' )
```

IDL’s internal COM subsystem instantiates the COM object within an `IDLcomIDDispatch` object with one of the following the dynamic class names

```
IDLcomIDDispatch$CLSID$A77BC2B2_88EC_4D2A_B2B3_F556ACB52E52
```

or

```
IDLcomIDDispatch$PROGID$RSIDemoComponent_RSIDemoObj1_1
```

and sets up communication between the object and IDL. You can work with the `IDLcomIDDispatch` object just as you would with any other IDL object; calling the object’s methods, and getting and setting its properties.

See “[IDLcomIDDispatch](#)” (*IDL Reference Guide*) for additional details.

Method Calls on IDLcomIDDispatch Objects

IDL allows you to call the underlying COM object's methods by calling methods on the IDLcomIDDispatch object. IDL handles conversion between IDL data types and the data types used by the component, and any results are returned in IDL variables of the appropriate type.

As with all IDL objects, the general syntax is:

```
result = ObjRef -> Method([Arguments])
```

or

```
ObjRef -> Method[, Arguments]
```

where `ObjRef` is an object reference to an instance of a dynamic subclass of the IDLcomIDDispatch class.

Function vs. Procedure Methods

In COM, all object methods are *functions*. IDL's implementation of the IDLcomIDDispatch object maps COM methods that supply a return value using the `retval` attribute as IDL functions, and COM methods that do not supply a return value via the `retval` attribute as procedures. See [“Displaying Interface Information using the Object Viewer”](#) on page 33 for more information on determining which methods use the `retval` attribute.

The IDLcomIDDispatch::GetProperty and IDLcomIDDispatch::SetProperty methods are special cases. These methods are IDL object methods — not methods of the underlying COM object — and they use *procedure* syntax. The process of getting and setting properties on COM objects encapsulated in IDLcomIDDispatch objects is discussed in [“Managing COM Object Properties”](#) on page 37.

Note

The IDL object system uses method names to identify and call object lifecycle methods (Init and Cleanup). If the COM object underlying an IDLcomIDDispatch object implements Init or Cleanup methods, they will be overridden by IDL's lifecycle methods, and the COM object's methods will be inaccessible from IDL. Similarly, IDL implements the GetProperty and SetProperty methods for the IDLcomIDDispatch object, so any methods of the underlying COM object that use these names will be inaccessible from IDL.

What Happens When a Method Call Is Made?

When a method is called on an IDLcomIDDispatch object, the method name and arguments are passed to the internal IDL COM subsystem, where they are used to construct the appropriate IDispatch method calls for the underlying COM object.

From the point of view of an IDL user issuing method calls on the IDLcomIDDispatch object, this process is completely transparent. The IDL user simply calls the COM object's method using IDL syntax, and IDL handles the translation.

Data Type Conversions

IDL and COM use different data types internally. While you should be aware of the types of data expected by the COM object's methods and the types it returns, you do not need to worry about converting between IDL data types and COM data types manually. IDL's dynamic type conversion facilities handle all conversion of data types between IDL and the COM system. The data type mappings are described in [“COM-IDL Data Type Mapping”](#) on page 44.

For example, if the COM object that underlies an IDLcomIDDispatch object has a method that requires a value of type INT as an input argument, you would supply the value as an IDL Long. If you supplied the value as any other IDL data type, IDL would first convert the value to an IDL Long using its normal data type conversion mechanism before passing the value to the COM object as an INT.

Similarly, if a COM object returns a BOOL value, IDL will place the value in a variable of Byte type, with a value of 1 (one) signifying True or a value of 0 (zero) signifying False.

Optional Arguments

Like IDL routines, COM object methods can have *optional arguments*. Optional arguments eliminate the need for the calling program to provide input data for all possible arguments to the method for each call. The COM optional argument functionality is passed along to COM object methods called on IDLcomIDDispatch objects, and to the IDLcomIDDispatch::GetProperty method. This means that if an argument is not required by the underlying COM object method, it can be omitted from the method call used on the IDLcomIDDispatch object.

Note

Only method arguments defined with the `optional` token in the object's interface definition are optional. See [“Displaying Interface Information using the Object Viewer”](#) on page 33 for more information regarding the object's interface definition file.

Warning

If an argument that is *not* optional is omitted from the method call used on the `IDLcomIDispatch` object, IDL will generate an error.

Argument Order

Like IDL, COM treats arguments as *positional parameters*. This means that it makes a difference where in the argument list an argument occurs. (Contrast this with IDL's handling of keywords, which can occur anywhere in the argument list after the routine name.) COM enforces the following ordering for arguments to object methods:

1. Required arguments
2. Optional arguments for which default values are defined
3. Optional arguments for which no default values are defined

The same order applies when the method is called on an `IDLcomIDispatch` object.

Default Argument Values

COM allows objects to specify a default value for any method arguments that are optional. If a call to a method that has an optional argument with a default value omits the optional argument, the default value is used. IDL behaves in the same way as COM when calling COM object methods on `IDLcomIDispatch` objects, and when calling the `IDLcomIDispatch::GetProperty` method.

Method arguments defined with the `defaultvalue()` token in the object's interface definition are optional, and will use the specified default value if omitted from the method call. See [“Displaying Interface Information using the Object Viewer”](#) on page 33 for more information regarding the object's interface definition file.

Argument Skipping

COM allows methods with optional arguments to accept a subset of the full argument list by specifying which arguments are not present. This allows the calling routine to supply, for example, the first and third arguments to a method, but not the second. IDL provides the same functionality for COM object methods called on

IDLcomIDDispatch objects, but not for the IDLcomIDDispatch::GetProperty or SetProperty methods.

To skip one or more arguments from a list of optional arguments, include the SKIP keyword in the method call. The SKIP keyword accepts either a scalar or a vector of numbers specifying which arguments are not provided.

Note

The indices for the list of method arguments are zero-based — that is, the first method argument (either optional or required) is argument 0 (zero), the next is argument 1 (one), *etc.*

For example, suppose a COM object method accepts four arguments, of which the second, third, and fourth are optional:

```
ObjMethod, arg1, arg2-optional, arg3-optional, arg4-optional
```

To call this method on the IDLcomIDDispatch object that encapsulates the underlying COM object, skipping `arg2`, use the following command:

```
objRef->ObjMethod, arg1, arg3, arg4, SKIP=1
```

Note that the SKIP keyword uses the index value 1 to indicate the second argument in the argument list. Similarly, to skip `arg2` and `arg3`, use the following command:

```
objRef->ObjMethod, arg1, arg4, SKIP=[1,2]
```

Finally, note that you do not need to supply the SKIP keyword if the arguments are supplied in order. For example, to skip `arg3` and `arg4`, use the following command:

```
objRef->ObjMethod, arg1, arg2
```

Finding Object Methods

In most cases, when you incorporate a COM object into an IDL program, you will know what the COM object's methods are and what arguments and data types those methods take, either because you created the COM object yourself or because the developer of the object provided you with the information.

If for some reason you do not know what methods the COM object supports, you may be able to determine which methods are available and what parameters they accept using the *OLE/COM Object Viewer* application provided by Microsoft. (See [“Finding COM Class and Program IDs”](#) on page 26 for information on acquiring the OLE/COM Object Viewer.)

Warning

Finding information about a COM object's methods using the OLE/COM Object Viewer requires a moderately sophisticated understanding of COM programming, or at least COM interface definitions. While we provide some hints in this section on how to interpret the interface definition, if you are not already familiar with the structure of COM objects you may find this material inadequate. If possible, consult the developer of the COM object you wish to use rather than attempting to determine its structure using the object viewer.

Displaying Interface Information using the Object Viewer

You can use the OLE/COM Object Viewer to view the interface definitions for any COM object on your Windows machine. Select a COM object in the leftmost panel of the object viewer, click the right mouse button, and select "View Type Information..." A new window titled "ITypeLib Viewer" will be displayed, showing all of the component's interfaces (Figure 3-2).

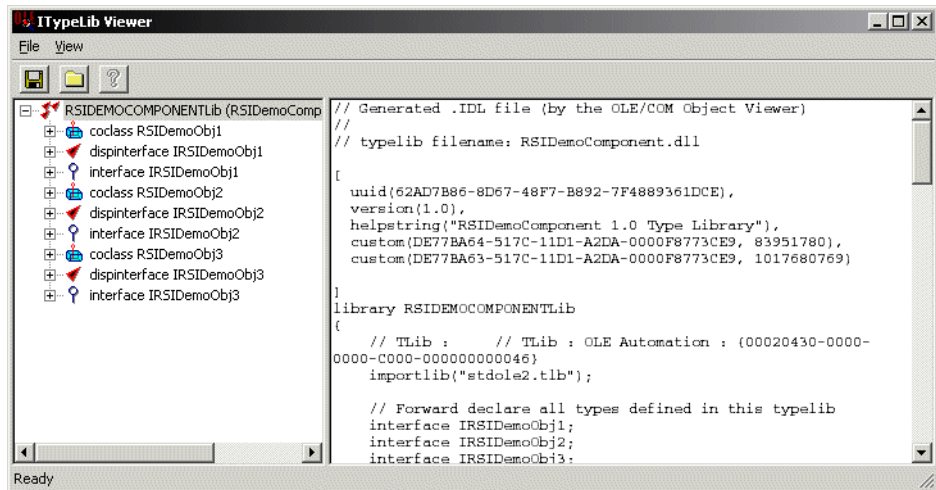


Figure 3-2: Viewing a COM Object's Interface Definition

Note

The top lines in the right-hand panel will say something like:

```
// Generated .IDL file (by the OLE/COM Object Viewer)
//
```

```
// typelib filename: RSIDemoComponent.dll
```

The “.IDL file” in this case has nothing to do with IDL, the Interactive Data Language. Here “IDL” stands for *Interface Description Language* — a language used to define component interfaces. If you are familiar with the Interface Description Language, you can often determine what a component is designed to do.

With the top-level object selected in the left-hand pane of the ITypelib Viewer, scroll down in the right-hand pane until you find the section that defines the IDispatch interface for the object in question. The definition will look something like this:

```
interface IRSIDemoObj1 : IDispatch {
    [id(0x00000001)]
    HRESULT GetCLSID([out, retval] BSTR* pBstr);
    [id(0x00000002), propput]
    HRESULT MessageStr([in] BSTR pstr);
    [id(0x00000002), propget]
    HRESULT MessageStr([out, retval] BSTR* pstr);
    [id(0x00000003)]
    HRESULT DisplayMessageStr();
    [id(0x00000004)]
    HRESULT Msg2InParams(
        [in] BSTR str,
        [in] long val,
        [out, retval] BSTR* pVal);
    [id(0x00000005)]
    HRESULT GetIndexObject(
        [in] long ndxObj,
        [out, retval] IDispatch** ppDisp);
    [id(0x00000006)]
    HRESULT GetArrayOfObjects(
        [out] long* pObjCount,
        [out, retval] VARIANT* psaObjs);
};
```

Method definitions look like this:

```
[id(0x00000001)]
HRESULT GetCLSID([out, retval] BSTR* pBstr);
```

where the line including the `id` string is an identifier used by the object to refer to its methods and the following line or lines (usually beginning with `HRESULT`) define the method’s interface.

Again, while it is beyond the scope of this manual to discuss COM object methods in detail, the following points may assist you in determining how to use a COM object:

- Methods whose definitions include the `retval` attribute will appear in IDL as functions.

```
[id(0x00000001)]
HRESULT GetCLSID([out, retval] BSTR* pBstr);
```

- Methods that do not include the `retval` attribute will appear in IDL as procedures.

```
[id(0x00000003)]
HRESULT DisplayMessageStr();
```

- Methods whose definitions include the `propget` attribute allow you to retrieve an object property using the `IDLcomIDispatch::GetProperty` method. You cannot call these methods directly in IDL; see [“Managing COM Object Properties”](#) on page 37 for additional details.

```
[id(0x00000002), propget]
HRESULT MessageStr([out, retval] BSTR* pstr);
```

- Methods whose definitions include the `propput` attribute allow you to set an object property using the `IDLcomIDispatch::SetProperty` method. You cannot call these methods directly in IDL; see [“Managing COM Object Properties”](#) on page 37 for additional details.

```
[id(0x00000002), propput]
HRESULT MessageStr([in] BSTR pstr);
```

- Methods that accept optional input values will include the `optional` token in the argument’s definition. For example, the following definition indicates that the second input argument is optional:

```
[id(0x00000004)]
HRESULT Msglor2InParams(
    [in] BSTR str,
    [in, optional] int val,
    [out, retval] BSTR* pVal);
```

- Methods that provide default values for optional arguments replace the `optional` token with the `defaultvalue()` token, where the default value of the argument is supplied between the parentheses. For example, the following definition indicates that the second input argument is optional, and has a default value of 15:

```
HRESULT Msglor2InParams(
    [in] BSTR str,
    [in, defaultvalue(15)] int val,
    [out, retval] BSTR* pVal);
```

- While methods generally return an `HRESULT` value, this is not a requirement.

Displaying Interface Information Using the IDL HELP Procedure

If you have an IDL program that instantiates a COM object running on your computer, you can determine either the class ID or the program ID by using the `HELP` command with the `OBJECTS` keyword. IDL displays a list of objects, along with their methods, with function and procedure methods in separate groups for each object class.

Managing COM Object Properties

As a convenience to the IDL programmer, COM object methods that have been defined using the `propget` and `propput` attributes are accessible via the IDLcomIDispatch object's `GetProperty` and `SetProperty` methods. This means that rather than calling the COM object's methods directly to get and set property values, you use the standard IDL syntax.

Note

If a COM object method's interface definition includes either the `propget` or the `propput` attribute, you *must* use the IDL `GetProperty` and `SetProperty` methods to get and set values. IDL does not allow you to call these methods directly.

As with all IDL objects, the IDLcomIDispatch object's `GetProperty` and `SetProperty` methods use procedure syntax. Keywords to the methods represent the names of the properties being retrieved or set, and the keyword values represent either an IDL variable into which the property value is placed or an IDL expression that is the value to which the property is set. You must use the procedure syntax when calling either method, even if the underlying COM object methods being used are functions rather than procedures.

Setting Properties

To set a property value on a COM object, use the following syntax:

```
ObjRef->SetProperty, KEYWORD=Expression
```

where `ObjRef` is the IDLcomIDispatch object that encapsulates the COM object, `KEYWORD` is the COM object property name, and `Expression` is an IDL expression representing the property value to be set.

If the underlying COM object's `propput` method requires additional arguments, the arguments are supplied in the `setProperty` method call, using the following syntax:

```
ObjRef->SetProperty [, arg0, arg1, ... argn], KEYWORD=Expression
```

Note

`KEYWORD` must map exactly to the full name of the underlying COM object's property setting method. The partial keyword name functionality provided by IDL is not valid with IDLcomIDispatch objects.

You can set multiple property values in a single statement by supplying multiple `KEYWORD=Expression` pairs.

IDL lets you to set multiple properties at once in the same SetProperty call. For example:

```
ObjRef->SetProperty, OPTION=1, INDEX=99L
```

This command is equivalent to the following lines:

```
ObjRef->SetProperty, OPTION=1
ObjRef->SetProperty, INDEX=99L
```

If you pass parameters when setting multiple properties, the parameter or parameters are sent to each property being set. For example:

```
ObjRef->SetProperty, 'Parm1', 24L, oRef, OPTION=1, INDEX=99L
```

This command is equivalent to the following lines:

```
ObjRef->SetProperty, 'Parm1', 24L, oRef, OPTION=1
ObjRef->SetProperty, 'Parm1', 24L, oRef, INDEX=99L
```

Thus, when you are setting multiple properties at the same time and passing parameters, all the properties that are set at the same time must be defined as receiving the same sets of parameters.

Getting Properties

To retrieve a property value from a COM object, use the following syntax:

```
ObjRef->GetProperty, KEYWORD=Variable
```

where *ObjRef* is the IDLcomIDispatch object that encapsulates the COM object, *KEYWORD* is the COM object property name, and *Variable* is the name of an IDL variable that will contain the retrieved property value.

Note

KEYWORD must map exactly to the full name of the underlying COM object's property getting method. The partial keyword name functionality provided by IDL is not valid with IDLcomIDispatch objects.

You can get multiple property values in a single statement by supplying multiple *KEYWORD=Variable* pairs.

Because some of the underlying COM object's *propget* methods may require arguments, the IDLcomIDispatch object's *GetProperty* method will accept optional arguments. To retrieve a property using a method that takes arguments, use the following syntax:

```
ObjRef->GetProperty [, arg0, arg1, ... argn], KEYWORD=Variable
```

Note, however, that if arguments are required, you can only specify one property to retrieve.

Passing Parameter Arrays by Reference

By default, IDL arrays are passed to and received from the COM subsystem “by value”, meaning the array is copied. When dealing with large arrays or a large number of arrays, performance may suffer due to the by value passing scheme. However, you can implement “by reference” array passing, which passes an IDL array to a COM object in such a way that the COM object can directly alter the IDL array memory without the cost of marshaling (copying) the array to or from the COM object. This can increase performance and save system memory allocation.

An IDL array parameter is passed by reference to a COM method when the parameter is defined as an IDL pointer to an array. For example:

```
myarr = LINDGEN(100)
myptr = PTR_NEW(myarr, /NO_COPY)
```

or

```
myptr = PTR_NEW(LINDGEN(100), /NO_COPY)
```

Then, the pointer is passed like a normal parameter:

```
PRINT, *myptr           ; array before call
obj->UseArrayRef, myptr
PRINT, *myptr           ; altered array after call
```

The IDL array must be large enough for the client's use. On the COM side:

- The COM object cannot resize the array (although the COM object does not have to use or set all the elements in the array)
- The COM object cannot change the type of elements
- The COM object cannot change the dimensionality of the array

Thus, for multidimensional arrays, IDL must define the source array with the same dimensions as the COM client expects.

In order for the IDL-COM subsystem to know that an IDL array should be passed by reference, it looks at the source IDL variable to make sure it is a pointer to an array, and that the destination COM method parameter is also declared as an array. Thus, it is important to properly declare the destination COM parameter as a `SAFEARRAY(<type>)`, when implementing in C++.

For example, if the desire is to pass an IDL array of 32-bit integer values to a COM client, the COM method parameter needs to be declared like this:

```
[in, out] SAFEARRAY(long) psa
```


For the code example above, the full method signature in C++/ATL is:

```
HRESULT UseArrayRef( [in, out] SAFEARRAY(long) psa);
```

When implementing a COM-callable class in C# and passing in an array of 32-bit integers, declare the method as:

```
public void UseArrayRef( [MarshalAs(UnmanagedType.SafeArray,  
SafeArraySubType=System.Runtime.InteropServices.VarEnum.VT_I4)]  
ref long [] arr)  
{  
    arr[0] = 10;  
    arr[1] = 11;  
    // etc  
}
```

It is critical to make sure that the element size of the IDL array matches the element size declared in the COM method signature. If they don't, a marshaling error occurs because the marshaler checks for consistency between the source and destination. This issue is notorious for causing problems with element types of “int” and “long”. For example, trying to call either of the two COM method signatures above with an IDL “integer” array would cause an error since IDL “integers” are 16-bits by default and C++/COM “ints” are 32-bits. Thus, in the code above, we declared the IDL array as “long” values, which are 32-bits and match the C++/COM “long” value in size.

Unsupported Array Types

You cannot pass an array by reference if the array consists of one of the following types:

- Strings
- Object references
- IDL pointers
- IDL structures

References to Other COM Objects

It is not uncommon for COM objects to return references to other COM objects, either as a property value or via an object method. If an IDLcomIDispatch object returns a reference to another COM object's IDispatch interface, IDL automatically creates an IDLcomIDispatch object to contain the object reference.

For example, suppose the `GetOtherObject` method to the COM object encapsulated by the IDLcomIDispatch object `Obj1` returns a reference to another COM object.

```
Obj2 = Obj1->GetOtherObject()
```

Here, `Obj2` is an IDLcomIDispatch object that encapsulates some other COM object. `Obj2` behaves in the same manner as any IDLcomIDispatch object.

Note that IDLcomIDispatch objects created in this manner are not linked in any way to the object whose method created them. In the above example, this means that destroying `Obj1` does not destroy `Obj2`. If the COM object you are using creates new IDLcomIDispatch objects in this manner, you must be sure to explicitly destroy the automatically-created objects along with those you explicitly create, using the `OBJ_DESTROY` procedure.

Destroying IDLcomIDDispatch Objects

Use the OBJ_DESTROY procedure to destroy an IDLcomIDDispatch object.

When OBJ_DESTROY is called with an IDLcomIDDispatch object as an argument, the underlying reference to the COM object is released and IDL resources relating to that object are freed.

Note

Destroying an IDLcomIDDispatch object does not automatically cause the destruction of the underlying COM object. COM employs a reference-counting methodology and expects the COM object to destroy itself when there are no remaining references. When an IDLcomIDDispatch object is destroyed, IDL simply decrements the reference count on the underlying COM object.

Note

IDL does not automatically destroy an object when the object variable goes out of scope (e.g., when a procedure returns). If the IDLcomIDDispatch object is not explicitly destroyed, the COM reference count is not decremented, which could keep the object instantiated and never released.

COM-IDL Data Type Mapping

When data moves from IDL to a COM object and back, IDL handles conversion of variable data types automatically. The data type mappings are shown in [Table 3-1](#).

COM Type	IDL Type
BOOL (VT_BOOL)	Byte (true =1, false=0)
ERROR (VT_ERROR)	Long
CY (VT_CY)	Double (see note below)
DATE (VT_DATE)	Double
I1 (VT_I1)	Byte
INT (VT_INT)	Long
UINT (VT_UINT)	Unsigned Long
VT_USERDEFINED	The IDL type is passed through
VT_UI1	Byte
VT_I2	Integer
VT_UI2	Unsigned integer
VT_ERROR	Long
VT_I4	Long
VT_UI4	Unsigned Long
VT_I8	Long64
VT_UI8	Unsigned Long 64
VT_R4	Float
VT_BSTR	String
VT_R8	Double

Table 3-1: IDL-COM Data Type Mapping

COM Type	IDL Type
VT_DISPATCH	IDLcomIDispatch
VT_UNKNOWN	IDLcomIDispatch

Table 3-1: IDL-COM Data Type Mapping (Continued)

Note on the COM CY Data Type

The COM CY data type is a scaled 64-bit integer, supporting exactly four digits to the right of the decimal point. To provide an easy-to-use interface, IDL automatically scales the integer as part of the data conversion that takes place between COM and IDL, allowing the IDL user to treat the number as a double-precision floating-point value. When the value is passed back to the COM object, it will be truncated if there are more than four significant digits to the right of the decimal point.

For example, the IDL double-precision value 234.56789 would be passed to the COM object as 234.5678.

Example: RSIDemoComponent

This example uses a COM component included in the IDL distribution. The RSIDemoComponent is included purely for demonstration purposes, and does not perform any useful work beyond illustrating how IDLcomIDispatch objects are created and used.

The RSIDemoComponent is contained in a file named `RSIDemoComponent.dll` located in the `examples\doc\bridges\COM` subdirectory of the IDL distribution. Before attempting to execute this example, make sure the component is registered on your system as described in [“Registering COM Components on a Windows Machine”](#) on page 23.

There are three objects defined by the RSIDemoComponent. The example begins by using RSIDemoObj1, which has the program ID:

```
RSIDemoComponent.RSIDemoObj1
```

and the class ID:

```
{A77BC2B2-88EC-4D2A-B2B3-F556ACB52E52}
```

Example Code

This complete example, `IDispatchDemo.pro`, is located in the `examples\doc\bridges\COM` subdirectory of the IDL distribution. It develops an IDL procedure called `IDispatchDemo` that illustrates use of the RSIDemoComponent. Run the example procedure by entering `IDispatchDemo` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT IDispatchDemo.pro`.

1. Begin by creating an IDLcomIDispatch object from the COM object. You can use either the class ID or the program ID. Remember that if you use the class ID, you must remove the braces ({ }) and replace the hyphens with underscores.

```
obj1 = OBJ_NEW( $
    'IDLcomIDispatch$PROGID$RSIDemoComponent_RSIDemoObj1' )
```

or (with Class ID):

```
obj1 = OBJ_NEW( $
    'IDLcomIDispatch$CLSID$A77BC2B2_88EC_4D2A_B2B3_F556ACB52E52' )
```

2. The COM object implements the `GetCLSID` method, which returns the class ID for the component. You can retrieve this value in an IDL variable and

print it. The string should be ' {A77BC2B2-88EC-4D2A-B2B3-F556ACB52E52} '.

```
strCLSID = obj1->GetCLSID()
PRINT, strCLSID
```

Note

The GetCLSID method returns the class identifier of the object using the standard COM separators (-).

3. The COM object has a property named MessageStr. To retrieve the value of the MessageStr property, enter:

```
obj1 -> GetProperty, MessageStr = outStr
PRINT, outStr
```

IDL should print 'RSIDemoObj1'.

4. You can also set the MessageStr property of the object and display it using the object's DisplayMessageStr method, which displays the value of the MessageStr property in a Windows dialog:

```
obj1 -> SetProperty, MessageStr = 'Hello, world'
obj1 -> DisplayMessageStr
```

5. The Msg2InParams method takes two input parameters and concatenates them into a single string. Executing the following commands should cause IDL to print 'The value is: 25'.

```
instr = 'The value is: '
val = 25L
outStr = obj1->Msg2InParams(instr, val)
PRINT, outStr
```

6. To view all known information about the IDLcomIDDispatch object, including its dynamic subclass name and the names of its methods, use the IDL HELP command with the OBJECTS keyword:

```
HELP, obj1, /OBJECTS
```

7. The GetIndexObject() method returns an object reference to one of the following three possible objects:
 - RSIDemoObj1 (index = 1)
 - RSIDemoObj2 (index = 2)
 - RSIDemoObj3 (index = 3)

Note

If the index is not 1, 2, or 3, the `GetIndexObject` method will return an error.

To get a reference to `RSIDemoObj3`, use the following command:

```
obj3 = obj1->GetIndexObject(3)
```

8. All three objects have the `GetCLSID` method. You can use this method to verify that the desired object was returned. The output of the following commands should be `{13AB135D-A361-4A14-B165-785B03AB5023}`.

```
obj3CLSID = obj3->GetCLSID()
PRINT, obj3CLSID
```

9. Remember to destroy a retrieved object when you are finished with it:

```
OBJ_DESTROY, obj3
```

10. Next, use the COM object's `GetArrayOfObjects()` method to return a vector of object references to `RSIDemoObj1`, `RSIDemoObj2`, and `RSIDemoObj3`, respectively. The number of elements in the vector is returned in the first parameter; the result should 3.

```
objs = obj1->GetArrayOfObjects(cItems)
PRINT, cItems
```

11. Since each object implements the `GetCLSID` method, you could loop through all the object references and get its class ID:

```
FOR i = 0, cItems-1 do begin
    objCLSID = objs[i] -> GetCLSID()
    PRINT, 'Object[' , i , ']' CLSID: ', objCLSID
ENDFOR
```

12. Remember to destroy object references when you are finished with them:

```
OBJ_DESTROY, objs
OBJ_DESTROY, obj1
```




Chapter 4

Using ActiveX Controls in IDL

This chapter discusses the following topics:

About Using ActiveX Controls in IDL	50	ActiveX Widget Events	57
ActiveX Control Naming Scheme	52	Destroying ActiveX Controls	60
Method Calls on ActiveX Controls	55	Example: Calendar Control	61
Managing ActiveX Control Properties	56	Example: Spreadsheet Control	65

About Using ActiveX Controls in IDL

If you want to incorporate a COM object that presents a user interface (that is, an ActiveX control) into your IDL application, use IDL's `WIDGET_ACTIVEX` routine to place the control in an IDL widget hierarchy. IDL provides the same object method and property manipulation facilities for ActiveX controls as it does for COM objects incorporated using the `IDLcomIDDispatch` object interface, but adds the ability to process events generated by the ActiveX control using IDL's widget event handling mechanisms.

Note

IDL can only incorporate ActiveX controls on Windows 2000/XP (and later) platforms. See [“Feature Support by Operating System”](#) (*What's New in IDL 6.4*) for details.

When you use the `WIDGET_ACTIVEX` routine, IDL automatically creates an `IDLcomActiveX` object that encapsulates the ActiveX control. `IDLcomActiveX` objects are a subclass of the `IDLcomIDDispatch` object class, and share all of the `IDLcomIDDispatch` methods and mechanisms discussed in [Chapter 3, “Using COM Objects in IDL”](#). You should be familiar with the material in that chapter before attempting to incorporate ActiveX controls in your IDL programs.

Note

If the COM object you want to use in your IDL application is *not* an ActiveX control, use the `IDLcomIDDispatch` object class.

Warning: Modeless Dialogs

When displaying an ActiveX form or dialog box, it is the responsibility of the COM object to pump messages. Modal dialogs pump messages themselves, but modeless dialogs do not. IDL's COM subsystem does not provide the ability to pump messages explicitly, giving IDL no way to pump messages while a modeless dialog is displayed. As a result, calling a modeless dialog from IDL will result in an error.

Registering COM Components on a Windows Machine

Before a COM object or ActiveX control can be used by a client program, it must be *registered* on the Windows machine. In most cases, components are registered by the program that installs them on the machine. If you are using a component that is not

installed by an installation program that handles the registration, you can register the component manually. For a description of the registration process, see [“Registering COM Components on a Windows Machine”](#) on page 23.

ActiveX Control Naming Scheme

When you incorporate an ActiveX control into an IDL widget hierarchy using the `WIDGET_ACTIVEX` routine, IDL automatically creates an `IDLcomActiveX` object that instantiates the control and handles all communication between it and IDL. You tell IDL which ActiveX control to instantiate by passing the COM class or program ID for the ActiveX control to the `WIDGET_ACTIVEX` routine as a parameter.

IDL automatically creates a *dynamic subclass* of the `IDLcomActiveX` class (which is itself a subclass of the `IDLcomIDispatch` class) to contain the ActiveX control. The resulting class name looks like

```
IDLcomActiveX$ID_type$ID
```

where *ID_type* is one of the following:

- `CLSID` if the object is identified by its COM class ID
- `PROGID` if the object is identified by its COM program ID

and *ID* is the COM object's actual class or program identifier string.

For more on COM class and program IDs see [“Class Identifiers”](#) on page 24 and [“Program Identifiers”](#) on page 25.

While you will never need to use this dynamic class name directly, you may see it reported by IDL via the `HELP` routine or in error messages. Note that when IDL reports the name of the dynamic subclass, it will replace the hyphen characters in a class ID and the dot characters in a program ID with underscore characters. This is because neither the hyphen nor the dot character are valid in IDL object names.

Finding COM Class and Program IDs

In general, if you wish to incorporate an ActiveX object into an IDL widget hierarchy, you will know the COM class or program ID, either because you created the control yourself or because the developer of the control provided you with the information.

If you do now know the class or program ID for the COM object you want to use, you may be able to determine them using the *OLE/COM Object Viewer* application provided by Microsoft. For more information, see [“Finding COM Class and Program IDs”](#) on page 26.

Creating ActiveX Controls

To include an ActiveX control in an IDL application, use the `WIDGET_ACTIVEX` function, supplying the COM class or program ID of the ActiveX control as the `COM_ID` argument.

Note

If the object you want to use in your IDL application is *not* an ActiveX control, use the `IDLcomIDDispatch` object class as described in [Chapter 3, “Using COM Objects in IDL”](#). Instantiating a non-ActiveX component using the `WIDGET_ACTIVEX` function is not supported, and may lead to unpredictable results.

Once the ActiveX object has been instantiated within an IDL widget hierarchy, you can call the control’s native methods as described in [“Method Calls on ActiveX Controls”](#) on page 55, and access or modify its properties as described in [“Managing ActiveX Control Properties”](#) on page 56. IDL widget events generated by the control are discussed in [“ActiveX Widget Events”](#) on page 57.

For example, suppose you wished to include an ActiveX control with the class ID:

```
{0002E510-0000-0000-C000-000000000046}
```

and the program ID:

```
OWC.Spreadsheet.9
```

in an IDL widget hierarchy. Use either of the following calls the `WIDGET_ACTIVEX` function:

```
wAx = WIDGET_ACTIVEX(wBase, $
    '0002E510-0000-0000-C000-000000000046')
```

or

```
wAx = WIDGET_ACTIVEX(wBase, 'OWC.Spreadsheet.9', ID_TYPE=1)
```

where `wBase` is the widget ID of the base widget that will contain the ActiveX control.

Note

When instantiating an ActiveX control using the `WIDGET_ACTIVEX` function, you do not need to modify the class or program ID as you do when creating an `IDLcomIDDispatch` object using the `OBJ_NEW` function. Be aware, however, that when IDL creates the underlying `IDLcomActiveX` object, the dynamic class name will replace the hyphens from a class ID or the dots from a program ID with underscore characters.

IDL's internal COM subsystem instantiates the ActiveX control within an IDLcomActiveX object with one of the following dynamic class names

```
IDLcomActiveX$CLSID$0002E510_0000_0000_C000_000000000046
```

or

```
IDLcomActiveX$PROGID$OWC_Spreadsheet_9
```

and sets up communication between the object and IDL. IDL also places the control into the specified widget hierarchy and prepares to accept widget events generated by the control.

See “[WIDGET_ACTIVEX](#)” (*IDL Reference Guide*) for additional details.

Method Calls on ActiveX Controls

IDL allows you to call the underlying ActiveX control's methods by calling methods on the IDLcomActiveX object that is automatically created when you call the WIDGET_ACTIVEX function. IDL handles conversion between IDL data types and the data types used by the component, and any results are returned in IDL variables of the appropriate type. As with all IDL objects, the general syntax is:

```
result = ObjRef->Method([Arguments])
```

or

```
ObjRef -> Method[, Arguments]
```

where ObjRef is an object reference to an instance of a dynamic subclass of the IDLcomActiveX class.

The IDLcomActiveX object class is a direct subclass of the IDLcomIDDispatch object class and provides none of its own methods. As a result, method calls on IDLcomActiveX objects follow the same rules as calls on IDLcomIDDispatch objects. You should read and understand [“Method Calls on IDLcomIDDispatch Objects”](#) on page 29 before calling an ActiveX control's methods.

Retrieving the Object Reference

Unlike IDLcomIDDispatch objects, which you create explicitly with a call to the OBJ_NEW function, IDLcomActiveX objects are created automatically by IDL. To obtain an object reference to the automatically created IDLcomActiveX object, use the GET_VALUE keyword to the WIDGET_CONTROL procedure.

For example, consider the following lines of IDL code:

```
wBase = WIDGET_BASE()
wAx = WIDGET_ACTIVEX(wBase, 'myProgram.myComponent.1', ID_TYPE=1)
WIDGET_CONTROL, wBase, /REALIZE
WIDGET_CONTROL, wAx, GET_VALUE=oAx
```

The first line creates a base widget that will hold the ActiveX control. The second line instantiates the ActiveX control using its program ID and creates an IDLcomActiveX object. The third line realizes the base widget and the ActiveX control it contains; note that the ActiveX widget must be realized before you can retrieve a reference to the IDLcomActiveX object. The fourth line uses the WIDGET_CONTROL procedure to retrieve an object reference to the IDLcomActiveX object in the variable oAx. You can use this object reference to call the ActiveX control's methods and set its properties.

Managing ActiveX Control Properties

As a convenience to the IDL programmer, ActiveX control methods that have been defined using the `propget` and `propput` attributes are accessible via the `IDLcomActiveX` object's `GetProperty` and `SetProperty` methods, which are inherited directly from the `IDLcomIDispatch` object class. This means that rather than calling the ActiveX control's methods directly to get and set property values, you use the standard IDL syntax.

The `IDLcomActiveX` object class is a direct subclass of the `IDLcomIDispatch` object class and provides none of its own methods. As a result, IDL's facilities for managing the properties of ActiveX controls follow the same rules as for `IDLcomIDispatch` objects. You should read and understand [“Managing COM Object Properties”](#) on page 37 before working with an ActiveX control's properties.

ActiveX Widget Events

Events generated by an ActiveX control are dispatched using the standard IDL widget methodology. When an ActiveX event is passed into IDL, it is packaged in an anonymous IDL structure that contains the ActiveX event parameters.

While the actual structure of an event generated by an ActiveX control will depend on the control itself, the following gives an idea of the structure's format:

```
{ ID           : 0L,
  TOP          : 0L,
  HANDLER      : 0L,
  DISPID       : 0L, ; The DISPID of the callback method
  EVENT_NAME    : "", ; The name of the callback method
  <Param1 name> : <Param1 value>,
  <Param2 name> : <Param2 value>,

  <ParamN name> : <ParamN value>
}
```

As with other IDL Widget event structures, the first three fields are standard. ID is the widget id of the widget generating the event, TOP is the widget ID of the top level widget containing ID, and HANDLER contains the widget ID of the widget associated with the handler routine.

The DISPID field contains the decimal representation of the *dispatch ID* (or DISPID) of the method that was called. Note that in the OLE/COM Object Viewer, this ID number is presented as a *hexadecimal* number. Other applications (Microsoft Visual Studio among them) may display the decimal representation.

The EVENT_NAME field contains the name of the method that was called.

The *Param name* fields contain the values of parameters returned by the called method. The actual parameter name or names displayed, if any, depend on the method being called by the ActiveX control.

Using the ActiveX Widget Event Structure

Since the widget event structure generated by an ActiveX control depends on the method that generated the event, it is important to check the type of event before processing values in IDL. Successfully parsing the event structure requires a detailed understanding of the dispatch interface of the ActiveX control; you must know either the DISPID or the method name of the method, and you must know the names and data types of the values returned.

For example, suppose the ActiveX control you are incorporating into your IDL application includes two methods named `Method1` and `Method2` in a dispatch interface that looks like this:

```
dispinterface MyDispInterface {
    properties:
    methods:
        [id(0x00000270)]
        void Method1([in] EventInfo* EventInfo);
        [id(0x00000272)]
        HRESULT Method2([out, retval] BSTR* EditData);
};
```

A widget event generated by a call to `Method1`, which has no return values, would look something like:

```
** Structure <3fb7288>, 5 tags, length=32, data length=32:
ID                LONG                13
TOP               LONG                12
HANDLER           LONG                12
DISPID            LONG                624
EVENT_NAME        STRING              'Method1'
```

Note that the `DISPID` is 624, the decimal equivalent of 270 hexadecimal.

A widget event generated by a call to `Method2`, which has one return value, would look something like:

```
** Structure <3fb7288>, 6 tags, length=32, data length=32:
ID                LONG                13
TOP               LONG                12
HANDLER           LONG                12
DISPID            LONG                626
EVENT_NAME        STRING              'Method2'
EDITDATA          STRING              'some text value'
```

An IDL event-handler routine could use the value of the `DISPID` field to check which of these two ActiveX control methods generated the event before attempting to use the value of the `EDITDATA` field:

```
PRO myRoutine_event, event
    IF(event.DISPID eq 626) THEN BEGIN
        PRINT, event.EDITDATA
    ENDIF ELSE BEGIN
        <do something else>
    ENDELSE
END
```

Dynamic Elements in the ActiveX Event Structure

Parameter data included in an event structure generated by an ActiveX control can take the form of an array. If this happens, the array is placed in an IDL pointer, and the pointer, rather than the array itself, is included in the IDL event structure.

Similarly, an ActiveX control may return a reference to another COM object, as described in [“References to Other COM Objects”](#) on page 42, in its event structure.

IDL pointers and objects created in this way are not automatically removed; it is the IDL programmer’s responsibility free them using a routine such as [PTR_FREE](#), [HEAP_FREE](#), or [OBJ_DESTROY](#).

If it is unclear whether the event structure will contain dynamic elements (objects or pointers) it is best to pass the ActiveX event structure to the [HEAP_FREE](#) routine when your event-handler routine has finished with the event. This will ensure that all dynamic portions of the structure are released.

Destroying ActiveX Controls

An ActiveX control incorporated in an IDL widget hierarchy is destroyed when any of the following occurs:

- When the widget hierarchy to which the ActiveX widget belongs is destroyed.
- When a call to `WIDGET_CONTROL, wAx, /DESTROY` is made, where `wAx` is the widget ID of the ActiveX widget.
- When the underlying `IDLcomActiveX` object is destroyed by a call to `OBJ_DESTROY`.

In most cases, cleanup of an application that includes an ActiveX control is not different from an application using only IDL native widgets. However, because it is possible for an ActiveX control to return references to other COM objects to IDL, you must be sure to keep track of all objects created by your application and destroy them as necessary. See [“References to Other COM Objects”](#) on page 42 for details.

In addition, it is possible for the widget event structure generated by an ActiveX control to include IDL pointers or object references. Pointers and object references included in the event structure are not automatically destroyed. See [“Dynamic Elements in the ActiveX Event Structure”](#) on page 59 for more information.

Example: Calendar Control

This example uses an ActiveX control that displays a calendar interface. The control, contained in the file `mscal.ocx`, is installed along with a typical installation of Microsoft Office 97, and may also be present on your system if you have upgraded to a more recent version of Microsoft Office. If the control is *not* present on your system (you'll know the control is not present if the example code does not display a calendar similar to the one shown in [Figure 4-1](#)), you can download a the control as part of a package of sample ActiveX controls included in the file `actxsamp.exe`, discussed in Microsoft Knowledge Base Article 165437.

If you download the control, place the file `mscal.exe` in a known location and execute the file; you will be prompted for a directory in which to place `mscal.ocx`. Open a command prompt window in the directory you chose and register the control as described in [“Registering COM Components on a Windows Machine”](#) on page 23.

The calendar control has the program ID:

```
MSCAL.Calendar.7
```

and the class ID:

```
{8E27C92B-1264-101C-8A2F-040224009C02}
```

Example Code

This example, `ActiveXCal.pro`, is included in the `examples\doc\bridges\COM` subdirectory of the IDL distribution and develops an IDL routine called `ActiveXCal` that illustrates use of the calendar ActiveX control within an IDL widget hierarchy. Run the example procedure by entering `ActiveXCal` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT ActiveXCal.pro`.

1. Create the `ActiveXCal` procedure. (Remember that in the `ActiveXCal.pro` file, this procedure occurs last.)

```
PRO ActiveXCal
```

2. Create a top-level base widget to hold the ActiveX control.

```
wBase = WIDGET_BASE(COLUMN = 1, SCR_XSIZE = 400, $
    TITLE='IDL ActiveX Widget Calendar Control')
```

3. Create base widgets to hold labels for the selected month, day, and year. Set the initial values of the labels.

```

wSubBase = WIDGET_BASE(wBase, /ROW)
wVoid = WIDGET_LABEL(wSubBase, value = 'Month: ')
wMonth = WIDGET_LABEL(wSubBase, value = 'October')
wSubBase = WIDGET_BASE(wBase, /ROW)
wVoid = WIDGET_LABEL(wSubBase, VALUE = 'Day: ')
wDay = WIDGET_LABEL(wSubBase, VALUE = '22')
wSubBase = WIDGET_BASE(wBase, /ROW)
wVoid = WIDGET_LABEL(wSubBase, VALUE = 'Year: ')
wYear = WIDGET_LABEL(wSubBase, VALUE = '1999')

```

4. Instantiate the ActiveX Control, using the control's class ID.

```

wAx=WIDGET_ACTIVEX(wBase, $
    '{8E27C92B-1264-101C-8A2F-040224009C02}')

```

5. Realize the top-level base widget.

```

WIDGET_CONTROL, wBase, /REALIZE

```

6. Set the top-level base's user value to an anonymous structure containing widget IDs of the month, day, and year label widgets.

```

WIDGET_CONTROL, wBase, $
    SET_UVALUE = {month:wMonth, day:wDay, year:wYear}

```

7. Retrieve the object ID of the IDLcomActiveX object that encapsulates the ActiveX control. Use the GetProperty method to retrieve the current values of the month, day, and year from the control.

```

WIDGET_CONTROL, wAx, GET_VALUE = oAx
oAx->GetProperty, month=month, day=day, year=year

```

8. Set the values of the label widgets to reflect the current date, as reported by the ActiveX control.

```

WIDGET_CONTROL, wMonth, SET_VALUE=STRTRIM(month, 2)
WIDGET_CONTROL, wDay, SET_VALUE=STRTRIM(day, 2)
WIDGET_CONTROL, wYear, SET_VALUE=STRTRIM(year, 2)

```

9. Call XMANAGER to manage the widget events, and end the procedure.

```

XMANAGER, 'ActiveXCal', wBase

```

```

END

```

10. Now create an event-handling routine for the calendar control. (Remember that in the ActiveXCal.pro file, this procedure occurs before the ActiveXCal procedure.)

```

PRO ActiveXCal_event, ev

```

11. The ActiveX widget is the only widget in this application that generates widget events, so the ID field of the event structure is guaranteed to contain the widget

ID of that widget. Use the GET_VALUE keyword to retrieve an object reference to the IDLcomActiveX object that encapsulates the control.

```
WIDGET_CONTROL, ev.ID, GET_VALUE = oCal
```

12. The user value of the top-level base widget is an anonymous structure that holds the widget IDs of the month, day, and year label widgets (see step 6 above). Retrieve the structure into a variable named `state`.

```
WIDGET_CONTROL, ev.TOP, GET_UVALUE = state
```

13. Use the `GetProperty` method on the IDLcomActiveX object to retrieve the current values of the month, day, and year from the calendar control.

```
ocal->GetProperty, month=month, day=day, year=year
```

14. Use `WIDGET_CONTROL` to set the values of the month, day, and year label widgets.

```
WIDGET_CONTROL, state.month, SET_VALUE = STRTRIM(month,2)
WIDGET_CONTROL, state.day, SET_VALUE = STRTRIM(day,2)
WIDGET_CONTROL, state.year, SET_VALUE = STRTRIM(year,2)
```

15. Call `HEAP_FREE` to ensure that dynamic portions of the event structure are released, and end the procedure.

```
HEAP_FREE, ev
```

```
END
```

Running the ActiveXCal procedure displays a widget that looks like the following:

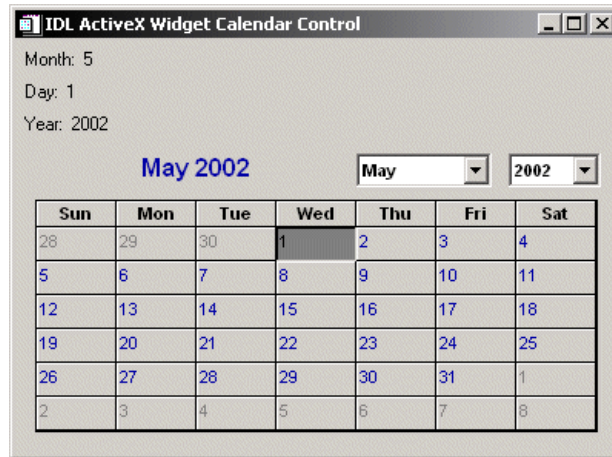


Figure 4-1: An IDL widget program Using an ActiveX Calendar Control

Example: Spreadsheet Control

This example uses an ActiveX control that displays a spreadsheet interface. The control, contained in the file `msowc.dll`, is installed along with a typical installation of Microsoft Office. If the control is *not* present on your system (you'll know the control is not present if the example code fails when trying to realize the widget hierarchy), the example will not run.

The spreadsheet control has the program ID:

```
OWC.Spreadsheet.9
```

and the class ID:

```
{0002E510-0000-0000-C000-000000000046}
```

Note

The spreadsheet control used in this example is included with older versions of Microsoft Office; it is discussed in Microsoft Knowledge Base Article 248822. Newer versions of Microsoft Office may include spreadsheet controls with updated program and class IDs.

Information about the spreadsheet control's properties and methods was gleaned from *Microsoft Excel 97 Visual Basic Step by Step* by Reed Jacobson (Microsoft Press, 1997) and by inspection of the control's interface using the *OLE/COM Object Viewer* application provided by Microsoft. It is beyond the scope of this manual to describe the spreadsheet control's interface in detail.

Example Code

This example, `ActiveXExcel.pro`, is included in the `examples\doc\bridges\COM` subdirectory of the IDL distribution and develops an IDL routine called `ActiveXExcel` that illustrates use of the spreadsheet ActiveX control within an IDL widget hierarchy. Run the example procedure by entering `ActiveXExcel` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT ActiveXExcel.pro`.

1. Create a function that will retrieve data from cells selected in the spreadsheet control. The function takes two arguments: an object reference to the `IDLcomActiveX` object that instantiates the spreadsheet control, and a variable to contain the data from the selected cells.

```
FUNCTION excel_getSelection, oExcel, aData
```

2. Retrieve an object that represents the selected cells. Note that when the ActiveX control returns this object, IDL automatically creates an IDLcomActiveX object that makes it accessible within IDL.

```
oExcel->GetProperty, SELECTION=oSel
```

3. Retrieve the total number of cells selected.

```
oSel->GetProperty, COUNT=nCells
```

4. If no cells are selected, destroy the selection object and return zero (the failure code).

```
IF (nCells LT 1) THEN BEGIN
    OBJ_DESTROY, oSel
    RETURN, 0
ENDIF
```

5. Retrieve objects that represent the dimensions of the selection.

```
oSel->GetProperty, COLUMNS=oCols, ROWS=oRows
```

6. Get the dimensions of the selection, then destroy the column and row objects.

```
oCols->GetProperty, COUNT=nCols
OBJ_DESTROY, oCols
oRows->GetProperty, COUNT=nRows
OBJ_DESTROY, oRows
```

7. Create a floating point array with the same dimensions as the selection.

```
aData = FLTARR(nCols, nRows, /NOZERO);
```

8. Iterate through the cells, doing the following:

- Retrieve an object that represents the cell. Note that the numeric index of the FOR loop is passed to the GetProperty method as an argument.
- Get the value contained in the cell.
- Set the appropriate element of the aData array to the cell's value.
- Destroy the object.

```
FOR i=1, nCells DO BEGIN
    oSel->GetProperty, ITEM=oItem, i
    oItem->GetProperty, VALUE=vValue
    aData[i-1] = vValue
    OBJ_DESTROY, oItem
ENDFOR
```

9. Destroy the selection object.

```
OBJ_DESTROY, oSel
```

10. Return one (the success code) and end the function definition.

```
RETURN, 1
```

```
END
```

11. Next, create a procedure that sets the values of the cells in the spreadsheet. This procedure takes one argument: an object reference to the IDLcomActiveX object that instantiates the spreadsheet control.

```
PRO excel_setData, oExcel
```

12. Define the size of the data array.

```
nX = 20
```

13. Get an object representing the active spreadsheet.

```
oExcel->GetProperty, ActiveSheet=oSheet
```

14. Get an object representing the cells in the spreadsheet.

```
oSheet->GetProperty, CELLS=oCells
```

15. Generate some data.

```
im = BESELJ(DIST(nX))
```

16. Iterate through the elements of the data array, doing the following:

- Retrieve an object that represents the cell that corresponds to the data element. Note that the numeric indices of the FOR loops are passed to the GetProperty method as arguments.
- Set the value of the cell.
- Destroy the object.

```
FOR i=0, nX-1 DO BEGIN
  FOR j=0, nX-1 DO BEGIN
    oCells->GetProperty, ITEM=oItem, i+1, j+1
    oItem->SetProperty, VALUE=im(i,j)
    OBJ_DESTROY, oItem
  ENDFOR
ENDFOR
```

17. Destroy the spreadsheet and cell objects, and end the procedure.

```
OBJ_DESTROY, oSheet
OBJ_DESTROY, oCells
```

```
END
```

18. Next, create a procedure to handle events generated by the widget application.

```
PRO ActiveXExcel_event, ev
```

19. The user value of the top-level base widget is set equal to a structure that contains the widget ID of the ActiveX widget. Retrieve the structure into the variable sState.

```
WIDGET_CONTROL, ev.TOP, GET_UVALUE=sState, /NO_COPY
```

20. Use the value of the DISPID field of the event structure to sort out “selection changing” events.

```
IF (ev.DISPID EQ 1513) THEN BEGIN
```

21. Place data from selected cells in variable aData, using the excel_getSelection function defined above. Check to make sure that the function returns a success value (one) before proceeding.

```
IF (excel_getSelection(sState.oExcel, aData) NE 0) THEN BEGIN
```

22. Get the dimensions of the aData variable.

```
szData = SIZE(aData)
```

23. If aData is two-dimensional, display a surface, otherwise, plot the data.

```
IF (szData[0] GT 1 AND szData[1] GT 1 AND szData[2] GT 1) $
  THEN SURFACE, aData $
ELSE $
  PLOT, aData
ENDIF
```

```
ENDIF
```

24. Reset the state variable sState and end the procedure.

```
WIDGET_CONTROL, ev.TOP, SET_UVALUE=sState, /NO_COPY
```

```
END
```

25. Create the main widget creation routine.

```
PRO ActiveXExcel
```

```
!EXCEPT=0 ; Ignore floating-point underflow errors.
```

26. Create a top-level base widget.

```
wBase = WIDGET_BASE(COLUMN=1, $
  TITLE="IDL ActiveX Spreadsheet Example")
```

27. Instantiate the ActiveX spreadsheet control in a widget.

```
wAx=WIDGET_ACTIVEX(wBase, $
```

```
{0002E510-0000-0000-C000-000000000046}', $
SCR_XSIZE=600, SCR_YSIZE=400)
```

28. Realize the widget hierarchy.

```
WIDGET_CONTROL, wBase, /REALIZE
```

29. The value of an ActiveX widget is an object reference to the IDLcomActiveX object that encapsulates the ActiveX control. Retrieve the object reference in the variable oExcel.

```
WIDGET_CONTROL, wAx, GET_VALUE=oExcel
```

30. Turn off the TitleBar property on the spreadsheet control.

```
oExcel->SetProperty, DisplayTitleBar=0
```

31. Populate the spreadsheet control with data, using the excel_setData function defined above.

```
excel_setData, oExcel
```

32. Set the user value of the top-level base widget to an anonymous structure that contains the widget ID of the spreadsheet ActiveX widget.

```
WIDGET_CONTROL, wBase, SET_UVALUE={oExcel:oExcel}
```

33. Call XMANAGER to manage the widgets, and end the procedure.

```
XMANAGER, 'ActiveXExcel', wBase, /NO_BLOCK
END
```

Running the ActiveXExcel procedure display widgets that look like the following:

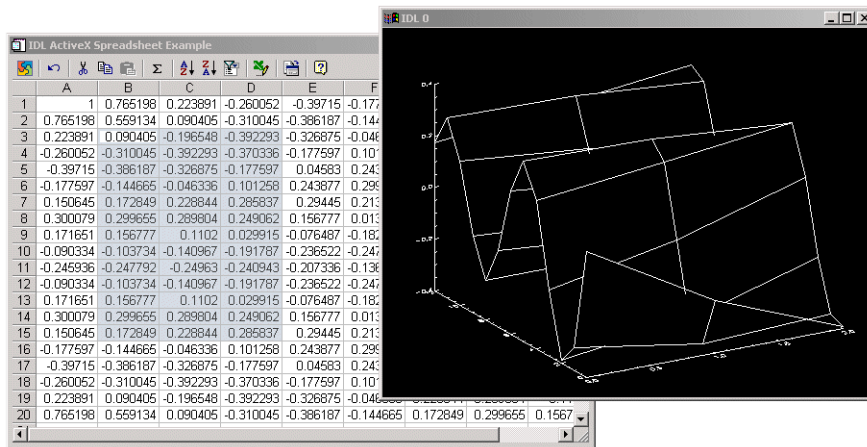


Figure 4-2: An IDL Widget Program Using an ActiveX Spreadsheet Control



Chapter 5

Using Java Objects in IDL

The following topics are covered in this chapter:

Overview of Using Java Objects	72	Destroying IDL-Java Objects	91
Initializing the IDL-Java Bridge	75	Showing IDL-Java Output in IDL	92
IDL-Java Bridge Data Type Mapping	78	The IDLJavaBridgeSession Object	94
Creating IDL-Java Objects	84	Java Exceptions	96
Method Calls on IDL-Java Objects	87	IDL-Java Bridge Examples	99
Managing IDL-Java Object Properties	89	Troubleshooting Your Bridge Session ...	118

Overview of Using Java Objects

Java is an object-oriented programming language developed by Sun Microsystems that is commonly used for web development and other programming needs. It is beyond the scope of this chapter to describe Java in detail. Numerous third-party books and electronic resources are available. The Java website (<http://java.sun.com>) may be useful.

The IDL-Java bridge allows you to access Java objects within IDL code. Java objects imported into IDL behave like normal IDL objects. See “[Creating IDL-Java Objects](#)” on page 84 for more information. The IDL-Java bridge allows the arrow operator (→) to be used to call the methods of these Java objects just as with other IDL objects, see “[Method Calls on IDL-Java Objects](#)” on page 87 for more information. The public data members of a Java object are accessed through `GetProperty` and `SetProperty` methods, see “[Managing IDL-Java Object Properties](#)” on page 89 for more information. These objects can also be destroyed with the `OBJ_DESTROY` routine, see “[Destroying IDL-Java Objects](#)” on page 91 for more information.

Note

IDL requires an evaluation or permanent IDL license to use this functionality. This functionality is not available in demo mode.

The bridge also provides IDL with access to exceptions created by the underlying Java object. This access is provided by the `IDLJavaBridgeSession` object, which is a Java object that maintains exceptions (errors) during a Java session, see “[The IDLJavaBridgeSession Object](#)” on page 94 for more information.

Note

Visual Java objects cannot be embedded into IDL widgets.

Note

On Solaris, there are potential problems creating graphical windows from the IDL-Java bridge using Java versions before 1.5. We recommend using the `XToolkit` option, which the IDL-Java bridge will use by default.

Java Runtime Environment Requirements

IDL supports version 1.5 and greater of the Java runtime environment.

Note

On Macintosh machines, the version of Java installed along with the operating system should be sufficient, whatever its version number.

Java Terminology

You should become familiar with the following terms before trying to understand how IDL works with Java objects:

Java Virtual Machine (JVM) - A software execution engine for executing the byte codes in Java class files on a microprocessor.

Java Native Interface (JNI) - Standard programming interface for accessing Java native methods and embedding the JVM into native applications. For example, JNI may be used to call C/C++ functionality from Java or JNI can be used to call Java from C/C++ programs.

Java Invocation API - An API by which one embeds the Java Virtual Machine into your native application by linking the native application with the JVM shared library.

Java Reflection API - Provides a small, type-safe, and secure API that supports introspection about the classes and objects. The API can be used to:

- Construct new class instances and new arrays
- Access and modify fields of objects and classes
- Invoke methods on objects and classes
- Access and modify elements of arrays

IDL-Java Bridge Architecture

The IDL-Java bridge uses the Java Native Interface (JNI), the reflection API, and the JVM to enable the connection between IDL and the underlying Java system.

The IDL OBJ_NEW function can be used to create a Java object. A Java-specific class token identifies the Java class used to create a Java proxy object. IDL parses this class name and creates the desired object within the underlying Java environment.

The Java-specific token is a case-insensitive form of the name of the Java class. Besides the token, the case-sensitive form of the name of the Java class is provided because Java itself is case-sensitive while IDL is not. IDL uses the case-insensitive form to create the object definition while Java uses the case-sensitive form.

After creation, the object can then be used and manipulated just like any other IDL object. Method calls are the same as any other IDL object, but they are vectored off to an IDL Java system, which will call the appropriate Java method using JNI.

The OBJ_DESTROY procedure in IDL is used to destroy the object. This process releases the internal Java object and frees any resources associated with it.

Initializing the IDL-Java Bridge

The IDL-Java bridge must be configured before trying to create and use Java objects within IDL. The IDL program initializes the bridge when it first attempts to create an instance of `IDLjavaObject`. Initializing the bridge involves starting the Java Virtual Machine, creating any internal Java bridge objects (both C++ and Java) including the internal `IDLJavaBridgeSession` object. See “[The IDLJavaBridgeSession Object](#)” on page 94 for more information on the session object.

Configuring the Bridge

The `.idljavabrc` file on UNIX or `idljavabrc` on Windows contains the IDL-Java bridge configuration information. Even though the IDL installer attempts to create a valid working configuration file based on IDL location, the file should be verified before trying to create and use Java objects within IDL.

The IDL-Java bridge looks for the configuration file in the following order:

1. If the environment variable `IDLJAVAB_CONFIG` is set, the file it indicates is used.

Note

This environment variable must include both the path *and* the file name of the configuration file.

2. If the environment variable `IDLJAVAB_CONFIG` is not set or the file indicated by that variable is not found in that location, the path specified in the `HOME` environment variable is used to try to locate the configuration file.
3. If the file is not found in the path indicated by the `HOME` environment variable, the `<IDL_DEFAULT>/resource/bridges/import/java` path is used to try to locate the configuration file.

The configuration file contains the following settings. With a text editor, open your configuration file to verify these settings are correct for your system.

- The `JVM Classpath` setting specifies additional locations for user classes. It must point to the location of any class files to be used by the bridge. On Windows, paths should be separated by semi-colons. On UNIX, colons should separate paths.

This path may contain folders that contain class files or specific jar files. It follows the same rules for specifying `'-classpath'` when running `java` or

javac. You can also include the CLASSPATH environment variable in the JVM Classpath:

```
JVM Classpath = $CLASSPATH:/home/johnd/myClasses.jar
```

which allows any class defined in the CLASSPATH environment variable to be used in the IDL-Java bridge.

On Windows, an example of a typical JVM Classpath setting is:

```
JVM Classpath = E:\myClasses.jar;$CLASSPATH
```

On UNIX, an example of a typical JVM Classpath setting is:

```
JVM Classpath = /home/johnd/myClasses.jar:$CLASSPATH
```

- The JVM LibLocation setting tells the Windows IDL-Java bridge which JVM shared library within a given Java version to use. Various versions of Java ship with different types of JVM libraries. For example, Java 1.3 on Windows ships with a “classic” JVM, a “hotspot” JVM, and a “server” JVM. Other versions and platforms have different JVM types.

On Windows, an example of a typical JVM LibLocation setting is:

```
JVM LibLocation = E:\jdk1.3.1_02\jre\bin\hotspot
```

On UNIX, you should not set JVM LibLocation in the configuration file. Instead, set the IDLJAVAB_LIB_LOCATION environment variable for the session that will use the IDL-Java bridge. The following is a typical command to set the environment variable:

```
setenv IDLJAVAB_LIB_LOCATION  
/usr/java/j2re1.4.0_02/lib/sparc/client
```

Note

You can also set the IDLJAVAB_LIB_LOCATION environment variable on Windows platforms, rather than specifying the value in the configuration file.

Note

On Macintosh platforms, IDL is hard-coded to use the Java VM 1.3.1, and so the system ignores any value you place in IDLJAVAB_LIB_LOCATION.

- The JVM Option# (where # is any whole number) setting allows you to send additional parameters to the Java Virtual machine upon initialization. These settings must be specified as string values. When these settings are encountered in the initialization, the options are added to the end of the options that the bridge sets by default.

- The `Log Location` setting indicates the directory where IDL-Java bridge log files will be created. The default location provided by the IDL installer is `/tmp` on Unix and `C:\temp` on Windows.
- The `Bridge Logging` setting indicates the type of bridge debug logging to be sent to a file called `jb_log<pid>.txt` (where `<pid>` is a process ID number) located in the directory specified by the `Log Location` setting.

Acceptable values (from least verbose to most verbose) are `SEVERE`, `CONFIG`, `CONFIGFINE`. The default value is `SEVERE`, which specifies that bridge errors are logged. The `CONFIG` value indicates the configuration settings are also logged. The `CONFIGFINE` value is the same as `CONFIG`, but provides more detail.

No log file is created if this setting is set to `OFF`.

The IDL-Java bridge usually only uses the configuration file once during an IDL session. The file is used when the first instance of the `IDLjavaObject` class is created in the session. If you edit the configuration file after the first instance is created, you must exit and restart IDL to update the IDL-Java bridge with the changes you made to the file.

IDL-Java Bridge Data Type Mapping

When data moves between IDL and a Java object, IDL automatically converts variable data types.

The following table maps how Java data types correlate to IDL data types.

Java Type (# bytes)	IDL Type	Notes
boolean (1)	Integer	True becomes 1, false becomes 0
byte (1)	Byte	
char (2)	Byte	The bridge handles Java UTF characters
short (2)	Integer	
int (4)	Long	
long (8)	Long64	
float (4)	Float	
double (8)	Double	
Java.lang.String	String	Java has the notion of a NULL string (the java.lang.String reference equals null) and the concept of an empty string. IDL makes no such differentiation, so both are identically converted.
Arrays of the above types	IDL array of the same dimensions (from 1 to 8 dimensions) and corresponding type.	

Table 5-1: Java to IDL Data Type Conversion

Java Type (# bytes)	IDL Type	Notes
Java.lang.Object (or array of java.lang.Object) and any subclass of java.lang.Object	IDL array of primitives or IDL array of IDLjavaObjects	In Java, everything is a subclass of Object. If the Java object is an array of primitives, an IDL array of the same dimensions and corresponding type (shown in this table) is created. IDL similarly converts arrays of primitives, arrays of strings, arrays of other Java objects to an IDL Java object of the same dimensions. If the Object is some single Java object, IDL creates an object reference of the IDLjavaObject class.
Null object	IDL Null object	

Table 5-1: Java to IDL Data Type Conversion (Continued)

The following table shows how data types are mapped from IDL to Java.

IDL Type	Java Type (# bytes)	Notes
Byte	byte (1)	IDL bytes range from 0 to 255, Java bytes are -128 to 127. IDL bytes converted to Java bytes will retain their binary representation but values greater than 127 will change. For example, BYTE(255) becomes a Java byte of -1. If BYTE is converted to wider Java value, the sign and value is preserved.
Integer	short (2)	
Unsigned integer	short (2)	IDL unsigned integers range from 0 to 65535, Java shorts are -32768 to 32767. IDL unsigned integers converted to Java shorts will retain their binary representation but values greater than 32768 will change. For example, UINT(65535) becomes a Java short of -1. If UINT is converted to wider Java value, the sign and value is preserved.
Long	int (4)	

Table 5-2: IDL to Java Data Type Conversion

IDL Type	Java Type (# bytes)	Notes
Unsigned long	int (4)	IDL unsigned longs range from 0 to 4294967295, Java ints are -2147483648 to 2147483647. IDL unsigned longs converted to Java ints will retain their binary representation but values greater than 2147483647 will change. For example, ULONG(4294967295) becomes a Java int of -1. If ULONG is converted to wider Java value, the sign and value is preserved.
Long64	long (8)	
Unsigned Long64	long (8)	IDL unsigned long64 range from 0 to 18446744073709551615, Java ints range from -9223372036854775808 to 9223372036854775807. IDL unsigned long64 converted to Java longs will retain their binary representation values greater than 9223372036854775807 will change. For example, ULONG64(18446744073709551615) becomes a Java long of -1.
Float	float (4)	
Double	double (8)	
String	Java.lang.String	
Arrays of the above types	Java array of the same dimensions and corresponding type	

Table 5-2: IDL to Java Data Type Conversion (Continued)

IDL Type	Java Type (# bytes)	Notes
IDLjavaObject	Object of corresponding Java class	
Arrays of objects	Java array of the same dimensions, consisting of corresponding Java proxy objects	Only objects of type IDLjavaObject are converted.
Null object	Java null	

Table 5-2: IDL to Java Data Type Conversion (Continued)

When calling a Java method or constructor from IDL, the data parameters are promoted as little as possible based on the signature of the given method. The following table shows how data types are promoted within Java relative to IDL.

Note

When strings and arrays are passed between IDL and Java, the array must be copied. Depending upon the size of the array, this copy may be time intensive. Care should be taken to minimize array copying.

IDL Type	Java Type (to order of desired promotion)	Notes
Byte	byte, char, short, int, long, float, double, boolean	
Integer	short, int, long, float, double, boolean	
Unsigned integer	short, int, long, float, double, boolean	
Long	int, long, float, double, boolean	
Unsigned Long	int, long, float, double, boolean	
Long64	long, float, double, boolean	
Unsigned Long64	long, float, double, boolean	

Table 5-3: Java Data Type Promotion Relative to IDL

IDL Type	Java Type (to order of desired promotion)	Notes
Float	float, double	
Double	double	
String	Java.lang.String	
IDLjavaObject	Java.lang.Object	

Table 5-3: Java Data Type Promotion Relative to IDL (Continued)

Creating IDL-Java Objects

As with all IDL objects, a Java object is created using the IDL `OBJ_NEW` function. Keying off the provided Java class name, the underlying implementation uses the IDL Java subsystem to call the constructor on the desired Java object. The following line of code demonstrates the basic syntax for calling `OBJ_NEW` to create a Java object within IDL:

```
oJava = OBJ_NEW(IDLjavaObject$JAVACLASSNAME, JavaClassName $
    [, Arg1, Arg2, ..., ArgN])
```

where *JAVACLASSNAME* is the class name token used by IDL to create the object, *JavaClassName* is the class name used by Java to initialize the object, and *Arg1* through *ArgN* are any data parameters required by the constructor. See [“Java Class Names in IDL”](#) for more information.

Example Code

The example `hellojava.pro` is located in the `resource/bridges/import/java/examples` directory of the IDL distribution and shows a simple example of an IDL-Java object creation. Run the example procedure by entering `hellojava` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT hellojava.pro`.

Note

If you edit and recompile a Java class used by IDL during an IDL-Java bridge session, you must first exit and restart IDL before your modified Java class will be recognized by IDL.

The IDL-Java bridge also provides the ability to access static Java methods and data members. See [“Java Static Access”](#) on page 85 for more information.

Java Class Names in IDL

The underlying Java interpreter recognizes the Java class name including all objects contained within the Java interpreter’s class path.

To identify a proper Java object, the fully-qualified package name should be used when creating the IDL class name. For example, a class of type `String` would be referred to as `java.lang.String`.

In the IDL class name, the Java class separator (`'.'`) should be replaced with an underscore (`'_'`). If a Java class of type `String` were created, the following IDL `OBJ_NEW` call would be used:

```
oJString = OBJ_NEW('IDLJavaObject$JAVA_LANG_STRING', $
    'java.lang.String', 'My String')
```

The class name is provided twice because IDL is case-insensitive whereas Java is case-sensitive, see [“IDL-Java Bridge Architecture”](#) on page 73 for more information.

Note

IDL objects use method names (INIT and CLEANUP) to identify and call object lifecycle methods. As such, these method names should be considered reserved. If an underlying Java object implements a method using either INIT or CLEANUP, those methods will be overridden by the IDL methods and not accessible from IDL. In Java, you can wrap these methods with different named methods to work around this limitation.

Java Static Access

In Java, a program can call a static method or access static data members on a Java class without first having to create the object.

IDL contains a special wrapper object type for calling static methods. This IDL object wrapper references the underlying Java class, allowing the object to call static methods on the class or allowing the object to use the Get/Set Property calls to access static data members. The following line of code demonstrates the basic syntax for calling OBJ_NEW to create a static proxy within IDL:

```
oJava = OBJ_NEW(IDLjavaObject$Static$JAVACLASSNAME, JavaClassName)
```

where *JAVACLASNAME* is the class name token used by IDL to create the object and *JavaClassName* is the class name used by Java to initialize the object. See [“Java Class Names in IDL”](#) on page 84 for more information.

A special static object would not need to be created to call an instantiated IDLJavaObject with static methods:

```
oNotStatic = OBJ_NEW('IDLjavaObject$JAVACLASSNAME', $
    'JavaClassName')
oNotStatic -> aStaticMethod ; this is OK
```

Example Code

The `javaprops.pro` file is located in the `resource/bridges/import/java/examples` directory of the IDL distribution and shows an example of working with static data members. Run the example procedure by entering `javaprops` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT javaprops.pro`.

Note

All restrictions on creating Java objects apply to this static object.

Method Calls on IDL-Java Objects

When a method is called on a Java-based IDL object, the method name and arguments are passed to the IDL-Java subsystem and the Java Reflection API to construct and invoke the method call on the underlying object.

IDL handles conversion between IDL and Java data types. Any results are returned in IDL variables of the appropriate type.

As with all IDL objects, the general syntax in IDL for an underlying Java method that returns a value (known as a function method in IDL) is:

```
result = ObjRef->Method([Arguments])
```

and the general syntax in IDL for an underlying Java method that does not return a value, a void method, (known as a procedure method in IDL) is:

```
ObjRef->Method[, Arguments]
```

where `ObjRef` is an object reference to an instance of a dynamic subclass of the `IDLjavaObject` class.

Note

Besides other Java based objects, the value of an argument may be an IDL primitive type, an `IDLjavaObject`, or an IDL primitive type array. No complex types (structures, pointers, etc.) are supported as parameters to method calls.

What Happens When a Method Call Is Made?

When a method is called on an instance of `IDLjavaObject`, IDL uses the method name and arguments to construct the appropriate method calls for the underlying Java object.

From the point of view of an IDL user issuing method calls on an instance of `IDLjavaObject`, this process is completely transparent. IDL handles the translation when the IDL user calls the Java object's method.

Due to case-sensitivity incompatibilities between IDL and Java, Java's ability to overload methods, and the fact that Java might promote certain data types, the Java bridge uses an algorithm to match the IDL method name and parameters to the corresponding Java object method.

Before the algorithm starts, IDL provides a case-insensitive `<METHODNAME>` and a reference to the Java object. For a given object and its parent classes, the Java bridge obtains a list of all the public method names, including static methods. This algorithm performs the following steps:

1. If the Java class has one method name matching the IDL `<METHODNAME>` (except for case insensitivity), this Java method name is used. At this point, signatures and overloaded functions are not taken into account.
2. If the Java class has several method names that differ only in case and one is all uppercase, the uppercase name is used. Otherwise, the IDL-Java bridge issues an error that it has no method named `<METHODNAME>`.
3. Once the method name has been determined, a promotion algorithm then matches the Java data parameters as closely as possible with the IDL parameters. Minimum data promotion from IDL to Java is preferred and only widening promotion is allowed. If no match is found, an error is issued.

Data Type Conversions

IDL and Java use different data types. IDL's dynamic type conversion facilities handle all conversion of data types between IDL and the Java system. The data type mappings are described in [“IDL-Java Bridge Data Type Mapping”](#) on page 78.

For example, if the Java object has a method that requires a value of type `int` as an input argument, IDL would supply the value as an IDL Long. For any other IDL data type, IDL would first convert the value to an IDL Long using its normal data type conversion mechanism before passing the value to the Java object as an `int`.

Managing IDL-Java Object Properties

Property names and arguments are also passed to the IDL Java subsystem and are used in conjunction with the Java Reflection API to construct and access public data members on the underlying object. These public data members (known as properties in IDL) are identified through arguments to the `GetProperty` and `SetProperty` methods. See “[Getting and Setting Properties](#)” on page 90 for more information.

Note

Only public data members may be accessed.

Due to case-sensitivity incompatibilities between IDL and Java and the fact that Java might promote certain data types, the Java bridge uses an algorithm to match the IDL properties name to the corresponding Java object data members.

Before the algorithm starts, IDL provides a case-insensitive `<PROPERTYNAME>` and a reference to the Java object. For the given object and its parent classes, the Java bridge obtains a list of all the public data members including static members. This algorithm performs the following steps:

1. If the Java class has one data member name matching the IDL `<PROPERTYNAME>` (except for case insensitivity), this Java data member is used. At this point, data types are not yet taken into account; this algorithm only matches the data member names.
2. If the Java class has several member names that differ only in case, the data member name that exactly matches the IDL `<PROPERTYNAME>` (i.e. the one that is all caps) is called. Otherwise, the IDL-Java bridge issues an error that the class has no data members named `< PROPERTYNAME >`.
3. When setting a property with the `SetProperty` method, a promotion algorithm matches the provided IDL parameter with the Java data parameter as closely as possible. If the IDL value can be promoted to the same type as the data member, this data member is used. Otherwise, an error is issued.

When retrieving a property with the `GetProperty` method, this step is skipped and the value is returned to IDL.

Example Code

The `allprops.pro` and `publicmembers.pro` files in the `resource/bridges/import/java/examples` directory of the IDL distribution provide information about data members associated with given Java classes. Run the example procedures by entering `allprops` and `publicmembers` at the IDL

command prompt or view the files in an IDL Editor window by entering `.EDIT allprops.pro` or `.EDIT publicmembers.pro`.

Getting and Setting Properties

The IDL-Java bridge follows the standard IDL property interface to support data member access on Java objects and classes.

To retrieve a property value from a Java object, use the following syntax:

```
ObjRef->GetProperty, PROPERTY=variable
```

where *ObjRef* is an instance of `IDLjavaObject` that encapsulates the Java object, *PROPERTY* is the name of the Java object's data member (property), and *variable* is the name of an IDL variable that will contain the retrieved property value.

To retrieve multiple property values in a single statement supply multiple *PROPERTY=variable* pairs separated by commas.

To set a property value on a Java object, use the following syntax:

```
ObjRef->SetProperty, Property=value
```

where *ObjRef* is an instance of `IDLjavaObject` that encapsulates the Java object, *PROPERTY* is the name of the Java object's data member, and *value* is value of the property to be set.

To set multiple property values in a single statement supply multiple *PROPERTY=value* pairs separated by commas.

Note

The provided *PROPERTY* must map directly to a data member name. Any name passed into either of the property routines is assumed to be a fully qualified Java property name. As such, the partial property name functionality provided by IDL is not valid with IDL Java based objects.

The *variable* or *value* part may be an IDL primitive type, an instance of `IDLJavaObject`, or an array of an IDL primitive type. See [“IDL-Java Bridge Data Type Mapping”](#) on page 78 for more information.

Note

Besides other Java-based objects, no complex types (structures, pointers, etc.) are supported as parameters to property calls.

Destroying IDL-Java Objects

The OBJ_DESTROY routine is used to destroy instances of IDLjavaObject. When OBJ_DESTROY is called with a Java-based object as an argument, IDL releases the underlying Java object and frees IDL resources relating to that object.

Note

Destruction of the IDL object does not automatically cause the destruction of the underlying Java object. Because Java utilizes a garbage collection mechanism to release any information allocated for a particular object, the resources utilized by the underlying Java object will persist until the Java virtual machine's garbage collector runs.

Showing IDL-Java Output in IDL

By default, IDL prints the output from Java (the `System.out` and `System.err` output streams).

For example, given the following Java code:

```
public class helloWorld
{
    // ctor
    public helloWorld() {
        System.out.println("helloWorld ctor");
    }

    public void sayHello() {
        System.out.println("Hello! (from the helloWorld object)");
    }
}
```

The following output occurs in IDL:

```
IDL> oJHello = OBJ_NEW('IDLjavaObject$HelloWorld', 'helloWorld')
% helloWorld ctor
IDL> oJHello -> SayHello
% Hello! (from the helloWorld object)
IDL> OBJ_DESTROY, oJHello
```

Example Code

This example code is also provided in the `helloJava.java` and `hellojava2.pro` files, which are in the `resource/bridges/import/java/examples` directory of the IDL distribution. Run these example procedures by entering `helloJava` and `hellojava2` at the IDL command prompt or view the files in an IDL Editor window by entering `.EDIT helloJava.pro` and `.EDIT hellojava2.pro`.

Note

Due to restrictions in IDL concerning receiving standard output from non-main threads, the bridge will only send `System.out` and `System.err` information to IDL from the main thread. Other threads' output will be ignored.

Note

A `print()` in Java will not have a carriage return at the end of the line (as opposed to `println()`, which does). However, when outputting to Java both `print()` and `println()` will print to IDL followed by a carriage return. You can change this

result by having the Java-side application buffer its data up into the lines you wish to see on the IDL-side.

The IDLJavaBridgeSession Object

Java exceptions are handled within IDL through an IDL-Java bridge session object, `IDLJavaBridgeSession`. This Java object can be queried to determine the status of the bridge, including information on any exceptions. For example, one important Java object available through the session object is the last issued Java exception.

The session object is a proxy to an internal Java object, which is created during the IDL-Java bridge initialization process. You can connect an `IDLJavaObject` to this object using `OBJ_NEW`:

```
oJSession = OBJ_NEW('IDLjavaObject$IDLJAVABRIDGESESSION')
```

Note

Only one Java session object needs to be created during an IDL session. Subsequent calls to this object will point to the same internal object.

When an exception occurs, the `GetException` function method indicates what exception occurred:

```
oJException = oJSession->GetException()
```

where `oJSession` is a reference to the session object and `oJException` is a proxy object to a `java.lang.Throwable` object, which is the class used in Java to manage exceptions. The session object also has a `ClearException` method that clears the session object's last exception. The `GetException` method always calls `ClearException` method.

The `IDLJavaBridgeSession` object also has the `GetVersionObject` method, which retrieves the `IDLJavaVersion` object:

```
oJVersion = oJSession->GetVersionObject()
```

where `oJSession` is a reference to the session object and `oJVersion` is a proxy object to an `IDLJavaVersion` object. This object determines version information about the IDL-Java bridge and the underlying Java system.

The `IDLJavaVersion` object provides the following function methods, which do not require any arguments:

- `GetBuildDate()` - a `java.lang.String` object specifying the build date. For example, Apr 1 2003.
- `GetJavaVersion()` - a `java.lang.String` object specifying the Java version. For example, 1.3.1_02.

- `GetBridgeVersion()` - a `java.lang.String` object specifying the IDL-Java bridge version.

Example Code

An example of the version object is provided in the `bridge_version.pro` file, which is in IDL's `resource/bridges/import/java/examples` directory. Run the example procedure by entering `bridge_version` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT bridge_version.pro`.

Java Exceptions

During the operation of the bridge, an error may occur when initializing the bridge, creating an `IDLjavaObject`, calling methods, setting properties, or getting properties. Typically, these errors will be fixed by changing your IDL or Java code (or by changing the bridge configuration). Java bridge errors operate like other IDL errors in that they stop execution of IDL and post an error message. These errors can be caught like any other IDL error.

On the other hand, Java uses the exception mechanism to report errors. For example, in Java, if we attempt to create a `java.lang.StringBuffer` of negative length, a `java.lang.NegativeArraySizeException` is issued.

Java exceptions are handled much like bridge errors. They stop IDL execution (if uncaught) and they report an error message containing a line number. In addition, a mechanism is provided to grab the exception object (a subclass of `java.lang.Throwable`) via the session object. Once connected with the exception object, IDL can call any of the methods provided by this Java object. For example, IDL can query the exception name to determine how to handle it, or print a stack trace of where the exception occurred in your Java code.

The exception object is provided through the `GetException` method to the `IDLJavaBridgeSession` object. See [“The IDLJavaBridgeSession Object”](#) on page 94 for more information about this object.

Uncaught Exceptions

If a Java exception is not caught, IDL will stop execution and display an `Exception` thrown error message. For example, when the following program is saved as `ExceptIssued.pro`, compiled, and ran in IDL:

```
PRO ExceptIssued

; This will throw a Java exception
oJStrBuffer = OBJ_NEW($
    'IDLJavaObject$java_lang_StringBuffer', $
    'java.lang.StringBuffer', -2)

END
```

IDL issues the following output:

```
IDL> ExceptIssued
% Exception thrown
% Execution halted at: EXCEPTISSUED 4 ExceptIssues.pro
%                               $MAIN$
```


From the IDL command line, you can then use the session object to help debug the problem:

```
IDL> oJSession = OBJ_NEW('IDLJavaObject$IDLJAVABRIDGESESSION')
IDL> oJExc = oJSession->GetException()
IDL> oJExc->PrintStackTrace
% java.lang.NegativeArraySizeException:
%   at java.lang.StringBuffer.<init>(StringBuffer.java:116)
```

Example Code

A similar example is also provided in the `exception.pro` file, which is in the `resource/bridges/import/java/examples` directory of the IDL distribution. The `exception.pro` example shows how to use the utility routine provided in the `showexcept.pro` file. This `showexcept` utility routine can be re-used to provide consist error messages when Java exceptions occur. The `showexcept.pro` file is also provided in the `resource/bridges/import/java/examples` directory of the IDL distribution. Run the example procedure by entering `exception` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT exception.pro`.

Caught Exceptions

Java exceptions can be caught just like IDL errors. Consult the documentation of the Java classes that you are using to ensure IDL is catching any expected exceptions. For example:

```
PRO ExceptCaught

; Grab the special IDLJavaBridgeSession object
oJBridgeSession = OBJ_NEW('IDLJavaObject$IDLJAVABRIDGESESSION')

bufferSize = -2
; Our Java constructor might throw an exception, so let's catch it
CATCH, error_status
IF (error_status NE 0) THEN BEGIN
    ; Use session object to get our Exception
    oJExc = oJBridgeSession->GetException()
    ; should be of type
    ; IDLJAVAOBJECT$JAVA_LANG_NEGATIVEARRAYSIZEEXCEPTION
    HELP, oJExc
    ; Now we can access the members java.lang.Throwable
    PRINT, 'Exception thrown:', oJExc->ToString()
    oJExc->PrintStackTrace
    ; Cleanup
    OBJ_DESTROY, oJExc
    ; Increase the buffer size to avoid the exception.
```

```
        bufferSize = bufferSize + 100
    ENDIF

    ; This throws a Java exception the 1st time, but pass the 2nd time.
    oJStrBuffer = OBJ_NEW('IDLJavaObject$java_lang_StringBuffer', $
        'java.lang.StringBuffer', bufferSize)

    OBJ_DESTROY, oJStrBuffer
    OBJ_DESTROY, oJBridgeSession

END
```

Example Code

A similar example is also provided in the `exception.pro` file, which is in the `resource/bridges/import/java/examples` directory of the IDL distribution. The `exception.pro` example shows how to use the utility routine provided in the `showexcept.pro` file. This `showexcept` utility routine can be re-used to provide consist error messages when Java exceptions occur. The `showexcept.pro` file is also provided in the `resource/bridges/import/java/examples` directory of the IDL distribution. Run the example procedure by entering `showexcept` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT showexcept.pro`.

IDL-Java Bridge Examples

The following examples demonstrate how to access data through the IDL-Java bridge:

- [“Accessing Arrays Example”](#)
- [“Accessing URLs Example”](#) on page 102
- [“Accessing Grayscale Images Example”](#) on page 104
- [“Accessing RGB Images Example”](#) on page 108

Note

If IDL is not able to find any Java class associated with these examples, make sure your IDL-Java bridge is properly configured. See [“Configuring the Bridge”](#) on page 75 for more information.

Accessing Arrays Example

This example creates a two-dimensional array within a Java class, which is contained in a file named `array2d.java`. IDL then accesses this data through the `ArrayDemo` routine, which is in a file named `arraydemo.pro`.

Example Code

These files are located in the `resource/bridges/import/java/examples` directory of the IDL distribution. Run this example procedure by entering `arraydemo` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT arraydemo.pro`.

The `array2d.java` file contains the following text for creating a two-dimensional array in Java:

```
public class array2d {
    short[][] m_as;
    long[][] m_aj;

    // ctor
    public array2d() {
        int SIZE1 = 3;
        int SIZE2 = 4;

        // default ctor creates a fixed number of elements
        m_as = new short[SIZE1][SIZE2];
    }
}
```

```

        m_aj = new long[SIZE1][SIZE2];

        for (int i=0; i<SIZE1; i++) {
            for (int j=0; j<SIZE2; j++) {
                m_as[i][j] = (short)(i*10+j);
                m_aj[i][j] = (long)(i*10+j);
            }
        }

    }

    public void setShorts(short[][] _as) {
        m_as = _as;
    }

    public short[][] getShorts() {return m_as;}

    public short getShortByIndex(int i, int j) {
        return m_as[i][j];
    }

    public void setLongs(long[][] _aj) {
        m_aj = _aj;
    }

    public long[][] getLongs() {return m_aj;}
    public long getLongByIndex(int i, int j) {return m_aj[i][j];}

}

```

The `arraydemo.pro` file contains the following text for accessing the two-dimensional array within IDL:

```

PRO ArrayDemo

; The Java class array2d creates 2 initial arrays, one
; of longs and one of shorts. We can interrogate and
; change this array.
oJArr = OBJ_NEW('IDLJavaObject$ARRAY2D', 'array2d')

; First, let's see what is in the short array at index
; (2,3).
PRINT, 'array2d short(2, 3) = ', $
    oJArr -> GetShortByIndex(2, 3), $
    '      (should be 23)'

; Now, let's copy the entire array from Java to IDL.
shortArrIDL = oJArr->GetShorts()
HELP, shortArrIDL

```

```

PRINT, 'shortArrIDL[2, 3] = ', shortArrIDL[2, 3], $
      '      (should be 23)'

; Let's change this value...
shortArrIDL[2, 3] = 999
; ...and copy it back to Java...
oJArr->SetShorts, shortArrIDL
; ...now its value should be different.
PRINT, 'array2d short(2, 3) = ', $
      oJArr->GetShortByIndex(2, 3), '      (should be 999)'

; Let's set our array to something different.
oJArr -> SetShorts, INDGEN(10, 8)
PRINT, 'array2d short(0, 0) = ', $
      oJArr->GetShortByIndex(0, 0), '      (should be 0)'
PRINT, 'array2d short(1, 0) = ', $
      oJArr->GetShortByIndex(1, 0), '      (should be 1)'
PRINT, 'array2d short(2, 0) = ', $
      oJArr->GetShortByIndex(2, 0), '      (should be 2)'
PRINT, 'array2d short(0, 1) = ', $
      oJArr->GetShortByIndex(0, 1), '      (should be 10)'

; Array2d has a setLongs method, but b/c arrays do not
; (currently) promote, the first call to setLongs works
; but the second fails.
oJArr->SetLongs, L64INDGEN(10, 8)
PRINT, 'array2d long(0, 1) = ', $
      oJArr->GetLongByIndex(0, 1), '      (should be 10)'

;PRINT, '(expecting an error on the next line...)'
;oJArr->SetLongs, INDGEN(10,8)

; Cleanup our object.
OBJ_DESTROY, oJArr

END

```

After saving and compiling the above files (array2d.java in Java and ArrayDemo.pro in IDL), update the jbxamples.jar file in the resource/bridges/import/java directory with the new compiled class and run the ArrayDemo routine in IDL. The routine should produce the following results:

```

array2d short(2, 3) = 23 (should be 23)
SHORTARRIDL  INT = Array[3, 4]
shortArrIDL[2, 3] = 23 (should be 23)
array2d short(2, 3) = 999 (should be 999)
array2d short(0, 0) = 0 (should be 0)
array2d short(1, 0) = 1 (should be 1)
array2d short(2, 0) = 2 (should be 2)

```

```
array2d short(0, 1) = 10 (should be 10)
array2d long(0, 1) = 10 (should be 10)
```

Accessing URLs Example

This example finds and reads a given URL, which is contained in a file named `URLReader.java`. IDL then accesses this data through the `URLRead` routine, which is in a file named `urlread.pro`.

Example Code

These files are located in the `resource/bridges/import/java/examples` directory of the IDL distribution. Run this example procedure by entering `urlread` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT urlread.pro`.

The `URLReader.java` file contains the following text for reading a given URL in Java:

```
import java.io.*;
import java.net.*;

public class URLReader
{
    private ByteArrayOutputStream m_buffer;

    // *****
    //
    // Constructor. Create the reader
    //
    // *****
    public URLReader() {
        m_buffer = new ByteArrayOutputStream();
    }

    // *****
    //
    // readURL: read the data from the URL into our buffer
    //
    // returns: number of bytes read (0 if invalid URL)
    //
    // NOTE: reading a new URL clears out the previous data
    //
    // *****
    public int readURL(String sURL) {
        URL url;
        InputStream in = null;
```

```

m_buffer.reset(); // reset our holding buffer to 0 bytes

int total_bytes = 0;
byte[] tempBuffer = new byte[4096];
try {
    url = new URL(sURL);
    in = url.openStream();

    int bytes_read;
    while ((bytes_read = in.read(tempBuffer)) != -1) {
        m_buffer.write(tempBuffer, 0, bytes_read);
        total_bytes += bytes_read;
    }
} catch (Exception e) {
    System.err.println("Error reading URL: "+sURL);
    total_bytes = 0;
} finally {
    try {
        in.close();
        m_buffer.close();
    } catch (Exception e) {}
}

return total_bytes;
}

// *****
//
// getData: return the array of bytes
//
// *****
public byte[] getData() {
    return m_buffer.toByteArray();
}

// *****
//
// main: reads URL and reports # of byts reads
//
// Usage: java URLReader <URL>
//
// *****

public static void main(String[] args) {
    if (args.length != 1)
        System.err.println("Usage: URLReader <URL>");
    else {
        URLReader o = new URLReader();
        int b = o.readURL(args[0]);
    }
}

```

```

        System.out.println("bytes="+b);
    }
}
}

```

The `urlread.pro` file contains the following text for inputting an URL as an IDL string and then accessing its data within IDL:

```

FUNCTION URLRead, sURLName

; Create an URLReader.
oJURLReader = OBJ_NEW('IDLjavaObject$URLReader', 'URLReader')

; Read the URL data into our Java-side buffer.
nBytes = oJURLReader->ReadURL(sURLName)

;PRINT, 'Read ', nBytes, ' bytes'

; Pull the data into IDL.
byteArr = oJURLReader->GetData()

; Cleanup Java object.
OBJ_DESTROY, oJURLReader

; Return the data.
RETURN, byteArr

END

```

After saving and compiling the above files (`URLReader.java` in Java and `urlread.pro` in IDL), you can run the `URLRead` routine in IDL. This routine is a function with one input argument, which should be a IDL string containing an URL. For example:

```

address = 'http://www.ittvis.com'
data = URLRead(address)

```

Accessing Grayscale Images Example

This example creates a grayscale ramp image within a Java class, which is contained in a file named `GreyBandsImage.java`. IDL then accesses this data through the `ShowGreyImage` routine, which is in the `showgreyimage.pro` file.

Example Code

These files are located in the `resource/bridges/import/java/examples` directory of the IDL distribution. Run this example procedure by entering

showgreyimage at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT showgreyimage.pro`.

The `GreyBandsImage.java` file contains the following text for creating a grayscale image in Java:

```
import java.awt.*;
import java.awt.image.*;

public class GreyBandsImage extends BufferedImage
{
    // Members
    private int m_height;
    private int m_width;

    //
    // ctor
    //
    public GreyBandsImage() {
        super(100, 100, BufferedImage.TYPE_INT_ARGB);
        generateImage();
        m_height = 100;
        m_width = 100;
    }

    //
    // private method to generate the image
    //
    private void generateImage() {
        Color c;
        int width = getWidth();
        int height = getHeight();
        WritableRaster raster = getRaster();
        ColorModel model = getColorModel();

        int BAND_PIXEL_WIDTH = 5;
        int nBands = width/BAND_PIXEL_WIDTH;
        int greyDelta = 255 / nBands;
        for (int i=0 ; i < nBands; i++) {
            c = new Color(i*greyDelta, i*greyDelta, i*greyDelta);
            int argb = c.getRGB();
            Object colorData = model.getDataElements(argb, null);

            for (int j=0; j < height; j++)
                for (int k=0; k < BAND_PIXEL_WIDTH; k++)
                    raster.setDataElements(j, (i*5)+k, colorData);
        }
    }

    //

```

```

// mutators
//
public int[] getRawData() {
    Raster oRaster = getRaster();
    Rectangle oBounds = oRaster.getBounds();
    int[] data = new int[m_height * m_width * 4];

    data = oRaster.getPixels(0,0,100,100, data);
    return data;
}
public int getH() {return m_height; }
public int getW() {return m_width; }
}

```

The `showgreyimage.pro` file contains the following text for accessing the grayscale image within IDL:

```

PRO ShowGreyImage

; Construct the GreyBandImage in Java. This is a sub-class of
; BufferedImage. It is actually a 4 band image that happens to
display bands in greyscale. It is 100x100 pixels.
oGrey = OBJ_NEW('IDLjavaObject$GreyBandsImage', 'GreyBandsImage')

; Get the 4 byte pixel values.
data = oGrey -> GetRawData()

; Get the height and width.
h = oGrey -> GetH()
w = oGrey -> GetW()

; Display the graphic in an IDL window
WINDOW, 0, XSIZE = 100, YSIZE = 100
TV, REBIN(data, h, w)

; Cleanup
OBJ_DESTROY, oGrey

END

```

After saving and compiling the above files (`GreyBandsImage.java` in Java and `showgreyimage.pro` in IDL), you can run the `ShowGreyImage` routine in IDL. The routine should produce the following image:

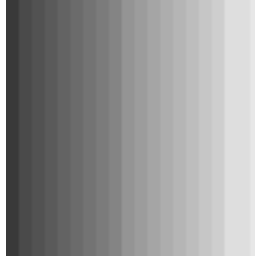


Figure 5-1: Java Grayscale Image Example

Accessing RGB Images Example

This example imports an RGB (red, green, and blue) image from the IDL distribution into a Java class. The image is in the `glowing_gas.jpg` file, which is in the `examples/data` directory of the IDL distribution. The Java class also displays the image in a Java Swing user-interface. Then, the image is accessed into IDL and displayed with the new `iImage` tool.

Example Code

The Java and IDL code for this example is provided in the `resource/bridges/import/java/examples` directory, but the Java code has not been built as part of the `jbexamples.jar` file.

Note

This example uses functionality only available in Java 1.4 and later.

Note

Due to a Java bug, this example (and any other example using Swing on AWT) will not work on Linux platforms.

The first and main Java class is `FrameTest`, which creates the Java Swing application that imports the image from the `glowing_gas.jpg` file. Copy and paste the following text into a file, then save it as `FrameTest.java`:

```
import java.awt.*;
import java.awt.image.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import java.io.File;

public class FrameTest extends JFrame {

    RSIIImageArea c_imgArea;
    int m_xsize;
    int m_ysize;
    Box c_controlBox;

    public FrameTest() {

        super("This is a JAVA Swing Program called from IDL");
        // Dispose the frame when the sys close is hit
        setDefaultCloseOperation(DISPOSE_ON_CLOSE);
        m_xsize = 350;
        m_ysize = 371;
        buildGUI();

    }

    public void buildGUI() {

        c_controlBox = Box.createVerticalBox();

        JLabel l1 = new JLabel("Example Java/IDL Interaction");
        JButton bLoadFile = new JButton("Load new file");
        bLoadFile.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JFileChooser chooser = new JFileChooser(new
                    File("c:\\ITT\\IDL63\\EXAMPLES\\DATA"));
                chooser.setDialogTitle("Enter a JPEG file");
                if (chooser.showOpenDialog(FrameTest.this) ==
                    JFileChooser.APPROVE_OPTION) {

                    java.io.File fname = chooser.getSelectedFile();
                    String filename = fname.getPath();
                    System.out.println(filename);
                    c_imgArea.setImageFile(filename);
                }
            }
        });
    }
}
```

```

});

JButton b1 = new JButton("Close this example");
b1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        dispose();
    }
});

c_imgArea = new
    RSImageArea("c:\\itt\\idl63\\examples\\data\\glowing_gas.jpg",
        new Dimension(m_xsize,m_ysize));

Box mainBox = Box.createVerticalBox();
Box rowBox = Box.createHorizontalBox();
rowBox.add(b1);
rowBox.add(bLoadFile);

c_controlBox.add(l1);
c_controlBox.add(rowBox);
mainBox.add(c_controlBox);
mainBox.add(c_imgArea);

getContentPane().add(mainBox);

pack();
setVisible(true);
c_imgArea.displayImage();
c_imgArea.addResizeListener(new RSImageAreaResizeListener() {
    public void areaResized(int newx, int newy) {
        Dimension cdim = c_controlBox.getSize(null);
        Insets i = getInsets();
        newx = i.left + i.right + newx;
        newy = i.top + cdim.height + newy + i.bottom;
        setSize(new Dimension(newx, newy));
    }
});
}

public void setImageData(int [] imgData, int xsize, int ysize) {
    MemoryImageSource ims = new MemoryImageSource(xsize, ysize,
        imgData, 0, ysize);
    Image imgtmp = createImage(ims);
    Graphics g = c_imgArea.getGraphics();
    g.drawImage(imgtmp, 0, 0, null);
}

public void setImageData(byte [][][] imgData, int xsize,

```

```

    int ysize) {

    System.out.println("SIZE = "+xsize+"x"+ysize);
    int newArray [] = new int[xsize*ysize];
    int pixi = 0;
    int curpix = 0;
    short [] currrgb = new short[3];
    for (int i=0;i<m_xsize;i++) {
        for (int j=0;j<m_ysize;j++) {
            for (int k=0;k<3;k++) {
                currrgb[k] = (short) imgData[k][i][j];
                currrgb[k] = (currrgb[k] < 128) ? (short) currrgb[k] : (short)
                    (currrgb[k]-256);
            }
            curpix = (int) currrgb[0] * +
                ((int) currrgb[1] * (int) Math.pow(2,8)) +
                ((int) currrgb[2] * (int) Math.pow(2,16));
            if (pixi % 1000 == 0)
                System.out.println("PIXI = "+pixi+" "+curpix);
            newArray[pixi++] = curpix;
        }
    }

    MemoryImageSource ims = new MemoryImageSource(xsize, ysize,
        newArray, 0, ysize);
    c_imgArea.setImageObj(c_imgArea.createImage(ims));

}

public byte[][][] getImageData()
{
    int width = 1;
    int height = 1;
    PixelGrabber pGrab;

    width = m_xsize;
    height = m_ysize;

    // pixarray for the grab - 3D bytearray for display
    int [] pixarray = new int[width*height];
    byte [][][] bytearray = new byte[3][width][height];

    // create a pixel grabber
    pGrab = new PixelGrabber(c_imgArea.getImageObj(),0,0,
        width,height, pixarray, 0, width);

    // grab the pixels from the image
    try {

```

```

        boolean b = pGrab.grabPixels();
    } catch (InterruptedException e) {
        System.err.println("pixel grab interrupted");
        return bytearray;
    }

    // break down the 32-bit integers from the grab into 8-bit bytes
    // and fill the return 3D array
    int pixi = 0;
    int curpix = 0;
    for (int j=0;j<m_ysize;j++) {
        for (int i=0;i<m_xsize;i++) {
            curpix = pixarray[pixi++];
            bytearray[0][i][j] = (byte) ((curpix >> 16) & 0xff);
            bytearray[1][i][j] = (byte) ((curpix >> 8) & 0xff);
            bytearray[2][i][j] = (byte) (curpix & 0xff);
        }
    }
    return bytearray;
}

public static void main(String [] args) {
    FrameTest f = new FrameTest();
}

}

```

Note

The above text is for the `FrameTest` class that accesses the `glowing_gas.jpg` file in the `examples/data` directory of a default installation of IDL on a Windows system. The file's location is specified as `c:\ITT\IDL70\EXAMPLES\DATA` in the above text. If the `glowing_gas.jpg` file is not in the same location on system, edit the text to change the location of this file to match your system.

The `FrameTest` class uses two other user-defined classes, `RSIImageArea` and `RSIImageAreaResizeListener`. These classes help to define the viewing area and display the image in Java. Copy and paste the following text into a file, then save it as `RSIImageArea.java`:

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Vector;
import java.io.File;

```



```

public class RSIIImageArea extends JComponent implements
    MouseMotionListener, MouseListener {

    Image c_img;
    int m_boxw = 100;
    int m_boxh = 100;
    Dimension c_dim;
    boolean m_pressed = false;
    int m_button = 0;
    Vector c_resizelisteners = null;

    public RSIIImageArea(String imgFile, Dimension dim) {

        c_img = getToolkit().getImage(imgFile);
        c_dim = dim;
        setPreferredSize(dim);
        setSize(dim);
        addMouseMotionListener(this);
        addMouseListener(this);

    }

    public void addResizeListener(RSIIImageAreaResizeListener l) {
        if (c_resizelisteners == null) c_resizelisteners = new Vector();
        if (! c_resizelisteners.contains(l)) c_resizelisteners.add(l);
    }
    public void removeResizeListener(RSIIImageAreaResizeListener l) {
        if (c_resizelisteners == null) return;
        if (c_resizelisteners.contains(l)) c_resizelisteners.remove(l);
    }

    public void displayImage() {
        repaint();
    }

    public void paint(Graphics g) {

        int xsize = c_img.getWidth(null);
        int ysize = c_img.getHeight(null);
        if (xsize != -1 && ysize != -1) {
            if (xsize != c_dim.width || ysize != c_dim.height) {
                c_dim.width = xsize;
                c_dim.height = ysize;
                setPreferredSize(c_dim);
                setSize(c_dim);
            }
            if (c_resizelisteners != null) {
                RSIIImageAreaResizeListener l = null;
                for (int j=0;j<c_resizelisteners.size();j++) {

```

```

        l = (RSIImageAreaResizeListener)
            c_resizelisteners.elementAt(j);
        l.areaResized(xsize, ysize);
    }
}
}
g.drawImage(c_img, 0, 0, null);
}

public void setImageFile(String fileName) {
    c_img = null;
    c_img = getToolkit().getImage(fileName);
    repaint();
}

public Image getImageObj() {
    return c_img;
}

public void setImageObj(Image img) {
    c_img = img;
    repaint();
}

public void drawZoomBox(MouseEvent e) {
    int bx = e.getX() - m_boxw/2;
    bx = (bx >=0) ? bx :0;
    int by = e.getY() - m_boxh/2;
    by = (by >=0) ? by :0;
    int ex = bx + m_boxw;
    if (ex > c_dim.width) {
        ex = c_dim.width;
        bx = c_dim.width-m_boxw;
    }
    int ey = by + m_boxh;
    if (ey > c_dim.height) {
        ey = c_dim.height;
        by = c_dim.height-m_boxh;
    }

    repaint();
    Graphics g = getGraphics();
    g.drawImage(c_img, bx, by, ex, ey, bx+(m_boxw/4), by+(m_boxh/4),
        ex-(m_boxw/4), ey-(m_boxh/4), null);
    g.setColor(Color.white);
    g.drawRect(bx, by, m_boxw, m_boxh);
}

```

```

    }

    public void mouseDragged(MouseEvent e) {
        drawZoomBox(e);
    }

    public void mouseMoved(MouseEvent e) {

        Graphics g = getGraphics();
        if (m_pressed && (m_button == 1)) {
            drawZoomBox(e);
            g.setColor(Color.white);
            g.drawString("DRAG", 10,10);
        } else {

            g.setColor(Color.white);
            String s = "("+e.getX()+","+e.getY()+")";
            repaint();
            g.drawString(s, e.getX(), e.getY());
        }

    }

    public void mouseClicked(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}

    public void mousePressed(MouseEvent e) {
        m_pressed = true;
        m_button = e.getButton();
        repaint();
        if (m_button == 1) drawZoomBox(e);
    }

    public void mouseReleased(MouseEvent e) {
        m_pressed = false;
        m_button = 0;
    }
}

```

And copy and paste the following text into a file, then save it as `RSIImageAreaResizeListener.java`:

```

public interface RSIImageAreaResizeListener {
    public void areaResized(int newx, int newy);
}

```

Compile these classes in Java. Then either update the `jbexamples.jar` file in the `resource/bridges/import/java` directory with the new compiled class, place

the resulting compiled classes in your Java class path, or edit the JVM Classpath setting in the IDL-Java bridge configuration file to specify the location (path) of these compiled classes. See “[Configuring the Bridge](#)” on page 75 for more information.

With the Java classes compiled, you can now access them in IDL. Copy and paste the following text into the IDL Editor window, then save it as `ImageFromJava.pro`:

```
PRO ImageFromJava
; Create a Swing Java object and have it load image data
; into IDL.

; Create the Java object first.
oJSwing = OBJ_NEW('IDLjavaObject$FrameTest', 'FrameTest')

; Get the image from the Java object.
image = oJSwing -> GetImageData()
PRINT, 'Loaded Image Information:'
HELP, image

; Delete the Java object.
OBJ_DESTROY, oJSwing

; Interactively display the image.
IIMAGE, image

END
```

After compiling the above routine, you can run it in IDL. This routine produces the following Java Swing application.

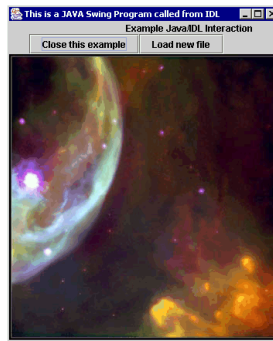


Figure 5-2: Java Swing Application Example

Then, the routine produces the following iImage tool.

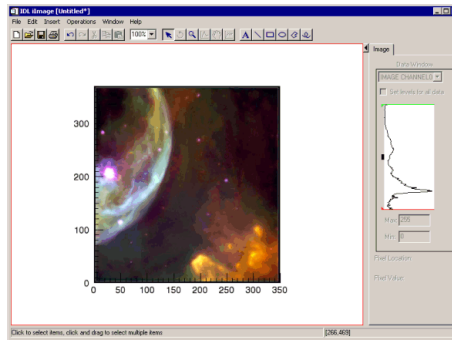


Figure 5-3: iImage Tool from Java Swing Example

Note

After IDL starts the Java Swing application, the two displays are independent of each other. If a new image is loaded into the Java application, the IDL iImage tool is not updated. If the iImage tool modifies the existing image or opens a new image, the Java Swing application is not updated.

Troubleshooting Your Bridge Session

The IDL-Java bridge provides error messages for specific types of operations. These messages can be used to determine when these errors occur, how these errors happen, and what solutions can be applied. The following sections pertain to these error messages and their possible solutions for each type of operation:

- [“Calling System.exit”](#)
- [“Errors When Initializing the Bridge”](#)
- [“Errors When Creating Objects”](#) on page 119
- [“Errors When Calling Methods”](#) on page 120
- [“Errors When Accessing Data Members”](#) on page 121

Calling System.exit

The Java method `System.exit` terminates the process in which the Java Virtual Machine is running. When the Java Virtual Machine is initialized by IDL, terminating its process also terminates IDL.

Errors When Initializing the Bridge

The IDL-Java bridge initializes when the first Java object in IDL is created. If the bridge is not configured correctly, an error message is issued and the IDL stops. The following errors occur because the IDL-Java bridge cannot find the Java Virtual Machine on your system. On UNIX, check the `IDLJAVAB_LIB_LOCATION` environment variable, and on Windows, check the `IDLJAVAB_LIB_LOCATION` environment variable. If this environment variable does not exist on your system, create it and set it equal to the location of the Java Virtual Machine on your system. See [“Configuring the Bridge”](#) on page 75 for details:

- Bad JVM Home value: `'path'`, where *path* is the location of Java Virtual Machine on your system.
- *JVM shared lib* not found in path `'JVM LibLocation'`, where *JVM shared lib* is the location of the Java Virtual Machine shared library and *JVM LibLocation* is the value of the `IDLJAVAB_LIB_LOCATION` environment variable.
- No valid JVM shared library exists at location pointed to by `$IDLJAVAB_LIB_LOCATION`

- `idljavab.jar` not found in path '*path*', where *path* is the location of the `/resource/bridges/import/java` directory in the IDL distribution.
- Bridge cannot determine which JVM to run
- Java virtual machine failed to start
- Failure loading JVM: *path/JVM shared lib name*, where *path* is the location of the Java Virtual Machine and *JVM shared lib name* is the name of the main Java shared library, which is usually `libjvm.so` on UNIX and `jvm.dll` on Windows.

If IDL catches an error and continues, subsequent attempts to call the bridge will generate the following message:

- IDL-Java bridge is not running

If this message occurs, fix the error and restart IDL.

Errors When Creating Objects

The following error messages can occur while creating a Java object in IDL. Possible solutions for these errors are also provided:

- Wrong number of parameters - occurs if `OBJ_NEW` does not have 2 or more parameters. Make sure you are specifying the class name twice; once in uppercase with periods replaced by underscores for IDL, and another with periods for Java. See [“Java Class Names in IDL”](#) on page 84 for details.
- Second parameter must be the Java class name - occurs if 2nd parameter is not an IDL string. When using `OBJ_NEW`, make sure the Java class name parameter is an IDL string. In other words, the class name has a single quote mark before and after it. See [“Java Class Names in IDL”](#) on page 84 for details.
- Class *classname* not found, where *classname* is the class name you specified in the first two parameters to `OBJ_NEW` - occurs if the IDL-Java bridge cannot find the class name specified. Check the spelling of each class name parameter and make sure the class name specified for IDL is referring to the same type of object specified for the Java class name. If the parameters are correct, check the Classpath setting in the IDL-Java bridge configuration file. Make sure the Classpath is set to the correct path for the class files containing the *classname* class. See [“Configuring the Bridge”](#) on page 75 for details.
- Class *classname* is not a public class, where *classname* is the class name you specified in the first two parameters to `OBJ_NEW` - occurs if

specified class is not a public class. Edit your Java code to make sure the class you want to access is public.

- Constructor `class::class(signature)` not found, where *class* is the class name - occurs if the IDL-Java bridge cannot find the class constructor with the given parameters. Check the spelling of the specified parameters and look in your Java code to see if you are specifying the correct arguments for the class you are trying to create. Also check to ensure your IDL data can be promoted to the data types in the Java signature. See “[Java Class Names in IDL](#)” on page 84 for details.
- Illegal IDL value in parameter *n*, where *n* is the position of the parameter - occurs if an illegal parameter type is provided. For example, an IDL structure is not allowed as a parameter to an IDLjavaObject.
- Exception thrown - occurs if an exception occurs in Java. Either correct or handle the Java exception. The Java exception can be determined with the IDLJavaBridgeSession object. See “[The IDLJavaBridgeSession Object](#)” on page 94 for details.

Errors When Calling Methods

The following error messages can occur while calling methods to Java objects in IDL. Possible solutions for these errors are also provided:

- Illegal IDL value in parameter *n*, where *n* is the position of the parameter - occurs if an illegal parameter type is provided. For example, an IDL structure are not allowed as a parameter to an IDLjavaObject.
- Class *class* has no method named *method*, where *class* is the class name and *method* is the method name specified when trying to call the Java method - occurs if the method of given name does not exist. Check the spelling of the method name. Also compare the method name in the Java class source file with the method name provided when calling the method in IDL. See “[What Happens When a Method Call Is Made?](#)” on page 87 for details.
- `class::method(signature)` is a void method. Must be called as a procedure, where *class* is the class name and *method* is the method name specified when a void Java method is called as an IDL function. Change the syntax of the method call. See “[Method Calls on IDL-Java Objects](#)” on page 87 for details.
- Method `class::method(signature)` not found, where *class* is the class name and *method* is the method name specified when trying to call the Java method - occurs if the IDL-Java bridge cannot find the method with a matching

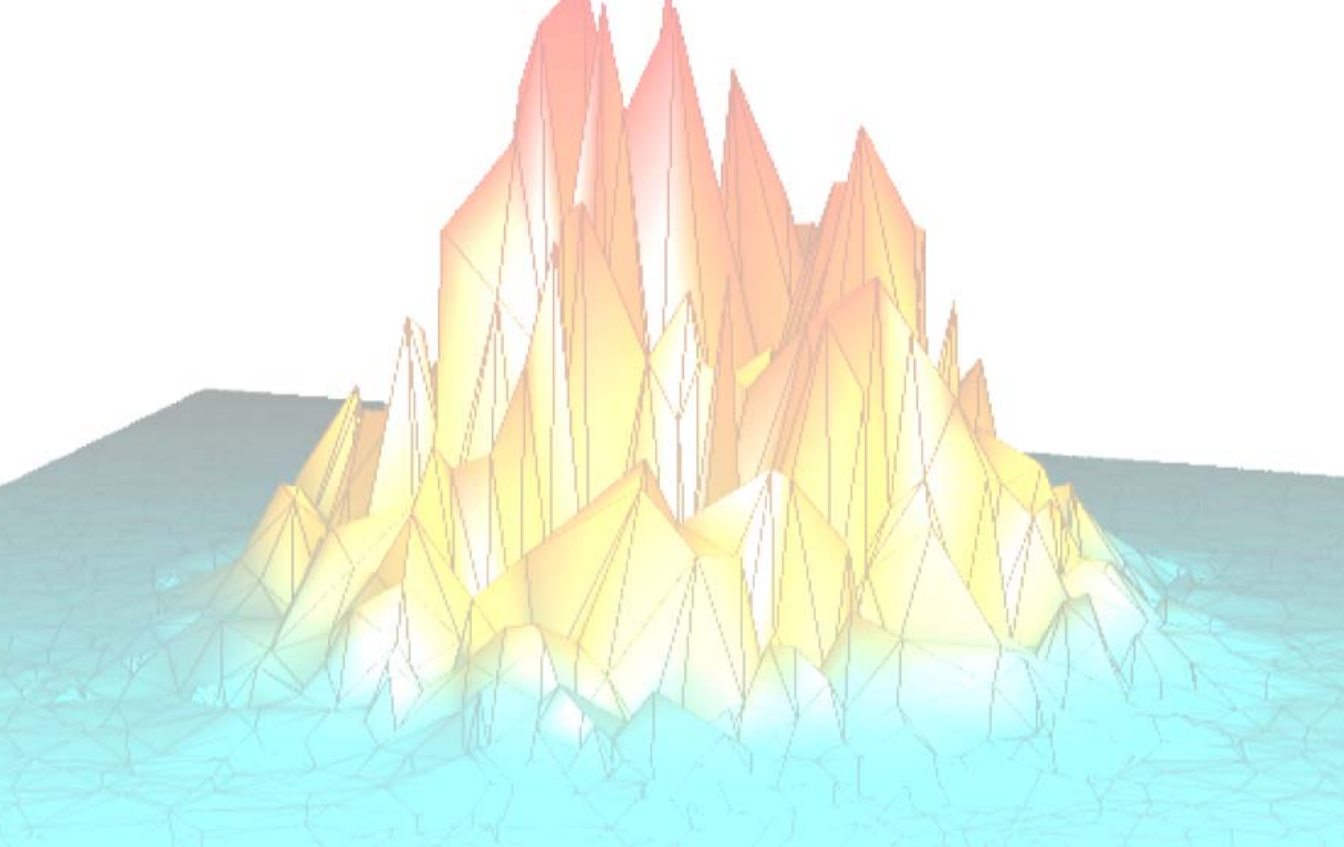
signature. Check the spelling of the method name. Also compare the method name in the Java class source file with the method name provided when calling the method in IDL. Also check to ensure your IDL data can be promoted to the Java signature. See [“What Happens When a Method Call Is Made?”](#) on page 87 for details.

- `Exception thrown` - occurs if an exception occurs in Java. Either correct or handle the Java exception. The Java exception can be determined with the `IDLJavaBridgeSession` object. See [“The IDLJavaBridgeSession Object”](#) on page 94 for details.

Errors When Accessing Data Members

The following error messages can occur while accessing data members to Java objects in IDL. Possible solutions for these errors are also provided:

- `Illegal IDL value in parameter n`, where *n* is the position of the parameter - occurs if an illegal parameter type is provided. For example, an IDL structure is not allowed as a parameter to an `IDLjavaObject`.
- `Class class has no data member named property`, where *class* is the class name and *property* is the data member name specified when trying to access the Java data member - occurs if the data member of the given name does not exist. Check the spelling of the property name. Also compare the data member name in the Java class source file with the property name provided when accessing it in IDL. See [“Managing IDL-Java Object Properties”](#) on page 89 for details.
- `Property class::property of type type not found`, where *class* is the class name, *property* is the data member name specified, and *type* is *property*’s data type when trying to access the Java data member - occurs if the IDL-Java bridge cannot find the Java data member of the given type. Check the data type of Java data member and make sure you are trying to use a similar type in IDL. See [“Getting and Setting Properties”](#) on page 90 for details.
- `Exception thrown` - occurs if an exception occurs in Java. Either correct or handle the Java exception. The Java exception can be determined with the `IDLJavaBridgeSession` object. See [“The IDLJavaBridgeSession Object”](#) on page 94 for details.



Part II: Exporting from IDL



Chapter 6

Exporting IDL Objects

This chapter discusses the following topics.

Overview of Exporting IDL Objects	126	Parameter Passing and Type Conversion .	136
Wrapper Objects	127	Event Handling	139
Object Lifecycle	130	Supported Platforms and IDL Modes	140
IDL Access	132	Configuring Build and Client Machines .	142

Overview of Exporting IDL Objects

IDL's *Export Bridge* technology allows you to easily integrate IDL technology into external environments using the latest component based frameworks and technology. Unlike the Callable IDL interface, which lets you create applications that exchange data with IDL through IDL variables and issue commands to the IDL interpreter but which requires familiarity with both C/C++ and IDL's own internal semantics and syntax, the export bridge technology allows you to create IDL objects that can be called directly from Java and COM applications.

Interaction with IDL is through native Java and COM *wrapper objects* that are generated for each IDL object with which client applications want to interact. The wrapper objects manage all aspects of IDL loading, initialization, process management, and cleanup, so you only need to be familiar with the client language (for embedding the wrapper in the client application) and the basics of IDL (for accessing and manipulating IDL data and processes).

The key to the Export Bridge is the Export Bridge Assistant, which generates these native wrapper objects from IDL objects. For more information on the Assistant, see [“Using the Export Bridge Assistant”](#) on page 147. For more information on wrapper objects, see [“Wrapper Objects”](#) on page 127.

Note

Before attempting to create and use wrapper objects, you should be familiar with the information in [“Configuring Build and Client Machines”](#) on page 142.

Wrapper Objects

The main concept used when exporting IDL objects for use in a client application is that of a *wrapper object*. A wrapper object is a native-language object (COM or Java) that exposes an IDL object's behavior to a client. The client interacts with an instance of the wrapper object using native-language constructs and native-language data types.

A wrapper object is built using the Export Bridge Assistant, in which you can choose which methods and properties of the IDL object to expose to the client. During the wrapper creation, you must specify the language-dependent variable types for all the parameters of the methods and properties to be exported. This is required since IDL has dynamically typed variables, whereas Java and COM do not. You can leave some properties or methods unimplemented in the wrapper object. For more information, see [“Using the Export Bridge Assistant”](#) on page 147.

When the Assistant exports an IDL object, it creates a language-specific wrapper object for the IDL object. The wrapper exposes methods and properties of the underlying IDL object it wraps, and the client interacts with the wrapper. When the client calls a method or modifies a property on a wrapper object, it is translated through a series of abstraction layers, and the underlying IDL object's method is called or property modified.

Every wrapper object has a collection of stock methods that are common to all wrapper objects as described in this document. Additionally, the underlying abstraction layers also handle creating the IDL object in another process. This use of multiple processes provides for IDL object pooling and isolation. For more information on these processes, see [“IDL Access”](#) on page 132.

For COM object wrappers, a `.dll` file is created for nondrawable objects; an `.ocx` file is created for drawable objects. In addition, a `.tlb` file is generated. The user registers the component and references the COM type library and property accessors (put/get) on the objects using native language constructs. A COM wrapper provides an IDispatch-based interface for client use.

For Java object wrappers, java files (`.java`) and class files (`.class`) are created. The user references the Java class definition in their code projects and calls methods and property accessors (set/get) on the objects using native language constructs. The Java wrapper is exposed as a standard Java object.

The actual use of the generated wrapper objects depends on the structure and patterns used for the client environment. For more information, see [“Using Exported COM Objects”](#) on page 189 and [“Using Exported Java Objects”](#) on page 215.

IDL Connector Objects and Custom Wrapper Objects

Access to IDL functionality from an external programming environment is available through connector and custom wrapper objects. The prebuilt connector wrapper object provides the ability to communicate with the IDL process from an external application. A custom wrapper object incorporates the functionality of your own IDL object.

Connector Objects

The connector object (distributed with IDL) provides access to IDL's processing capabilities through a number of methods that let you communicate with the IDL process. Using these methods, you can:

- Create and destroy instances of the connector object in your application
- Pass data to and retrieve data from IDL
- Get and set the IDL process name (see [“IDL Access”](#) on page 132 for more information)
- Execute IDL commands

Although the connector object does not provide support for graphics, it provides an easy way to access the processing power of IDL in an external environment. See [“Stock COM Wrapper Methods”](#) on page 192 (COM) and [“Stock Java Wrapper Methods”](#) on page 218 (Java) for complete language-specific method reference information. For examples using the connector object, see [Chapter 10, “Using the Connector Object”](#).

Note

There are no stock properties.

Custom Wrapper Objects

A custom wrapper object is an IDL object that is exported using the Export Bridge Assistant. A custom wrapper object contains the stock methods (referenced above) in addition to the specific methods and properties of the IDL object being wrapped. For information about how to create an IDL object that can be successfully exported, see [Chapter 11, “Writing IDL Objects for Exporting”](#). Examples of creating and using custom objects are available in:

- [Chapter 12, “Creating Custom COM Export Objects”](#)
- [Chapter 13, “Creating Custom Java Export Objects”](#)

Note

For more information on the language-specific wrapper objects, see “[COM Wrapper Objects](#)” on page 191 (COM) and “[Java Wrapper Objects](#)” on page 217 (Java).

Drawable and Nondrawable Objects

Custom wrapper objects can encapsulate either drawable or nondrawable IDL objects. To create a custom drawable wrapper object, the IDL source object must subclass from an `IDLitWindow`, `IDLgrWindow`, or `IDLitDirectWindow` visualization class and implement a set of callback routines for event handling. When events are detected for that window object, the callback methods are called with the information specific to the event detected. By subclassing from one of the drawable objects, a visualization written for use in an iTool visualization, Object Graphics display, or Direct Graphics display will seamlessly operate in an external environment via an export bridge. See “[Exporting Drawable Objects](#)” on page 264 for important information about creating and using drawable objects.

Nondrawable IDL objects are not derived from the `IDLitWindow`, `IDLgrWindow`, or `IDLitDirectWindow` classes and do not render to the screen. Nondrawable IDL objects do not have to inherit from any superclass, though derivation from `IDLitComponent` is necessary to fire IDL notifications.

Note

Java drawable objects are not supported on the Macintosh OS X platform.

Object Lifecycle

Object lifecycle means the duration in which an object is valid for use between the time it is instantiated or created and then released or destroyed. There are two lifecycles to understand when dealing with the Export Bridge's wrapper objects: the lifecycle of an instance of the wrapper object and the lifecycle of the underlying IDL object being wrapped.

The lifecycle of a wrapper object begins when an instance of the wrapper object is created within the client's application. However, the underlying IDL object is not created until the `CreateObject` stock method is called on the wrapper object instance. Every wrapper object has a set of stock methods, including `CreateObject` and `DestroyObject`, which are used to manage the object lifecycle. (For more information, see [“Object Creation”](#) and [“Object Release”](#) below.)

Note

For Java objects, the method is `createObject`, which is a more Java-like method-naming scheme. Assume that when this chapter mentions method calls, COM capitalizes the first word, but Java does not.

When the `CreateObject` method is called, the underlying IDL process is created (if necessary), and an instance of the IDL object is created. The lifecycle of the IDL object continues until the `DestroyObject` stock method is called on the wrapper object instance. The lifecycle of the client's wrapper object instance continues until it is released or destroyed using native language constructs.

Object Creation

Calling the `CreateObject` method on the wrapper object instance creates an instance of the underlying IDL object and calls its `Init` method with the specified parameters, if any. See [“CreateObject”](#) on page 194 (COM) and [“createObject”](#) on page 220 (Java) for language-specific calling conventions.

Object Release

Calling the `DestroyObject` method calls the underlying IDL object's `Cleanup` method, if present; then the underlying IDL object itself is destroyed. Calling `DestroyObject` does not release or destroy the wrapper object instance within the client space. This happens when the release method is called on the wrapper instance. See [“DestroyObject”](#) on page 199 (COM) and [“destroyObject”](#) on page 223 (Java) for language-specific calling conventions.

Java uses a garbage-collection scheme to clean up memory. It is important that there are no references to the wrapper object remaining in the client application; otherwise, the Java Virtual Machine (JVM) will not garbage-collect the wrapper object.

Note

There can be a period of time between the call to the `DestroyObject` method and when the wrapper instance itself is released. During that period, no method calls on the wrapper instance can be made because the underlying IDL object no longer exists.

IDL Access

Calling a method or accessing a property on a wrapper object instance calls into the underlying IDL object's method or property. Each wrapper object is associated with an IDL *process*, controlled by the IDL *main process*, by giving it a process name during wrapper creation by the Export Bridge Assistant. All wrapper objects that use the same process name have their underlying IDL objects created within the same IDL process. For each wrapper object that provides a unique process name, a new IDL process is created.

As a COM or Java developer, you do not need to worry about IDL process creation or destruction. Creating a new object creates a new process for it (unless a process already exists and the new object is being added to it), and destroying the last object in a process also destroys the process.

The code for the IDL object must be available because the bridge's process layers call it. The wrapper does not contain the IDL object, only provides an interface for it, and if you modify the IDL object after generation of its wrapper object, the wrapper might not work as expected. For more information, see [“Modifying a Source Object After Export”](#) on page 181.

Note

See [“Configuring Build and Client Machines”](#) on page 142 for information on setting up machines for building and using wrapper objects.

Note

Stock wrapper methods allow you to work with IDL processes. For COM, see [“GetProcessName”](#) on page 205 and [“SetProcessName”](#) on page 207. For Java, see [“getProcessName”](#) on page 228 and [“setProcessName”](#) on page 231. To take effect, you must set a process name before creating an object in order for the object to exist in that process.

Consider the following diagram:

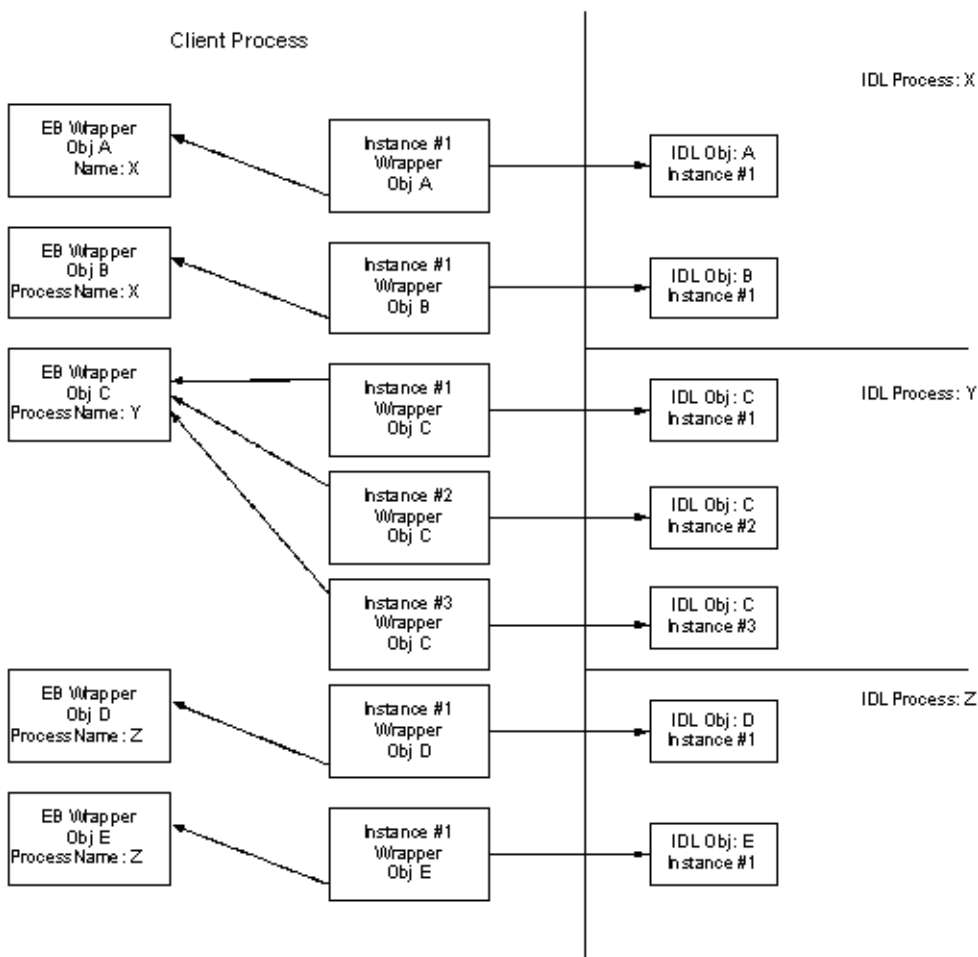


Figure 6-1: Example of Wrapper and Process Use

In the diagram, the client has created instances of several different wrapper objects: A, B, C, D, and E. Wrapper objects A and B have their process name set to X, and thus all instances of A and B create their underlying IDL objects in the same IDL process called X. Wrapper object C uses a different process, Y. Since there are three instances of the same wrapper object C, there are three instances of the IDL object C created in the process, Y. Wrapper objects E and D use an entirely different process, Z.

IDL Ownership and Blocking

During a method call, the client-side wrapper object instance becomes the owner of the IDL process that contains the underlying IDL object and remains the owner until the method call returns. An IDL process can only have one owner at a time. If there is a current owner of an IDL process and another wrapper object attempts to access the same IDL process, an IDL busy indication is returned through the wrapper object.

COM and Java handle error conditions differently: COM method calls return an HRESULT error value, whereas Java method calls throw an exception. In COM, this results in an IDL_BUSY condition; however, in Java, the requests are queued so that no busy condition occurs. See [“Error Handling”](#) on page 211 (COM) and [“Error Handling”](#) on page 242 (Java) for more information.

However, if one wrapper object instance owns a particular IDL process, another client process is free to make calls on other wrapper object instances that map to different IDL processes. In other words, the client can have multiple method calls executing at the same time as long as each method call maps to a different process.

For example, using the diagram in [Figure 6-1](#), if Instance #1 of wrapper object A is the current owner of the IDL process named X, and then another thread calls a method on Instance #1 of wrapper object B, it will return an IDL busy error, since it will try to use the same process as the wrapper object A. However, another thread can call a method on any instances of wrapper objects C, D, and E since they map to a different processes that are not currently owned.

IDL Licensing Modes

By default, when a client COM or Java application initializes the IDL object, the IDL export bridge checks to see what type of license is available on the client machine. If an IDL development license is available, it is used and the IDL object runs in full development mode. If a development license is not found, the export bridge checks for a runtime license; if the IDL object runs in runtime mode, normal runtime limitations (no compilation of `.pro` code, for example) are enforced. If no development or runtime license is found, the IDL object runs in Virtual Machine mode; normal Virtual Machine limitations (no compilation of `.pro` code, use of EXECUTE disabled, etc.) are enforced.

COM and Java applications can explicitly set IDL process initialization parameters to specify which licensing mode the IDL object will use. See the description of the *initializer* argument to the [createObject](#) method for details on initializing IDL objects from a Java application. See the description of the *flags* argument to the

[CreateObjectEx](#) method for details on initializing IDL objects from a COM application.

Parameter Passing and Type Conversion

The following topics contain important information that must be kept in mind when passing objects, arrays and variables between IDL and an external programming environment:

- [“Object Reference Use”](#) below
- [“Array Order Conversion”](#) on page 137
- [“Type Conversion”](#) on page 137

Object Reference Use

It is possible to pass an object reference to another wrapper object as a method parameter, with the following restrictions.

- The object reference must be a reference to another Export Bridge wrapper object instance of the same wrapper language type (COM or Java) — that is, COM to COM or Java to Java
- You cannot pass in object references to non-Export Bridge wrapper objects
- The object reference is “in-only,” meaning that methods and properties cannot return or modify a reference to an object
- Both objects (the object being referred to and the object using the reference) must have their underlying IDL objects contained within the same IDL process.

For example, using the diagram in [Figure 6-1](#), wrapper object A can have a method that takes an object reference. But the only valid object reference that can be specified is to an instance of wrapper object B, since both have their underlying IDL objects living in the same process, X.

If you attempt to pass in an object reference to an IDL object contained in different processes, the method call returns an error. An error is also returned if you attempt to pass in an object reference that does not reference an instance of an Export Bridge wrapper object.

Arrays of Object References

You can also create an array of object references as long as all the objects being referenced are in the same IDL process as the object using the array.

When creating an array of object references for COM, it must be defined as a `SAFEARRAY` of variants, with each variant containing the `IUnknown` or `IDispatch` pointer to a COM or ActiveX wrapper object instance.

When creating an array of object references for Java, it must be defined as a `JIDLArray` containing an array of `JIDLObjectI` references.

Array Order Conversion

A method parameter or property value can be an array. When dealing with multidimensional arrays, one must always be aware of the array ordering. See [“Multidimensional Array Storage and Access”](#) on page 493 for a complete discussion of the issues.

However, you must take into account the array ordering of the client-side array and the array order expected by IDL. The wrapper objects will convert array ordering when designated to do so in the Export Bridge Assistant. During wrapper object construction, the Export Bridge Assistant lets you designate a method parameter as an array and then indicate if the array needs to be converted (see [“Converting Array Majority”](#) on page 165 for details). If the array parameter is marked for conversion, the client array is converted during the method call before being sent to the underlying IDL object. If the parameter is also marked with In/Out mutability (meaning that the parameter is not constant and can be set by the caller and pass the value back to the caller), the array is also converted on the way back to the client. For more information on mutability, see [“Parameter Information”](#) on page 176.

However, there are certain cases where arrays are automatically converted and the user does not have the option to designate conversion. When calling the `GetIdlVariable` and `SetIdlVariable` methods on a wrapper object, or when an IDL function returns an array value, the array is always converted into the order expected by COM. (For Java, the user has the option to designate conversion.)

Type Conversion

IDL is a dynamically typed language that lets variables change type after creation. Java and COM are strongly typed languages, which require a variable to be given a fixed type when it is created. This difference can lead to type-conversion errors during method calls because the IDL object can redefine the data type of a parameter. When a method parameter is marked In/Out, the updated parameter value is returned to the client upon return of the method. During the method return, the wrapper compares the data type of the input value against the data type of the output value.

The wrapper will perform a *loose type conversion* in which:

- Any scalar type can be converted to any other scalar type (e.g., a short integer to a long integer)
- A scalar string to a scalar string (e.g., a string of one length to a different length)
- An array to an array (e.g., any dimensionality and type to any other dimensionality and type)

Loose type conversion attempts to convert the variables returned by the wrapped IDL object to the types expected by the wrapper object.

A data conversion error is returned when the above rules are not met. For example:

- A scalar changes to a string
- A scalar changes to an array
- A string changes to a scalar
- A string changes to an array
- An array changes to a scalar
- An array changes to a string

See [“Supported Data Types”](#) on page 166 for data types supported by COM and Java.

IDL Error State and Successful Method Return

If your client creates an instance of a COM/Java Export Bridge wrapper object, and calls an object method whose code throws an error, the wrapped method will return an error unless the referenced code resets the internal IDL error state.

In this circumstance, it is best if the wrapped code catches its own error, handles it, and resets the IDL error state. You can reset the IDL error state in the error handling catch block by calling the MESSAGE procedure:

```
MESSAGE, /RESET
```

This procedure call sets the !ERROR_STATE system variable back to the “success” state.

Event Handling

There are three main types of events that the clients of wrapper objects care about: user-interface events (e.g., mouse click and mouse move), IDL output, and IDL notifications. User-interface events are only available for drawable wrapper objects. The IDL output and notifications are available for drawable and nondrawable wrapper objects. The mechanism for the clients to receive wrapper-object events is different for the different wrapper-object languages, as described in “[Event Handling](#)” on page 208 (COM) and “[Event Handling](#)” on page 232 (Java).

An IDL notification is a way for an IDL object to relay information back to wrapper object instances while in the middle of a method call. This can be used for things like updating the status of lengthy operations. In order for a wrapper object to receive an IDL notification, the IDL object must inherit from the `IDLitComponent` object, and the client must subscribe to the wrapper instance’s events. All IDL graphic objects automatically inherit from `IDLitComponent`. For nondrawable objects, if the IDL object needs to send out a notification, it must explicitly inherit from `IDLitComponent`.

The `IDLitComponent::NotifyBridge` method sends the notification. It takes any two strings as parameters. For example, in the pro code below, assume that the object is derived from `IDLitComponent` and the user wants to inform the client of the status of a lengthy computation.

```
pro IDLmyObject::DoLongComputation

    for I = 0, 10000000 do begin
        ...
        percentDone = CalcPercentDone()
        ; Send client some status
        self->NotifyBridge, 'Completion Status', STRING(percentDone)
    endfor

end
```

Note

IDL objects must derive from `IDLitComponent` if IDL notifications will be used.

Supported Platforms and IDL Modes

The IDL Export Bridge technology is available on the following platforms:

Feature	Windows		OS X	Linux		Solaris	
	32-bit	64-bit	32-bit	32-bit	64-bit	32-bit	64-bit
COM Object –Export (via Export Bridge Assistant)	•						
Java Object –Export (via Export Bridge Assistant)	•	•	• ^a	•	•	•	•

Table 6-1: Export Bridge Platform Support

^a Graphical Java objects cannot be exported under Macintosh OS X.

Supported Compilers

The IDL Export Bridge requires the following compilers for building COM and Java wrapper objects.

Wrapper Object Type	Compilers Supported
COM	Use Visual Studio 2005 for both the machine running the Export Bridge Assistant and the machine building an application using the wrapper objects (if different). VB.NET, C#, C++ Managed, and C++ Unmanaged are all supported.
Java	Use the Java Developer's Kit (JDK) and Java Runtime Environment (JRE) version 1.5 or higher Note - On Macintosh machines, the version of Java installed along with the operating system should be sufficient, whatever its version number.

Table 6-2: Export Bridge Wrapper Object Compiler Support

Client Machine Requirements

Client machines (those running applications that incorporate a wrapper object) have separate requirements. See [“Configuring the Machine Running the Wrapper Client”](#) on page 144 for details

Output Destinations

Windows allows output to both COM and Java. On other supported platforms, only Java is supported (*not* COM). For a COM project on non-Windows platforms, the Build menu in the Export Bridge Assistant is disabled.

IDL Licensing

Build machines must have an IDL license and an Export Bridge Assistant license. Client machines must have either a licensed installation of IDL or a copy of the IDL Virtual Machine. (Note that the `ExecuteString` methods are disabled for applications running in the IDL Virtual Machine.)

Export Bridge Assistant Licensing

The Export Bridge Assistant is an IDL application. While the Assistant will run and allow you to create export projects with any IDL license, an additional-cost license is required to build the Java or COM native wrapper objects.

The Export Bridge Assistant cannot be run in runtime mode or in the IDL Virtual Machine. Attempting to run the Assistant in various licensing modes will have the following effects:

- Runtime mode — the Assistant will issue an error and exit
- IDL demo mode or no IDL license — the Save and Build operations are disabled
- No Export Bridge Assistant license — the Build operation is disabled without an Assistant licence feature (`idl_bridge_assist`)

Configuring Build and Client Machines

This section describes how to configure *build machines*:

- Machines that run the Export Bridge Assistant
- Machines that use the wrapper objects created by the Assistant in an external development environment (if different)

and *client machines*:

- Machines running applications that rely on wrapper objects

As a developer of applications that use wrapped IDL objects, you should be familiar with all of the information in this section.

Configuring the Machine Running the Assistant

The computer that runs the Export Bridge Assistant must meet the following requirements:

Item	Description
General Requirements	<ul style="list-style-type: none">• IDL must be installed.• The IDL source object does not need to be in the IDL path to be used by the Assistant, but does to be used by the client application (see below). Any IDL code referenced by the source object must be in the same directory as the source object or in the IDL path.• Drawable IDL objects that inherit from IDLgrWindow, IDLitWindow, and IDLitDirectWindow have special requirements as described in “Requirements for Drawable Objects” on page 264.
COM Requirements	Visual Studio must be installed.
Java Requirements	Java must be installed, and <code>javac</code> must be in the execution path.

Note

See “[Supported Platforms and IDL Modes](#)” on page 140 for supported COM and Java versions.

Configuring the Machine Using Wrapper Objects

If different from the machine running the Assistant, the machine using wrapper objects in application development must meet the following requirements in addition to the requirements listed for the Assistant (“[Configuring the Machine Running the Assistant](#)” on page 142).

COM Registration Requirements

The wrapper object generated by the Assistant must be registered using `regsvr32 <wrapperName>.DLL` for non-drawable objects or `regsvr32 <wrapperName>.OCX` for drawable objects. To register a file:

1. Select **Start** → **Run**, type `cmd` in the text box and click **OK** to open the Command Prompt window.
2. Use the `cd` command to change to the directory containing the file to be registered.
3. Enter `regsvr32 <wrapperName>.DLL` or `<wrapperName>.OCX` to register the file.

A message box will report the successful registration of the file.

Note

If needed, you can unregister a file by using the `-u` flag as in
`regsvr32 -u <wrapperName>.DLL`

See “[Wrapper Generation Example](#)” on page 182 for a short example that exports and uses a simple IDL object.

Java Requirements

Java must be installed. Both `javac` and `java` must be in the execution path.

Note

The Java runtime environment installation does not provide `javac`.

For compilation and execution, the file

```
IDL_DIR/resource/bridges/export/java/javaidlb.jar
```

must be in the Java classpath.

For Java routines to use the exported java objects, they must use the following import statement:

```
import com.idl.javaidl.*
```

On UNIX systems, the [LD_LIBRARY_PATH](#) environment variable (DYLD_LIBRARY_PATH on Mac OS X) must include the IDL bin.<platform>.<arch> directory. The [PATH](#) environment variable must also include this directory.

The [IDL_PATH](#) environment variable must include the directory containing the IDL source object source or SAVE file. In most cases, the variable should also include the default IDL library so that IDL routines can be resolved.

See “[Wrapper Generation Example](#)” on page 182 for a short example that exports and uses a simple IDL object.

The bridge_setup Script

On UNIX platforms, source the <IDL_DIR>/bin/bridge_setup script to set the appropriate values for the [IDL_DIR](#), [LD_LIBRARY_PATH](#), and [CLASSPATH](#) environment variables. (The <IDL_DIR>/bin directory also contains versions of this script for use with the korn or bash shells.)

Note

On 64-bit Solaris platforms, the bridge_setup script will specify the 32-bit version of IDL by default, since most Solaris systems use the 32-bit version of Java as the default. To explicitly specify that the 64-bit version of IDL should be used, set the IDL_PREFER_64 environment variable. (The value to which this environment variable is set is not important; if it is defined at all the 64-bit version of IDL will be used.)

There is no bridge_setup script for Windows platforms. In most cases, setting the CLASSPATH environment variable (or specifying the class path along with the java or javac command at the command line) is the only configuration necessary.

Configuring the Machine Running the Wrapper Client

The machine running the COM or Java application that uses a wrapper object must have either a licensed version of IDL or a copy of the IDL Virtual Machine installed. (Note that applications that use the ExecuteString method will not work in the IDL

Virtual Machine.) Additionally, the IDL `.pro` or `.sav` file containing the object definition must be in the IDL path. This requirement also applies to any IDL files called by code in the source object.

COM Applications

For a COM application:

- The executable file (`.exe`), and any `.dlls` generated during the Visual Studio build process must be made available to the client.
- The `.dll` or `.ocx` file associated with a custom wrapper object must be registered on the client machine. The client need not have Visual Studio installed.

Note

Applications using the connector wrapper object need not register the connector object `.dll`. This file is automatically registered upon IDL installation.

- For an application built in a .NET language (such as Visual Basic .NET or C#), the Microsoft .NET Framework must be installed on the client machine.

Java Applications

For a Java application:

- The Java Runtime Environment (JRE) must be installed (see “[Supported Platforms and IDL Modes](#)” on page 140 for supported version information)
- The executable `.class` file must be made available to the client
- `IDL_DIR/resource/bridges/export/java/javaidlb.jar` must be in the Java classpath

Note

On UNIX systems, it is advisable to execute the `bridge_setup` script on the client machine as part of the Java application initialization. This ensures that IDL is properly configured on the client machine. See “[The bridge_setup Script](#)” on page 144 for details.



Chapter 7

Using the Export Bridge Assistant

This chapter discusses the following topics.

Export Bridge Assistant Overview	148	Specifying Information for Exporting ...	164
Running the Assistant	149	Information Skipped During Export	178
Using the Assistant	150	Exporting a Source Object's Superclasses	180
Working with a Project	157	Modifying a Source Object After Export .	181
Building an Object	161	Wrapper Generation Example	182
Exporting an Object	162		

Export Bridge Assistant Overview

The Export Bridge technology lets an IDL object be accessed from Java or COM through the use of wrapper objects. The Export Bridge Assistant helps to automate the process of creating the Java or COM wrapper object from the IDL source object.

The Assistant obtains as much information as possible about the IDL object directly from IDL. Since IDL is loosely typed, the return types of functions and the types of object properties and method parameters cannot be determined from IDL. Other information such as the output destination (Java or COM) and destination specific properties are not available from IDL and must be specified by the user.

The Assistant lets you specify the information described above for each item that is to be exported. Note that you can choose not to export some properties, methods, or parameters of the IDL source object. Any items that are both fully specified and marked for export are built in the exported Java or COM object.

The Export Bridge Assistant can produce an IDL SAVE file containing a specification of the IDL source object that is to be exported. This SAVE file, called a wrapper definition file, preserves the state of your work between invocations of the Assistant. You can stop the Assistant before the specification is complete and reopen it at a later time to continue building.

Note

There are special requirements for IDL source object that are to be exported including data type limitations, structural requirements, and methods that need to be included for drawable objects. See [Chapter 11, “Writing IDL Objects for Exporting”](#) for complete details.

Platform Support and Machine Configuration

See [“Supported Platforms and IDL Modes”](#) on page 140 for information on the platforms on which you can use the Export Bridge Assistant to create wrapper objects. See [“Configuring Build and Client Machines”](#) on page 142 for details on configuring computers to build and run wrapper objects.

Running the Assistant

Start the Export Bridge Assistant from the IDL Workbench by entering the command

`IDLEXBR_ASSISTANT`

at the IDL command line. For more information, see “[IDLEXBR_ASSISTANT](#)” (*IDL Reference Guide*).

Using the Assistant

You can use the Export Bridge Assistant to create COM or Java wrapper objects from native IDL objects. The Assistant is a system-wide dialog; for information on launching it, see [“Running the Assistant”](#) on page 149.

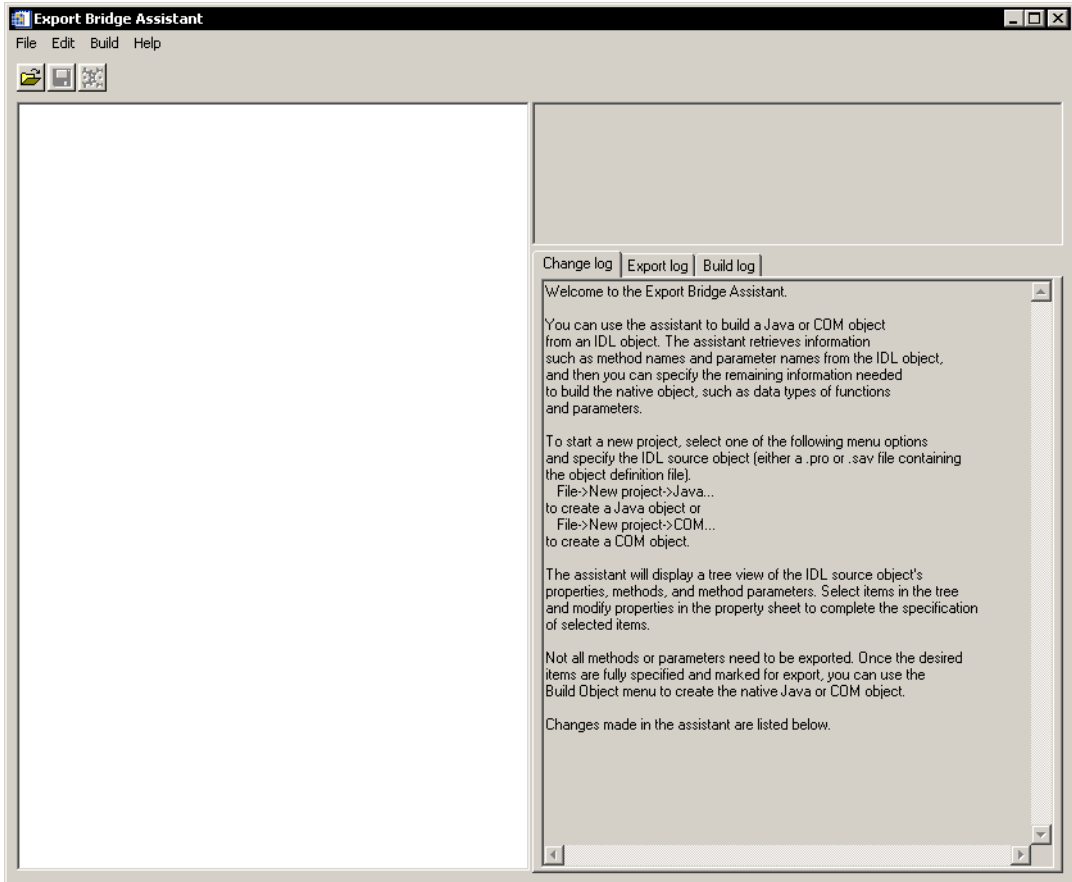


Figure 7-1: The Export Bridge Assistant When Launched

[Figure 7-1](#) shows the Assistant when it is first launched, without a project open.

Understanding the Assistant Interface

The Assistant consists of three panels, a menu bar, and a toolbar. The panels are a tree view of the current project (if any), a property view of the current selected item (if any), and a view of the three informational logs available in the Assistant.

The Menu Bar

The following menus appear on the Assistant menu bar:

- File
- Edit
- Build
- Help

The **File** menu, shown in [Table 7-1](#), contains tools for creating, importing, exporting, and saving projects.

Menu Selection	Function
New Project	For COM or Java (selected in a sub-menu), creates a new project by selecting an IDL source file (.pro) or SAVE file containing an object definition. See “Working with a Project” on page 157 for details.
Open Project...	Opens an existing project. See “Opening a Project” on page 157 for details.
Close Project	Closes the current project, prompting you to save any unsaved changes.
Save Project	Saves the current project to an IDL SAVE file. If the project has not been previously saved, the behavior matches that of Save Project As... (below). See “Saving a Project” on page 157 for details.
Save Project As...	Prompts you to select a name for the project’s IDL SAVE file. See “Saving a Project” on page 157 for details.
Revert To Saved	Prompts you to discard changes made to the current project and revert to its most recent saved version.

Table 7-1: The File menu

Menu Selection	Function
Update From Source...	Prompts you to select an IDL source file or SAVE file containing an object definition, which is compared to the source object in the current project. See “Updating a Project” on page 158 for details.
Save Log...	Saves the contents of the current log (change, export, or build). The menu selection’s name changes to reflect that of the current log (e.g., Save Change Log...). See “The Logs Panel” on page 154 for details.
Exit	Closes the Assistant, prompting you to save any unsaved project changes.

Table 7-1: The File menu

The **Edit** menu contains only one operation: **Clear Log**, which clears the contents of the current log (change, export, or build). The operation’s name changes to reflect that of the current log (e.g., Clear Change Log). See [“The Logs Panel”](#) on page 154 for details.

The **Build** menu contains only one operation: **Build Object**, which builds the current object. See [“Building an Object”](#) on page 161 for details.

The **Help** menu opens the online help for the following topics:

- Using the Export Bridge Assistant
- Configuring the Export Bridge Assistant
- Exporting IDL objects to COM and Java
- Using exported COM objects
- Using exported Java objects
- Help on IDL

The Toolbar

The following buttons appear on the Assistant toolbar:

- Open Project
- Save Project
- Build Object

These buttons match the menu operations of the same name. See [“The Menu Bar”](#) on page 151 for details.

The Project Tree View

The project tree displays a hierarchical view of the project and the contained IDL source object with its properties, methods, method parameters, and superclasses. (See [“Specifying Information for Exporting”](#) on page 164 for more information.) Clicking on an item in the tree fills in the property sheet for that selected item.

Multiple selection is enabled. This can be very useful for setting object properties efficiently. A property is only applied to a selected item if it implements the property, allowing a selection to span disparate items.

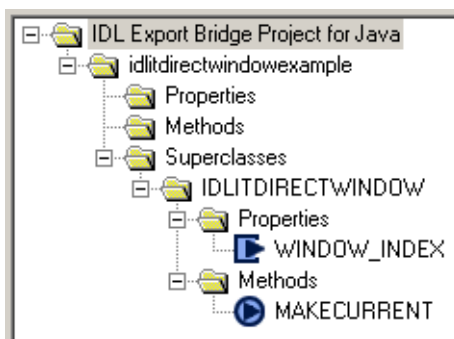


Figure 7-2: The Project Tree View of the Export Bridge Assistant

The icons next to items in the project tree indicate their readiness for export to a wrapper object. For more information, see [“Exporting an Object”](#) on page 162.

The Property Sheet View

The property view displays the properties of items selected in the project tree view. You can change the properties using this view. Multiple selection is not enabled.

WINDOW_INDEX	
Type	JIDLNumber
Array	False
Convert majority	True
Export	True

Figure 7-3: The Property Sheet View of the Export Bridge Assistant

The Logs Panel

The logs panel has three tabs: Change Log, Export Log, and Build Log.

The Change Log

This text field initially contains welcome text that is cleared when a project is created or closed. When a project is open, the field displays a running log of property settings made by the user, including property changes and the following actions: Save Project, Update From Source, and Revert To Saved Project. [Figure 7-4](#) shows an example of a change log in progress.

The text is saved with the project, and when an existing project is opened, it is re-displayed.

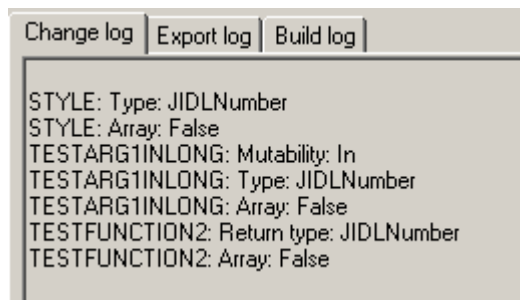


Figure 7-4: The Change Log of the Export Bridge Assistant

The Export Log

This text field contains a description of the items that are to be exported (those items that are both fully specified and marked for export). It is cleared when a project is created or closed. [Figure 7-5](#) shows an example of an export log in progress.

The text is saved with the project, and when an existing project is opened, it is re-displayed.

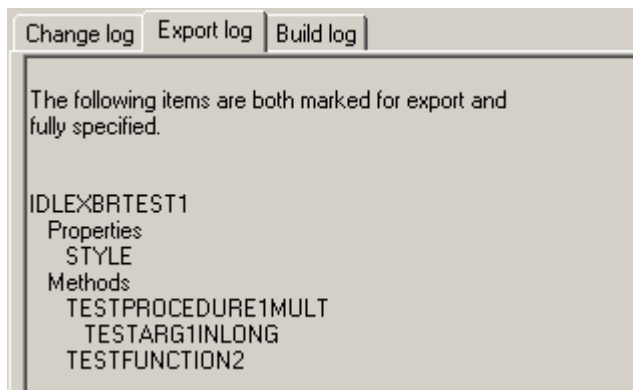


Figure 7-5: The Export Log of the Export Bridge Assistant

The Build Log

This text field displays the results of the build operation. It is cleared when a project is created, opened or closed. [Figure 7-6](#) shows an example of a build log in progress.

The text is saved with the project, and when an existing project is opened, it is re-displayed.

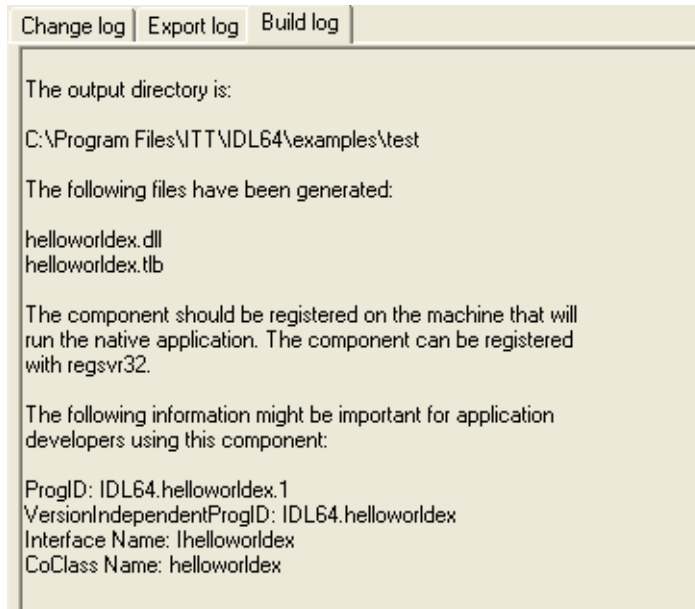


Figure 7-6: The Build Log of the Export Bridge Assistant

Note

The ProgID identifies the exported IDL object, and is displayed on the Build Log tab. You may need this identifier if you handle the exported object directly in a custom application.

Working with a Project

The Export Bridge Assistant works with a project that contains an IDL source object to be exported. You can create a new project or open an existing one, modify or update it, and save it.

Opening a Project

If you are creating a new project, you have the choice of making it COM or Java. For both object types, you must specify the IDL source object by selecting either an IDL source file (`<idlObject>__define.pro`) or a SAVE file containing an object definition (`<idlObject>__define.sav`).

To open an existing project, you must select an existing wrapper definition file (`<idlObject>_<dest>_wrapdef.sav`) created by a previous invocation of the Assistant.

Note

You can create or open a COM project on UNIX, but you cannot build any COM objects. See [“Output Destinations”](#) on page 141 for details.

Once the source object is specified, the IDL object is resolved. Note that the source object file does not have to be in the path. However, any supporting or referenced source file must be in the same directory or in the IDL path so that it can be resolved.

When the object is resolved, the Assistant populates the project tree with property names, routine names, and parameter names from the object. You can use this view to specify information about the object necessary for creation of the wrapper objects.

If you create a new project or open an existing project while you already have a project open, you will be prompted to save any changes made to the current project before the new or existing project opens. You can cancel instead to continue working on the current project.

Saving a Project

You can save your work in the current project at any time. The Assistant stores the information in an IDL SAVE file. You can save a project without having an Export Bridges license (see [“Running the Assistant”](#) on page 149 for details).

If you are saving a project for the first time, the Assistant prompts you for the SAVE file’s name and location. The default name is based on the source object class name

as follows: `<idlObject>_<dest>_wrapdef.sav`, where `<dest>` is either `java` or `com`.

Note

This filename is the default created by the Assistant, but you can save project files in SAVE files with any name.

Updating a Project

You might have used the Assistant to generate a wrapper object's specification, make changes to the original IDL source object, and want to merge these changes into the existing object specification without losing the initial work done in the Assistant. You can do this by bringing in the modified source object and having the Assistant respond with both automated and manual update functionality.

The following list provides some common cases where an update might be useful:

- Changes to the IDL object
 - Changes to method names in IDL object, parameters unchanged
 - Methods added
 - Methods removed
 - Method parameters added
 - Method parameters removed
- Changes to object specification
 - Method data modified (e.g., from function to procedure, the return type, whether the return value is an array or not)
 - Parameter data modified (e.g., parameter type, array)

When you select an object definition using the **File → Update From Source...** command, the Assistant compares it to the object in the current project and ensures that the object class of the file selected matches the class of the existing project.

Updating an existing project with an IDL source object redefines the project based on the definition of the source object. When applicable, attributes from the existing project are applied to matching items from the update. This application takes place both automatically in the Assistant and manually through interaction with a dialog that launches to guide the update.

First, the project tree is populated with routine names and parameter names from the updated source (the master). Next, information from the IDL source object is

compared to the existing definition. Property, method, and parameter information is copied when the item is present in both existing definition and the updated source object. The matching functionality is triggered if there are both added and removed methods. The matching dialog is displayed (if applicable) so you can match names of methods that were renamed. If matched, parameter information that matches exactly is copied to the new wrapper definition.

The following dialog shows a method that has been renamed in the updated source (marked with '`_CHANGED`'). The method `TESTPROCEDURE1MULT` from the old methods has been linked to the new method `TESTPROCEDURE1MULT_CHANGED`, which updates the display of linked methods.

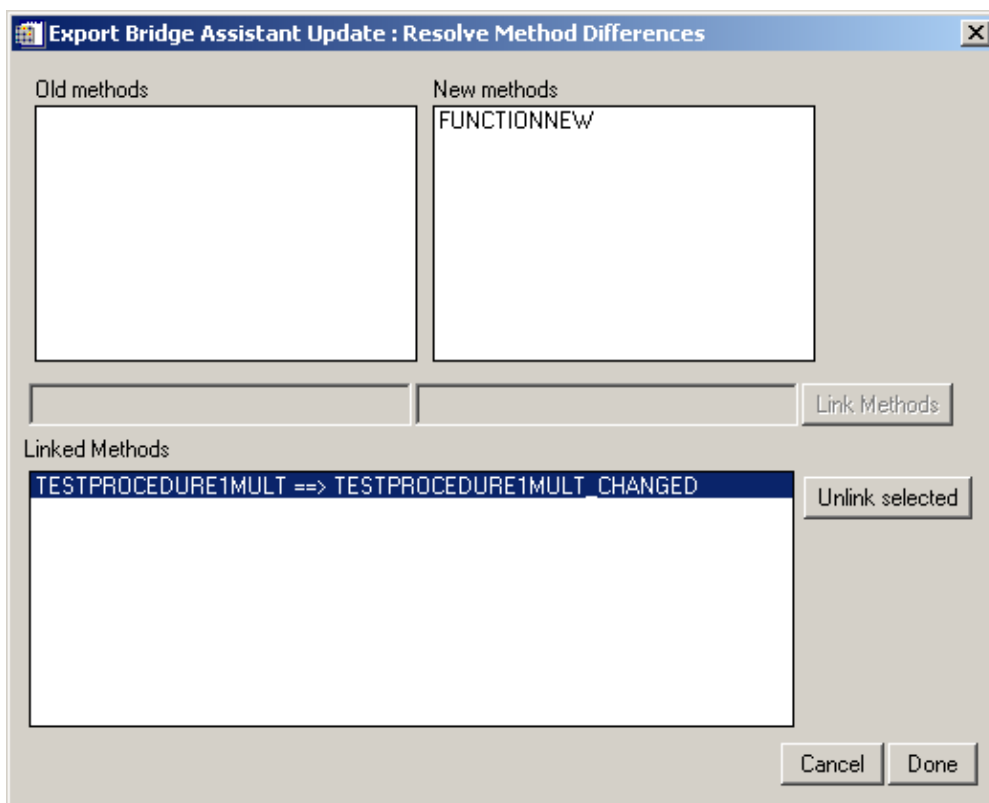


Figure 7-7: The Export Bridge Assistant's Update Dialogue

Table 7-2 summarizes the details of object modification and project update.

Object Modifications	Effect of Modification in Assistant	Manual Action Taken	Automatic Action Taken
Method renamed, parameters unchanged	Both Methods Added and Methods Removed are true	Object method added, object definition method missing; you can match old method name with new method name	New method added, information from old method copied to new method, old method removed.
Method added	Object has a method not in the project	If Methods Removed is false, add method; otherwise, see the method-renamed information (above)	New method added
Method removed	Object lacks a method in the project	If Methods Added is false, remove method; otherwise, see the method-renamed information (above)	Old method removed
Parameter renamed	N/A	None	New parameter added, old parameter removed
Parameter added	Updated object has a parameter not in the project	None	New parameter added
Parameter removed	Object lacks a parameter in the project	None	Old parameter removed
Property added	Updated object has a property not in the project	None	New property added
Property removed	Object lacks a property in the project	None	Old property removed

Table 7-2: Resolving an Update from Source in the Export Bridge Assistant

Building an Object

To build a wrapper object, you need to create an object specification about the exported object in the Export Bridge Assistant. This wrapper object is what your client application needs to use the IDL source object's functionality.

Note that the wrapper object is only an interface between your client application and the IDL source object. That is, the Assistant does not *include* the entire IDL object in a wrapper object generated from it, but creates a COM or Java layer to interact with the source object. Furthermore, if you change the source object, you might affect an existing wrapper object exported from it (see [“Modifying a Source Object After Export”](#) on page 181).

Important topics regarding building an object include the following:

- Understanding the object status for exporting ([“Exporting an Object”](#) on page 162)
- What information you need to specify when exporting an object ([“Specifying Information for Exporting”](#) on page 164)
- Java and COM types supported by the Export Bridge technology ([“Supported Data Types”](#) on page 166)
- What gets skipped for exporting ([“Information Skipped During Export”](#) on page 178)
- How to export superclasses ([“Exporting a Source Object's Superclasses”](#) on page 180)
- What to do with a modified object after exporting ([“Modifying a Source Object After Export”](#) on page 181)

Exporting an Object

The Assistant lets you set data types for parameters and other values needed for creation of the wrappers. In addition, the interface for the Assistant indicates visually the progress made so far. The icons representing properties, methods, and parameters in the assistant indicate the following:

- Which parts of the source object will be used:
 - Methods that will be exported
 - Methods that will be not be exported
- Which parts of the source object are completed:
 - Methods that are fully specified
 - Methods requiring further information









These two aspects of the state of the source object are independent from one another. For example, a method might be fully specified, but Export could be False because you want to test the exported object without generating the method. You might want to set Export to True for several methods, fill out the information for only some of them, and then create the exported object. The wrappers would be generated only for those items that have Export set to true and are fully specified.

Note

Changing the export or completion status of a parameter could affect the status of the method containing the parameter.

To make the process as simple as possible, default values and behaviors have been specified when possible. For example, all methods start out with Export set to False, but as soon as you specify information, such as a return type on the method, the value of the Export property is set to True. (For more information, see [“About the Export Property”](#) on page 165.)

The project tree icons indicate the status of an item. The icons shown below represent all of the permutations of the states described above.

Meaning	Icons
Export is False, Incomplete (initial default)	<ul style="list-style-type: none">• Method: • Property or parameter: 
Export is True, Incomplete	<ul style="list-style-type: none">• Method: • Property or parameter: 
Export is False, Fully Specified	<ul style="list-style-type: none">• Method: • Property or parameter: 
Export is True, Fully Specified	<ul style="list-style-type: none">• Method: • Property or parameter: 

Specifying Information for Exporting

When the Assistant creates a new project, it supplies default values for the attributes that must be specified. Most of these values are set to UNSPECIFIED to indicate that you must modify this attribute. Some attributes do not have a default value because there is no reasonable one; also, supplying a default value could cause the wrappers to be built with incorrect values.

The one value that is set by default in most cases is the Convert Majority flag, used if the value is an array. The default setting for this attribute (True) provides the most expected behavior. For more information, see [“Converting Array Majority”](#) on page 165.

Note that in the IDL language, parameters are optional, so the Assistant does not require the user to export every parameter that is retrieved from the IDL source object and presented in the Assistant. It is up to the user to decide which parameters should be exported. This might require defensive programming in the IDL source object to ensure that parameters are not used if they are not supplied.

Information that can be specified includes:

- [“Bridge Information”](#) on page 167 — defines general wrapper object information, the output directory, package name (Java only) and GUIDS settings (COM only)
- [“Source Object Information”](#) on page 170 — indicates whether the object is drawable or not (this cannot be changed)
- [“Property Information”](#) on page 171 — defines the property data type, whether it is an array (and the array majority if it is), and whether or not it is to be exported
- [“Method Information”](#) on page 173 — defines the export characteristics of a procedure or function method, and defines the return value data type and array characteristics if the method is a function
- [“Parameter Information”](#) on page 176 — defines the mutability, data type, array characteristics and export selection for method parameters

Note

See [“Parameter Passing and Type Conversion”](#) on page 136 for important information about passing objects, arrays and variables as parameters.

About the Export Property

The first (and only the first time) any attribute of a property, method, or parameter other than Export is set, the item has its Export property set to True. This behavior is provided as a convenience.

Converting Array Majority

The Convert Majority property may be an option for a property, function return value or method parameter that is defined as an array (the Array property is True). The rules for the Convert Majority property vary depending on destination (COM or Java) and whether the array is a property value, function return value or method parameter. The settings and default values are described in [Table 7-3](#).

Where to Specify	Can Specify in COM?	Can Specify in Java?
Get Property	No (arrays always converted)	Yes (default is to convert)
Set Property	Yes (default is to convert)	Yes (default is to convert)
Function return values	No (arrays always converted)	Yes (default is to convert)
Procedure parameters	Yes (default is to convert)	Yes (default is to convert)

Table 7-3: Rules for Specifying the Convert Majority Property

See the following for more information on these rules:

- [Table 7-8](#) in “[Property Information](#)” on page 171
- [Table 7-10](#) in “[Method Information](#)” on page 173
- [Table 7-11](#) in “[Parameter Information](#)” on page 176

Note

For COM wrappers, you could theoretically set the Convert Majority flag for the property setting call but not the property retrieval call. In practice, the Assistant uses one flag to control the Convert Majority setting for both Get and Set Property, and so for COM, the setting for properties is always Convert Majority, which is set to True and disabled.

For more information on array majority, see “[Multidimensional Array Storage and Access](#)” on page 493. Also see “[Array Order Conversion](#)” on page 137.

Supported Data Types

The following data types are supported with the Export Bridge technology.

COM	<ul style="list-style-type: none">• Unsigned char• Char• Short• Unsigned short• Long• Unsigned long• LONGLONG	<ul style="list-style-type: none">• ULONGLONG• Float• Double• BSTR• IUnknown*• VARIANT
Java	<ul style="list-style-type: none">• JIDLNumber• JIDLObjectI• JIDLString	

Note — See [Appendix A, “IDL Java Object API”](#) for information on JIDL* objects.

Bridge Information

The project has general information about the bridge being used (COM or Java).

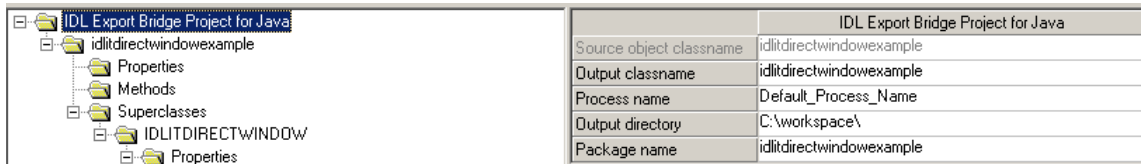


Figure 7-8: The Export Bridge Assistant: General Bridge Information

Table 7-5 describes the general bridge information's properties and values

Property	Value
Name	Defaults to "IDL Export Bridge Project for <dest>" where <dest> is "COM" or "Java." Displayed in the sheet header only, it is distinct from the project filename and source object classname and cannot be modified.
Source object classname	Specified by selection of IDL object definition file. The file must be selected, rather than specification of the object by name only. Because this value is obtained from the source object filename, the capitalization is the same as the filename.
Output classname	Defaults to IDL source object class name; must be non-null and a valid IDL identifier. Because this value is obtained from the source object filename, the capitalization is the same as the filename.
Process name	Defaults to 'Default_Process_Name'; must be non-null and valid IDL identifier
Output directory	Defaults to location of source object file (.pro or .sav); independent from location of project file and source file (except for initial default to source location).

Table 7-4: General Bridge Information's Properties

The other properties displayed for the project depends on which bridge it is using: COM or Java. The following describes COM-specific values.

IDL Export Bridge Project for COM	
Source object classname	idlexbrtest1
Output classname	idlexbrtest1
Process name	Default_Process_Name
Output directory	C:\workspace\idl63\bridge\wizard\test
Regenerate GUIDs	False
Explicit CreateObject	True

Figure 7-9: The Export Bridge Assistant: COM Bridge Information

Property	Value
Regenerate GUIDs	On the first build operation, GUIDS are always generated, so this property is desensitized until after the first build. On subsequent builds, if Regenerate GUIDs is False, the existing GUIDs are used, allowing a developer to use the newly built object without re-registering. If Regenerate GUIDs is true, new GUIDS will be created during a build operation. Any time GUIDs are regenerated during a build operation. they are saved and can be used by setting Regenerate GUIDs to False.
Explicit CreateObject	By default, graphical COM objects (ActiveX objects) call the createObject method automatically when the control is created. If this property is set to True, the automatic call to the createObject method is removed; the client code must then explicitly call one of the CreateObject or CreateObjectEx methods to create the IDL process. This property is disabled for nongraphical COM objects, which always require an explicit call to one of the CreateObject methods.

Table 7-5: COM Bridge Information's Property Values

The following describes Java-specific properties and values.

IDL Export Bridge Project for Java	
Source object classname	idlexbrtest1
Output classname	idlexbrtest1
Process name	Default_Process_Name
Output directory	C:\workspace\idl63\bridge\wizard\test
Package name	idlexbrtest1

Figure 7-10: The Export Bridge Assistant: Java Bridge Information

Property	Value
Package name	<p>Defaults to source object class name. Because this value is obtained from the source object filename, the capitalization is the same as the filename.</p> <p>This property is optional and can be blank. If blank, the Java file and class file will be created in the output directory. If not blank, the name is used to create one or more subdirectories below the output directory. Period characters are separator characters that produce a directory hierarchy in the resulting subdirectory for the result. Each segment between period characters must be a valid identifier.</p>

Table 7-6: Java Bridge Information's Property Values

Source Object Information

The source object for which you are making a wrapper has its own set of properties.

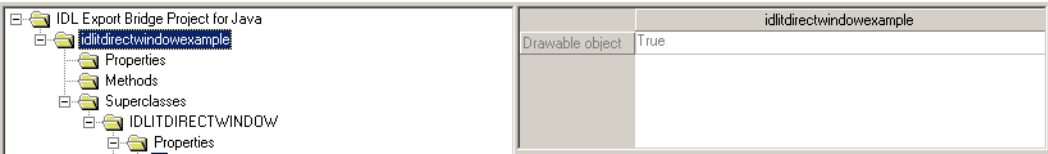


Figure 7-11: The Export Bridge Assistant: Source Object Information

Table 7-7 describes the characteristics of the source object.

Property	Value
Name	Name of this IDL source object; specified when project was created and shown in the sheet header only
Drawable object	True if IDL source object is a subclass of IDLitWindow, IDLgrWindow, or IDLitDirectWindow; otherwise False.

Table 7-7: Source Object Information’s Property Values

Property Information

Each property of the source object has its own set of properties.

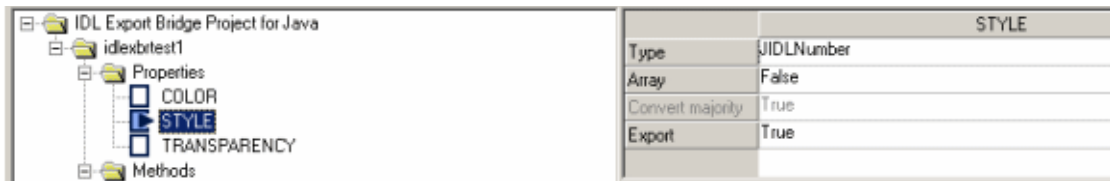


Figure 7-12: The Export Bridge Assistant: Source Object Properties

Table 7-8 describes the properties and values of the source object's properties.

Property	Value
Name	Name of the object's property; shown in the sheet header only.
Type	One of the types supported by the Export Bridge technology. For the list, see “Supported Data Types” on page 166.
Array	Indicates if property is of type array: True if it is, False otherwise. If True, Convert Majority is sensitive.
Convert Majority	Sensitive only if Array is True. Set to True if the property value is an array and needs to be converted when setting the property. (For COM, when retrieving a property value, the majority is always converted regardless of this attribute setting.) The default value for both COM and Java is True. For more information, see “Converting Array Majority” on page 165.
Export	Indicates if the Assistant will export this property: True if it will, False otherwise.

Table 7-8: Source Object Property Information's Property Values

About Property Extraction

The object properties are extracted from the IDL source object by compiling the list of all keywords on either or both of the SetProperty and GetProperty methods of the object.

The following factors are *not* used to determine source object properties:

- Whether a property is registered or not (the export bridges do not require that an object uses the component framework)
- The presence of a member variable in the source object's definition structure
- Keywords to the object's Init method

Note that properties of built-in superclasses are not extracted (see [“Exporting a Source Object’s Superclasses”](#) on page 180). To obtain wrapper routines to get or set a superclass property, you must add an explicit property handler to your SetProperty and/or GetProperty methods for the superclass property.

Method Information

Each method of the source object has its own set of properties. [Figure 7-13](#) displays a procedure's property information. [Figure 7-14](#) displays the property information for a function.

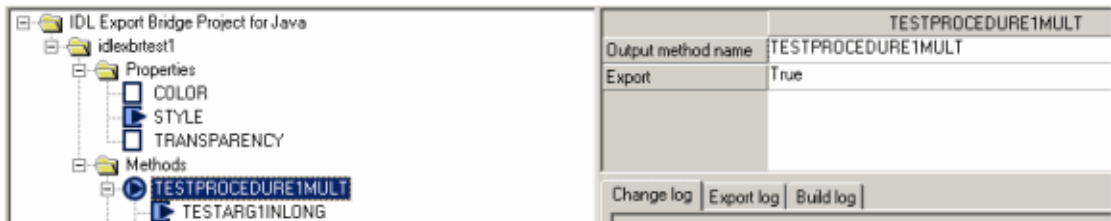


Figure 7-13: The Export Bridge Assistant: Procedure Information

[Table 7-9](#) describes the procedure information's properties and values.

Property	Value
Name	Name of the procedure; shown in the sheet header only.
Output Method Name	The name of the wrapper method in the wrapper object. Defaults to the method name obtained from the source object, but can be changed to reflect native platform naming conventions and case. Regardless of the output method name, the wrapper method will call through the Export Bridge technology layers to the original source object method name in the IDL object.
Export	Indicates if the Assistant will export this property: True if it will, False otherwise.

Table 7-9: Procedure Information's Property Values

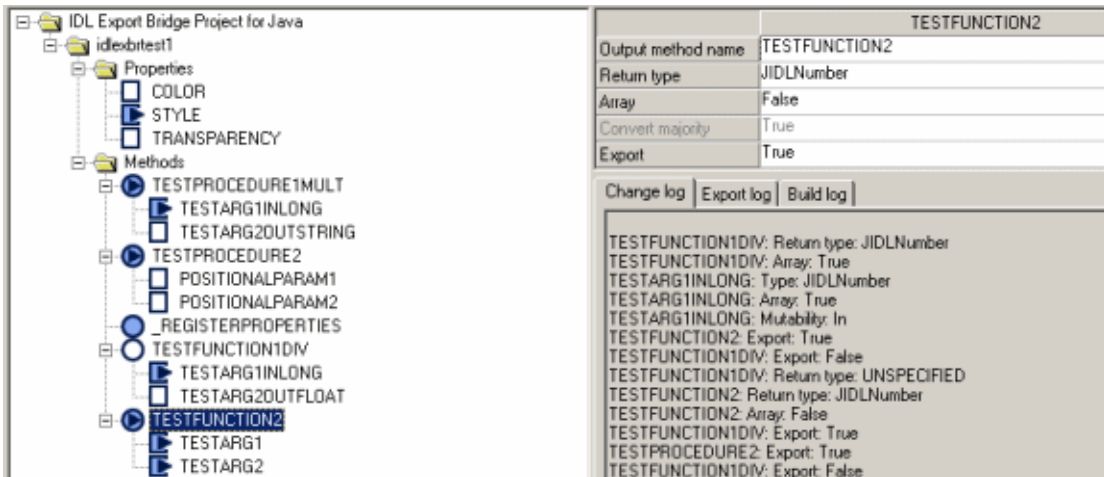


Figure 7-14: The Export Bridge Assistant: Function Information

Table 7-10 describes the function information's properties and values. In addition to the values that can be specified for procedure methods (Table 7-9), the following can also be defined.

Property	Value
Return Type	One of the types supported by the Export Bridge technology. For the list, see “Supported Data Types” on page 166.
Array	Indicates if property is of type array: True if it is, False otherwise. If True and the destination is Java, Convert Majority is sensitive.

Table 7-10: Function Information's Property Values

Property	Value
Convert Majority	<p>Sensitive only if Array is True and the destination is Java. Set to True if the property value is an array and needs to be converted when setting the property. (For COM, when retrieving a property value, the majority is always converted regardless of this setting, which is why this property does not appear with COM.) The default value is True.</p> <p>For more information, see “Converting Array Majority” on page 165.</p>

Table 7-10: Function Information’s Property Values (Continued)

Parameter Information

Each parameter of the source object's methods has its own set of properties.

Note

If a method parameter has its Export property set to True, all parameters of the method to the left of the current parameter are marked for export as well so as to not leave holes in the parameters list and cause parameters to be out of sequence.

If a parameter has its Export property set to False, all parameters to the right will also have their Export property set to False. If a parameter has its Export property set to True, the parent method will have its Export property set to True.

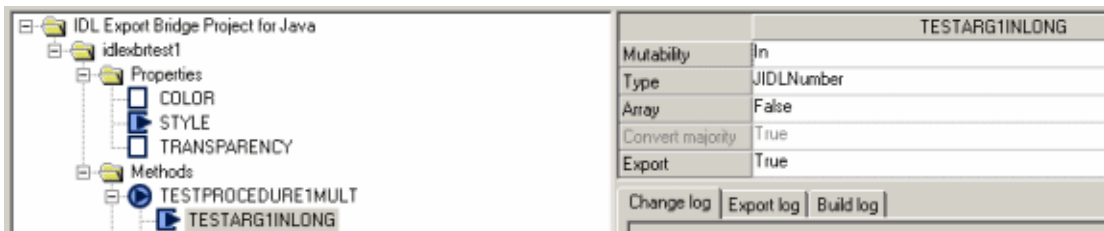


Figure 7-15: The Export Bridge Assistant: Parameter Information

Table 7-11 describes the parameter information's properties and values.

Property	Value
Name	Name of the procedure; shown in the sheet header only.
Mutability	Either In or In/Out. Use In for parameters that are constant (In-Only, meaning that their values cannot be changed). Use In/Out for parameters that are not constant and require a value to be passed back to the caller (can be In/Out or Out-only).
Return Type	One of the types supported by the Export Bridge technology. For the list, see “Supported Data Types” on page 166.
Array	Indicates if property is of type array: True if it is, False otherwise. If True, Convert Majority is sensitive.

Table 7-11: Parameter Information's Property Values

Property	Value
Convert Majority	<p>Sensitive only if Array is True. Set to True if the property value is an array and needs to be converted when setting the property. The default value is True.</p> <p>For more information, see “Converting Array Majority” on page 165.</p>
Export	<p>Indicates if the Assistant will export this property: True if it will, False otherwise.</p>

Table 7-11: Parameter Information's Property Values (Continued)

Information Skipped During Export

The Assistant skips certain information when creating an object specification for exporting because such information is unnecessary or unavailable for a wrapper object.

Lifecycle Methods

The lifecycle methods of the IDL source object, `Init` and `Cleanup`, are not presented in the list of methods to export in the Assistant. These methods are called through the bridge when the wrapper object stock methods `createObject` and `destroyObject` are called. (Note that Java capitalization is used here, COM names are different.) It is not useful to export a wrapper method explicitly for either of these routines.

For information on the stock methods, see [“Stock COM Wrapper Methods”](#) on page 192 (COM) and [“Stock Java Wrapper Methods”](#) on page 218 (Java).

Get Property and Set Property Methods

The `GetProperty` and `SetProperty` methods of the IDL source object are not presented in the list of methods to export in the Assistant. These methods will be called through the Export Bridge when the wrapper object routines for setting or retrieving a specific property are called. It is not useful to export a wrapper method explicitly for either of these routines.

Drawable Object Event Handlers

For drawable objects (objects subclassed from `IDLitWindow`, `IDLgrWindow`, or `IDLitDirectWindow`) as well as `IDLitDirectWindow` superclass itself, the following methods are not typically needed in the exported object:

<ul style="list-style-type: none">• <code>OnEnter</code>• <code>OnExit</code>• <code>OnExpose</code>• <code>OnKeyboard</code>	<ul style="list-style-type: none">• <code>OnMouseDown</code>• <code>OnMouseMove</code>• <code>OnMouseUp</code>• <code>OnResize</code>
--	--

By default these methods are not presented in the Assistant for export from either the original IDL source object or its superclasses.

These routines in the source object are called directly by the Export Bridge when events are being handled, and so they are typically not needed in the exported object.

Exporting these routines would be unnecessary and confusing to most users since they might assume that the methods in the exported object would be called, but under default conditions they are unused. The sophisticated user might actually want to call these in the client application, however, and so they can be presented in the assistant by starting the application with the `DRAWABLE_EVENTHANDLERS` keyword set (in addition to the `OBJECT_FILE` keyword). See [“Running the Assistant”](#) on page 149 for details.

Typically, the methods found in `.pro` code object definition files will appear in the Export Bridge Assistant. Since `IDLgrWindow` and `IDLitWindow` object definition files are built-in, they do not appear as superclasses and their methods are not presented in the Assistant.

Exporting a Source Object's Superclasses

You might want to set properties or call methods that are implemented in the superclass of the source object. The Assistant interrogates the IDL source object to obtain the properties, methods and method parameters. It also uses the `OBJ_CLASS` method to obtain the superclasses of the source object class, and for each user class also obtains the properties, methods and method parameters. This is a recursive process that requires interrogating each superclass for its superclasses. Built-in IDL superclasses will not be included in the wrapper definition.

The routine used to extract object information, IDL's `ROUTINE_INFO` function, can obtain the methods of a built-in object class. However, because `ROUTINE_INFO` does not provide parameter information for built-in routines, the Assistant is unable to extract the properties (parameters to `SetProperty` or `GetProperty`) or the parameters of object methods for a built-in superclass. The built-in superclasses are not presented in the project tree view.

To obtain wrapper routines to modify properties of built-in superclasses, you must add an explicit property handler to the `SetProperty` and/or `GetProperty` methods for the superclass property. To obtain wrapper routines to call methods of built-in superclasses, you must add an explicit method to their source object that calls the superclass method.

Modifying a Source Object After Export

Modifications to the IDL source object can affect the operation of an existing wrapper object even if the wrapper is not rebuilt because the wrapper object uses the source object in its current state, not a state cached at the time the Assistant generates the wrapper object.

In general:

- Adding properties or methods has no impact on an existing wrapper object.
- Removing properties or methods or changing method interfaces can invalidate an existing wrapper object.
- Modifying behavior in a property handler or method causes the new behavior to be in effect for the next invocation of the application using the wrapper client. This can be useful because the wrapper does not need to be regenerated for the client to pick up IDL source modifications.

Wrapper Generation Example

The following example exports a simple IDL object that has no properties or methods and demonstrates the configuration necessary to initialize a COM or Java client application to use the exported object. First, create the IDL source object.

1. Create a file named `helloworld__define.pro` (within your IDL path) containing the following code:

```
FUNCTION helloworld::INIT
    RETURN, 1
END

PRO helloworld__define
    struct = {helloworld, $
        dummy:0b $; dummy structure field, not a property
    }
END
```

This is the source object definition file that you will export using the Export Bridge Assistant.

2. Open the Assistant by entering `IDLEXBR_ASSISTANT` at the command line.

See one of the following:

- [“COM Wrapper Object Generation and Use”](#) below
- [“Java Wrapper Object Generation and Use”](#) on page 184

COM Wrapper Object Generation and Use

The following example exports and uses the `helloworld` object in a simple Visual Basic .NET console application. After creating the object definition file and launching the Assistant as described in [“Wrapper Generation Example”](#) on page 182, complete the following steps.

1. Select to create a COM export object by selecting **File** → **New Project** → **COM** and browse to select the `helloworld__define.pro` file. Click **Open** to load the file into the Export Assistant.

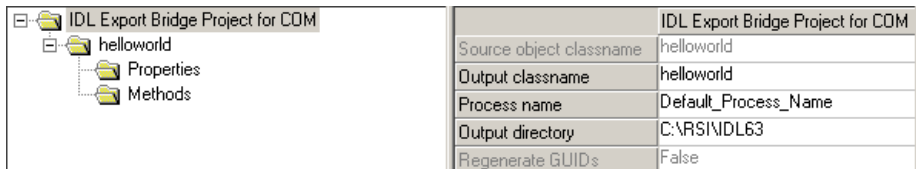


Figure 7-16: Helloworld COM Export Project

- The top-level project entry in the left-hand tree panel is selected by default. There is no need to modify the default properties shown in the right-hand property panel, but you can enter different values if desired. Select the tree view item listed in the left column to configure the related properties in the right column.

Tree View Item	Parameter Configuration
IDL Export Bridge Project	Accept the default value or make changes as desired: <ul style="list-style-type: none"> • Output classname • Process name • Output directory
helloworld	Drawable object equals False

Table 7-12: Example Export Object Parameters

For this simple example, the source object has no properties or methods, so none are exported.

Note

See [“Specifying Information for Exporting”](#) on page 164 for details on configuring export values.

- Save the project by selecting **File** → **Save project**. Accept the default name and location or make changes as desired.
- Build the export object by selecting **Build** → **Build object**. The **Build log** panel shows the results of the build process. For a nondrawable object, `.t1b`

and .dll files (named based on the object name) are created in the **Output directory**.

5. Register the .dll using `regsvr32 helloworld.dll`. See [“COM Registration Requirements”](#) on page 143 for details if needed.
6. Create a new Visual Basic .NET console application and add a reference to the COM library named `helloworldLib 1.0 Type Library`. Select **Project → Add Reference**, and click on the **COM** tab. Select the `helloworld.dll` and click **Ok**.
7. Replace the default module code with the following text:

```
Imports helloworldLib
Module Module1
    Dim oHello As New helloworldLib.helloworldClass
    Sub Main()
        Try
            oHello.CreateObject(0, 0, 0)
            Catch ex As Exception
                Console.WriteLine(oHello.GetLastError())
            Return
        End Try
        AddHandler oHello.OnIDLOutput, AddressOf evOutput
        oHello.ExecuteString("Print, 'Hello World'")
    End Sub
    Sub evOutput(ByVal ss As String)
        Console.WriteLine(ss)
    End Sub
End Module
```

In this example, the stock [ExecuteString](#) method is used to print the hello world message. By adding a handler for the `OnIDLOutput` method, the console application is able to capture and output the information that would typically be printed to the Output window of IDL. After building the solution and starting without debugging, the console window appears with the output messages.

Java Wrapper Object Generation and Use

The following example exports and uses the `helloworld` object in a simple Java application. After creating the object definition file and launching the Assistant as described in [“Wrapper Generation Example”](#) on page 182, complete the following steps.

1. Select to create a Java export object by selecting **File** → **New Project** → **Java** and browse to select the `helloworld__define.pro` file. Click **Open** to load the file into the Export Assistant.

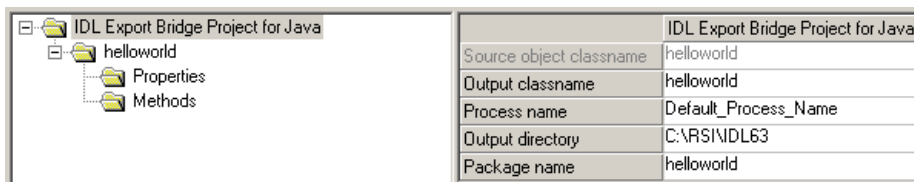


Figure 7-17: Helloworld Java Export Project

2. The top-level project entry in the left-hand tree panel is selected by default. There is no need to modify the default properties shown in the right-hand property panel, but you can enter different values if desired. Select the tree view item listed in the left column to configure the related properties in the right column.

Tree View Item	Parameter Configuration
IDL Export Bridge Project	Accept the default value or make changes: <ul style="list-style-type: none"> • Output classname • Process name • Output directory (paths in later parts of this example assume this field equals the main IDL installation directory, which is typically <code>C:\ITT\IDLxx</code> on Windows)
helloworld	Drawable object equals False

Table 7-13: Example Export Object Parameters

For this simple example, the source object has no properties or methods, so none are exported.

Note

See “[Specifying Information for Exporting](#)” on page 164 for details on configuring export values.

3. Save the project by selecting **File** → **Save project**. Accept the default name and location or make changes as desired.
4. Build the export object by selecting **Build** → **Build object**. The **Build log** panel shows the results of the build process. A subdirectory, named `helloworld` (based on the object name), contains the `.java` and `.class` files, and is located in the **Output directory**.
5. Create a file named `helloworld_example.java` that contains the following code and save the file in the `helloworld` directory.

```
package helloworld;
import com.idl.javaidl.*;
public class helloworld_example extends helloworld
implements JIDLOutputListener
{
    private helloworld hwObj;

    // Constructor
    public helloworld_example() {
        hwObj = new helloworld();
        hwObj.createObject();
        hwObj.addIDLOutputListener(this);
        hwObj.executeString("print, 'Hello World'");
    }

    // implement JIDLOutputListener
    public void IDLOutput(JIDLObjectI obj, String sMessage) {
        System.out.println("IDL: "+sMessage);
    }

    public static void main(String[] argv) {
        helloworld_example example = new helloworld_example();
    }
}
```

Note

By default, the Assistant generates a package so any Java routine using an exported wrapper object must include the package name. The second statement, `import com.idl.javaidl.*;` is also required.

For example purposes, the stock method `executeString` is called, and an output listener is registered to retrieve the IDL output.

The wrapper is compiled and run using the commands below:

- “[Windows Commands to Build and Run the Client](#)” on page 187
- “[UNIX Commands to Build and Run the Client](#)” on page 188

Windows Commands to Build and Run the Client

The following commands build and run this Java wrapper example on Windows.

1. To compile and run the Java routine, open the Windows Command window by selecting **Start** → **Run** and enter `cmd` in the textbox.
2. Use the `cd` command to change to the directory containing the `helloworld` directory. For a default Windows installation, the command would be similar to the following:

```
cd C:\ITT\IDL63
```

3. Reference the classpath of `javaidl.b.jar` in the compile statement. Enter the following commands (each as a single line), replacing `IDL_DIR` with the IDL installation directory, for example `ITT\IDL63`:

```
javac -classpath  
    ".;IDL_DIR\resource\bridges\export\java\javaidl.b.jar"  
    helloworld\helloworld_example.java  
java -classpath  
    ".;IDL_DIR\resource\bridges\export\java\javaidl.b.jar"  
    helloworld.helloworld_example
```

In both commands, the `.` character includes the current directory in the classpath.

The first command uses `javac` to compile the example client. The path to the `helloworld_example.java` file is specified using a backslash character as a directory separator.

The second command uses `java` to run the example client. The final argument specifies the package path to the `helloworld_example` class file. Note that a `.` character is used as a separator in the package path. The final argument to the second command intentionally omits the suffix.

After compiling and running the project, the output message will appear in the command window.

UNIX Commands to Build and Run the Client

The following commands build and run this Java wrapper example on UNIX:

```
source IDL_DIR/bin/bridge_setup
javac helloworld/helloworld_example.java
java helloworld.helloworld_example
```

Note

See [“Java Requirements”](#) on page 143 for more information on the `bridge_setup` file.

The `source` command adds the necessary directories to the dynamic library path and the classpath.

The second command uses `javac` to compile the example client. The third command uses `java` to run the example client. The final argument specifies the package path to the `helloworld_example.class` file. Note that a `.` character is used as a separator in the package path. The final argument to the second command intentionally omits the suffix.

After compiling and running the project, the output message will appear.



Chapter 8

Using Exported COM Objects

This chapter discusses the following topics.

Overview of COM Export Objects	190	Event Handling	208
COM Wrapper Objects	191	Error Handling	211
Stock COM Wrapper Methods	192	Debugging	213

Overview of COM Export Objects

Once you have chosen to use a connector object or have exported a custom IDL source object using the Assistant, use the method and event reference information described here to create an instance of the object and interact with the IDL process from an external COM environment.

This chapter presents important background information on using IDL objects exported into COM:

- [“COM Wrapper Objects”](#) on page 191
- [“Stock COM Wrapper Methods”](#) on page 192
- [“Event Handling”](#) on page 208
- [“Error Handling”](#) on page 211

For examples that use the methods and events described here, see:

- [“Using the Connector Object”](#) on page 245, which describes how to use the connector object in COM environments
- [“Creating Custom COM Export Objects”](#) on page 269, which provides examples of using custom object methods (in addition to the stock wrapper methods) in COM environments

COM Wrapper Objects

A *COM wrapper object* is defined as one that wraps a nondrawable IDL object, and an *ActiveX control* as one that wraps a drawable IDL object. Typically, only ActiveX controls handle (user) events, but COM wrapper objects can also fire events so that the client can receive IDL output and notifications.

To use a COM wrapper object, the client instantiates one or more instances of the wrapper objects and then calls its methods and properties. An ActiveX control must be created in a host window before its methods and properties can be called.

ActiveX controls are typically hosted on GUI forms. These forms are generally built in a GUI-based development environment such as Visual Basic or Visual Studio .NET. The user creates a form by dragging and dropping controls onto the form. ActiveX controls usually interrogate the host window to determine what user mode they are in: design or runtime. While in design mode, the ActiveX control usually displays a static image whereas in runtime mode, the ActiveX control is executing and dynamically drawing to the screen.

The Export Bridge ActiveX wrapper controls also check for the user mode. In design mode, a static image with the IDL Export Bridge logo is displayed. In runtime mode, the ActiveX control internally calls the `CreateObject` method, the underlying IDL object is created, and IDL begins rendering to the ActiveX window. When the application is stopped and transitioned back to design mode, the ActiveX control internally calls the `DestroyObject` method, and the static image is once again displayed. See [“Stock COM Wrapper Methods”](#) on page 192 for information on these methods.

Note

Not all ActiveX host windows provide the user mode. If the host window does not provide the user mode, the Export Bridge ActiveX wrapper controls assume that they are in runtime mode, and they immediately begin to render to the screen as soon as they are instantiated.

Stock COM Wrapper Methods

This section describes the stock methods in the COM wrapper objects created by the Export Bridge Assistant:

- [“Abort”](#) on page 193
- [“CreateObject”](#) on page 194
- [“CreateObjectEx”](#) on page 196
- [“DestroyObject”](#) on page 199
- [“ExecuteString”](#) on page 200
- [“GetIDLObjectClassName”](#) on page 201
- [“GetIDLObjectVariableName”](#) on page 202
- [“GetIDLVariable”](#) on page 203
- [“GetLastError”](#) on page 204
- [“GetProcessName”](#) on page 205
- [“SetIDLVariable”](#) on page 206
- [“SetProcessName”](#) on page 207

Every connector object and custom COM wrapper object has these methods in addition to those defined by the wrapped IDL object.

Abort

The Abort method requests that the IDL process containing the underlying IDL object abort its current activity. This method is useful if a given IDL method call is busy for a very long time (e.g., a very long image processing command).

Note

The request is only that, a request, and IDL might take a long time before it actually stops or might completely finish its current activity. Such a wait is an effect of the IDL interpreter.

The client can only abort the current IDL activity if that wrapper object is the current owner of the underlying IDL process.

Syntax

HRESULT Abort(void)

Parameters

None

CreateObject

The CreateObject method creates the actual underlying IDL object. The *argc*, *argv*, and *argpal* parameters are used to supply parameters to the underlying IDL object's Init method. If the Init method does not have any parameters, the caller sets *argc*, *argv*, and *argpal* to 0, NULL, and NULL, respectively.

This method creates IDL objects that use a default licensing algorithm (see “[IDL Licensing Modes](#)” on page 134 for details). To use a specific IDL licensing mode, use the [CreateObjectEx](#) method.

Note

By default, ActiveX controls call the CreateObject method implicitly. In an ActiveX control, calls to the CreateObject method in client code will be ignored if the **Explicit CreateObject** property in the Export Bridge Assistant project was set to False when the ActiveX control was built.

Syntax

```
HRESULT CreateObject ([in] int argc, [in] VARIANT argv, [in] VARIANT argpal)
```

Parameters

argc

An integer that specifies the number of elements in the *argv* and *argpal* arrays.

argv

A VARIANT containing a COM SafeArray of VARIANT types, one for each parameter to Init. The elements in the array are given in order of the parameters listed in Init, ordered from left to right.

argpal

A VARIANT containing a COM SafeArray of 32-bit integer flag values, which can be a combination of the IDLBML_PARMFLAG_CONST and IDLBML_PARMFLAG_CONVMAJORITY values ORed together. The latter value is only used when an *argv* element is an array itself. For parameters that are not arrays, the *argpal*[*n*] value must be 0.

The following constant values defined in the typlib information of a wrapped IDL object can be used:

IDLBML_PARMFLAG_CONST	Use for parameters that are constant (In-Only, meaning that their values cannot be changed).
IDLBML_PARMFLAG_CONVMAJORITY	Include if the property value is an array. For more information, see “Converting Array Majority” on page 165.

Example

Note

See [Appendix B, “COM Object Creation”](#) for examples of creating objects from a variety of COM programming languages.

The Init method of the IDL object being wrapped has the following signature:

```
PRO IDLexFoo::INIT, rect, filename
```

where `rect` is an array of 4 integers and `filename` is a string.

The COM client code that creates an instance of the wrapper object, and calls the `CreateObject()` method with the `rect` and `filename` parameters, would look like the following:

```
CComSafeArray<int> csa(4);
csa[0] = 0; csa[1] = 0; csa[2] = 5; csa[3] = 10;

CComVariant argv[2];
int          argp[2];

argv[0] = csa.Detach();
argp[0] = IDLBML_PARMFLAG_CONST;
argv[1] = "someFilename.txt";
argp[1] = IDLBML_PARMFLAG_CONST;

CComPtr<IMyWrapper> spWrapper;
spWrapper.CoCreateInstance(__uuidof(MyWrapper));

spWrapper->CreateObject(2, argv, argp);
```

CreateObjectEx

The `CreateObjectEx` method creates the actual underlying IDL object; it differs from the `CreateObject` method in that it allows the specification of flag values that control the way the IDL process is initialized. The *argc*, *argv*, and *argpal* parameters are used to supply parameters to the underlying IDL object's `Init` method. If the `Init` method does not have any parameters, the caller sets *argc*, *argv*, and *argpal* to 0, `NULL`, and `NULL`, respectively. The *flags* parameter specifies one or more initialization flags governing the way the IDL process is initialized; currently, the available flags control the method used to license the IDL session. (See “[IDL Licensing Modes](#)” on page 134 for details on the default licensing mechanism.)

Note

By default, ActiveX controls call the `CreateObject` method implicitly. In an ActiveX control, calls to the `CreateObject` method in client code will be ignored if the **Explicit CreateObject** property in the Export Bridge Assistant project was set to `False` when the ActiveX control was built.

Syntax

```
HRESULT CreateObjectEx ([in] int argc, [in] VARIANT argv, [in] VARIANT  
argpal, [in] long flags))
```

Parameters

argc

An integer that specifies the number of elements in the *argv* and *argpal* arrays.

argv

A `VARIANT` containing a COM `SafeArray` of `VARIANT` types, one for each parameter to `Init`. The elements in the array are given in order of the parameters listed in `Init`, ordered from left to right.

argpal

A `VARIANT` containing a COM `SafeArray` of 32-bit integer flag values, which can be a combination of the `IDLBML_PARMFLAG_CONST` and `IDLBML_PARMFLAG_CONVMAJORITY` values ORed together. The latter value is only used when an *argv* element is an array itself. For parameters that are not arrays, the *argpal*[*n*] value must be 0.

The following constant values defined in the typlib information of a wrapped IDL object can be used:

IDLBML_PARMFLAG_CONST	Use for parameters that are constant (In-Only, meaning that their values cannot be changed).
IDLBML_PARMFLAG_CONVMAJORITY	Include if the property value is an array. For more information, see “Converting Array Majority” on page 165.

flags

Flag values that control the way the IDL process is initialized. The following constant values defined in the typlib information of a wrapped IDL object can be used:

IDLBML_LIC_FULL	The application requires that a licensed copy of IDL be installed on the local machine. If IDL is installed but no license is available, the application will run in IDL Demo (7-minute) mode.
IDLBML_LIC_LICENSED_SAV	The application looks for an embedded license in the save file being restored.
IDLBML_LIC_RUNTIME	The application looks for a runtime IDL license. If no runtime license is available, the application will run in Virtual Machine mode.
IDLBML_LIC_VM	The application will run in Virtual Machine mode.

Example

Note

See [Appendix B, “COM Object Creation”](#) for examples of creating objects from a variety of COM programming languages.

The Init method of the IDL object being wrapped has the following signature:

```
PRO IDLexFoo::INIT, rect, filename
```

where `rect` is an array of 4 integers and `filename` is a string.

The COM client code that creates an instance of the wrapper object and calls the `CreateObjectEx()` method with the `rect` and `filename` parameters, and which explicitly specifies that it should run in IDL Virtual Machine mode, would look like the following:

```
CComSafeArray<int> csa(4);
csa[0] = 0; csa[1] = 0; csa[2] = 5; csa[3] = 10;

CComVariant argv[2];
int          argp[2];

argv[0] = csa.Detach();
argp[0] = IDLBML_PARMFLAG_CONST;
argv[1] = "someFilename.txt";
argp[1] = IDLBML_PARMFLAG_CONST;

CComPtr<IMyWrapper> spWrapper;
spWrapper.CoCreateInstance(__uuidof(MyWrapper));

spWrapper.CreateObjectEx(2, argv, argp, IDLBML_LIC_VM);
```

DestroyObject

The DestroyObject method destroys the underlying IDL object. If the object being destroyed is the last object within an OPS process, the OPS process is also destroyed.

Note

Trying to re-create an object after it has been destroyed is not supported. You must re-define the variable and then re-create the object.

Syntax

HRESULT DestroyObject(void)

Parameters

None

ExecuteString

The ExecuteString method executes the specified command in the IDL process containing the underlying IDL object.

Note

This method is disabled for applications running in the IDL Virtual Machine.

Syntax

```
HRESULT ExecuteString([in] BSTR bstrCmd)
```

Parameters

bstrCmd

A string containing the IDL command to be executed.

Examples

See [“IDL Command Line with a COM Connector Object”](#) on page 252 for an example that executes any IDL command entered into one textbox and writes IDL output or error information to a second textbox.

GetIDLObjectClassName

The GetIDLObjectClassName method returns the IDL class name of the underlying IDL object.

Syntax

```
HRESULT GetIDLObjectClassName([out,retval] BSTR* Name)
```

Return Value

A string containing the class name of the IDL object.

GetIDLObjectVariableName

When the underlying IDL object was created in the IDL process, it was assigned a variable name. The GetIDLObjectVariableName method returns that name.

Syntax

```
HRESULT GetIDLObjectVariableName([out,retval] BSTR* Name)
```

Return Value

A string containing the variable name of the IDL object.

GetIDLVariable

The GetIDLVariable method retrieves a named variable from the IDL process containing the underlying IDL object.

Note

This method is disabled for applications running in the IDL Virtual Machine.

Syntax

```
HRESULT GetIDLVariable([in] BSTR bstrVar, [out,retval] VARIANT* Value)
```

Parameters

bstrVar

A string containing the name of the variable to retrieve from the IDL process.

Return Value

The variable's values. If the variable is an array, the array is always converted from IDL majority to the standard COM SAFEARRAY majority ordering.

Examples

See [“Data Manipulation with a COM Connector Object”](#) on page 251 for an array manipulation example that uses the GetIDLVariable, SetIDLVariable and ExecuteString methods.

GetLastError

The GetLastError method gets the error string for the last error that has occurred. It is called after a method call returns an error. The returned error string is usually the actual IDL error message, if IDL generated the error message.

Syntax

```
HRESULT GetLastError([out,retval] BSTR* LastError)
```

Return Value

The error string for the last error that occurred.

GetProcessName

The GetProcessName method returns the name of the process that contains the underlying IDL object.

Syntax

```
HRESULT GetProcessName([out,retval] BSTR* Name)
```

Return Value

A string containing the name of the process that contains the IDL object.

SetIDLVariable

The SetIDLVariable method sets the specified variable name to the specified value in the IDL process containing the underlying IDL object. If the value is a SAFEARRAY, it is always converted from the standard COM SAFEARRAY majority ordering to IDL majority.

Note

This method is disabled for applications running in the IDL Virtual Machine.

Syntax

HRESULT SetIDLVariable([in] BSTR bstrVar, [in] VARIANT Value)

Parameters

bstrVar

A string identifying the variable in the IDL process to be set to *Value*.

Value

The value for the variable.

Examples

See [“Data Manipulation with a COM Connector Object”](#) on page 251 for an array manipulation example that uses the GetIDLVariable, SetIDLVariable and ExecuteString methods.

SetProcessName

The SetProcessName method sets the name of the process that will contain the IDL object. This can *only* be called before making the CreateObject call. Once the object is created, the process name cannot be reset and calling this method after CreateObject has no effect.

Syntax

```
HRESULT SetProcessName([in] BSTR Name)
```

Parameters

Name

A string containing the name of the process that will contain the IDL object.

Event Handling

Clients subscribe to wrapper instance events through a process called advising. The wrapper object defines an outgoing source interface (event interface) containing the event methods, and the client implements that interface. During advising, the client passes a reference to its event interface to the wrapper. When an event occurs within the wrapper, it fires the event to the client by calling the appropriate event method on the client's event interface.

ActiveX controls fire events in the classical way via an outgoing source interface. The Export Bridge ActiveX wrapper controls define the outgoing source interface `_DIDLWrapperEvents`, as described below. Any client that wants to receive the events must subscribe to events by calling the wrapper object's `IConnectionPoint::Advise()` method. Once advised, the client unsubscribes to events by calling `IConnectionPoint::Unadvise()`.

```
HRESULT Advise      ([in] IUnknown* pUnk,
                    [out,retval] DWORD* pdwCookie);
HRESULT Unadvise    ([in] DWORD dwCookie);
```

The client implements the `_DIDLWrapperEvents` interface and calls the wrapper object's `Advise()` method with its `_DIDLWrapperEvents` interface reference, and receives a cookie for that connection. When the clients wants to disconnect, the client calls `Unadvise()` with the connection cookie.

In the classical sense, only ActiveX controls fire events, which are typically UI events. However, a client using an Export Bridge COM wrapper object may be interested in IDL output and notification. So, we carry the concept of firing events over onto COM objects. Clients of COM wrapper objects can receive events by advising to the same outgoing source interface in the same way that clients advise for events on the ActiveX wrapper controls.

Mouse and Keyboard Events in COM Export Objects

For UI events generated by ActiveX wrapper object, the client receives the events first before IDL receives them. The client then has the option to “eat” the event and prevent IDL from ever seeing the event. Each UI event has a `ForwardToIdl` parameter, which is initially set to `TRUE` (1). If the event handler in the client code clears the value to `FALSE` (0), then the wrapper does not forward the event to IDL.

Note

For a COM example that passes keyboard events to IDL, see [“COM IDLItWindow Surface Manipulation”](#) on page 281.

The event interface is defined below and uses the following values. The mouse Button parameter can have any of the following values ORed together:

IDLBML_MBUTTON_LEFT	0x1,
IDLBML_MBUTTON_RIGHT	0x2,
IDLBML_MBUTTON_MIDDLE	0x4,

The KeyState parameter can have any of the following values ORed together:

IDLBML_KEYSTATE_SHIFT	0x1,
IDLBML_KEYSTATE_CTRL	0x2,
IDLBML_KEYSTATE_CAPSLOCK	0x4,
IDLBML_KEYSTATE_ALT	0x8,

For the KeyCode parameters, if the key pressed is an ASCII character, then KeyCode is the ASCII value; otherwise it is one of these values:

IDLBML_KEYBOARD_EVENT_SHIFT	1	
IDLBML_KEYBOARD_EVENT_CONTROL	2	
IDLBML_KEYBOARD_EVENT_CAPSLOCK	3	
IDLBML_KEYBOARD_EVENT_ALT	4	
IDLBML_KEYBOARD_EVENT_LEFT	5	
IDLBML_KEYBOARD_EVENT_RIGHT	6	
IDLBML_KEYBOARD_EVENT_UP	7	
IDLBML_KEYBOARD_EVENT_DOWN	8	
IDLBML_KEYBOARD_EVENT_PAGE_UP	9	
IDLBML_KEYBOARD_EVENT_PAGE_DOWN	10	
IDLBML_KEYBOARD_EVENT_HOME	11	
IDLBML_KEYBOARD_EVENT_END	12	
IDLBML_KEYBOARD_EVENT_DEL	127	// isASCII is set to 1 when // this code is given

Note

The constants above are defined in the typelib information contained within each wrapper object and are used with the `_DIDLWrapperEvents` interface defined below.

```
dispinterface _DIDLWrapperEvents
{
    HRESULT OnMouseDown      (long Button, long KeyState, long x,
                              long y, [in,out]long* ForwardToIdl);
    HRESULT OnMouseUp        (long Button, long KeyState, long x,
                              long y, [in,out]long* ForwardToIdl);
    HRESULT OnMouseMove      (long Button, long KeyState, long x,
                              long y, [in,out]long* ForwardToIdl);
    HRESULT OnMouseWheel     (long KeyState, long WheelDelta, long x,
                              long y, [in,out]long* ForwardToIdl);
    HRESULT OnMouseDoubleClick (long Button, long KeyState, long
                              x, long y, [in,out]long* ForwardToIdl);
}
```

```

        HRESULT OnMouseEnter    (void);
        HRESULT OnMouseExit    (void);
        HRESULT OnKeyDown      (long KeyCode, long KeyState,
[in,out]long* ForwardToIdl);
        HRESULT OnKeyUp        (long KeyCode, long KeyState,
[in,out]long* ForwardToIdl);
        HRESULT OnSize          (long width, long height, [in,out]long*
ForwardToIdl);
        HRESULT OnIDLNotify     (BSTR bstr1, BSTR bstr2);
        HRESULT OnIDLOutput     (BSTR bstrOutput);
};

```

Note

For the OnMouseWheel event, the value of WheelDelta is a positive or negative value that indicates the amount that the wheel was rotated forward or backward, *e.g.* +/- 1, +/- 2, *etc.*

Note

Since the COM wrapper uses the same event interface, only the OnIDLNotify and OnIDLOutput events will be fired to subscribers of COM “events.” The UI events in the _DIDLWrapperEvents interface have no meaning in a nondrawable COM wrapper context, and therefore will not be fired to the client.

Error Handling

Each method of a wrapper object returns an HRESULT value. If the method call was successful, it returns S_OK; otherwise it returns a standard COM failure. If any of the methods calls return an HRESULT error code, the client can then call the GetLastError method to retrieve an error string, which is generally the actual IDL error message string.

The table below describes the error return values and their meaning when they are returned within the context of the wrapper method calls.

Error Code	Meaning
E_ACCESSDENIED	<p>This error return value occurs in one of two situations:</p> <ul style="list-style-type: none">• IDL is busy. The client made a method call on a wrapper object, but the underlying IDL is still busy processing a previous request (method call) and has not finished yet. For more information, see “IDL Ownership and Blocking” on page 134.• The client called the Abort method on a wrapper object, but that wrapper object is not the current owner of the underlying IDL; therefore it is not allowed to abort IDL.

Table 8-1: HRESULT Error Codes

Error Code	Meaning
E_ABORT	<p>This error return value occurs in one of two situations:</p> <ul style="list-style-type: none"> • It is returned from the original wrapper method call whose operation was aborted by a successful call to the Abort method. • It occurs when the client has created several wrapper instances whose underlying IDL objects all live in the same IDL process. During a method call on one of those wrapper instances, the IDL pro code issues the IDL <code>EXIT</code> command. When this occurs, the OPS process is destroyed, which also destroys all the underlying IDL objects. However, the client needs to be notified of this event so that it can consider all those wrapper instances as invalid and not use them again. First, each listener (event subscriber) for each wrapper instances will receive an <code>OnIDLNotify</code> callback with the first string set to “OPS_NOTIFY_EXIT”. Then, the method call (if any) that is in progress at the time of the <code>EXIT</code> command will return with the specified error code. <p>Upon receiving the notification and after receiving this error code, the user must not make any other method calls on the wrapper instance, as the underlying IDL object no longer exists.</p>
E_FAIL	<p>This error return value occurs in one of two situations:</p> <ul style="list-style-type: none"> • The client called the Abort method on a wrapper object, but the underlying IDL is not currently busy, so there is nothing to abort. • An IDL error occurred. In this case, the error message will be the same as the IDL error message.

Table 8-1: HRESULT Error Codes (Continued)

Debugging

When running an application that relies on a COM wrapper object, it is often difficult to determine when errors occur in the associated IDL object or IDL process. Since the instance of the wrapper object is invoked outside of IDL, the normal debugging capabilities of the IDL Workbench are not available.

However, it is possible to obtain this output by setting the `IDL_BRIDGE_DEBUG` environment variable as described in “[IDL_BRIDGE_DEBUG](#)” (Chapter 1, *Using IDL*). This debug information is usually written to `stdout` on Unix and Windows (unless noted otherwise in the following table). This output can also be captured in Visual Studio, the Debug Monitor (`DBMON.exe`) or WinDbg debugger on Windows. In each instance, the output depends on the value of the `IDL_BRIDGE_DEBUG` environment variable:

Value	Behavior
0	Turn off debug output
1	Turn on debug output, which includes output from library load errors, IDL execution errors, and PRINT statement output

The expected behavior in common debugging environments is described in the following table.

Application	Debug Output
Console Application	Command window — debug information is presented in line with any console window output when an <code>.exe</code> is executed from the Windows command window
	Visual Studio — debug output does not appear
	DBMON — debug information appears in the debug monitor window as it is generated

Table 8-2: Type and Location of Debug Information Output

Application	Debug Output
Windows Application	Command window — no debug output since the window that is launched has no knowledge of the debugging environment variable
	Visual Studio — debug output appears in the Debug Output window only when the application window is closed
	DBMON — debug information appears in the debug monitor window as it is generated

Table 8-2: Type and Location of Debug Information Output (Continued)

Note

In Windows, the environment variable is read when an application or command window is initially instantiated. For example, if you open Visual Studio and then change the value of the environment variable, you must re-launch Visual Studio to see the change in debug output behavior. DBMON is an exception as it always listens for debug information output and immediately reflects changes in content level.



Chapter 9

Using Exported Java Objects

This chapter discusses the following topics.

Overview of Java Export Objects	216	Event Handling	232
Java Wrapper Objects	217	Error Handling	242
Stock Java Wrapper Methods	218	Debugging	244

Overview of Java Export Objects

Once you have chosen to use a connector object or have exported a custom IDL source object using the Assistant, use the method and event reference information described here to create an instance of the object and interact with the IDL process from an external Java environment.

This chapter presents important background information on using IDL objects exported into Java:

- [“Java Wrapper Objects”](#) on page 217
- [“Stock Java Wrapper Methods”](#) on page 218
- [“Event Handling”](#) on page 232
- [“Error Handling”](#) on page 242

Java Wrapper Objects

There are two different types of objects created by the Java Export Bridge: drawable and non-drawable.

- Drawable wrapper objects contain a Java AWT Canvas object to which IDL draws. These wrapper objects inherit from the `JIDLCanvas` object.
- Nondrawable objects provide an interface between Java and IDL to call methods and pass data. However, these objects do not provide a Canvas on which IDL can draw. They inherit from the `JIDLObject` object.

Note

Drawable Java objects are not supported on Macintosh OS X.

`JIDLCanvas` objects extend `java.awt.Canvas`, which means that they are a heavyweight component. They will work fine with AWT components. However, Swing introduces the concept of lightweight components, which presents some issues in Java when heavyweight objects are mixed with lightweight components. (Swing does not provide a lightweight Canvas. If Swing users require the use of a Canvas, they use an `awt.Canvas`). Where possible, the `JIDLCanvas` attempts to work around these problems. However, Swing developers should be aware of them. The following article provides background on this problem and describes the various problems that may occur when mixing lightweight and heavyweight components:

<http://java.sun.com/products/jfc/tsc/articles/mixing/>.

Java is a highly multi-threaded language, especially in GUI applications, which can lead to problems with event handling. For example, event handling can happen in a different thread from the main thread that started the GUI. Thus, a complicated GUI could start processing events after the GUI has been initialized, but before the `createObject` method is called to instantiate the wrapper object for client use. It is therefore important not to start handling events before a successful object creation, which can be accomplished by calling the `isObjectCreated` method available for all Java wrapped objects to make sure the `createObject` call has finished successfully.

In addition to the wrapper methods created by the Export Bridge Assistant (see “[Stock Java Wrapper Methods](#)” on page 218 for details), exported Java objects have access to the interfaces and classes included in the IDL Java package itself. See [Appendix A, “IDL Java Object API”](#) for details.

Stock Java Wrapper Methods

This section describes the stock methods in the Java wrapper objects created by the Export Bridge Assistant:

- [“abort”](#) on page 219
- [“createObject”](#) on page 220
- [“destroyObject”](#) on page 223
- [“executeString”](#) on page 224
- [“getIDLObjectClassName”](#) on page 225
- [“getIDLObjectVariableName”](#) on page 226
- [“getIDLVariable”](#) on page 227
- [“getProcessName”](#) on page 228
- [“isObjectCreated”](#) on page 229
- [“setIDLVariable”](#) on page 230
- [“setProcessName”](#) on page 231

Every Java connector object and custom Java wrapper object has these methods in addition to those defined by the wrapped IDL object.

abort

The abort method requests that the IDL process containing the underlying IDL object abort its current activity. This method is useful if a given IDL method call is busy for a very long time (e.g., a very long image processing command).

Note

The request is only that, a request, and IDL might take a long time before it actually stops or might completely finish its current activity. Such a wait is an effect of the IDL interpreter.

Note that the client can only call abort from a different thread than the one currently executing because the method executing is, by its nature, blocked. The caller cannot abort IDL activity that is occurring from an execution call in another wrapper object. The client can only abort the current IDL activity if that wrapper object is the current owner of the underlying IDL process. For more information on error return code relating to aborting, see [“Error Handling”](#) on page 242.

Syntax

```
public void abort()
```

Arguments

None

createObject

The `createObject` method creates the actual underlying IDL object. The `argc`, `argv`, and `argpal` arguments are used to supply parameters to the underlying IDL object's `Init` method. If the `Init` method does not have any parameters, the caller sets `argc`, `argv`, and `argpal` to 0, null, and null, respectively. The `initializer` argument is used to specify IDL process initialization parameters (notably the IDL licensing mode).

The `createObject` method does the following:

1. It calls the `Init` method for the IDL object.
2. It calls the superclass `initListeners` method (either `JIDLCanvas::initListeners` or `JIDLObject::initListeners`) to initialize any event handlers.

The `initListeners` method has default behavior, which is different for drawable and nondrawable objects (see “[Event Handling](#)” on page 232 for more information). If the default behavior is not desired, a subclass to modify the listener initialization can override the `initListeners` method.

Note

Registering or unregistering listeners for events should happen in the `initListeners` method or AFTER the `createObject` method.

Syntax

```
public void createObject( )  
public void createObject(int argc, Object[] argv, int[] argpal)  
public void createObject(int argc, Object[] argv, int[] argpal,  
    JIDLProcessInitializer initializer)  
public void createObject(JIDLProcessInitializer initializer)
```

Arguments

argc

The number of parameters to be passed to `Init`.

argv

The array of objects to be passed to IDL. This array should be of length *argc* and should contain objects of type [JIDLNumber](#), [JIDLObjectI](#), [JIDLString](#), or [JIDLArray](#).

argpal

An array of *argc* flags denoting whether each *argv* parameter that is of type array should be convolved or not. For parameters that are not arrays, the value within the array will always be 0.

initializer

A [JIDLProcessInitializer](#) object that encapsulates the IDL process initialization parameters. (Process initialization parameters allow the Java programmer to control the licensing mode of the IDL process. See “[IDL Licensing Modes](#)” on page 134 for details on the default licensing mechanism.)

Example

The Init method of the IDL object being wrapped has the following signature:

```
PRO IDLexFoo::INIT, rect, filename
```

where *rect* is an array of four integers and *filename* is a string.

The following is an example of Java client code that creates an instance of the wrapper object and calls the `createObject` method with the *rect* and *filename* parameters:

```
// These are the Java types we want to pass to the ::Init method
int[] rect = {0, 0, 5, 10};
String file = "someFilename.txt";

// Wrap the Java types using Export Bridge data types
JIDLArray bRect = new JIDLArray(rect);
JIDLString bFile = new JIDLString(file);

// Create the wrapper object
MyWrapper wrapper = new MyWrapper();

// Set up parameters to pass to createObject
final int ARGV = 2;
Object[] argv = new Object[ARGV];
int[] argp = new int[ARGV];
```

```
argv[0] = bRect;
argp[0] = JIDLConst.PARMFLAG_CONST; // "in-only" parameter
argv[1] = bFile;
argp[1] = JIDLConst.PARMFLAG_CONST; // "in-only" parameter

// Create the underlying IDL object and call
// its ::Init method with parameters and default IDL
// process initialization settings
wrapper.createObject(ARGC, argv, argp);
```

Note

See [Appendix C, “Java Object Creation”](#) for additional examples of creating Java wrapper objects with and without parameters.

destroyObject

The `destroyObject` method destroys the underlying IDL object associated with the wrapper. If the object being destroyed is the last object within a process, the process is also destroyed.

Note that this method does not destroy the actual wrapper object. Because the wrapper object is a Java object, it follows all the Java reference-counting and garbage-collection schemes. Once all references to the wrapper object are released from Java code and once the JVM calls the garbage collector, the wrapper object may be deleted from memory.

Note

Trying to re-create an object after it has been destroyed is not supported. You must re-define the variable and then re-create the object.

Syntax

```
public void destroyObject()
```

Arguments

None

executeString

The executeString method executes the specified command in the IDL process containing the underlying IDL object

Note

This method is disabled for applications running in the IDL Virtual Machine.

Syntax

```
public void executeString(String sCmd)
```

Arguments

sCmd

The command to be executed.

Examples

See [“IDL Command Line with Java Connector Object”](#) on page 258 for an example that executes an IDL command entered into one textbox and writes IDL output or error information to a second textbox.

getIDLObjectClassName

The `getIDLObjectClassName` method returns the IDL object class name of the underlying IDL object.

Syntax

```
public String getIDLObjectClassName()
```

Arguments

None

getIDLObjectVariableName

When the underlying IDL object was created in the IDL process, it was assigned a variable name. The `getIDLObjectVariableName` method returns that name.

Syntax

```
public String getIDLObjectVariableName()
```

Arguments

None

getIDLVariable

The `getIDLVariable` method retrieves the named variable from the IDL process associated with the underlying IDL object.

Note

This method is disabled for applications running in the IDL Virtual Machine.

Syntax

```
public Object getIDLVariable(String sVar)
```

Arguments

sVar

The named variable to be retrieved. The returned object is of type [JIDLNumber](#), [JIDLString](#), [JIDLObjectI](#), or [JIDLArray](#).

If the variable is an array, the array is always converted from IDL majority to the standard Java array majority. (For more information on implications of array majority, see “[Multidimensional Array Storage and Access](#)” on page 493.)

Examples

See “[Data Manipulation with a Java Connector Object](#)” on page 256 for an array manipulation example that uses the `getIDLVariable`, `setIDLVariable` and `executeString` methods.

getProcessName

The `getProcessName` method returns the name of the process associated with the underlying IDL object.

Syntax

```
public String getProcessName()
```

Arguments

None

isObjectCreated

The `isObjectCreated` method returns `True` if the object has been created successfully and returns `False` if the object has not yet been created or if the object creation was unsuccessful. This call is often useful in a multi-threaded environment to check that an object is created before making a method call on that object.

Syntax

```
public boolean isObjectCreated()
```

Arguments

None

setIDLVariable

The `setIDLVariable` method sets the specified variable name to the specified value in the IDL process containing the underlying IDL object. If the type is [JIDLArray](#), it is always converted to IDL majority.

Note

This method is disabled for applications running in the IDL Virtual Machine.

Syntax

```
public void setIDLVariable(String sVar, Object value)
```

Arguments

sVar

A string identifying the variable in the IDL process to be set to *value*.

value

The value for *sVar*. The value should be an object of type [JIDLNumber](#), [JIDLObjectI](#), [JIDLString](#) or [JIDLArray](#). If the variable does not exist, it is created.

Examples

See “[Data Manipulation with a Java Connector Object](#)” on page 256 for an array manipulation example that uses the `getIDLVariable`, `setIDLVariable` and `executeString` methods.

setProcessName

The `setProcessName` method sets the name of the process that will contain the IDL object. This can *only* be called before making the `createObject` call. Once the object is created, the process name cannot be reset and calling this method after `createObject` has no effect.

Syntax

```
public void setProcessName(String sProcess)
```

Arguments

sProcess

A string containing the name of the process that will contain the IDL object.

Event Handling

Events in Java are handled by registered listener objects (often referred to as the Observer design pattern). The object interested in listening to a given event must implement the proper Java interface and then register to receive the events.

Any Java object can register to listen to any other object's events, but it is often useful for a wrapper object to listen to its own GUI and notify events. It usually makes most sense for a client object to listen to IDL output events.

Note

Registering or unregistering listeners for events should happen in the `initListeners` method or AFTER the `createObject` method.

Nondrawable Java Objects

Nondrawable objects, which inherit from `JIDLObject`, can be notified of the following events:

- IDL notify events (by implementing `JIDLNotifyListener`)
- IDL output events (by implementing `JIDLOutputListener`)

The default behavior as provided by the `JIDLObject` superclass is that they are not wired to listen to any events.

Drawable Java Objects

Drawable objects, which inherit from `JIDLCanvas`, are wired by default to listen to the following events:

- Mouse events (by implementing `JIDLMouseListener`)
 - Mouse enter canvas
 - Mouse exit canvas
 - Mouse pressed
 - Mouse released
- Mouse motion events (by implementing `JIDLMouseMotionListener`)
 - Mouse dragged
 - Mouse moved

- Mouse wheel events (by implementing [JIDLMouseWheelListener](#))
- Key events (by implementing [JIDLKeyListener](#))
 - Key pressed
 - Key released
- Component events (by implementing [JIDLComponentListener](#))
 - Canvas exposed
 - Canvas resized

In addition, drawable objects can also listen to the following events, but they do not listen to them by default:

- IDL notify events (by implementing [JIDLNotifyListener](#))
- IDL output events (by implementing [JIDLOutputListener](#))

IDL Notification

As mentioned above, IDL objects that subclass `itComponent` can trigger a notification from the IDL object level by calling [IDLitComponent::NotifyBridge](#). Both drawable ([JIDLCanvas](#)) and nondrawable ([JIDLObject](#)) wrapper objects handle IDL notifications.

To receive a notification, a class must implement the [JIDLNotifyListener](#) interface and register with the wrapper object by calling its `addIDLNotifyListener` method to register itself as a listener. The listener class can unregister itself by calling the `removeIDLNotifyListener` method.

The following is the definition of the `JIDLNotifyListener` interface:

```
public interface JIDLNotifyListener {  
  
    // obj: a reference to the wrapper object that triggered notify  
    // s1 and s2 are strings sent from IDLitComponent::NotifyBridge  
    void OnIDLNotify(JIDLObjectI obj, String s1, String s2);  
}
```

These methods are available to `JIDLCanvas` and `JIDLObject`:

```
public void addIDLNotifyListener(JIDLNotifyListener l);  
public void removeIDLNotifyListener(JIDLNotifyListener l);
```

IDL Output

In general, IDL output can be listened to by any class that implements the [JIDLOutputListener](#) interface and registers itself as a listener by calling `addIDLOutputListener`. The listener class can unregister itself by calling `removeIDLOutputListener`. Both drawable ([JIDLCanvas](#)) and non-drawable ([JIDLObject](#)) wrapper objects handle IDL output.

The following is the definition of the `JIDLOutputListener` interface:

```
public interface JIDLOutputListener {  
  
    // obj: a reference to the wrapper object that triggered notify  
    // s is the IDL output string  
    void IDLoutput(JIDLObjectI obj, String s);  
}
```

These methods are available to `JIDLCanvas` and `JIDLObject`:

```
public void addIDLOutputListener(JIDLOutputListener l);  
public void removeIDLOutputListener(JIDLOutputListener l);
```

Handling Specific Events

This section describes how client applications can listen to and handle the following events: mouse, mouse motion, keyboard, and component.

Mouse Events

Mouse events include a mouse entering the canvas, the mouse exiting the canvas, a mouse press in the canvas, and a mouse release in the canvas. Drag and move events are handled as mouse motion events (see “[Mouse Motion Events](#)” on page 235).

In general, mouse events may be listened to by any class that implements the [JIDLMouseListener](#) interface and registers itself as a listener by calling the `addIDLMouseListener` method. The listener class can unregister itself by calling the `removeIDLMouseListener` method. Only drawable ([JIDLCanvas](#)) wrapper objects handle this event type.

The following is the definition of the `JIDLMouseListener` interface:

```
public interface JIDLMouseListener {  
  
    // obj is a reference to the wrapper object  
    // e is a java.awt.event.MouseEvent  
    void IDLmouseEntered (JIDLObjectI obj, MouseEvent e);  
    void IDLmouseExited (JIDLObjectI obj, MouseEvent e);  
    void IDLmousePressed (JIDLObjectI obj, MouseEvent e);  
}
```

```
void IDLmouseReleased(JIDLObjectI obj, MouseEvent e);
}
```

These methods are available to JIDLCanvas:

```
public void addIDLMouseListener(JIDLMouseListener l);
public void removeIDLMouseListener(JIDLMouseListener l);
```

The default behavior of drawable wrappers is that they automatically register to listen to themselves and provide default event handlers for each of these events. The following table describes the default behavior for each event type.

Event	Action
IDLmousePressed	Triggered when a mouse button is pressed inside the canvas. The default behavior passes the event to the IDL method OnMouseDown.
IDLmouseReleased	Triggered when a mouse button is released inside the canvas. The default behavior passes the event to the IDL method OnMouseUp.
IDLmouseEntered	Triggered when the mouse enters the canvas. Default implementation does nothing. The default behavior calls the IDL method OnEnter.
IDLmouseExited	Triggered when the mouse exits the canvas. Default implementation does nothing. The default behavior calls the IDL method OnExit.

Table 9-1: The Default Behavior of Mouse Event Types

Mouse Motion Events

Mouse motion events include a mouse being moved or dragged inside the canvas. In general, mouse motion can be listened to by any class that implements the [JIDLMouseMotionListener](#) interface and registers itself as a listener by calling the `addIDLMouseMotionListener` method. The listener class can unregister itself by calling the `removeIDLMouseMotionListener` method. Only drawable ([JIDLCanvas](#)) wrapper objects handle this event type.

The following is the definition of the `JIDLMouseMotionListener` interface:

```
public interface JIDLMouseMotionListener {

    // obj is a reference to the wrapper object
```

```

// e is a java.awt.event.MouseEvent
void IDLmouseDragged(JIDLObjectI obj, MouseEvent e);
void IDLmouseMoved(JIDLObjectI obj, MouseEvent e);
}

```

These methods are available to JIDLCanvas:

```

public void addIDLMouseMotionListener(JIDLMouseMotionListener l);
public void removeIDLMouseMotionListener(JIDLMouseMotionListener
l);

```

The default behavior of drawable wrappers is that they automatically register to listen to themselves and provide default event handlers for each of these events. The following table describes the default behavior for each event type.

Event	Action
IDLmouseDragged	Triggered when the mouse is moved while its left button is pressed inside the canvas. The default behavior passes the event to the IDL method OnMouseMove.
IDLmouseMoved	Triggered when the mouse is moved (while no button is pressed) inside the canvas. The default behavior passes the event to the IDL method OnMouseMove.

Table 9-2: The Default Behavior of Mouse Motion Event Types

Mouse Wheel Events

Mouse wheel events include the scroll wheel of the mouse being rolled inside the canvas. In general, mouse wheel motion can be listened to by any class that implements the [JIDLMouseWheelListener](#) interface and registers itself as a listener by calling the `addIDLMouseWheelListener` method. The listener class can unregister itself by calling the `removeIDLMouseWheelListener` method. Only drawable ([JIDLCanvas](#)) wrapper objects handle this event type.

The following is the definition of the `JIDLMouseWheelListener` interface:

```

public interface JIDLMouseWheelListener {
    /**
     * A mouse wheel has moved inside the JIDLCanvas.
     * obj is a reference to the wrapper object
     * e is a java.awt.event.MouseWheelEvent
     */
    void IDLmouseWheelMoved(JIDLObjectI obj, MouseWheelEvent e);
}

```

These methods are available to JIDLCanvas:

```
public void addIDLMouseWheelListener(JIDLMouseWheelListener l);
public void removeIDLMouseWheelListener(JIDLMouseWheelListener l);
```

The default behavior of drawable wrappers is that they automatically register to listen to themselves and provide default event handlers for each of these events. The following table describes the default behavior for each event type.

Event	Action
IDLMouseWheelMoved	Triggered when the mouse wheel is rolled. The default behavior passes the event to the IDL method OnWheel.

Table 9-3: The Default Behavior of Mouse Wheel Event Type

Keyboard Events

Keyboard events include a key being pressed or released when the Canvas has focus. In general, keyboard events can be listened to by any class that implements the [JIDLKeyListener](#) interface and registers itself as a listener by calling the `addIDLKeyListener` method. The listener class can unregister itself by calling the `removeIDLKeyListener` method. Only drawable ([JIDLCanvas](#)) wrapper objects handle this event type.

The following is the definition of the JIDLKeyListener interface:

```
public interface JIDLKeyListener {

    // obj is a reference to the wrapper object
    // e is a java.awt.event.KeyEvent
    // (x,y) is the location of the mouse in the Canvas
    void IDLkeyPressed (JIDLObjectI obj, KeyEvent e, int x, int y);
    void IDLkeyReleased(JIDLObjectI obj, KeyEvent e, int x, int y);
}
```

These methods are available to JIDLCanvas:

```
public void addIDLKeyListener(JIDLKeyListener l);
public void removeIDLKeyListener(JIDLKeyListener l);
```

The default behavior of drawable wrappers is that they automatically register to listen to themselves and provide default event handlers for each of these events. The following table describes the default behavior for each event type.

Event	Action
IDLkeyPressed	Triggered when a key is pressed when the canvas has focus. The default behavior passes the event to the IDL method OnKeyboard.
IDLkeyReleased	Triggered when a key is released when the canvas has focus. The default behavior passes the event to the IDL method OnKeyboard.

Table 9-4: The Default Behavior of Keyboard Event Types

Component Events

Component events include the drawable canvas being resized and being exposed (uncovered or redrawn). Typically, these events are not handled by the client, but are handled behind the scenes by the Java Export Bridge, which resizes and repaints the canvas automatically. However, these events can be of interest to the client.

In general, component events can be listened to by any class that implements the [JIDLComponentListener](#) interface and registers itself as a listener by calling the `addComponentListener` method. The listener class can unregister itself by calling the `removeComponentListener` method. Only drawable ([JIDLCanvas](#)) wrapper objects handle this event type, and these methods are available only to [JIDLCanvas](#) objects.

The following is the definition of the [JIDLComponentListener](#) interface:

```
public interface JIDLComponentListener {
    void IDLcomponentResized(JIDLObjectI obj, ComponentEvent e);
    void IDLcomponentExposed(JIDLObjectI obj);
}
```

These methods are available to [JIDLCanvas](#):

```
public void addIDLComponentListener(JIDLComponentListener l)
public void removeIDLComponentListener(JIDLComponentListener l)
```

Specifically, drawable wrapper objects (those that inherit from [JIDLCanvas](#)) automatically register to listen to their own component events and provide default handlers for each of these events. The following table describes the methods and default implementations for the events.

Event	Action
<code>IDLcomponentResized</code>	Triggered when the canvas is resized. The default behavior calls the IDL method <code>OnResize</code> .
<code>IDLcomponentExposed</code>	Triggered when the canvas is exposed. The default behavior calls the IDL <code>OnExpose</code> method, which is expected to call the IDL object's <code>draw</code> method.

Table 9-5: The Default Behavior of Component Event Types

Subclassing to Change Behavior

There are two ways to change the event-handling behavior of listener objects: subclassing the wrapper object and handling the events in the subclass, or allowing a client object to handle events. Typically, GUI events and notifications are handled through subclassing and IDL output through client objects.

When a client calls the (drawable or nondrawable) wrapper object's `createObject` method, the wrapper object calls its `initListeners` method internally. This method, automatically generated by the Export Bridge Assistant, determines which events the wrapper object will listen to. As explained above, the wrapper object also has a set of methods generated to provide the default handling of these events.

To change what the object is listening to, subclass the generated wrapper object and override the `initListeners` method. The subclassed `initListeners` method can now register for whatever listeners in which it is interested.

For example, automatically generated drawable wrapper objects handle mouse, mouse motion, keyboard, and component events. Suppose you have a wrapper object called `canvasWrapper`, generated by the Assistant. You could subclass a wrapper object called `myCanvasWrapper` that would only handle mouse motion events. (The mouse motion events would still be handled in the default manner, but mouse, keyboard, and component listening would not be enabled.) This new wrapper object would look like this:

```
class myCanvasWrapper extends canvasWrapper {
    public void initListeners() {
        addIDLMouseMotionListener(this);
    }
}
```

To change the behavior of the listener handlers, subclass the generated wrapper object and override the event handling method whose behavior you want change. To get the default behavior, simply pass the event to the superclass.

Consider the following example. Given the same generated canvasWrapper class, you could ignore mouse drags and, on a mouse press, print information to a console object before passing up to the IDL object to handle. This class would look like this:

```
class myCanvasWrapper2 extends canvasWrapper {
    public void IDLmousePressed(JIDLObjectI o, MouseEvent e) {
        console.printMouseEvent(e);
        super.IDLmousePressed(o, e); // pass to IDL
    }

    public void IDLmouseDragged(JIDLObjectI o, MouseEvent e) {
        // do nothing
    }
}
```

Listening from Other Java Objects

Any Java object that implements the proper listener interface and registers itself with the wrapper object as a listener can also listen to events of interest. When more than one object is registered to listen to a given event, all listeners receive the event without a guarantee of order.

The steps are as follows:

1. The class implements the proper listener interface.
2. The class registers to listen to events.
3. The class handles the event in the listener interface method (or methods).

As an example, use the same canvasWrapper in a class called myClient that listens to IDL output. First, implement the JIDLOutputListener interface. Next, use the constructor to have the client register itself as a listener of the wrapper's IDL output. Finally, implement the IDLoutput to act on the output. The code is shown below:

```
import com.itt.javaidl.*;

class myClient implements JIDLOutputListener {
    canvasWrapper m_wrapper;
    public myClient() {
        m_wrapper = new canvasWrapper();
        m_wrapper.createObject();
        m_wrapper.addIDLOutputListener(this);
    }
    public void IDLoutput(JIDLObjectI obj, String s) {
```



```
        // do something with the IDL output
    }
    ...
}
```

Error Handling

When an error occurs in a Java wrapper object, it throws an unchecked exception of type `JIDLException` (or a subclass of `JIDLException`), which means that calls into a wrapper object should be wrapped in try-catch blocks, as is standard in Java. `JIDLException` provides the following method for getting the IDL error code:

```
public long getErrorCode();
```

In addition, because `JIDLException` inherits from `java.lang.Error`, other Java exception methods such as `getMessage` and `printStackTrace` are available.

The table below describes the error return values and their meaning when they are returned within the context of the wrapper method calls. The Java errors are encapsulated in a `JIDLException` object or a subclass of `JIDLException`, as noted in the table.

Error Exception/Code	Meaning
<code>JIDLBusyException</code> (a subclass of <code>JIDLException</code>) with <code>JIDLConst.IDL_BUSY</code> error code	IDL is busy. The client made a method call on a wrapper object, but the underlying IDL process is still busy with a previous request (method call) and has not finished yet. For more information, see “IDL Ownership and Blocking” on page 134.
<code>JIDLException</code> with <code>JIDLConst.IDL_ABORT_NOT_OWNER</code> error code	The client called the abort method on a wrapper object, but that wrapper object is not the current owner of the underlying IDL process. Therefore, it is not allowed to abort IDL.
<code>JIDLException</code> with <code>JIDLConst.IDL_NOTHING_TO_ABORT</code> error code	The client called the abort method on a wrapper object, but the underlying IDL process is not currently busy, so there is nothing to abort.
<code>JIDLAbortedException</code> (a subclass of <code>JIDLException</code>) with <code>JIDLConst.IDL_ABORTED</code> error code	This error is returned from the original wrapper method call whose operation was aborted by a successful call to the abort method.

Table 9-6: *JIDLException* Error Codes

Error Exception/Code	Meaning
JIDLException with JIDLConst.OPS_NOTICE_PROCESS_ABORTED error code	<p>This error occurs when the client has created several wrapper instances whose underlying IDL objects all live in the same IDL process. During a method call on one of those wrapper instances, the IDL pro code issues the IDL exit command. When this occurs, the process is destroyed, which also destroys all the underlying IDL objects. However, the client needs to be notified of this event so that it can consider all those wrapper instances as invalid and not use them again.</p> <p>First, each listener (event subscriber) for each wrapper instance receives an OnIDLNotify callback with the first string set to “OPS_NOTIFY_EXIT”. Then, the method call (if any) that is in progress at the time of the EXIT command will return with the specified error code.</p> <p>Upon receiving the notification and after receiving this error code, the user must not make any other method calls on the wrapper instance, as the underlying IDL object no longer exists.</p>
JIDLException with IDL error code	A specific IDL error occurred. The error code is the same as the IDL error code.

Table 9-6: JIDLException Error Codes (Continued)

Debugging

When running an application that relies on a Java wrapper object, it is often difficult to determine when errors occur in the associated IDL object. Since the instance of the wrapper object is invoked outside of IDL, the normal debugging capabilities of the IDL Workbench are not available.

However, it is possible to obtain this output by setting the `IDL_BRIDGE_DEBUG` environment variable as described in “[IDL_BRIDGE_DEBUG](#)” (Chapter 1, *Using IDL*). For example, if you set this environment variable to 1, you can see library load errors (on Windows), IDL execution errors, and output from IDL print commands. The appearance of debug information printed to `stdout` on Windows or UNIX depends upon the value set for the `IDL_BRIDGE_DEBUG` environment variable:

Value	Behavior
0	Turn off debug output
1	Turn on debug output, which includes output from library load errors, IDL execution errors, and PRINT statement output

To get additional Java-side diagnostics related to finding and loading the native libraries, define the `IDL_LOAD_DEBUG` parameter on the command line when starting a Java application, as follows:

```
java -DIDL_LOAD_DEBUG <class-to-run>
```



Chapter 10

Using the Connector Object

This chapter discusses how to use the prebuilt connector object that is included in the IDL distribution in COM and Java applications.

About the IDL Connector Object	246	Connector Object COM Examples	249
Preparing to Use the IDL Connector Object . .	247	Connector Object Java Examples	253

About the IDL Connector Object

The prebuilt IDL connector export object that is shipped with the IDL distribution lets you quickly incorporate the processing power of IDL into an application developed in an external, object-oriented environment such as COM or Java. The connector object definition provides the basis for a nondrawable COM or Java connector wrapper object that includes the ability to get and set IDL variables and execute command statements in the associated IDL process. These connector wrapper objects expose all of the standard wrapper object methods. See [“Stock COM Wrapper Methods”](#) on page 192 (COM) and [“Stock Java Wrapper Methods”](#) on page 218 (Java) for details.

Use a connector wrapper object if you need basic IDL processing capabilities. If you need the flexibility of custom object methods, an interactive IDL drawing interface, and/or associated mouse events, you should create an IDL object with the needed functionality and export it using the Export Bridge Assistant as described in [Chapter 7, “Using the Export Bridge Assistant”](#).

Note

Using the connector object provides exactly the same functionality as creating and exporting the simplest IDL object, which could consist of code similar to the following:

```
FUNCTION simpleobj::INIT
    RETURN, 1
END

PRO simpleobj__define
    struct = {simpleobj, $
        dummy:0b $ ; dummy structure field, not a property
    }
END
```

Preparing to Use the IDL Connector Object

All of the files needed to use a connector object are provided in the IDL distribution. You can locate the files in the following directory locations where *IDL_DIR* is where you have installed IDL:

Files	File Descriptions
COM	<p>Resource files:</p> <ul style="list-style-type: none"> • <code>COM_idl_connect.dll</code> • <code>COM_idl_connect.tlb</code> <p>are located in <i>IDL_DIR/resource/bridges/export/COM</i></p>
Java	<p>The <code>java_IDL_connect</code> class is included in the <code>javaidl.jar</code> file, which is located in the <i>IDL_DIR/resource/bridges/export/java</i> directory.</p> <p>The <code>javaidl.jar</code> file must be included in the Java classpath in order to use the Java connector wrapper object. This file contains the <code>com.idl.javaidl</code> package, which defines the Java class files needed by the Java export bridge.</p>
IDL Object	<p>The connector object definition is stored in a SAVE file named <code>idl_connect_define.sav</code> located in the <i>IDL_DIR/lib/bridges</i> directory.</p> <p>This is the only object definition file, and since it is contained within a SAVE file, it can be used with runtime IDL. Unlike custom IDL object definition files, there is no need to distribute this definition file with your application; it is already included in the IDL distribution.</p>

Table 10-1: Connector Object Files

To use the connector object with a COM application, you must reference the `COM_idl_connectLib 1.0 Type Library` library in your application. There is no need to register the `COM_idl_connect.dll` as described in [“COM Registration Requirements”](#) on page 143 since this is automatically registered upon IDL installation.

To use the connector object within a Java application, you must include the correct `import` statement in your Java application and set the classpath and as described in [“Java Requirements”](#) on page 143.

Connector Object COM Examples

The following examples show how to use the connector object in Visual Basic .NET Console and Windows applications. These examples contain important information about how to access messages sent from IDL in a COM application and how to communicate with the IDL process. In COM clients, the IDL output and notification methods are part of the default outgoing event interface.

- [“Hello World Example with a COM Connector Object”](#) on page 250 — shows how to use the [ExecuteString](#) method of the wrapper object to print a statement such as “Hello World” in a console application.
- [“Data Manipulation with a COM Connector Object”](#) on page 251 — uses [SetIDLVariable](#), [GetIDLVariable](#) and [ExecuteString](#) methods during array manipulation within a Console application.
- [“IDL Command Line with a COM Connector Object”](#) on page 252 — provides an interactive “IDL command line” in a Windows application.

Hello World Example with a COM Connector Object

To create a Visual Basic .NET console application using the connector object wrapper methods to print “Hello World” in a console application window, complete the following steps:

1. Create a new Visual Basic .NET console application and add a reference to the COM_idl_connectLib 1.0 Type Library.
2. Replace the default module definition with the code referenced below. See code comments for details.

Example Code

The text file for this example, `com_export_hello_doc.txt`, is located in the `examples/doc/bridges/COM` subdirectory of the IDL distribution. This Visual Basic .NET code can be copied from the text file and adopted for use in your COM environment.

After building and running the project, a simple console window will appear and “Hello World” will be output to this location.

Note

An expanded “Hello World” example that allows you to optionally say hello from someone can be found in [“Hello World COM Example with Custom Method”](#) in Chapter 12. This example uses a custom IDL object with a method and the Export Bridge Assistant to create the necessary wrapper object files.

Data Manipulation with a COM Connector Object

The following Visual Basic .NET example creates two arrays and passes them to IDL using the [SetIDLVariable](#) method. An [ExecuteString](#) command then multiplies the two arrays and [GetIDLVariable](#) returns the result to the COM application. The product of the array multiplication is printed to the console window.

1. Create a new Visual Basic .NET console application and add a reference to the COM_idl_connectLib 1.0 Type Library.
2. Replace the default module definition with the following code. See code comments for useful information.

Example Code

The text file for this example, `com_export_arrays_doc.txt`, is located in the `examples/doc/bridges/COM` subdirectory of the IDL distribution. This Visual Basic .NET code can be copied from the text file and adopted for use in your COM environment.

Building and running this program outputs the following to the console window.

```

C:\com\stockObjTest1\stockObjTest1\bin\stockObjTest1.exe
HELP, c /FULL equals:
C LONG = Array[6, 6]

Number of elements in 1st dimension:
6

The results of multiplying aArray
times bArray equals the following:
  0  1  2  3  4  5
  5  4  3  2  1  0
  0  4  8  12  16  20
  0  3  6  9  12  15
  0  2  4  6  8  10
  0  1  2  3  4  5
  0  0  0  0  0  0
Press any key to continue
  
```

Figure 10-1: Console Output of Array Multiplication

IDL Command Line with a COM Connector Object

The following example creates a simple Windows application in Visual Basic .NET that includes two text boxes. An IDL command typed in the top textbox is passed to the IDL process through the use of the [ExecuteString](#) method. Command output and any error messages are printed in the bottom textbox.

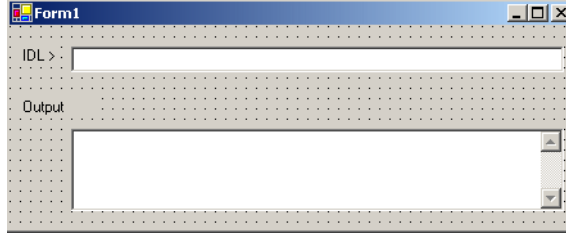


Figure 10-2: Design-time View of Simple Command Line Example

To replicate this example, complete the following steps:

1. Create a new Visual Basic .NET Windows application and add a reference to the `COM_idl_connectLib 1.0 Type Library`.
2. Replace the default form definition with the code referenced below. See code comments for details.

Example Code

The text file for this example, `com_export_commandline_doc.txt`, is located in the `examples/doc/bridges/COM` subdirectory of the IDL distribution. This Visual Basic .NET code can be copied from the text file and adopted for use in your COM environment.

After building and running the project, enter IDL commands in the top textbox. Pressing the **Enter** key sends the command to the IDL process.

Connector Object Java Examples

The following examples introduce the capabilities of the Java connector object:

- [“Hello World Example with a Java Connector Object”](#) on page 254
- [“Data Manipulation with a Java Connector Object”](#) on page 256
- [“IDL Command Line with Java Connector Object”](#) on page 258

Note

The Java class files needed for the Export Bridge are found in the `com.idl.javaidl` package, which is stored in the `javaidlb.jar` file. See [“Preparing to Use the IDL Connector Object”](#) on page 247 for more information.

Note on Running the Java Examples

Examples in this chapter provide Windows-style compile `javac` (compile) and `java` (run) commands. If you are running on a platform other than Windows, use your platform’s path and directory separators and see [“Java Requirements”](#) on page 143 for information about the `bridge_setup` file, which sets additional information.

Hello World Example with a Java Connector Object

To create a Java application that uses the connector object's `executeString` method to print "Hello World" in the command window, complete the following steps.

Example Code

The file for this example, `hello_example.java`, is located in the `examples/doc/bridges/java` subdirectory of the IDL distribution.

1. Open the file named `hello_example.java` in the location referenced above.
2. Open the Windows Command window by selecting **Start** → **Run** and enter `cmd` in the textbox.
3. Use the `cd` command to change to the directory containing the `hello_example.java` file. For a default Windows installation, the command would be:

```
cd IDL_DIR\examples\doc\bridges\java
```

where `IDL_DIR` is the directory where you have installed IDL.

4. Reference the classpath of `javaidl.jar` in the compile statement. This automatically accesses the connector object, `java_IDL_connect`, which is contained within the file. Enter the following two commands (as single lines) to compile and execute the program, replacing `IDL_DIR` with the IDL installation directory:

```
javac -classpath
      ".;IDL_DIR\resource\bridges\export\java\javaidl.jar"
      hello_example.java

java -classpath
     ".;IDL_DIR\resource\bridges\export\java\javaidl.jar"
     hello_example
```

Tip

See [“Note on Running the Java Examples”](#) on page 253 for information on executing Java commands on a non-Windows platform.

After compiling and running the project, "Hello World!" will appear in the command window.

Note

An expanded "Hello World" example that allows you to optionally say hello from someone can be found in [“Hello World Java Example with Additional Method”](#) on

page 294. This example uses a custom IDL object with a method and the Export Bridge Assistant to create the necessary wrapper object files.

Data Manipulation with a Java Connector Object

The following Java example creates two arrays and passes them to IDL using the `setIDLVariable` method. An `executeString` command then multiplies the two arrays and `getIDLVariable` returns the result to the java application. The product of the array multiplication is printed to the command window.

Example Code

The file for this example, `arrays_example.java`, is located in the `examples/doc/bridges/java` subdirectory of the IDL distribution.

Complete the following steps:

1. Open the file named `arrays_example.java` in the location referenced above.
2. Open the Windows Command window by selecting **Start** → **Run** and enter `cmd` in the textbox.
3. Use the `cd` command to change to the directory containing the `arrays_example.java` file. For a default Windows installation, the command would be:

```
cd IDL_DIR\examples\doc\bridges\java
```

where `IDL_DIR` is the directory where you have installed IDL.

4. Reference the classpath of `javaidl.b.jar` in the compile statement. This automatically accesses the connector object, `java_IDL_connect`, which is contained within the file. Enter the following two commands (as single lines) to compile and execute the program, replacing `IDL_DIR` with the IDL installation directory:

```
javac -classpath
      ".;IDL_DIR\resource\bridges\export\java\javaidl.b.jar"
      arrays_example.java

java -classpath
     ".;IDL_DIR\resource\bridges\export\java\javaidl.b.jar"
     arrays_example
```

Tip

See “[Note on Running the Java Examples](#)” on page 253 for information on executing Java commands on a non-Windows platform.

After compiling and running the project, the result of the array manipulation is printed to the command window, a subset of which appears in the following figure.

```
C:\RSI\IDL63\examples\doc\bridges\java>java -classpath ".;C:\RSI\IDL63\resource\
bridge\export\java\javaidl.jar" arrays_example
C
      LONG      = Array[6, 6]

Results of multiplying aArray
0 1 2 3 4 5
times bArray
5 4 3 2 1 0
equals:
< 0 5 10 15 20 25 >
< 0 4 8 12 16 20 >
< 0 3 6 9 12 15 >
< 0 2 4 6 8 10 >
< 0 1 2 3 4 5 >
< 0 0 0 0 0 0 >
```

Figure 10-3: Java Array Manipulation Result

IDL Command Line with Java Connector Object

The following example creates a simple Java application that includes two text boxes. An IDL command typed in the top textbox is passed to the IDL process through the use of the [executeString](#) method. Command output and any error messages are printed in the bottom textbox.

Example Code

The file for this example, `JIDLCommandLine.java`, is located in the `examples/doc/bridges/java` subdirectory of the IDL distribution.

1. Open the file named `JIDLCommandLine.java` in the location referenced above:
2. Open the Windows Command window by selecting **Start** → **Run** and enter `cmd` in the textbox.
3. Use the `cd` command to change to the directory containing the `JIDLCommandLine.java` file. For a default Windows installation, the command would be:

```
cd IDL_DIR\examples\doc\bridges\java
```

where `IDL_DIR` is the directory where you have installed IDL.

4. Reference the classpath of `javaidl.jar` in the compile statement. This automatically accesses the connector object, `java_IDL_connect`, which is contained within the file. Enter the following two commands (as single lines) to compile and execute the program, replacing `IDL_DIR` with the IDL installation directory:

```
javac -classpath
      ".;IDL_DIR\resource\bridges\export\java\javaidl.jar"
      JIDLCommandLine.java

java -classpath
     ".;IDL_DIR\resource\bridges\export\java\javaidl.jar"
     JIDLCommandLine
```

Tip

See [“Note on Running the Java Examples”](#) on page 253 for information on executing Java commands on a non-Windows platform.

After compiling and running the project, a simple command line interface appears as shown in the following figure.

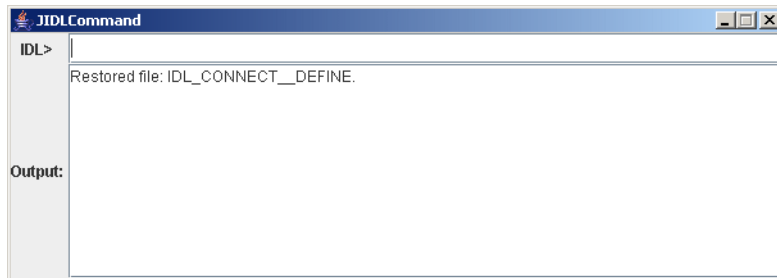


Figure 10-4: Java “IDL” Command Line Interface

Send commands to the IDL process by entering them in the top text box and pressing the Enter key. Any output or errors will appear in the lower text field.



Chapter 11

Writing IDL Objects for Exporting

This chapter discusses the following topics.

Overview	262	Exporting Drawable Objects	264
Programming Limitations	263	Drawable Object Canvas Examples	266

Overview

The objects you write in IDL can, in the vast number of cases, take full advantage of the Export Bridge technology, with only a few of IDL's capabilities not available. In addition, due to limitations imposed by external environments (COM and Java), certain restrictions exist for the method signatures that are exposed through the Export Bridge. This chapter outlines these functional limitations.

The chapter also provides a concise introduction to the object classes available to make drawable wrapper objects (objects that subclass from `IDLitWindow`, `IDLgrWindow`, and `IDLitDirectWindow`) and which to use when, with examples.

Programming Limitations

This section discusses the programming limitations required by the Export Bridge technology for successfully generating wrapper objects.

Keyword Parameters

Because COM and Java don't support the concept of keyword parameters, the Export Bridge does not support IDL keyword parameters in method signatures. If you want to export an IDL method that uses keyword parameters, you must wrap the method in another method that only implements positional parameters. The keyword parameters to IDL source object methods are ignored except for the SetProperty and GetProperty methods, in which keyword parameters are extracted to obtain the object's properties.

Unsupported Data Types

Properties and method parameters exported to a wrapper object class cannot include data of any of the following types:

- IDL Pointer
- Single- or double-precision complex data
- IDL Structure
- IDL objects, unless the object is an exported IDL object that exists in the same process space or pool as the object upon which the method is being called (as described in [“Object Reference Use”](#) on page 136)

Array Majority and Shape

The majority and shape of how data arrays are structured differs between the external environments supported by the Export Bridge (COM and Java) and IDL. Note that the Export Bridge technology might create a copy of array data when converting between external environments and IDL.

For further information on array majority, see [“Multidimensional Array Storage and Access”](#) on page 493. In addition, see [“Array Order Conversion”](#) on page 137 and [“Converting Array Majority”](#) on page 165 for details on how array majority is handled in the Assistant.

Exporting Drawable Objects

If you want to create a COM or Java application that uses a drawable wrapper object, you must subclass your IDL object from one of the following object classes before generating the wrapper:

- [IDLgrWindow](#) — provides a canvas for graphic objects
- [IDLitWindow](#) — provides a canvas for iTool visualizations
- [IDLitDirectWindow](#) — provides a canvas for Direct Graphic routine output

Note

Java drawable objects are not supported on the Macintosh OS X platform.

Requirements for Drawable Objects

Objects that inherit from [IDLgrWindow](#) must set the [GRAPHICS_TREE](#) property following creation of the objects hierarchy. This supports the automatic redraw capabilities of the `OnExpose` method.

Note

Common drawable object methods (such as `OnKeyboard` or `OnMouseMotion`) are typically not displayed in the Export Bridge Assistant when exporting a drawable object. See “[Drawable Object Event Handlers](#)” on page 178 for details.

In addition, IDL objects derived from [IDLitDirectWindow](#) must first provide a call of `self->makeCurrent` at the beginning of each method to ensure that the graphics rendering occurs in the wrapper’s drawable window, as described below.

Direct Graphics Support

To provide IDL Direct Graphics support, the export bridge uses an object to create an IDL Direct Graphics drawing surface. The Direct Graphics object, [IDLitDirectWindow](#), differs from standard IDL Direct Graphics in the following manner:

- The object implements the Active Window event handler callback methods to manage events. As such, to perform any event processing in IDL the user must sub-class this object and override the desired event callback methods.

- To make the window object current in the underlying direct graphics driver the user calls the `IDLitDirectWindow::MakeCurrent` method on the object. This is similar to a WSET operation in IDL, but no window index is required.
- Once a window is current, any IDL direct graphics routine can be called to draw graphics on the provided drawing surface. The user can add a method on the object they implement to render graphics or use the execute string functionality of the bridge to issue IDL commands.

While the Export Bridge implementation provides a different method to create and interact with a Direct Graphics Window, the differences are minor and let users rapidly port their IDL Direct Graphics implementation for use with this technology.

Drawable Object Canvas Examples

The following examples use the three object classes as canvases for drawable objects. You can use them with the Export Bridge by following “[Java Wrapper Example](#)” on page 267 or use them with the Export Bridge Assistant (for more information, see “[Using the Export Bridge Assistant](#)” on page 147). For information about a COM example, see “[COM Wrapper Example](#)” on page 268.

IDLgrWindow Example

The IDLgrWindow example uses object graphics to create a map that lets you click on and transform it with a trackball.

Example Code

The procedure file `idlgrwindowexample__define.pro`, located in the `examples/doc/bridges/` subdirectory of the IDL distribution, contains the example code. Run the example procedure by entering `idlgrwindowexample__define` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT idlgrwindowexample__define.pro`.

IDLitDirectWindow Example

The IDLitDirectWindow example uses direct graphics to create a palette on which you can draw and erase lines.

Example Code

The procedure file `idliddirectwindowexample__define.pro`, located in the `examples/doc/bridges/` subdirectory of the IDL distribution, contains the example code. Run the example procedure by entering `idliddirectwindowexample__define` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT idliddirectwindowexample__define.pro`.

IDLitWindow Example

The IDLitWindow example uses the `iSurface` tool to plot a Hanning transform on a surface.

Example Code

The procedure file `idlitwindowexample__define.pro`, located in the `examples/doc/bridges/` subdirectory of the IDL distribution, contains the

example code. Run the example procedure by entering
`idlitwindowexample__define` at the IDL command prompt or view the file in
an IDL Editor window by entering `.EDIT`
`idlitwindowexample__define.pro`.

Java Wrapper Example

An example Java wrapper that works with all three of the canvas types described above is included in the IDL distribution. The application accepts a parameter that specifies the name of the IDL class to use.

Note

Drawable Java objects are not supported on Macintosh OS X.

Note

The Export Bridge Assistant creates wrapper objects comparable to the code in this example. Your applications should not need to include such code if they are using Assistant-generated wrappers.

Note

The following steps assume you are working on a UNIX platform. If you are working on a Windows platform, substitute the appropriate paths and path-separator characters.

1. Copy the file `IDLWindowExample.java` from the

`IDL_DIR/examples/doc/bridges/java`

directory to a new directory where you will compile the Java code. In this example, we assume you will build the Java example in the
`/tmp/idljavatest` directory.

2. Change directories to the `/tmp/idljavatest` directory.
3. Source the `bridge_setup` file from the `bin` subdirectory of the IDL installation. If you use a C shell:

`source IDL_DIR/bin/bridge_setup`

where `IDL_DIR` is the path to your IDL installation. (There are `bridge_setup` files for the C shell, korn shell, and bash shell. Use the proper source command and `bridge_setup` file for your installation.)

4. Compile the `IDLWindowExample.java` file with the following command:

```
javac IDLWindowExample.java
```

This command creates two class files: `IDLWindow.class` and `IDLWindowExample.class`.

5. Execute the example code with the following command:

```
java IDLWindowExample <IDL_classname>
```

where `<IDL_classname>` is the name of one of the example classes described above.

Note

The `bridge_setup` file sets your `CLASSPATH` environment variable to include both the current directory (".") and the `IDL_DIR/resource/bridges/export/java/javaidlb.jar` file. See [“Java Requirements”](#) on page 143 for additional information about the class path

For example, if you sourced the `bridge_setup` file and compiled the `IDLWindowExample.java` file in the `/tmp/idljavatest` directory, the following commands would execute the three examples described above:

```
java IDLWindowExample IDLgrWindowExample
java IDLWindowExample IDLitDirectWindowExample
java IDLWindowExample IDLitWindowExample
```

COM Wrapper Example

See [“Tri-Window COM Export Example”](#) on page 284 for the steps needed to include controls based on the three drawable objects in a Visual Basic .NET Windows application.



Chapter 12

Creating Custom COM Export Objects

This chapter discusses the following topics.

About COM Export Object Examples	270	Drawable COM Export Examples	276
Nondrawable COM Export Example	272		

About COM Export Object Examples

An IDL object can be wrapped for use in a COM application using the Export Bridge Assistant as described in [Chapter 7, “Using the Export Bridge Assistant”](#). This chapter provides several Visual Basic .NET examples that use custom COM export objects. These include:

- Nondrawable examples — show how to access the processing power of IDL in a COM application by exchanging data with the IDL process, issuing IDL commands, and accessing IDL output
- Drawable examples — contain the elements needed to create interactive IDL drawing windows and to access keyboard and mouse events

Note

You can quickly incorporate the processing power of IDL in a COM application by including the pre-built COM connector wrapper object in your external application. Use this option if you do not need custom methods or an interactive drawing surface. See [Chapter 10, “Using the Connector Object”](#) for examples.

The general process for each of these examples involves the following:

1. Create the object in IDL.
2. Export the object using the Export Bridge Assistant, which creates the .dll, .tlb or .ocx files associated with the IDL object that is now wrapped in a COM export object wrapper.
3. Register the .dll or .ocx file.
4. Reference the appropriate library in your COM application before attempting to access the object functionality. This functionality automatically includes stock methods and events (described in [Chapter 8, “Using Exported COM Objects”](#)) in addition to custom methods you have chosen to export.

Note

See [“Writing IDL Objects for Exporting”](#) on page 261 for information on how to create custom IDL objects that can be successfully exported using the Export Bridge Assistant. There are important object method and data type requirements that must be met.

Note

When you distribute an application, you will also need to share:

- any application-specific .dll files generated during the build process
- the executable file (.exe)
- the .dll or .ocx files (generated by the Export Bridge Assistant)
- the .pro or .sav file that contains the custom object definition

Any .pro or .sav files included with your application must be located in the IDL path.

Debugging Applications Using Export Objects

It can be challenging to determine what is happening in the IDL process associated with a wrapper object without the debugging features of the IDL Workbench. For access to valuable debug information, consider using the IDL_BRIDGE_DEBUG environment variable, described in [“Debugging”](#) on page 213.

Nondrawable COM Export Example

Nondrawable objects provide access to the enormous processing power of IDL, but do not provide IDL drawing capabilities. This is useful for applications that need the data manipulation capabilities of IDL, but have no need for, or have independent drawing capabilities.

Hello World COM Example with Custom Method

The following simple example creates an IDL object with a single function method that accepts one argument, and walks through the process of exporting the object using the Export Bridge Assistant. Once the export files are created, a simple Visual Basic .NET console application shows how to access the object method and capture its output.

Complete the following steps to duplicate this example.

1. In an IDL Editor window, copy in the following code and save the file as `helloworldex__define.pro` in a directory in your IDL path:

```
; Method returns message based on presence or
; absence of argument.
FUNCTION helloworldex::HelloFrom, who
    IF (N_ELEMENTS(who) NE 0) THEN BEGIN
        message = "Hello World from " + who
        RETURN, message
    ENDIF ELSE BEGIN
        message = 'Hello World'
        RETURN, message
    ENDELSE
END

; Init returns object reference on successful
; initialization.
FUNCTION helloworldex::INIT
    RETURN, 1
END

; Object definition.
PRO helloworldex__define
    struct = {helloworldex, $
        who: ' ', $
        message: ' ' $
    }
END
```


Note

It is a good idea to test the functionality of an object before exporting it. After compiling the file, enter the following lines at the command line and make sure the output is what is expected for this object.

```
ohello = OBJ_NEW("HELLOWORLDEX")
PRINT, ohello->HelloFrom()
PRINT, ohello->HelloFrom('Mr. Bill')
```

2. Open the Export Bridge Assistant by entering IDLEXBR_ASSISTANT at the command line.
3. Select to create a COM export object by selecting **File** → **New Project** → **COM** and browse to select the helloworldex__define.pro file. Click **Open** to load the file into the Export Assistant.

Note

Export Bridge Assistant details are available in [“Specifying Information for Exporting”](#) on page 164. Refer to that section if you need more information about the following steps.

4. The top-level project entry in the left-hand tree panel is selected by default. There is no need to modify the default properties shown in the right-hand property panel, but you can enter different values if desired. Set other export object characteristics as described in the following table. Select the tree view item listed in the left column to configure the related properties in the right column.

Tree View Item	Parameter Configuration
IDL Export Bridge Project	Accept the default value or make changes as desired: <ul style="list-style-type: none"> • Output classname • Process name • Output directory
helloworldex	Drawable object equals False

Table 12-1: Example Export Object Parameters

Tree View Item	Parameter Configuration
HELLOFROM method	<p>Output method name — accept the default value, HELLOFROM</p> <p>Return Type — BSTR since this function method returns a string message (as defined in the IDL object definition structure)</p> <p>Array — False since this method returns a single string, not an array</p> <p>Export — True</p>
WHO argument	<p>Mutability — In since the argument is not passed back to the caller</p> <p>Type — BSTR since this argument is defined as a string in the IDL object definition</p> <p>Array — False</p> <p>Export — True</p>

Table 12-1: Example Export Object Parameters

- Save the project by selecting **File** → **Save project**. Accept the default name and location or make changes as desired.
- Verify that the object elements you want to export are listed in the Export log panel. If the expected items are not present, one or more items may still have an UNSPECIFIED field value that must be changed.
- Build the export object by selecting **Build** → **Build object**. The **Build log** panel shows the results of the build process. For a nondrawable object, .tlb and .dll files (named based on the object name) are created in the **Output directory**.
- Register the .dll using `regsvr32 helloworldex.dll`. See “[COM Registration Requirements](#)” on page 143 for details if needed.
- Create a new Visual Basic .NET console application and import a reference to the COM library named `helloworldexLib 1.0 Type Library`.
- Replace the default module code with the text in the file referenced below. See code comments for details.

Example Code

The text file for this example, `com_export_helloex_doc.txt`, is located in the `examples/doc/bridges/COM` subdirectory of the IDL distribution. This Visual Basic .NET code can be copied from the text file and adopted for use in your COM environment.

After building the solution and starting without debugging, the console window appears with the output messages.

Drawable COM Export Examples

A COM export object that supports graphics must be based on a custom IDL object that inherits from `IDLgrWindow`, `IDLitWindow` or `IDLitDirectWindow` (as described in [“Exporting Drawable Objects”](#) on page 264). Additionally, your IDL object must also implement a set of callback methods if you want to be able to respond to mouse or keyboard events in the graphics window. These are described in [“Event Handling”](#) on page 208. Examples in this section include:

- [“COM IDLgrWindow Based Histogram Plot Generator”](#) on page 277 — provides an object based on `IDLgrWindow` that creates a histogram plot for a selected image file and lets you change the plot linestyle property.
- [“COM IDLitWindow Surface Manipulation”](#) on page 281 — includes a drawable `IDLitWindow` example with `ISURFACE` functionality and a custom method lets you change the active manipulator. Delete key events are captured and passed to a custom `OnKeyboard` method that deletes selected visualizations.
- [“Tri-Window COM Export Example”](#) on page 284 — includes controls based on the three types of drawable objects (`IDLgrWindow`, `IDLitWindow`, and `IDLitDirectWindow`) in a single Visual Basic .NET Windows application. A subprocedure captures `IDLitComponent::NotifyBridge` output and prints it to a label on the form.

COM IDLgrWindow Based Histogram Plot Generator

This drawable object example inherits from IDLgrWindow and creates a histogram plot for a selected monochrome or RGB image file. While this example does contain several custom methods including those for opening a file, creating the plots, and changing plot characteristics, it does not use keyboard or mouse events. See “[COM IDLitWindow Surface Manipulation](#)” on page 281 for such an example.

Example Code

The object definition file, `export_grwindow_doc__define.pro` is located in the `examples/doc/bridges` subdirectory of the IDL distribution. Run the example procedure by entering `export_grwindow_doc__define` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT export_grwindow_doc__define.pro`.

Complete the following steps to duplicate this example.

1. In an IDL Editor window, open the object definition file by entering `.EDIT export_grwindow_doc__define.pro` at the command prompt.

Note

It is a good idea to test the functionality of an object before exporting it. After compiling the file, enter the following lines at the command line and make sure the output is what is expected for this object.

```
oPlotWindow = OBJ_NEW("export_grwindow_doc")
oPlotWindow->CHANGELINE, 2
```

This will display a three channel histogram plot and change the plot linestyle to dashed.

2. Open the Export Bridge Assistant by entering `IDLEXBR_ASSISTANT` at the command line.
3. Select to create a COM export object by selecting **File** → **New Project** → **COM** and browse to select `export_grwindow_doc__define.pro`. Click **Open** to load the file into the Export Assistant.

Note

Export Bridge Assistant details are available in “[Specifying Information for Exporting](#)” on page 164. Refer to that section if you need more information about the following items.

4. Set export object characteristics as described in the following table. When you select the tree view item listed in the left column, configure the related properties as noted in the right column.

Note

Set the Export parameter to True for all items in this list unless otherwise noted.

Tip

You can select multiple items in the tree view and set properties for the group.

Tree View Item	Property Configuration
IDL Export Bridge Project	Accept the default value or make changes as desired: <ul style="list-style-type: none"> • Output classname • Process name • Output directory
export_grwindow_doc	Drawable object equals True
OMODEL property	Type — IUnknown*
OVIEW property	Array — False
OXAXIS property	
OXTTEXT property	
OYAXIS property	
OYTEXT property	
OPLOTCOLL property	Type — IUnknown* Array — True
SFILE property	Type — BSTR Array — False
CHANGELINE method	Enter different name if desired and mark Export as True

Table 12-2: Example Export Object Parameters

Tree View Item	Property Configuration
STYLE argument	Mutability — In Type — Short Array — False
CREATEPLOTS method	Enter different name if desired and mark Export as True
IMAGE argument VROWS argument VCOLS argument	Mutability — In Type — Variant Array — True Convert majority — False
VRGB argument	Mutability — In Type — short Array — False
OPEN method	Enter different name if desired and mark Export as True
SFILE argument	Mutability — In Type — BSTR Array — False NOTE: You can choose not to export this parameter. If so, the method follows the path for cases where no argument is defined. (You will need to modify the Visual Basic code to read <code>Me.Axexport_grwindow_doc1.OPEN()</code> instead of passing an argument.) If you do choose to export this method, the argument must either be a null string or a full file path.

Table 12-2: Example Export Object Parameters

5. Save the project by selecting **File** → **Save project**. Accept the default name and location or make changes as desired.

6. Verify that the object elements you want to export are listed in the Export log panel. If the expected items are not present, one or more items may still have an UNSPECIFIED field value that must be changed.
7. Build the export object by selecting **Build** → **Build object**. The **Build log** panel shows the results of the build process. For a drawable object, .tlb and .ocx files (named based on the object name) are created in the **Output directory**.
8. Register the .ocx using `regsvr32 export_grwindow_doc.ocx`. See [“COM Registration Requirements”](#) on page 143 for details if needed.
9. Create a new Visual Basic .NET Windows Application and add the `export_grwindow_doc Class` file to the toolbox. Select **View** → **Toolbox** and select the desired tab. Right-click and select **Add/Remove Items**. Click on the **COM Components** tab, place a checkmark next to the class file and click **OK**.
10. Add the IDL `export_grwindow_doc` control to your form.
11. Replace the default form code with the text in the file referenced below. See code comments for details.

Example Code

The text file for this example, `com_export_grwindow_doc.txt`, is located in the `examples/doc/bridges/COM` subdirectory of the IDL distribution. This Visual Basic .NET code can be copied from the text file and adopted for use in your COM environment.

After building and running the project, a Windows application interface will display a histogram plot of an RGB image. You can change the linestyle of the plot by making a selection from the listbox. You can also create a histogram plot for a new image by clicking the button.

COM IDLitWindow Surface Manipulation

This drawable object example inherits from IDLitWindow and creates an ISURFACE display in a COM control. A listbox in a Visual Basic .NET Windows application is populated with manipulator string values that, when selected, allow you to draw annotations, rotate, or zoom within the exported IDLitWindow control. You should avoid exposing any manipulator that has an associated widget interface (such as a profile line manipulator) since such widget functionality is not supported in objects that subclass from IDLitWindow.

Example Code

The object definition file, `export_itwinmanip_doc__define.pro` is located in the `examples/doc/bridges` subdirectory of the IDL distribution. Run the example procedure by entering `export_itwinmanip_doc__define` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT export_itwinmanip_doc__define.pro`.

Complete the following steps to duplicate this example:

1. In an IDL Editor window, open the object definition file by entering `.EDIT export_itwinmanip_doc__define.pro` at the command prompt.

Note

It is a good idea to test the functionality of an object before exporting it. After compiling the file, enter the following lines at the command line and make sure the output is what is expected for this object.

```
oWin = OBJ_NEW("export_itwinmanip_doc")
oWin->CHANGEMANIPULATOR, "annotation/oval"
```

This will let you draw a oval annotation in the window. If you hit the Delete key, the annotation will be removed. The available manipulator strings are printed in the IDL output window.

2. Open the Export Bridge Assistant by entering `IDLEXBR_ASSISTANT` at the command line.
3. Select to create a COM export object by selecting **File** → **New Project** → **COM** and browse to select `export_itwinmanip_doc__define.pro`. Click **Open** to load the file into the Export Assistant.

Note

Export Bridge Assistant details are available in [“Specifying Information for Exporting”](#) on page 164. Refer to that section if you need more information about the following items.

4. Set export object characteristics as described in the following table. When you select the tree view item listed in the left column, configure the related properties as noted in the right column.

Tree View Item	Property Configuration
IDL Export Bridge Project	Accept the default value or make changes as desired: <ul style="list-style-type: none"> • Output classname • Process name • Output directory
export_itwinmanip_doc	Drawable object equals True
CHANGEMANIPULATOR method	Enter different name if desired and mark Export as True
MANIPID argument	Mutability — In Type — BSTR Array — False Export — True

Table 12-3: Example Export Object Parameters

5. Save the project by selecting **File** → **Save project**. Accept the default name and location or make changes as desired.
6. Verify that the object elements you want to export are listed in the Export log panel. If the expected items are not present, one or more items may still have an UNSPECIFIED field value that must be changed.
7. Build the export object by selecting **Build** → **Build object**. The **Build log** panel shows the results of the build process. For a drawable object, `.t1b` and `.ocx` files (named based on the object name) are created in the **Output directory**.

8. Register the .ocx using `regsvr32 export_itwinmanip_doc.ocx`. See [“COM Registration Requirements”](#) on page 143 for details if needed.
9. Create a new Visual Basic .NET Windows Application and add the `export_itwinmanip_doc Class` file to the toolbox. Select **View** → **Toolbox** and select the desired tab. Right-click and select **Add/Remove Items**. Select the **COM Components** tab, place a checkmark next to the class file, and click **OK**.
10. Add the IDL `export_itwinmanip_doc` control to your form.
11. Replace the default form code with the text in the file referenced below. See code comments for details.

Example Code

The text file for this example, `com_export_itwinmanip_doc.txt`, is located in the `examples/doc/bridges/COM` subdirectory of the IDL distribution. This Visual Basic .NET code can be copied from the text file and adopted for use in your COM environment.

Note

This example exposes keyboard events. The value of the **Delete** key and other standard keys are described in [“Mouse and Keyboard Events in COM Export Objects”](#) on page 208.

Build and run the project. Select a manipulator from the listbox to use it in the IDLItWindow display. If you hit the keyboard **Delete** key while visualizations are selected, they will be removed from the view.

Tri-Window COM Export Example

The `examples/doc/bridges` directory includes three object definition files that inherit from the three types of drawable objects: `IDLgrWindow`, `IDLitDirectWindow` and `IDLitWindow`. The following example uses the Export Bridge Assistant to create ActiveX controls from these object definition files and then creates a Windows application in Visual Basic .NET that includes the three controls.

Example Code

The object definition files, `idlgwindowexample__define.pro`, `idlitdirectwindowexample__define.pro`, and `idlitwindowexample__define.pro` are located in the `examples/doc/bridges` subdirectory of the IDL distribution.

Open the Export Bridge Assistant by entering `IDLEXBR_ASSISTANT` at the IDL command line and then complete the following steps to export the three drawable objects.

Note

Export Bridge Assistant details are available in [“Specifying Information for Exporting”](#) on page 164. Refer to that section if you need more information about the following items.

Wrap the IDLitDirectWindow Example

The object defined in `idlitdirectwindowexample__define.pro` inherits from [IDLitDirectWindow](#) and creates a drawing canvas that you can write on using your mouse.

1. Select **File** → **New Project** → **COM**, browse to select `idlitdirectwindowexample__define.pro` from the `examples/doc/bridges` directory, and click **Open**.
2. Set export object characteristics as described in the following table. When you select the tree view item, listed in the left column, configure the related properties as noted in the right column.

Tree View Item	Property Configuration
IDL Export Bridge Project	Accept the default value or make changes as desired: <ul style="list-style-type: none"> • Output classname • Process name • Output directory
idltdirectwindowexample	Drawable object equals True
WINDOW_INDEX property	You do not need to export the WINDOW_INDEX property as the control will always know it's own index number. You can leave all fields unchanged.
MAKECURRENT method	Export — False. This is only used within methods in the IDL source object definition file.

Table 12-4: Example Export Object Parameters

3. Save the project by selecting **File** → **Save project**. Accept the default name and location or make changes as desired.
4. Verify that the object elements you want to export are listed in the Export log panel. If the expected items are not present, one or more items may still have an UNSPECIFIED field value that must be changed.
5. Build the export object by selecting **Build** → **Build object**. The **Build log** panel shows the results of the build process. For a drawable object, `.tlb` and `.ocx` files (named based on the object name) are created in the **Output directory**.

Wrap the IDLgrWindow Example

The object defined in `idlgrwindowexample__define.pro` inherits from [IDLgrWindow](#) and displays a globe that can be rotated using your mouse.

1. Select **File** → **New Project** → **COM**, browse to select `idlgrwindowexample__define.pro` from the `examples/doc/bridges` directory, and click **Open**.

2. Set export object characteristics as described in the following table. When you select the tree view item, listed in the left column, configure the related properties as noted in the right column.

Tree View Item	Property Configuration
IDL Export Bridge Project	Accept the default value or make changes as desired: <ul style="list-style-type: none"> • Output classname • Process name • Output directory
idlgrwindowexample	Drawable object equals True
CREATEOBJECTS method	Export — False. This method is not called from the COM client.

Table 12-5: Example Export Object Parameters

3. Save the project by selecting **File** → **Save project**. Accept the default name and location or make changes as desired.
4. Verify that the object elements you want to export are listed in the Export log panel. If the expected items are not present, one or more items may still have an UNSPECIFIED field value that must be changed.
5. Build the export object by selecting **Build** → **Build object**. The **Build log** panel shows the results of the build process. For a drawable object, `.tlb` and `.ocx` files (named based on the object name) are created in the **Output directory**.

Wrap the IDLitWindow Example

The object defined in `idlitwindowexample__define.pro` inherits from [IDLitWindow](#) and displays a surface in a view in which you can pan and zoom.

1. Select **File** → **New Project** → **COM**, browse to select `idlitwindowexample__define.pro` from the `examples/doc/bridges` directory, and click **Open**.
2. There are no export object characteristics that must be modified, but you can make changes to the default items as described in the following table. When

you select the tree view item, listed in the left column, configure the related properties as noted in the right column.

Tree View Item	Property Configuration
IDL Export Bridge Project	Accept the default value or make changes as desired: <ul style="list-style-type: none"> • Output classname • Process name • Output directory
idlitwindowexample	Drawable object equals True

Table 12-6: Example Export Object Parameters

3. Save the project by selecting **File** → **Save project**. Accept the default name and location or make changes as desired.
4. Build the export object by selecting **Build** → **Build object**. The **Build log** panel shows the results of the build process. For a drawable object, `.tlb` and `.ocx` files (named based on the object name) are created in the **Output directory**.

Register the Controls and Create the Application

1. Register the `.ocx` files generated by the Export Bridge Assistant using `regsvr32` (see “[COM Registration Requirements](#)” on page 143 for details if needed). If you kept the default names, you will need to register `idlgrwindowexample.ocx`, `idlitdirectwindowexample.ocx`, and `idlitwindowexample.ocx`.
2. Create a new Visual Basic .NET Windows Application and add the `idlgrwindowexample Class`, `idlitdirectwindowexample Class`, and `idlitwindowexample Class` files to the toolbox. Select **View** → **Toolbox** and select the desired tab. Right-click and select **Add/Remove Items**. Select the **COM Components** tab, place a checkmark next to the class files, and click **OK**.
3. Add the three controls IDL to your form in the order of `idlgrwindowexample Class`, `idlitdirectwindowexample Class` and `idlitwindowexample Class` from left to right.

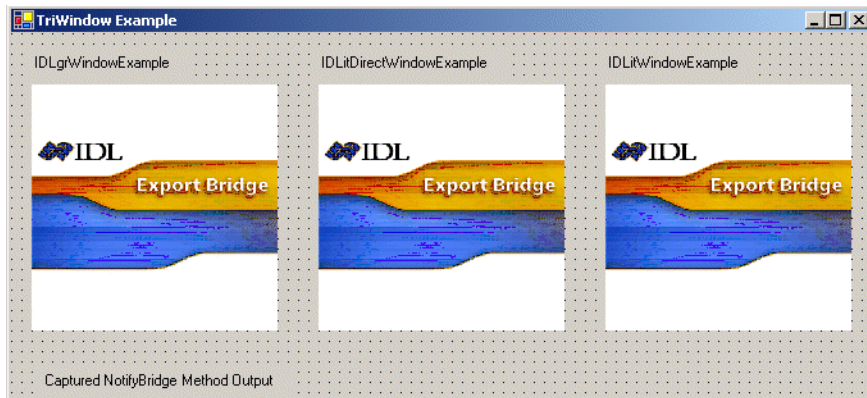


Figure 12-1: Design-time View of Three Drawable Window Controls

4. Replace the default module code with the text in the file referenced below. See code comments for details.

Example Code

The text file for this example, `com_export_triwindow_doc.txt`, is located in the `examples/doc/bridges/COM` subdirectory of the IDL distribution. This Visual Basic .NET code can be copied from the text file and adopted for use in your COM environment.

When you build and run the example, the output will appear similar to the following figure.

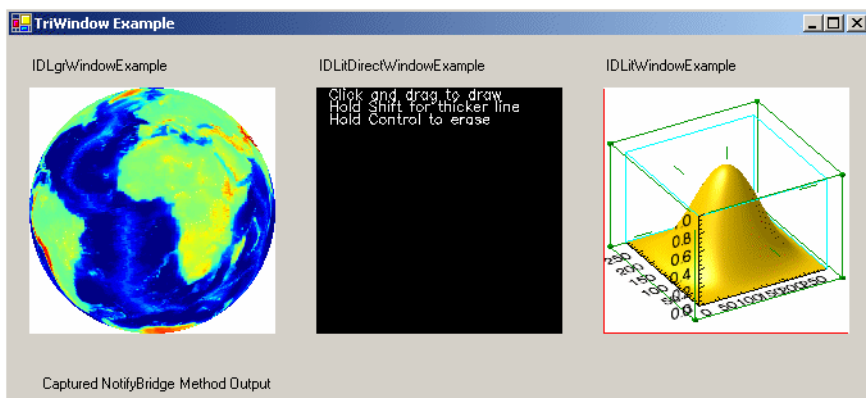


Figure 12-2: Runtime View of Three Drawable Window Controls

Left-click and drag in the IDLgrWindow control to rotate the globe and follow the instructions in the IDLitDirectWindow control to draw in the window. In the IDLitWindow, left-click on the surface and drag the mouse cursor to reposition the object, or left-click on one of the view handles and drag up or down to zoom in or out. The bottom label text will change when you move your mouse into or out of the IDLgrWindow- or IDLitDirectWindow-based controls as the label is updated with the NotifyBridge output from the IDL object definition files.



Chapter 13

Creating Custom Java Export Objects

This chapter discusses the following topics.

About Java Export Object Examples	292	Drawable Java Export Examples	298
Nondrawable Java Export Example	294		

About Java Export Object Examples

An IDL object can be wrapped for use in a Java application using the Export Bridge Assistant. For valuable information on the theory and architecture of a wrapper object created by the Export Bridge Assistant, see [Chapter 7, “Using the Export Bridge Assistant”](#).

This chapter provides several Java examples that incorporate the use of Java export objects. These include:

- Nondrawable examples — show how to access the processing power of IDL in a Java application by exchanging data with the IDL process, issuing IDL commands, accessing IDL output.
- Drawable examples — contain the elements needed to create interactive IDL drawing windows and access to mouse events.

Note

You can quickly incorporate the processing power of IDL in a Java application by including the pre-built Java connector wrapper object in your external application. Use this option if you do not need custom methods or an interactive drawing surface. See [Chapter 10, “Using the Connector Object”](#) for details and examples.

The general process for each of these examples involves the following:

1. Create the object in IDL.
2. Export the object using the Export Bridge Assistant, which creates the files associated with the IDL object that is now wrapped in a Java export object wrapper.
3. Access the object in a Java application. This functionality automatically includes stock methods (described in [Chapter 9, “Using Exported Java Objects”](#)) in addition to custom methods you have chosen to export.
4. Compile and execute the application with a classpath reference to `javaidlb.jar`.

Note

See [“Writing IDL Objects for Exporting”](#) on page 261 for information on how to create custom IDL objects that can be successfully exported using the Export Bridge Assistant. There are important object method and data type requirements that must be met.

Note

When you distribute an application, you will also need to share:

- the executable (.class) file(s) including those generated by the Assistant
- the .pro or .sav file that contains the custom IDL object definition

Any .pro or .sav files included with your application must be located in the IDL path. Also, `IDL_DIR/resource/bridges/export/java/javaidlb.jar` must be in the Java classpath.

Note on Running the Java Examples

Examples in this chapter provide Windows-style compile `javac` (compile) and `java` (run) commands. If you are running on a platform other than Windows, use your platform's path and directory separators and see [“Java Requirements”](#) on page 143 for information about the `bridge_setup` file, which sets additional information.

Debugging Applications Using Export Objects

It can be challenging to determine what is happening in the IDL process associated with a wrapper object without the debugging features of the IDL Workbench. For access to valuable debug information, consider using the `IDL_BRIDGE_DEBUG` environment variable, described in [“Debugging”](#) on page 244.

Nondrawable Java Export Example

Nondrawable objects provide access to the enormous processing power of IDL, but do not provide IDL drawing capabilities. This is useful for applications that need the data manipulation capabilities of IDL, but have no need for, or have independent drawing capabilities.

Hello World Java Example with Additional Method

The following simple example creates an IDL object with a single function method that accepts one argument, and walks through the process of exporting the object using the Export Bridge Assistant. Once the export files are created, a simple Java application shows how to access the object method and capture its output.

Complete the following steps to duplicate this example.

1. In an IDL Editor window, copy in the following code and save the file as `helloworldex__define.pro` in your working directory:

```
; Method returns message based on presence or
; absence of argument.
FUNCTION helloworldex::HelloFrom, who
    IF (N_ELEMENTS(who) NE 0) THEN BEGIN
        message = "Hello World from " + who
        RETURN, message
    ENDIF ELSE BEGIN
        message = 'Hello World'
        RETURN, message
    ENDELSE
END

; Init returns object reference on successful
; initialization.
FUNCTION helloworldex::INIT
    RETURN, 1
END

; Object definition.
PRO helloworldex__define
    struct = {helloworldex, $
        who: ' ', $
        message: ' ' $
    }
END
```

Note

It is a good idea to test the functionality of an object before exporting it. After compiling the file, enter the following lines at the command line and make sure the output is what is expected for this object.

```
ohello = OBJ_NEW("HELLOWORLDEX")
PRINT, ohello->HelloFrom()
PRINT, ohello->HelloFrom('Mr. Bill')
```

2. Open the Export Bridge Assistant by entering IDLEXBR_ASSISTANT at the command line.
3. Select to create a Java export object by selecting **File** → **New Project** → **Java** and browse to select the helloworldex__define.pro file. Click **Open** to load the file into the Export Assistant.

Note

Export Bridge Assistant details are available in [“Specifying Information for Exporting”](#) on page 164. Refer to that section if you need more information about the following steps.

4. The top-level project entry in the left-hand tree panel is selected by default. There is no need to modify the default properties shown in the right-hand property panel, but you can enter different values if desired. Set other export object characteristics as described in the following table. Select the tree view item listed in the left column to configure the related properties in the right column.

Tree View Item	Parameter Configuration
IDL Export Bridge Project	Accept the default value or make changes: <ul style="list-style-type: none"> • Output classname • Process name • Output directory
helloworldex	Drawable object equals False
Package name	helloworldex

Table 13-1: Example Export Object Parameters

Tree View Item	Parameter Configuration
HELLOFROM method	<p>Output method name — accept the default value, HELLOFROM</p> <p>Return Type — JIDLString since this function method returns a string message (as defined in the IDL object definition structure)</p> <p>Array — False since this method returns a single string, not an array</p> <p>Export — True</p>
WHO argument	<p>Mutability — In since the argument is not passed back to the caller</p> <p>Type — JIDLString since this argument is defined as a string in the IDL object definition</p> <p>Array — False</p> <p>Export — True</p>

Table 13-1: Example Export Object Parameters

5. Save the project by selecting **File** → **Save project**. Accept the default name and location or make changes as desired.
6. Verify that the object elements you want to export are listed in the Export log panel. If the expected items are not present, one or more items may still have an UNSPECIFIED field value that must be changed.
7. Build the export object by selecting **Build** → **Build object**. The **Build log** panel shows the results of the build process. A subdirectory, named `helloworldex` (based on the object name), contains the `.java` and `.class` files, and is located in the **Output directory**.

Using the Export Wrapper Object

The following simple Java application uses the wrapper object created in the previous section.

Example Code

The file for this example, `helloworldex_example.java`, is located in the `examples/doc/bridges/java` subdirectory of the IDL distribution.

1. Open the file named `helloworldex_example.java` in the previously referenced directory and save the file in the `helloworldex` directory.
2. Open the Windows Command window by selecting **Start** → **Run** and enter `cmd` in the textbox.
3. Use the `cd` command to change to the directory containing the `helloworldex` directory.
4. Reference the classpath of `javaidl.jar` in the compile statement. Enter the following two commands (as single lines) to compile and execute the program, replacing `IDL_DIR` with the IDL installation directory:

```
javac -classpath
      ".;IDL_DIR\resource\bridges\export\java\javaidl.jar"
      helloworldex\helloworldex_example.java
java -classpath
     ".;IDL_DIR\resource\bridges\export\java\javaidl.jar"
     helloworldex.helloworldex_example
```

Tip

See [“Note on Running the Java Examples”](#) on page 293 for information on executing Java commands on a non-Windows platform.

After compiling and running the project, the output message will appear in the command window.

Drawable Java Export Examples

A Java export object that supports graphics must be based on a custom IDL object that inherits from `IDLgrWindow`, `IDLitWindow`, or `IDLitDirectWindow` (as described in [“Exporting Drawable Objects”](#) on page 264). Additionally, your IDL object must also implement a set of listeners if you want to be able to respond to keyboard or mouse events in the graphics window. These are described in [“Event Handling”](#) on page 232. Examples in this section include:

- [“Java IDLgrWindow Based Histogram Plot Generator”](#) on page 299 — provides an object based on `IDLgrWindow` that creates a histogram plot for a selected image file and lets you change the plot linestyle property.
- [“Java IDLitWindow Surface Manipulation”](#) on page 304 — includes a drawable `IDLitWindow` example with `ISURFACE` functionality and a custom method lets you change the active manipulator. The main class is subclassed to pass key events to IDL. In the `OnKeyboard` method, Delete key events are captured and selected visualizations are deleted.

Java IDLgrWindow Based Histogram Plot Generator

This drawable object example inherits from IDLgrWindow and creates a histogram plot for a selected monochrome or RGB image file. While this example does contain several custom methods including those for opening a file, creating the plots, and changing plot characteristics, it does not use keyboard or mouse events. See [“Java IDLgrWindow Surface Manipulation”](#) on page 304 for such an example.

Example Code

The object definition file, `export_grwindow_doc__define.pro` is located in the `examples/doc/bridges` subdirectory of the IDL distribution. Run the example procedure by entering `export_grwindow_doc__define` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT .EDIT export_grwindow_doc__define.pro`.

Complete the following steps to duplicate this example.

1. In an IDL Editor window, open the object definition file by entering `.EDIT export_grwindow_doc__define.pro` at the command prompt.

Note

It is a good idea to test the functionality of an object before exporting it. After compiling the file, enter the following lines at the command line and make sure the output is what is expected for this object.

```
oPlotWindow = OBJ_NEW("export_grwindow_doc")
oPlotWindow->CHANGELINE, 2
```

This will display a three channel histogram plot and change the plot linestyle to dashed.

2. Open the Export Bridge Assistant by entering `IDLEXBR_ASSISTANT` at the command line.
3. Select to create a Java export object by selecting **File** → **New Project** → **Java** and browse to select `export_grwindow_doc__define.pro`. Click **Open** to load the file into the Export Assistant.

Note

Export Bridge Assistant details are available in [“Specifying Information for Exporting”](#) on page 164. Refer to that section if you need more information about the following items.

4. Set export object characteristics as described in the following table. When you select the tree view item listed in the left column, configure the related properties as noted in the right column.

Note

Set the Export parameter to True for all items in this list unless otherwise noted.

Tip

You can select multiple items in the tree view and set properties for the group.

Tree View Item	Property Configuration
IDL Export Bridge Project	Accept the default value or make changes as desired: <ul style="list-style-type: none"> • Output classname • Process name • Output directory
export_grwindow_doc	Drawable object equals True
Package name	export_grwindow_doc
OMODEL property OVIEW property OXAXIS property OXTTEXT property OYAXIS property OYTEXT property	Type — JIDLObjectI Array — False
OPLOTCOLL property	Type — JIDLObjectI Array — True
SFILE property	Type — JIDLString Array — False

Table 13-2: Example Export Object Parameters

Tree View Item	Property Configuration
CHANGELINE method	Enter different name if desired and mark Export as True
STYLE argument	Mutability — In Type — JIDLNumber Array — False
CREATEPLOTS method	Enter different name if desired and mark Export as True
IMAGE argument VROWS argument VCOLS argument	Mutability — In Type — JIDLNumber Array — True Convert majority — False
VRGB argument	Mutability — In Type — JIDLNumber Array — False
OPEN method	Enter different name if desired and mark Export as True
SFILE argument	Mutability — In Type — JIDLString Array — False NOTE: You can choose not to export this parameter. If so, the method follows the path for cases where no argument is defined. (You will need to modify the Java code to read <code>export_grwindow_doc.OPEN()</code> instead of passing an argument.) If you do choose to export this method, the argument must either be a null string or a full file path.

Table 13-2: Example Export Object Parameters

5. Save the project by selecting **File** → **Save project**. Accept the default name and location or make changes as desired.

6. Verify that the object elements you want to export are listed in the Export log panel. If the expected items are not present, one or more items may still have an UNSPECIFIED field value that must be changed.
7. Build the export object by selecting **Build** → **Build object**. The **Build log** panel shows the results of the build process. A subdirectory, named `export_grwindow_doc` (based on the object name), contains the `.java` and `.class` files, and is located in the **Output directory**.

Using the Java Export Object

The following section describes using the Java export object in a simple application.

Example Code

The file for this example, `export_grwindow_doc_example.java`, is located in the `examples/doc/bridges/java` subdirectory of the IDL distribution.

1. Open the file named `export_grwindow_doc_example.java` in the location referenced above and copy it to your `<output directory>/export_grwindow_doc` directory where `<output directory>` was the directory specified as the **Output directory** in the Assistant.
2. Open the Windows Command window by selecting **Start** → **Run** and enter `cmd` in the textbox.
3. Use the `cd` command to change to the directory containing the `export_grwindow_doc` directory.
4. Reference the classpath of `javaidl.jar` in the compile statement. Enter the following two commands (as single lines) to compile and execute the program, replacing `IDL_DIR` with the IDL installation directory:

```
javac -classpath
      ".;IDL_DIR\resource\bridges\export\java\javaidl.jar"
      export_grwindow_doc\export_grwindow_doc_example.java
java -classpath
     ".;IDL_DIR\resource\bridges\export\java\javaidl.jar"
     export_grwindow_doc.export_grwindow_doc_example
```

Tip

See “[Note on Running the Java Examples](#)” on page 293 for information on executing Java commands on a non-Windows platform.

After compiling and running the project, a Java interface will display a histogram plot of an RGB image. You can change the linestyle of the plot by making a selection from the listbox. You can also create a histogram plot for a new image by clicking the button.

Java IDLitWindow Surface Manipulation

This drawable object example inherits from IDLitWindow and creates an ISURFACE display in a Java application. A listbox is populated with manipulator string values that, when selected, allow you to draw annotations, rotate, or zoom within the exported IDLitWindow object. You should avoid exposing any manipulator that has an associated widget interface (such as a profile line manipulator) since such widget functionality is not supported in objects that subclass from IDLitWindow.

Example Code

The object definition file, `export_itwinmanip_doc__define.pro` is located in the `examples/doc/bridges` subdirectory of the IDL distribution. Run the example procedure by entering `export_itwinmanip_doc__define` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT export_itwinmanip_doc__define.pro`.

Complete the following steps to duplicate this example.

1. In an IDL Editor window, open the object definition file by entering `.EDIT export_itwinmanip_doc__define.pro` at the command prompt.

Note

It is a good idea to test the functionality of an object before exporting it. After compiling the file, enter the following lines at the command line and make sure the output is what is expected for this object.

```
oWin = OBJ_NEW("export_itwinmanip_doc")
oWin->CHANGEMANIPULATOR, "annotation/oval"
```

This will let you draw a oval annotation in the window. If you hit the Delete key, the annotation will be removed. The available manipulator strings are printed in the IDL output window.

2. Open the Export Bridge Assistant by entering `IDLEXBR_ASSISTANT` at the command line.
3. Select to create a Java export object by selecting **File** → **New Project** → **Java** and browse to select `export_itwinmanip_doc__define.pro`. Click **Open** to load the file into the Export Assistant.

Note

Export Bridge Assistant details are available in [“Specifying Information for Exporting”](#) on page 164. Refer to that section if you need more information about the following items.

4. Set export object characteristics as described in the following table. When you select the tree view item listed in the left column, configure the related properties as noted in the right column.

Tree View Item	Property Configuration
IDL Export Bridge Project	Accept the default value or make changes as desired: <ul style="list-style-type: none"> • Output classname • Process name • Output directory
export_itwinmanip_doc	Drawable object equals True
Package name	export_itwinmanip_doc
CHANGEMANIPULATOR method	Enter different name if desired and mark Export as True
MANIPID argument	Mutability — In Type — JIDLString Array — False Export — True

Table 13-3: Example Export Object Parameters

5. Save the project by selecting **File** → **Save project**. Accept the default name and location or make changes as desired.
6. Verify that the object elements you want to export are listed in the Export log panel. If the expected items are not present, one or more items may still have an UNSPECIFIED field value that must be changed.
7. Build the export object by selecting **Build** → **Build object**. The **Build log** panel shows the results of the build process. A subdirectory, named

`export_itwinmanip_doc` (based on the object name), contains the `.java` and `.class` files, and is located in the **Output directory**.

Using the Java Export Object

The following section describes using the Java export object in a simple application.

Example Code

The files for this example, `export_itwinmanip_doc_example.java`, and `export_itwinmanip_delete.java`, are located in the `examples/doc/bridges/java` subdirectory of the IDL distribution.

In this example, the `export_itwinmanip_doc_example.java` file contains the code to display the listbox and IDLitWindow drawing canvas. The `export_itwinmanip_delete.java` file subclasses the previous file and handles key press events, passing them on to the IDL object `OnKeyboard` method so that selected visualizations can be deleted.

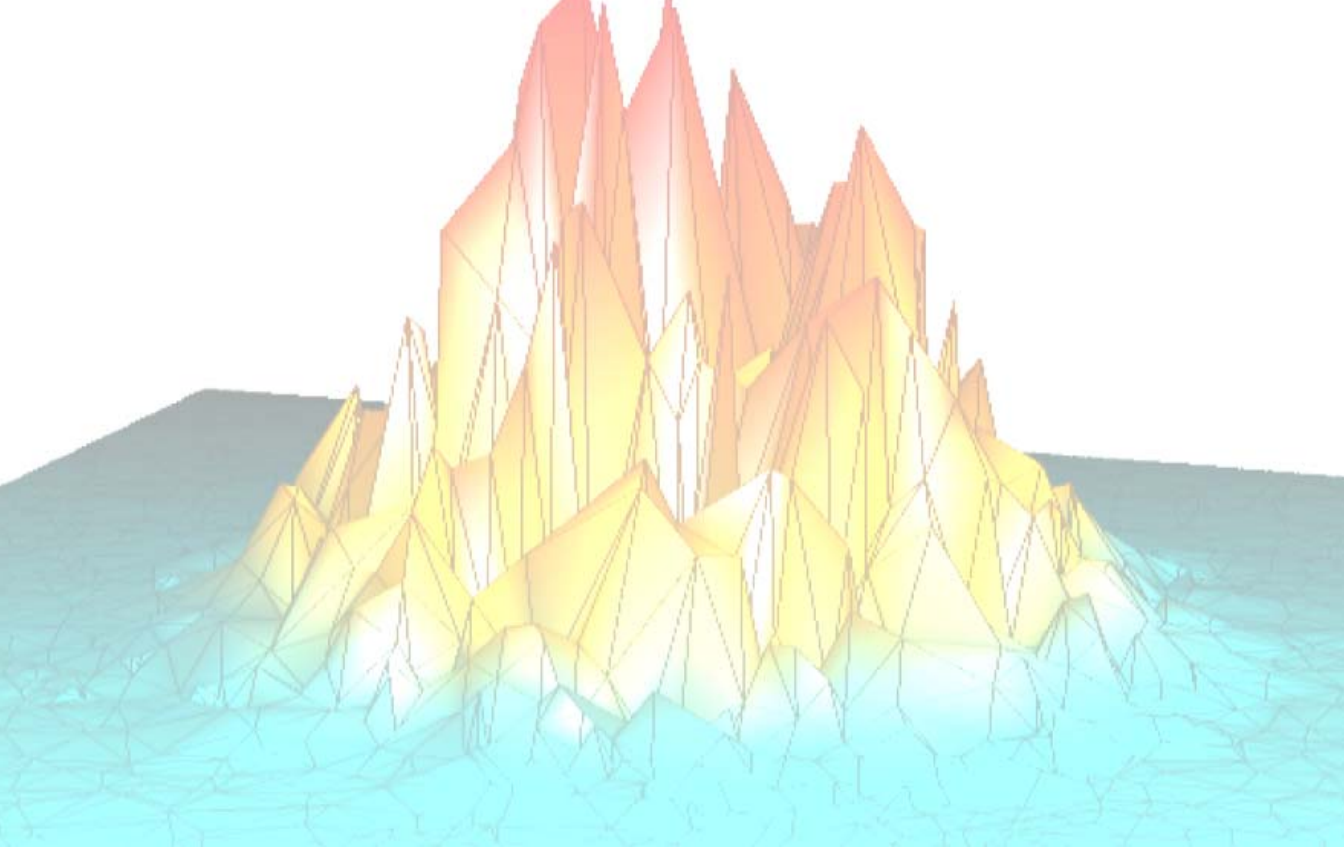
1. Open the files named `export_itwinmanip_doc_example.java` and `export_itwinmanip_delete.java` in the location referenced above and copy them to your `<output directory>/export_itwinmanip_doc` directory where `<output directory>` was the directory specified as the **Output directory** in the Assistant.
2. Open the Windows Command window by selecting **Start** → **Run** and enter `cmd` in the textbox.
3. Use the `cd` command to change to the directory containing the `export_itwinmanip_doc` directory.
4. Reference the classpath of `javaidl.jar` in the compile statement. Enter the following two commands (as single lines) to compile and execute the program, replacing `IDL_DIR` with the IDL installation directory:

```
javac -classpath
      ".;IDL_DIR\resource\bridges\export\java\javaidl.jar"
      export_itwinmanip_doc\*.java
java -classpath
     ".;IDL_DIR\resource\bridges\export\java\javaidl.jar"
     export_itwinmanip_doc.export_itwinmanip_doc_example
```

Tip

See [“Note on Running the Java Examples”](#) on page 293 for information on executing Java commands on a non-Windows platform.

After compiling and running the project, a Java interface will display a surface in an IDLitWindow. Select a manipulator from the listbox to use it in the IDLitWindow display. If you hit the keyboard **Delete** key while visualizations are selected, they will be removed from the view.



Part III: Appendices



Appendix A

IDL Java Object API

This chapter describes the IDL Java package interfaces, classes and errors.

Package Summary	312
---------------------------------------	-----

Package Summary

This chapter describes the IDL Java Package in a format similar to JavaDoc.

Class Summary	
Interfaces	
<code>JIDLComponentListener</code>	The listener interface for receiving component events (expose, resize) on a <code>JIDLCanvas</code> .
<code>JIDLKeyListener</code>	The listener interface for receiving keyboard events (key pressed, key released) on a <code>JIDLCanvas</code> .
<code>JIDLMouseListener</code>	The listener interface for receiving mouse events from IDL (press, release, enter, and exit) on a <code>JIDLCanvas</code> .
<code>JIDLMouseMotionListener</code>	The listener interface for receiving mouse motion events from IDL (move and drag) on a <code>JIDLCanvas</code> .
<code>JIDLMouseWheelListener</code>	The listener interface for receiving mouse wheel events from IDL on a <code>JIDLCanvas</code> .
<code>JIDLNotifyListener</code>	The listener interface for receiving notify events from IDL.
<code>JIDLNumber</code>	The <code>JIDLNumber</code> class wraps a primitive java number as a mutable object usable by the Java-IDL Export bridge.
<code>JIDLObjectI</code>	The interface that wrapped IDL objects must implement.
<code>JIDLOutputListener</code>	The listener interface for receiving output events from IDL.
Classes	
<code>JIDLArray</code>	The <code>JIDLArray</code> class wraps a Java array as an object usable by the Java-IDL Export bridge.

Class Summary	
<code>JIDLBoolean</code>	The <code>JIDLBoolean</code> class wraps a boolean as a mutable object usable by the Java-IDL Export bridge.
<code>JIDLByte</code>	The <code>JIDLByte</code> class wraps a byte as a mutable object usable by the Java-IDL Export bridge.
<code>JIDLCanvas</code>	This class wraps an IDL object of type <code>IDLitWindow</code> in a <code>java.awt.Canvas</code> providing direct rendering of the object from IDL.
<code>JIDLChar</code>	The <code>JIDLChar</code> class wraps a char as a mutable object usable by the Java-IDL Export bridge.
<code>JIDLConst</code>	Contains constants used by the Java-IDL wrapper classes.
<code>JIDLDouble</code>	The <code>JIDLDouble</code> class wraps a double as a mutable object usable by the Java-IDL Export bridge.
<code>JIDLFloat</code>	The <code>JIDLFloat</code> class wraps a float as a mutable object usable by the Java-IDL Export bridge.
<code>JIDLInteger</code>	The <code>JIDLInteger</code> class wraps an int as a mutable object usable by the Java-IDL Export bridge.
<code>JIDLLong</code>	The <code>JIDLLong</code> class wraps a long as a mutable object usable by the Java-IDL Export bridge.
<code>JIDLObject</code>	This class wraps an IDL object.
<code>JIDLProcessInitializer</code>	The <code>JIDLProcessInitializer</code> class provides a mechanism to pass licensing initialization parameters to the <code>JIDLCanvas</code> and <code>JIDLObject</code> <code>createObject</code> methods.
<code>JIDLShort</code>	The <code>JIDLShort</code> class wraps a short as a mutable object usable by the Java-IDL Export bridge.
<code>JIDLString</code>	The <code>JIDLString</code> class wraps a String as a mutable object usable by the Java-IDL Export bridge.
Errors	

Class Summary	
<code>JIDLAbortedException</code>	Thrown when a call to IDL is interrupted by an abort request.
<code>JIDLBusyException</code>	Thrown when a call to IDL is not executed because the current IDL process is busy.
<code>JIDLException</code>	An unchecked exception thrown when a call to IDL encounters an error.

JIDLAbortedException

Declaration

```
public class JIDLAbortedException extends JIDLException
implements java.io.Serializable

java.lang.Object
|
+--java.lang.Throwable
|
+--java.lang.Error
|
+--com.idl.javaidl.JIDLException
|
+--com.idl.javaidl.JIDLAbortedException
```

All Implemented Interfaces:

```
java.io.Serializable
```

Description

An unchecked exception thrown when a call to IDL is interrupted by an abort request.

Inherited Member Summary
Methods inherited from interface JIDLException
getErrorCode() , toString()
Methods inherited from class Object

Inherited Member Summary
<code>equals(Object), getClass(), hashCode(), notify(), notifyAll(), wait(long, int), wait(long, int), wait(long, int)</code>
Methods inherited from class <code>Throwable</code>
<code>fillInStackTrace(), getCause(), getLocalizedMessage(), getMessage(), getStackTrace(), initCause(Throwable), printStackTrace(PrintWriter), printStackTrace(PrintWriter), printStackTrace(PrintWriter), setStackTrace(StackTraceElement[])</code>

JIDLArray

Declaration

```
public class JIDLArray implements java.io.Serializable  
  
java.lang.Object  
|  
+---com.idl.javaidl.JIDLArray
```

All Implemented Interfaces:

```
java.io.Serializable
```

Description

The JIDLArray class wraps a Java array as an object usable by the Java-IDL Export bridge.

Wraps arrays of type boolean, byte, char, short, int, long, float, double, String, and JIDLObjectI.

When retrieving the object, the calling code must cast the Object wrapped by JIDLArray to the proper * array type. For example:

```
int[] myNativeArray = ...;  
// Create a wrapped array so it may be used in the bridge  
JIDLArray arr = new JIDLArray(myNativeArray)  
// ... do something in the bridge to modify the array ...  
  
// Now cast the resultant array to the expected type  
int[] newNative = (int[])arr.arrayValue();
```

Member Summary	
Constructors	
	JIDLArray(java.lang.Object arr) Construct a JIDLArray from a native array
Methods	

Member Summary	
java.lang.Object	<code>arrayValue()</code> Get the native array that is wrapped by this object
java.lang.String	<code>getClassName()</code> Get the classname of the wrapped array.
java.lang.Object	<code>getValue()</code> Get the native array that is wrapped by this object
void	<code>setValue(JIDLArray arr)</code> Set the native array that is wrapped by this object
void	<code>setValue(java.lang.Object arr)</code> Set the native array that is wrapped by this object
java.lang.String	<code>toString()</code>

Inherited Member Summary
Methods inherited from class Object
<code>equals(Object), getClass(), hashCode(), notify(), notifyAll(), wait(long, int), wait(long, int), wait(long, int)</code>

Constructors

JIDLArray(Object)

```
public JIDLArray(java.lang.Object arr)
```

Construct a JIDLArray from a native array

Parameters:

`arr` - the native array to wrap for use in the export bridge (Must be an array of type boolean, byte, char, short, int, long, float, double, String, or JIDLObjectI.)

Methods

arrayValue()

```
public java.lang.Object arrayValue()
```

Get the native array that is wrapped by this object

Returns:

the native array to wrap for use in the export bridge returned as an object. The array will be of type boolean, byte, char, short, int, long, float, double, String, or JIDLObjectI.

getClassName()

```
public java.lang.String getClassName()
```

Get the classname of the wrapped array.

Returns:

The classname of the wrapped array.

getValue()

```
public java.lang.Object getValue()
```

Get the native array that is wrapped by this object

Returns:

the native array to wrap for use in the export bridge returned as an object. The array will be of type boolean, byte, char, short, int, long, float, double, String, or JIDLObjectI.

setValue(JIDLArray)

```
public void setValue(com.idl.javaidl.JIDLArray arr)
```

Set the native array that is wrapped by this object

Parameters:

`arr` - the native array to wrap for use in the export bridge (Must be an array of type boolean, byte, char, short, int, long, float, double, String, or JIDLObjectI.)

setValue(Object)

```
public void setValue(java.lang.Object arr)
```

Set the native array that is wrapped by this object

Parameters:

`arr` - the native array to wrap for use in the export bridge (Must be an array of type boolean, byte, char, short, int, long, float, double, String, or JIDLObjectI.)

toString()

```
public java.lang.String toString()
```

Overrides:

toString in class Object

JIDLBoolean

Declaration

```
public class JIDLBoolean implements JIDLNumber,  
    java.io.Serializable  
  
    java.lang.Object  
    |  
    +---com.idl.javaidl.JIDLBoolean
```

All Implemented Interfaces:

[JIDLNumber](#), [java.io.Serializable](#)

Description

The JIDLBoolean class wraps a boolean as a mutable object usable by the Java-IDL Export bridge.

Member Summary	
Constructors	
	JIDLBoolean(boolean value) Construct a wrapper object.
	JIDLBoolean(JIDLNumber value) Construct a wrapper object.
Methods	
boolean	booleanValue() Return the value of the wrapped primitive
byte	byteValue() Return the value of the wrapped primitive
char	charValue() Return the value of the wrapped primitive
double	doubleValue() Return the value of the wrapped primitive

Member Summary	
float	<code>floatValue()</code> Return the value of the wrapped primitive
int	<code>intValue()</code> Return the value of the wrapped primitive
long	<code>longValue()</code> Return the value of the wrapped primitive
void	<code>setValue(boolean value)</code> Change the value of the wrapper object
void	<code>setValue(JIDLNumber value)</code> Change the value of the wrapper object
short	<code>shortValue()</code> Return the value of the wrapped primitive
java.lang.String	<code>toString()</code>

Inherited Member Summary
Methods inherited from class <code>Object</code>
<code>equals(Object)</code> , <code>getClass()</code> , <code>hashCode()</code> , <code>notify()</code> , <code>notifyAll()</code> , <code>wait(long, int)</code> , <code>wait(long, int)</code> , <code>wait(long, int)</code>

Constructors

JIDLBoolean(boolean)

```
public JIDLBoolean(boolean value)
```

Construct a wrapper object.

Parameters:

value - value to wrap for use in the export bridge

JIDLBoolean(JIDLNumber)

```
public JIDLBoolean(com.idl.javaidl.JIDLNumber value)
```

Construct a wrapper object.

Parameters:

value - JIDLNumber to wrap for use in the export bridge

Methods

booleanValue()

```
public boolean booleanValue()
```

Return the value of the wrapped primitive

Specified By:

[booleanValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

byteValue()

```
public byte byteValue()
```

Return the value of the wrapped primitive

Specified By:

[byteValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

charValue()

```
public char charValue()
```

Return the value of the wrapped primitive

Specified By:

[charValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

doubleValue()

```
public double doubleValue()
```

Return the value of the wrapped primitive

Specified By:

[doubleValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

floatValue()

```
public float floatValue()
```

Return the value of the wrapped primitive

Specified By:

[floatValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

intValue()

```
public int intValue()
```

Return the value of the wrapped primitive

Specified By:

[intValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

longValue()

```
public long longValue()
```

Return the value of the wrapped primitive

Specified By:

[longValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

setValue(boolean)

```
public void setValue(boolean value)
```

Change the value of the wrapper object

Parameters:

value - primitive value to wrap for use in the export bridge

setValue(JIDLNumber)

```
public void setValue(com.idl.javaidl.JIDLNumber value)
```

Change the value of the wrapper object

Specified By:

[setValue](#) in interface [JIDLNumber](#)

Parameters:

value - JIDLNumber to wrap for use in the export bridge

shortValue()

```
public short shortValue()
```

Return the value of the wrapped primitive

Specified By:

[shortValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

toString()

```
public java.lang.String toString()
```

Overrides:

[toString](#) in class [Object](#)

JIDLBusyException

Declaration

```
public class JIDLBusyException extends JIDLException implements
java.io.Serializable

java.lang.Object
|
+--java.lang.Throwable
|
+--java.lang.Error
|
+--com.idl.javaidl.JIDLException
|
+--com.idl.javaidl.JIDLBusyException
```

All Implemented Interfaces:

```
java.io.Serializable
```

Description

An unchecked exception thrown when a call to IDL is not executed because the current IDL process is busy.

Inherited Member Summary
Methods inherited from interface JIDLException
getErrorCode() , toString()
Methods inherited from class <code>Object</code>

Inherited Member Summary
<code>equals(Object), getClass(), hashCode(), notify(), notifyAll(), wait(long, int), wait(long, int), wait(long, int)</code>
Methods inherited from class <code>Throwable</code>
<code>fillInStackTrace(), getCause(), getLocalizedMessage(), getMessage(), getStackTrace(), initCause(Throwable), printStackTrace(PrintWriter), printStackTrace(PrintWriter), printStackTrace(PrintWriter), setStackTrace(StackTraceElement[])</code>

JIDLByte

Declaration

```
public class JIDLByte implements JIDLNumber,  
    java.io.Serializable  
  
    java.lang.Object  
    |  
    +--com.idl.javaidl.JIDLByte
```

All Implemented Interfaces:

[JIDLNumber](#), [java.io.Serializable](#)

Description

The JIDLByte class wraps a byte as a mutable object usable by the Java-IDL Export bridge.

Member Summary	
Constructors	
	JIDLByte(byte value) Construct a wrapper object.
	JIDLByte(JIDLNumber value) Construct a wrapper object.
Methods	
boolean	booleanValue() Return the value of the wrapped primitive.
byte	byteValue() Return the value of the wrapped primitive
char	charValue() Return the value of the wrapped primitive
double	doubleValue() Return the value of the wrapped primitive

Member Summary	
float	<code>floatValue()</code> Return the value of the wrapped primitive
int	<code>intValue()</code> Return the value of the wrapped primitive
long	<code>longValue()</code> Return the value of the wrapped primitive
void	<code>setValue(byte value)</code> Change the value of the wrapper object
void	<code>setValue(JIDLNumber value)</code> Change the value of the wrapper object
short	<code>shortValue()</code> Return the value of the wrapped primitive
java.lang.String	<code>toString()</code>

Inherited Member Summary
Methods inherited from class Object
<code>equals(Object)</code> , <code>getClass()</code> , <code>hashCode()</code> , <code>notify()</code> , <code>notifyAll()</code> , <code>wait(long, int)</code> , <code>wait(long, int)</code> , <code>wait(long, int)</code>

Constructors

JIDLByte(byte)

```
public JIDLByte(byte value)
```

Construct a wrapper object.

Parameters:

`value` - value to wrap for use in the export bridge

JIDLByte(JIDLNumber)

```
public JIDLByte(com.idl.javaidl.JIDLNumber value)
```

Construct a wrapper object.

Parameters:

value - JIDLNumber to wrap for use in the export bridge

Methods

booleanValue()

```
public boolean booleanValue()
```

Return the value of the wrapped primitive.

Specified By:

[booleanValue](#) in interface [JIDLNumber](#)

Returns:

true if non-zero, false otherwise

byteValue()

```
public byte byteValue()
```

Return the value of the wrapped primitive

Specified By:

[byteValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

charValue()

```
public char charValue()
```

Return the value of the wrapped primitive

Specified By:

[charValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

doubleValue()

```
public double doubleValue()
```

Return the value of the wrapped primitive

Specified By:

[doubleValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

floatValue()

```
public float floatValue()
```

Return the value of the wrapped primitive

Specified By:

[floatValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

intValue()

```
public int intValue()
```

Return the value of the wrapped primitive

Specified By:

[intValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

longValue()

```
public long longValue()
```

Return the value of the wrapped primitive

Specified By:

[longValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

setValue(byte)

```
public void setValue(byte value)
```

Change the value of the wrapper object

Parameters:

value - primitive value to wrap for use in the export bridge

setValue(JIDLNumber)

```
public void setValue(com.idl.javaidl.JIDLNumber value)
```

Change the value of the wrapper object

Specified By:

[setValue](#) in interface [JIDLNumber](#)

Parameters:

value - JIDLNumber to wrap for use in the export bridge

shortValue()

```
public short shortValue()
```

Return the value of the wrapped primitive

Specified By:

[shortValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

toString()

```
public java.lang.String toString()
```

Overrides:

[toString](#) in class [Object](#)

JIDLCanvas

Declaration

```
public abstract class JIDLCanvas extends java.awt.Canvas
implements JIDLObjectI, java.awt.event.ComponentListener,
java.awt.event.KeyListener, java.awt.event.MouseListener,
java.awt.event.MouseMotionListener, JIDLMouseListener,
JIDLMouseMotionListener, JIDLKeyListener, JIDLComponentListener,
JIDLCursorSupport

java.lang.Object
|
+--java.awt.Component
|
+---java.awt.Canvas
|
+---com.idl.javaidl.JIDLCanvas
```

All Implemented Interfaces:

```
javax.accessibility.Accessible,
java.awt.event.ComponentListener, java.util.EventListener,
java.awt.image.ImageObserver, JIDLComponentListener,
JIDLCursorSupport, JIDLKeyListener, JIDLMouseListener,
JIDLMouseMotionListener, JIDLObjectI,
java.awt.event.KeyListener, java.awt.MenuContainer,
java.awt.event.MouseListener,
java.awt.event.MouseMotionListener, java.io.Serializable
```

Description

This class wraps an IDL object of type IDLitWindow in a java.awtCanvas providing direct rendering of the object from IDL.

Note

JIDLCanvas is not supported on Macintosh OS X.

In many of the methods of this class, one or more flags are required to be specified for parameters being passed to or from the bridge. These flags follow the following guidelines:

For all types of parameters (subclasses of JIDLNumber, JIDLString, JIDLObjectI and JIDLArray), a flag should be set that determines whether the parameter is in-only

(const) or in-out (we expect it to be changed by IDL). The constants that determine this are either `JIDLConst.PARMFLAG_CONST` or `JIDLConst.PARMFLAG_IN_OUT`.

For parameters that are arrays, a flag should be set that tells the bridge whether the array is to be convolved when passed to IDL. If the `PARM_IN_OUT` flag is set, this flag will also tell the bridge whether to convolve the array when it is copied back to Java. The constants that determine this are either `JIDLConst.PARMFLAG_CONVMAJORITY` or `JIDLConst.PARMFLAG_NO_CONVMAJORITY`.

For example, if the parameter in question is an array that is to be modified by IDL (in-out) and needs to be convolved when passed to and from IDL, we would set its `argpal` array member as follows:

```
argpal[2] = JIDLConst.PARMFLAG_IN_OUT | JIDLConst.PARMFLAG_CONV
MAJORITY;
```

Member Summary	
Fields	
static int	<code>IDL_SOFTWARE_RENDERER</code> Internal use
static int	<code>OPENGL_RENDERER</code> Internal use
Constructors	
	<code>JIDLCanvas(java.lang.String sClass, int iOPSFlags, java.lang.String sProcessName)</code> Construct a <code>JIDLCanvas</code>
	<code>JIDLCanvas(java.lang.String sClass, java.lang.String sProcessName)</code> Construct a <code>JIDLCanvas</code> Note that constructing the <code>JIDLObject</code> does NOT create the object on the IDL-side of the bridge.
Methods	
void	<code>abort()</code> Requests that the IDL process containing the underlying IDL object abort its current activity.

Member Summary	
void	<code>addIDLComponentListener(JIDLComponentListener listener)</code> Adds the specified JIDLComponentListener to a list of listeners that receive notification of Component events.
void	<code>addIDLKeyListener(JIDLKeyListener listener)</code> Adds the specified JIDLKeyListener to a list of listeners that receive notification of Key events.
void	<code>addIDLMouseListener(JIDLMouseListener listener)</code> Adds the specified JIDLMouseListener to a list of listeners that receive notification of Mouse events.
void	<code>addIDLMouseMotionListener(JIDLMouseMotionListener listener)</code> Adds the specified JIDLMouseMotionListener to a list of listeners that receive notification of MouseMotion events.
void	<code>addIDLMouseWheelListener(JIDLMouseWheelListener listener)</code> Adds the specified JIDLMouseWheelListener to a list of listeners that receive notification of MouseWheel events.
void	<code>addIDLNotifyListener(JIDLNotifyListener listener)</code> Adds the specified IDL notify listener to receive IDL notification events on this object.
void	<code>addIDLOutputListener(JIDLOutputListener listener)</code> Adds the specified IDL output listener to receive IDL output events on this object.
java.lang.Object	<code>callFunction(java.lang.String sMethodName, int iPalFlag)</code> Call IDL function that accepts zero parameters.
java.lang.Object	<code>callFunction(java.lang.String sMethodName, int argc, java.lang.Object argv, int[] argpal, int iPalFlag)</code> Call IDL function.
void	<code>callProcedure(java.lang.String sMethodName)</code> Call IDL procedure that accepts zero parameters.

Member Summary	
void	<code>callProcedure(java.lang.String sMethodName, int argc, java.lang.Object argv, int[] argpal)</code> Call IDL procedure.
void	<code>componentHidden(java.awt.event.ComponentEvent e)</code> Called when the component is hidden.
void	<code>componentMoved(java.awt.event.ComponentEvent e)</code> Called when the component is moved.
void	<code>componentResized(java.awt.event.ComponentEvent e)</code> Internal use.
void	<code>componentShown(java.awt.event.ComponentEvent e)</code> Called when the component is shown.
void	<code>createObject()</code> Create the wrapped object by calling IDL's ::INIT method.
void	<code>createObject(int argc, java.lang.Object argv, int[] argpal)</code> Create the wrapped object by calling IDL's ::INIT method.
void	<code>createObject(int argc, java.lang.Object argv, int[] argpal, com.idl.javaidl.JIDLProcessInitializer initializer)</code> Create the wrapped object by calling IDL's ::INIT method.
void	<code>createObject(com.idl.javaidl.JIDLProcessInitializer initializer)</code> Create the wrapped object by calling IDL's ::INIT method.
void	<code>destroyObject()</code> Destroys the underlying IDL object associated with the wrapper.
void	<code>draw()</code> Internal use.
void	<code>executeString(java.lang.String sCmd)</code> Execute the given command string in IDL.
java.lang.String	<code>getClassName()</code> Get the class name of the object.

Member Summary	
long	<code>getCookie()</code> Internal use.
java.lang.String	<code>getIdLObjectClassName()</code> Retrieves the IDL object class name of the underlying IDL object.
java.lang.String	<code>getIdLObjectVariableName()</code> When the underlying IDL object was created in the IDL process, it was assigned a variable name.
java.lang.Object	<code>getIdLVariable(java.lang.String sVar)</code> Given a variable name, return the IDL variable.
java.lang.String	<code>getObjVariableName()</code> Get the IDL Variable name of the given object
java.lang.String	<code>getProcessName()</code> Returns the name of the process that contains the underlying IDL object.
java.lang.Object	<code>getProperty(java.lang.String sProperty, int iPalFlag)</code> Call IDL <code>getProperty</code> method to get named property.
void	<code>IDLcomponentExposed(JIDLObjectI obj)</code> Called when the JIDLCanvas is exposed.
void	<code>IDLcomponentResized(JIDLObjectI obj, java.awt.event.ComponentEvent e)</code> Called when the JIDLCanvas is resized.
void	<code>IDLkeyPressed(JIDLObjectI obj, java.awt.event.KeyEvent e, int x, int y)</code> Called when the JIDLCanvas has focus and a key is pressed.
void	<code>IDLkeyReleased(JIDLObjectI obj, java.awt.event.KeyEvent e, int x, int y)</code> Called when the JIDLCanvas has focus and a key is released.
void	<code>IDLmouseDragged(JIDLObjectI obj, java.awt.event.MouseEvent e)</code> Called when the mouse is dragged in a JIDLCanvas.

Member Summary	
void	<code>IDLmouseEntered(JIDLObjectI obj, java.awt.event.MouseEvent e)</code> Called when the mouse enters a JIDLCanvas.
void	<code>IDLmouseExited(JIDLObjectI obj, java.awt.event.MouseEvent e)</code> Called when the mouse exits a JIDLCanvas.
void	<code>IDLmouseMoved(JIDLObjectI obj, java.awt.event.MouseEvent e)</code> Called when the mouse is moved in a JIDLCanvas.
void	<code>IDLmousePressed(JIDLObjectI obj, java.awt.event.MouseEvent e)</code> Called when the mouse is pressed in a JIDLCanvas.
void	<code>IDLmouseReleased(JIDLObjectI obj, java.awt.event.MouseEvent e)</code> Called when the mouse is released in a JIDLCanvas.
void	<code>initListeners()</code> Initialize listeners.
boolean	<code>isFocusTraversable()</code> Internal use.
boolean	<code>isObjCreated()</code> Determine if object has been created successfully.
boolean	<code>isObjectCreated()</code> Determine if object has been created successfully.
boolean	<code>isObjectDisplayable()</code>
void	<code>keyPressed(java.awt.event.KeyEvent e)</code> Internal use.
void	<code>keyReleased(java.awt.event.KeyEvent e)</code> Internal use.
void	<code>keyTyped(java.awt.event.KeyEvent e)</code> Internal use.
int	<code>mapIDLCursorToJavaCursor(java.lang.String idlCursor)</code> Maps the IDL cursor to a suitable Java cursor.

Member Summary	
void	<code>mouseClicked(java.awt.event.MouseEvent e)</code> Internal use.
void	<code>mouseDragged(java.awt.event.MouseEvent e)</code> Internal use.
void	<code>mouseEntered(java.awt.event.MouseEvent e)</code> Internal use.
void	<code>mouseExited(java.awt.event.MouseEvent e)</code> Internal use.
void	<code>mouseMoved(java.awt.event.MouseEvent e)</code> Internal use.
void	<code>mousePressed(java.awt.event.MouseEvent e)</code> Internal use.
void	<code>mouseReleased(java.awt.event.MouseEvent e)</code> Internal use.
void	<code>paint(java.awt.Graphics g)</code> Internal use.
void	<code>removeIDLComponentListener</code> <code>(JIDLComponentListener listener)</code> Remove the specified JIDLComponentListener from a list of listeners that receive notification of Component events.
void	<code>removeIDLKeyListener(JIDLKeyListener listener)</code> Removes the specified JIDLKeyListener from a list of listeners that receive notification of Key events.
void	<code>removeIDLMouseListener(JIDLMouseListener listener)</code> Removes the specified JIDLMouseListener from a list of listeners that receive notification of Mouse events.
void	<code>removeIDLMouseMotionListener</code> <code>(JIDLMouseMotionListener listener)</code> Removes the specified JIDLMouseMotionListener from a list of listeners that receive notification of MouseMotion events.

Member Summary	
void	removeIDLMouseWheelListener (JIDLMouseWheelListener listener) Removes the specified JIDLMouseWheelListener from a list of listeners that receive notification of MouseWheel events.
void	removeIDLNotifyListener (JIDLNotifyListener listener) Removes the specified IDL notify listener so it no longer receives IDL notifications.
void	removeIDLOutputListener (JIDLOutputListener listener) Removes the specified IDL output listener on this object.
void	setCursor (java.lang.String idlCursor) Set the JIDLCanvas cursor.
void	setIDLVariable (java.lang.String sVar, java.lang.Object obj) Set/Create an IDL variable of the given name and value.
void	setProcessName (java.lang.String process) Set the process name that the object will be created in.
void	setProperty (java.lang.String sProperty, java.lang.Object obj, int iPalFlag) Call IDL setProperty method to set named property.
java.lang.String	toString() Returns a string representation of the object.
void	update (java.awt.Graphics g) Internal use.

Inherited Member Summary
Fields inherited from class Component
BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT
Fields inherited from interface ImageObserver

Inherited Member Summary
ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH
Methods inherited from class Canvas
<code>addNotify(), createBufferStrategy(int, BufferCapabilities), createBufferStrategy(int, BufferCapabilities), getAccessibleContext(), getBufferStrategy()</code>
Methods inherited from class Component

Inherited Member Summary

```

action(Event, Object), add(PopupMenu),
addComponentListener(ComponentListener),
addFocusListener(FocusListener),
addHierarchyBoundsListener(HierarchyBoundsListener),
addHierarchyListener(HierarchyListener),
addInputMethodListener(InputMethodListener),
addKeyListener(KeyListener),
addMouseListener(MouseListener),
addMouseMotionListener(MouseMotionListener),
addMouseWheelListener(MouseWheelListener),
addPropertyChangeListener(String,
PropertyChangeListener), addPropertyChangeListener(String,
PropertyChangeListener),
applyComponentOrientation(ComponentOrientation),
areFocusTraversalKeysSet(int), bounds(), checkImage(Image,
ImageObserver), checkImage(Image, ImageObserver),
contains(Point), contains(Point),
createImage(ImageProducer), createImage(ImageProducer),
createVolatileImage(int, int, ImageCapabilities),
createVolatileImage(int, int, ImageCapabilities),
deliverEvent(Event), disable(), dispatchEvent(AWTEvent),
doLayout(), enable(boolean), enable(boolean),
enableInputMethods(boolean), getAlignmentX(),
getAlignmentY(), getBackground(), getBounds(Rectangle),
getBounds(Rectangle), getColorModel(),
getComponentAt(Point), getComponentAt(Point),
getComponentListeners(), getComponentOrientation(),
getCursor(), getDropTarget(), getFocusCycleRootAncestor(),
getFocusListeners(), getFocusTraversalKeys(int),
getFocusTraversalKeysEnabled(), getFont(),
getFontMetrics(Font), getForeground(), getGraphics(),
getGraphicsConfiguration(), getHeight(),
getHierarchyBoundsListeners(), getHierarchyListeners(),
getIgnoreRepaint(), getInputContext(),
getInputMethodListeners(), getInputMethodRequests(),
getKeyListeners(), getListeners(Class), getLocale(),
getLocation(Point), getLocation(Point),
getLocationOnScreen(), getMaximumSize(), getMinimumSize(),
getMouseListeners(), getMouseMotionListeners(),
getMouseWheelListeners(), getName(), getParent(),
getPeer(), getPreferredSize(),
getPropertyChangeListeners(String),
getPropertyChangeListeners(String), getSize(Dimension),
getSize(Dimension), getToolkit(), getTreeLock(),
getWidth(), getX(), getY(), gotFocus(Event, Object),
handleEvent(Event), hasFocus(), hide(), imageUpdate(Image,

```

Inherited Member Summary

```

int, int, int, int, int), inside(int, int), invalidate(),
isBackgroundSet(), isCursorSet(), isDisplayable(),
isDoubleBuffered(), isEnabled(),
isFocusCycleRoot(Container), isFocusOwner(),
isFocusable(), isFontSet(), isForegroundSet(),
isLightweight(), isOpaque(), isShowing(), isValid(),
isVisible(), keyDown(Event, int), keyUp(Event, int),
layout(), list(PrintWriter, int), list(PrintWriter, int),
list(PrintWriter, int), list(PrintWriter, int),
list(PrintWriter, int), locate(int, int), location(),
lostFocus(Event, Object), minimumSize(), mouseDown(Event,
int, int), mouseDrag(Event, int, int), mouseEnter(Event,
int, int), mouseExit(Event, int, int), mouseMove(Event,
int, int), mouseUp(Event, int, int), move(int, int),
nextFocus(), paintAll(Graphics), postEvent(Event),
preferredSize(), prepareImage(Image, ImageObserver),
prepareImage(Image, ImageObserver), print(Graphics),
printAll(Graphics), remove(MenuComponent),
removeComponentListener(ComponentListener),
removeFocusListener(FocusListener),
removeHierarchyBoundsListener(HierarchyBoundsListener),
removeHierarchyListener(HierarchyListener),
removeInputMethodListener(InputMethodListener),
removeKeyListener(KeyListener),
removeMouseListener(MouseListener),
removeMouseMotionListener(MouseMotionListener),
removeMouseWheelListener(MouseWheelListener),
removeNotify(), removePropertyChangeListener(String,
PropertyChangeListener),
removePropertyChangeListener(String,
PropertyChangeListener), repaint(long, int, int, int,
int), repaint(long, int, int, int, int), repaint(long,
int, int, int, int), repaint(long, int, int, int, int),
requestFocus(), requestFocusInWindow(), reshape(int, int,
int, int), resize(Dimension), resize(Dimension),
setBackground(Color), setBounds(Rectangle),
setBounds(Rectangle),
setComponentOrientation(ComponentOrientation),
setCursor(Cursor), setDropTarget(DropTarget),
setEnabled(boolean), setFocusTraversalKeys(int, Set),
setFocusTraversalKeysEnabled(boolean),
setFocusable(boolean), setFont(Font),
setForeground(Color), setIgnoreRepaint(boolean),
setLocale(Locale), setLocation(Point), setLocation(Point),
setName(String), setSize(Dimension), setSize(Dimension),
setVisible(boolean), show(boolean), show(boolean), size(),

```

Inherited Member Summary
<code>transferFocus()</code> , <code>transferFocusBackward()</code> , <code>transferFocusUpCycle()</code> , <code>validate()</code>
Methods inherited from class <code>Object</code>
<code>equals(Object)</code> , <code>getClass()</code> , <code>hashCode()</code> , <code>notify()</code> , <code>notifyAll()</code> , <code>wait(long, int)</code> , <code>wait(long, int)</code> , <code>wait(long, int)</code>

Fields

IDL_SOFTWARE_RENDERER

```
public static final int IDL_SOFTWARE_RENDERER
```

Internal use

OPENGL_RENDERER

```
public static final int OPENGL_RENDERER
```

Internal use

Constructors

JIDLCanvas(String, int, String)

```
public JIDLCanvas(java.lang.String sClass, int iOPSFlags,  
java.lang.String sProcessName)
```

Deprecated.

Replaced by constructor taking 2 parameters

Construct a JIDLCanvas

Parameters:

`sClass` - IDL Class name

`iOPSFlags` - Unused. The process name determines the OPS flags.

`sProcessName` - The process name. If null or "", in-process is used.

JIDLCanvas(String, String)

```
public JIDLCanvas(java.lang.String sClass,  
java.lang.String sProcessName)
```

Construct a JIDLCanvas Note that constructing the JIDLObject does NOT create the object on the IDL-side of the bridge. This is done using the createObject method.

Parameters:

sClass - IDL Class name

sProcessName - The process name. If null or “”, in-process is used.

Methods

abort()

```
public void abort()
```

Requests that the IDL process containing the underlying IDL object abort its current activity.

This is only a request and IDL may take a long time before it actually stops.

The client can only Abort the current IDL activity if that wrapper object is the current “owner” of the underlying IDL.

Specified By:

[abort](#) in interface [JIDLObjectI](#)

Throws:

[JIDLException](#) - If IDL encounters an error.

See Also:

[JIDLAbortedException](#)

addIDLComponentListener(JIDLComponentListener)

```
public void  
addIDLComponentListener(com.idl.javaidl.JIDLComponentListener  
listener)
```

Adds the specified JIDLComponentListener to a list of listeners that receive notification of Component events.

Note that registering/unregistering for events should happen in the `initListeners` method or AFTER the `createObject` method.

Parameters:

`listener` - the listener

See Also:

[JIDLComponentListener](#)

addIDLKeyListener(JIDLKeyListener)

```
public void addIDLKeyListener(com.idl.javaidl.JIDLKeyListener  
listener)
```

Adds the specified `JIDLKeyListener` to a list of listeners that receive notification of Key events.

Note that registering/unregistering for events should happen in the `initListeners` method or AFTER the `createObject` method.

Parameters:

`listener` - the listener

See Also:

[JIDLKeyListener](#)

addIDLMouseListener(JIDLMouseListener)

```
public void addIDLMouseListener(com.idl.javaidl.JIDLMouseListener  
listener)
```

Adds the specified `JIDLMouseListener` to a list of listeners that receive notification of Mouse events.

Note that registering/unregistering for events should happen in the `initListeners` method or AFTER the `createObject` method.

Parameters:

`listener` - the listener

See Also:

[JIDLMouseListener](#)

addIDLMouseMotionListener(JIDLMouseMotionListener)

```
public void  
    addIDLMouseMotionListener(com.idl.javaidl.JIDLMouseMotionListener  
        listener)
```

Adds the specified JIDLMouseMotionListener to a list of listeners that receive notification of MouseMotion events.

Note that registering/unregistering for events should happen in the initListeners method or AFTER the createObject method.

Parameters:

listener - the listener

See Also:

[JIDLMouseMotionListener](#)

addIDLMouseWheelListener(JIDLMouseWheelListener)

```
public void  
    addIDLMouseWheelListener(com.idl.javaidl.JIDLMouseWheelListener  
        listener)
```

Adds the specified JIDLMouseWheelListener to a list of listeners that receive notification of MouseWheel events.

Note that registering/unregistering for events should happen in the initListeners method or AFTER the createObject method.

Parameters:

listener - the listener

See Also:

[JIDLMouseWheelListener](#)

addIDLNotifyListener(JIDLNotifyListener)

```
public void  
    addIDLNotifyListener(com.idl.javaidl.JIDLNotifyListener listener)
```

Adds the specified IDL notify listener to receive IDL notification events on this object.

Note that registering/unregistering for events should happen in the initListeners method or AFTER the createObject method.

Specified By:

[addIDLNotifyListener](#) in interface [JIDLObjectI](#)

Parameters:

`listener` - the listener

addIDLOutputListener(JIDLOutputListener)

```
public void  
    addIDLOutputListener(com.idl.javaidl.JIDLOutputListener listener)
```

Adds the specified IDL output listener to receive IDL output events on this object.

Note that registering/unregistering for events should happen in the `initListeners` method or AFTER the `createObject` method.

Specified By:

[addIDLOutputListener](#) in interface [JIDLObjectI](#)

Parameters:

`listener` - the listener

callFunction(String, int)

```
public java.lang.Object callFunction(java.lang.String  
    sMethodName, int iPalFlag)
```

Call IDL function that accepts zero parameters.

Parameters:

`sMethodName` - the function name

`iPalFlag` - a flag determining whether a returned array is convolved or not. If the returned value is not an array, this value is zero. See class description for more information.

Returns:

an Object of type [JIDLNumber](#), [JIDLString](#), [JIDLObject](#) or [JIDLArray](#). The caller must know the type of the Object being returned and cast it to its proper type. May also return null.

Throws:

[JIDLException](#) - If IDL encounters an error.

See Also:

[callFunction\(String, int, Object\[\], int\[\], int\)](#)

callFunction(String, int, Object[], int[], int)

```
public java.lang.Object callFunction(java.lang.String  
    sMethodName, int argc, java.lang.Object[] argv, int[] argpal,  
    int iPalFlag)
```

Call IDL function.

The argpal parameter is an array of flags created by OR-ing constants from class JIDLConst. Each array element corresponds to the equivalent parameter in argv.

Specified By:

[callFunction](#) in interface [JIDLObjectI](#)

Parameters:

sMethodName - the procedure name

argc - the number of parameters

argv - array of Objects to be passed to IDL. This array should be of length argc and should contain objects of type JIDLNumber, JIDLObject, JIDLString or JIDLObject.

argpal - array of flags denoting whether each argv parameter passed to be bridge is 1) in-out vs constant; or 2) a convolved or non-convolved array This array should be of length argc.

iPalFlag - a flag determining whether a returned array if convolved or not. If the returned value is not an array, this value is zero.

Returns:

an Object of type JIDLNumber, JIDLString, JIDLObjectI or JIDLArray. The caller must know the type of the Object being returned and cast it to its proper type.

Throws:

[JIDLException](#) - If IDL encounters an error.

See Also:

[JIDLNumber](#), [JIDLObject](#), [JIDLString](#), [JIDLArray](#),
[JIDLConst.PARMFLAG_CONST](#), [JIDLConst.PARMFLAG_IN_OUT](#),
[JIDLConst.PARMFLAG_CONVMAJORITY](#),
[JIDLConst.PARMFLAG_NO_CONVMAJORITY](#)

callProcedure(String)

```
public void callProcedure(java.lang.String sMethodName)
```

Call IDL procedure that accepts zero parameters.

Parameters:

sMethodName - the procedure name

Throws:

[JIDLException](#) - If IDL encounters an error.

See Also:

[callProcedure\(String, int, Object\[\], int\[\]\)](#)

callProcedure(String, int, Object[], int[])

```
public void callProcedure(java.lang.String sMethodName,  
    int argc, java.lang.Object[] argv, int[] argpal)
```

Call IDL procedure.

The argpal parameter is an array of flags created by OR-ing constants from class JIDLConst. Each array element corresponds to the equivalent parameter in argv.

Specified By:

[callProcedure](#) in interface [JIDLObjectI](#)

Parameters:

sMethodName - the procedure name

argc - the number of parameters

argv - array of Objects to be passed to IDL. This array should be of length argc and should contain objects of type JIDLNumber, JIDLObject, JIDLString or JIDLObject.

argpal - array of flags denoting whether each argv parameter passed to be bridge is 1) in-out vs constant; or 2) a convolved or non-convolved array This array should be of length argc.

Throws:

[JIDLException](#) - If IDL encounters an error.

See Also:

[JIDLNumber](#), [JIDLObject](#), [JIDLString](#), [JIDLArray](#),
[JIDLConst.PARMFLAG_CONST](#), [JIDLConst.PARMFLAG_IN_OUT](#),
[JIDLConst.PARMFLAG_CONVMAJORITY](#),
[JIDLConst.PARMFLAG_NO_CONVMAJORITY](#)

componentHidden(ComponentEvent)

```
public void componentHidden(java.awt.event.ComponentEvent e)
```

Called when the component is hidden.

This method does nothing because IDL does not care about this event. This could be overridden by a child of JIDLCanvas if these events were of interest to the client application

Specified By:

`componentHidden` in interface `ComponentListener`

See Also:

```
java.awt.event.ComponentListener
```

componentMoved(ComponentEvent)

```
public void componentMoved(java.awt.event.ComponentEvent e)
```

Called when the component is moved.

This method does nothing because IDL does not care about this event. This could be overridden by a child of JIDLCanvas if these events were of interest to the client application

Specified By:

`componentMoved` in interface `ComponentListener`

See Also:

```
java.awt.event.ComponentListener
```

componentResized(ComponentEvent)

```
public final void  
componentResized(java.awt.event.ComponentEvent e)
```

Internal use.

Called when the JIDLCanvas is resized.

If interested in resize events, use `IDLcomponentResized`. This method should NOT be overridden by a child of `JIDLCanvas`.

Specified By:

`componentResized` in interface `ComponentListener`

See Also:

[JIDLComponentListener](#), [IDLcomponentResized\(JIDLObjectI, ComponentEvent\)](#)

componentShown(ComponentEvent)

```
public void componentShown(java.awt.event.ComponentEvent e)
```

Called when the component is shown.

This method does nothing because IDL does not care about this event. This could be overridden by a child of `JIDLCanvas` if these events were of interest to the client application

Specified By:

`componentShown` in interface `ComponentListener`

See Also:

```
java.awt.event.ComponentListener
```

createObject()

```
public void createObject()
```

Create the wrapped object by calling IDL's `::INIT` method.

Used for `::INIT` methods that take zero parameters.

Throws:

[JIDLException](#) - If IDL encounters an error.

See Also:

[createObject\(int, Object\[\], int\[\]\)](#)

createObject(int, Object[], int[])

```
public void createObject(int argc, java.lang.Object[] argv,  
int[] argpal)
```

Create the wrapped object by calling IDL's `::INIT` method.

Note that the GUI that this Canvas lives in must be exposed *before* the `createObject` method is called.

`createObject` does the following:

- call `IDL ::INIT`
- attach the IDL Window to this Canvas
- call `initListeners` to hook up default event handling
- repaint the canvas

The `argpal` parameter is an array of flags created by OR-ing constants from class `JIDLConst`. Each array element corresponds to the equivalent parameter in `argv`. See the class description for more information.

Specified By:

`createObject` in interface `JIDLObjectI`

Parameters:

`argc` - the number of parameters

`argv` - array of Objects to be passed to IDL. This array should be of length `argc` and should contain objects of type `JIDLNumber`, `JIDLObject`, `JIDLString` or `JIDLObject`.

`argpal` - array of flags denoting whether each `argv` parameter passed to be bridge is 1) in-out vs constant; or 2) a convolved or non-convolved array This array should be of length `argc`.

Throws:

`JIDLException` - If IDL encounters an error.

See Also:

`JIDLConst`, `initListeners()`

`createObject(int, Object[], int[], JIDLProcessInitializer)`

```
public void createObject(int argc, java.lang.Object[] argv,
    int[] argpal, com.idl.javaidl.JIDLProcessInitializer initializer)
```

Create the wrapped object by calling IDL's `::INIT` method.

Note that the GUI that this Canvas lives in must be exposed *before* the `createObject` method is called.

`createObject` does the following:

- Calls `::Init` method in the IDL object
- Calls the superclass `initListeners` method to initialize any event handlers. The `initListeners` method has default behavior, which is different for graphical and non-graphical objects. If the default behavior is not desired, a sub-class to modify the listener initialization may override the `initListeners` method.

Specified By:

`createObject` in interface `JIDLObjectI`

Parameters:

`argc` - the number of parameters to be passed to `INIT`

`argv` - array of Objects to be passed to IDL. This array should be of length `argc` and should contain objects of type `JIDLNumber`, `JIDLObject`, `JIDLString` or `JIDLArray`.

`argpal` - array of flags denoting whether each `argv` parameter that is of type array should be convolved or not. For parameters that are not arrays, the value within the array will always be 0.

`initializer` - a `JIDLProcessInitializer` object that specifies IDL process initialization parameters such as the licensing mode to be used. See “[IDL Licensing Modes](#)” on page 134 for details on the default licensing mechanism used when no `JIDLProcessInitializer` is specified.

Throws:

[`JIDLException`](#) - If IDL encounters an error.

`createObject(JIDLProcessInitializer)`

```
public void createObject (com.idl.javaidl.JIDLProcessInitializer  
initializer)
```

Create the wrapped object by calling IDL’s `::INIT` method.

Used for `::INIT` methods that take zero parameters.

The `initializer` parameter is used to supply IDL process initialization values.

Note that the GUI that this Canvas lives in must be exposed *before* the `createObject` method is called.

Parameters:

`initializer` - a `JIDLProcessInitializer` object that specifies IDL process initialization parameters such as the licensing mode to be used. See “[IDL Licensing](#)”

[Modes](#)” on page 134 for details on the default licensing mechanism used when no `JIDLProcessInitializer` is specified.

Throws:

[JIDLException](#) - If IDL encounters an error.

destroyObject()

```
public void destroyObject()
```

Destroys the underlying IDL object associated with the wrapper.

If the object being destroyed is the last object within an OPS process, the OPS process is also destroyed.

Note that this does not destroy the actual wrapper object. Because the wrapper object is a Java object, it follows all the Java reference counting/garbage collection schemes. Once all references to the wrapper object are released from Java code and once the Java Virtual Machine calls the garbage collector, the wrapper object may be deleted from memory.

Specified By:

[destroyObject](#) in interface [JIDLObjectI](#)

draw()

```
public void draw()
```

Internal use.

Call IDL to inform the Canvas has been exposed to cause a redraw.

This in turn calls all the `JIDLComponentListeners`. Should not be overridden.

executeString(String)

```
public void executeString(java.lang.String sCmd)
```

Execute the given command string in IDL.

Specified By:

[executeString](#) in interface [JIDLObjectI](#)

Parameters:

`sCmd` - the single-line command to execute in IDL.

Throws:

[JIDLException](#) - If IDL encounters an error.

getClassName()

```
public java.lang.String getClassName()
```

Deprecated.

Replaced by [getIDLObjectClassName\(\)](#)

Get the class name of the object.

Returns:

class name ("" if object not created yet)

getCookie()

```
public long getCookie()
```

Internal use.

Specified By:

[getCookie](#) in interface [JIDLObjectI](#)

getIDLObjectClassName()

```
public java.lang.String getIDLObjectClassName()
```

Retrieves the IDL object class name of the underlying IDL object.

Specified By:

[getIDLObjectClassName](#) in interface [JIDLObjectI](#)

Returns:

the IDL object class name

getIDLObjectVariableName()

```
public java.lang.String getIDLObjectVariableName()
```

When the underlying IDL object was created in the IDL process, it was assigned a variable name. This method retrieves that name.

Specified By:

[getIDLObjectVariableName](#) in interface [JIDLObjectI](#)

Returns:

the variable name

getIDLVariable(String)

```
public java.lang.Object getIDLVariable(java.lang.String sVar)
```

Given a variable name, return the IDL variable.

Note that in the case of arrays, the array will ALWAYS be convolved when passed between Java and IDL.

Specified By:

[getIDLVariable](#) in interface [JIDLObjectI](#)

Parameters:

sVar - The IDL variable name

Returns:

an Object of type JIDLNumber, JIDLString, JIDLObject or JIDLArray. The caller must know the type of the Object being returned and cast it to its proper type. May also return null.

Throws:

[JIDLException](#) - If IDL encounters an error.

getObjVariableName()

```
public java.lang.String getObjVariableName()
```

Deprecated.

Replaced by [getIDLObjectVariableName\(\)](#)

Get the IDL Variable name of the given object

Returns:

a String representing the IDL Variable name

getProcessName()

```
public java.lang.String getProcessName()
```

Returns the name of the process that contains the underlying IDL object. For an in-process object, returns an empty string.

Specified By:

[getProcessName](#) in interface [JIDLObjectI](#)

Returns:

process name. Empty string if the process is in-process.

getProperty(String, int)

```
public java.lang.Object getProperty(java.lang.String
    sProperty, int iPalFlag)
```

Call IDL `getProperty` method to get named property.

Specified By:

[getProperty](#) in interface [JIDLObjectI](#)

Parameters:

`sProperty` - the property name

`iPalFlag` - a flag determining whether a returned array will be convolved or not. If the returned value is not is ignored.

Returns:

an Object of type [JIDLNumber](#), [JIDLString](#), [JIDLObject](#) or [JIDLArray](#). The caller must know the type of the Object being returned and cast it to its proper type. May also return null.

Throws:

[JIDLException](#) - If IDL encounters an error.

See Also:

[JIDLNumber](#), [JIDLObjectI](#), [JIDLString](#), [JIDLArray](#),
[JIDLConst.PARMFLAG_CONVMAJORITY](#),
[JIDLConst.PARMFLAG_NO_CONVMAJORITY](#)

IDLcomponentExposed(JIDLObjectI)

```
public void IDLcomponentExposed(com.idl.javaidl.JIDLObjectI obj)
```

Called when the [JIDLCanvas](#) is exposed.

The default behavior of this method is to lock the Canvas, pass the event on to IDL to handle (i.e. redraw), and then unlock the Canvas.

The behavior may be changed by overriding this method in a sub-class. For example, the sub-class may want to do something special before or after the redraw happens. The method would be implemented as follows:

```
public class mySubClass extends JIDLCanvas {
    public void IDLcomponentExposed() {
        // do something here before IDL is called
        super.IDLcomponentExposed();
        // do something if desired afterwards
    }
}
```

Specified By:

[IDLcomponentExposed](#) in interface [JIDLComponentListener](#)

See Also:

[JIDLComponentListener](#), [initListeners\(\)](#)

IDLcomponentResized(JIDLObjectI, ComponentEvent)

```
public void IDLcomponentResized(com.idl.javaidl.JIDLObjectI obj,
    java.awt.event.ComponentEvent e)
```

Called when the JIDLCanvas is resized.

The default behavior of this method is to send the resize event to IDL to handle.

Specified By:

[IDLcomponentResized](#) in interface [JIDLComponentListener](#)

See Also:

[JIDLComponentListener](#), [initListeners\(\)](#)

IDLkeyPressed(JIDLObjectI, KeyEvent, int, int)

```
public void IDLkeyPressed(com.idl.javaidl.JIDLObjectI obj,
    java.awt.event.KeyEvent e, int x, int y)
```

Called when the JIDLCanvas has focus and a key is pressed.

The default behavior of this method is pass the event to IDL which, if registered for the event will call `::OnKeyboard`.

The behavior may be changed by overriding this method in a sub-class. For example, the sub-class may want to ignore the event by providing an empty implementation of

the method. Or the sub-class may do something special before or after the event happens.

See `IDLcomponentExposed` for an example of how this would be done.

Specified By:

[IDLkeyPressed](#) in interface [JIDLKeyListener](#)

See Also:

[JIDLKeyListener](#), [IDLcomponentExposed\(JIDLObjectI\)](#), [initListeners\(\)](#)

IDLkeyReleased(JIDLObjectI, KeyEvent, int, int)

```
public void IDLkeyReleased(com.idl.javaidl.JIDLObjectI obj,  
    java.awt.event.KeyEvent e, int x, int y)
```

Called when the `JIDLCanvas` has focus and a key is released.

The default behavior of this method is pass the event to IDL which, if registered for the event will call `::OnKeyboard`. The behavior may be changed by overriding this method in a sub-class. For example, the sub-class may want to ignore the event by providing an empty implementation of the method. Or the sub-class may do something special before or after the event happens. See `IDLcomponentExposed` for an example of how this would be done.

Specified By:

[IDLkeyReleased](#) in interface [JIDLKeyListener](#)

See Also:

[JIDLKeyListener](#), [IDLcomponentExposed\(JIDLObjectI\)](#), [initListeners\(\)](#)

IDLmouseDragged(JIDLObjectI, MouseEvent)

```
public void IDLmouseDragged(com.idl.javaidl.JIDLObjectI obj,  
    java.awt.event.MouseEvent e)
```

Called when the mouse is dragged in a `JIDLCanvas`.

The default behavior of this method is pass the event to IDL which, if registered for the event, will call `::OnMouseMotion`.

The behavior may be changed by overriding this method in a sub-class. For example, the sub-class may want to ignore the event by providing an empty implementation of the method. Often our IDL `IDLitWindow` is only interested in one type of motion event and not another. Or the sub-class may do something special before or after the event happens.

See `IDLcomponentExposed` for an example of how this would be done.

Specified By:

[IDLmouseDragged](#) in interface [JIDLMouseMotionListener](#)

See Also:

[JIDLMouseMotionListener](#), [IDLcomponentExposed\(JIDLObjectI\)](#), [initListeners\(\)](#)

IDLmouseEntered(JIDLObjectI, MouseEvent)

```
public void IDLmouseEntered(com.idl.javaidl.JIDLObjectI obj,  
    java.awt.event.MouseEvent e)
```

Called when the mouse enters a `JIDLCanvas`.

The default behavior of this method is to ignore the event.

The behavior may be changed by overriding this method in a sub-class.

Specified By:

[IDLmouseEntered](#) in interface [JIDLMouseListener](#)

See Also:

[JIDLMouseListener](#), [initListeners\(\)](#)

IDLmouseExited(JIDLObjectI, MouseEvent)

```
public void IDLmouseExited(com.idl.javaidl.JIDLObjectI obj,  
    java.awt.event.MouseEvent e)
```

Called when the mouse exits a `JIDLCanvas`.

The default behavior of this method is to ignore the event.

The behavior may be changed by overriding this method in a sub-class.

Specified By:

[IDLmouseExited](#) in interface [JIDLMouseListener](#)

See Also:

[JIDLMouseListener](#), [initListeners\(\)](#)

IDLmouseMoved(JIDLObjectI, MouseEvent)

```
public void IDLmouseMoved(com.idl.javaidl.JIDLObjectI obj,  
    java.awt.event.MouseEvent e)
```

Called when the mouse is moved in a JIDLCanvas.

The default behavior of this method is pass the event to IDL which, if registered for the event, will call `::OnMouseMove`.

The behavior may be changed by overriding this method in a sub-class. For example, the sub-class may want to ignore the event by providing an empty implementation of the method. Often our IDL IDLitWindow is only interested in one type of motion event and not another. Or the sub-class may do something special before or after the event happens.

See `IDLcomponentExposed` for an example of how this would be done.

Specified By:

[IDLmouseMoved](#) in interface [JIDLMouseMotionListener](#)

See Also:

[JIDLMouseMotionListener](#), [IDLcomponentExposed\(JIDLObjectI\)](#), [initListeners\(\)](#)

IDLmousePressed(JIDLObjectI, MouseEvent)

```
public void IDLmousePressed(com.idl.javaidl.JIDLObjectI  
    obj, java.awt.event.MouseEvent e)
```

Called when the mouse is pressed in a JIDLCanvas.

The default behavior of this method is pass the event to IDL which, if registered for the event, will call `::OnMouseDown`.

The behavior may be changed by overriding this method in a sub-class. For example, the sub-class may want to ignore the event by providing an empty implementation of the method. Or the sub-class may do something special before or after the event happens.

See `IDLcomponentExposed` for an example of how this would be done.

Specified By:

[IDLmousePressed](#) in interface [JIDLMouseListener](#)

See Also:

[JIDLMouseListener](#), [IDLcomponentExposed\(JIDLObjectI\)](#), [initListeners\(\)](#)

IDLmouseReleased(JIDLObjectI, MouseEvent)

```
public void IDLmouseReleased(com.idl.javaidl.JIDLObjectI obj,  
    java.awt.event.MouseEvent e)
```

Called when the mouse is released in a `JIDLCanvas`.

The default behavior of this method is pass the event to IDL which, if registered for the event, will call `::OnMouseUp`.

The behavior may be changed by overriding this method in a sub-class. For example, the sub-class may want to ignore the event by providing an empty implementation of the method. Or the sub-class may do something special before or after the event happens.

See `IDLcomponentExposed` for an example of how this would be done.

Specified By:

[IDLmouseReleased](#) in interface [JIDLMouseListener](#)

See Also:

[JIDLMouseListener](#), [IDLcomponentExposed\(JIDLObjectI\)](#), [initListeners\(\)](#)

initListeners()

```
public void initListeners()
```

Initialize listeners.

This method is always called by `createObject`. The `JIDLCanvas` listens to the following events:

- `JIDLComponentListener`
- `JIDLKeyListener`
- `JIDLMouseListener`
- `JIDLMouseMotionListener`

The method may be overridden by sub-classes to initialize a different set of listeners (or none at all). For example if a sub-class of `JIDLCanvas` only wished to listen to key and component events, it would override `initListeners` as follows:

```
public void initListeners() {  
    addIDLComponentListener(this);  
    addIDLKeyListener(this);  
}
```

As another example, if a sub-class of `JIDLCanvas` wished to listen to key events, component events, and notify events, it would need to implement `JIDLNotifyListener` and register to listen for these events in `initListeners`, as follows:

```

    public class newCanvas extends JIDLCanvas implements JIDLNotifyLi
stener
    {
        public void initListeners() {
            addIDLComponentListener(this);
            addIDLKeyListener(this);
            addIDLNotifyListener(this);
        }
        void OnIDLNotify(JIDLObjectI obj, String s1, String s2) {
            // do something with the notify
        }
    }

```

Specified By:

[initListeners](#) in interface [JIDLObjectI](#)

See Also:

[JIDLComponentListener](#), [JIDLKeyListener](#), [JIDLMouseListener](#),
[JIDLMouseMotionListener](#), [JIDLNotifyListener](#), [JIDLOutputListener](#)

isFocusTraversable()

```
public boolean isFocusTraversable()
```

Internal use.

Overrides:

[isFocusTraversable](#) in class [Component](#)

isObjCreated()

```
public boolean isObjCreated()
```

Deprecated.

Replaced by [isObjectCreated\(\)](#)

Determine if object has been created successfully.

Returns:

true if object created successfully, or false if object not created or creation was unsuccessful.

isObjectCreated()

```
public boolean isObjectCreated()
```

Determine if object has been created successfully.

Specified By:

[isObjectCreated](#) in interface [JIDLObjectI](#)

Returns:

true if object created successfully, or false if object not created, destroyed, or creation was unsuccessful.

See Also:

[createObject\(\)](#)

isObjectDisplayable()

```
public boolean isObjectDisplayable()
```

Specified By:

[isObjectDisplayable](#) in interface [JIDLObjectI](#)

keyPressed(KeyEvent)

```
public final void keyPressed(java.awt.event.KeyEvent e)
```

Internal use.

Called when a key is pressed when the JIDLCanvas has focus.

If interested in this event, use IDLkeyPressed. This method should NOT be overridden by a child of JIDLCanvas.

Specified By:

[keyPressed](#) in interface [KeyListener](#)

See Also:

[JIDLKeyListener](#), [IDLkeyPressed\(JIDLObjectI, KeyEvent, int, int\)](#)

keyReleased(KeyEvent)

```
public final void keyReleased(java.awt.event.KeyEvent e)
```

Internal use.

Called when a key is released when the JIDLCanvas has focus.

If interested in this event, use IDLkeyReleased. This method should NOT be overridden by a child of JIDLCanvas.

Specified By:

keyReleased in interface `KeyListener`

See Also:

[JIDLKeyListener](#), [IDLkeyReleased\(JIDLObjectI, KeyEvent, int, int\)](#)

keyTyped(KeyEvent)

```
public void keyTyped(java.awt.event.KeyEvent e)
```

Internal use.

Called when a key is typed.

This method does nothing because IDL does not care about this event, using `keyPressed` to trigger its mouse events. This method should NOT be overridden by a child of `JIDLCanvas`.

Specified By:

keyTyped in interface `KeyListener`

mapIDLCursorToJavaCursor(String)

```
public int mapIDLCursorToJavaCursor(java.lang.String idlCursor)
```

Maps the IDL cursor to a suitable Java cursor. This is called internally by `setCursor` when the IDL drawable changes the cursor.

May be overridden to change the mapping. The default mapping is as follows:

- “ARROW” → `Cursor.DEFAULT_CURSOR`;
- “UP_ARROW” → `Cursor.DEFAULT_CURSOR`;
- “IBEAM” → `Cursor.TEXT_CURSOR`;
- “ICON” → `Cursor.TEXT_CURSOR`;
- “CROSSHAIR” → `Cursor.CROSSHAIR_CURSOR`;
- “ORIGINAL” → `Cursor.CROSSHAIR_CURSOR`;
- “HOURLASS” → `Cursor.WAIT_CURSOR`;
- “MOVE” → `Cursor.MOVE_CURSOR`;
- “SIZE_NW” → `Cursor.NW_RESIZE_CURSOR`;
- “SIZE_SE” → `Cursor.SE_RESIZE_CURSOR`;
- “SIZE_NE” → `Cursor.NE_RESIZE_CURSOR`;

- “SIZE_SW” → `Cursor.SW_RESIZE_CURSOR`;
- “SIZE_EW” → `Cursor.E_RESIZE_CURSOR`;
- “SIZE_NS” → `Cursor.N_RESIZE_CURSOR`;
- otherwise → `Cursor.DEFAULT_CURSOR`;

Specified By:

`mapIDLCursorToJavaCursor` in interface `JIDLCursorSupport`

Parameters:

`idlCursor` - a `String` representing the IDL cursor

Returns:

the `Cursor` constant representing the Java `Cursor` style

See Also:

[setCursor\(String\)](#)

mouseClicked(MouseEvent)

```
public void mouseClicked(java.awt.event.MouseEvent e)
```

Internal use.

Called when the mouse is clicked.

This method does nothing because IDL does not care about this event, using `mousePressed` to trigger its mouse events. This method should NOT be overridden by a child of `JIDLCanvas`.

Specified By:

`mouseClicked` in interface `MouseListener`

mouseDragged(MouseEvent)

```
public final void mouseDragged(java.awt.event.MouseEvent e)
```

Internal use.

Called when the mouse is dragged in the `JIDLCanvas`.

If interested in this event, use `IDLmouseDragged`. This method should NOT be overridden by a child of `JIDLCanvas`.

Specified By:

mouseDragged in interface `MouseMotionListener`

See Also:

[JIDLMouseMotionListener](#), [IDLmouseDragged\(JIDLObjectI, MouseEvent\)](#)

mouseEntered(MouseEvent)

```
public final void mouseEntered(java.awt.event.MouseEvent e)
```

Internal use.

Called when the mouse enters the `JIDLCanvas`.

If interested in this event, use `IDLmouseEntered`. This method should NOT be overridden by a child of `JIDLCanvas`.

Specified By:

mouseEntered in interface `MouseListener`

See Also:

[JIDLMouseListener](#), [IDLmouseEntered\(JIDLObjectI, MouseEvent\)](#)

mouseExited(MouseEvent)

```
public final void mouseExited(java.awt.event.MouseEvent e)
```

Internal use.

Called when the mouse exits the `JIDLCanvas`.

If interested in this event, use `IDLmouseExited`. This method should NOT be overridden by a child of `JIDLCanvas`.

Specified By:

mouseExited in interface `MouseListener`

See Also:

[JIDLMouseListener](#), [IDLmouseExited\(JIDLObjectI, MouseEvent\)](#)

mouseMoved(MouseEvent)

```
public final void mouseMoved(java.awt.event.MouseEvent e)
```

Internal use.

Called when the mouse moves in the `JIDLCanvas`.

If interested in this event, use `IDLmouseMoved`. This method should NOT be overridden by a child of `JIDLCanvas`.

Specified By:

`mouseMoved` in interface `MouseMotionListener`

See Also:

[JIDLMouseMotionListener](#), [IDLmouseMoved\(JIDLObjectI, MouseEvent\)](#)

mousePressed(MouseEvent)

```
public final void mousePressed(java.awt.event.MouseEvent e)
```

Internal use.

Called when the mouse is pressed.

If interested in this event, use `IDLmousePressed`. This method should NOT be overridden by a child of `JIDLCanvas`.

Specified By:

`mousePressed` in interface `MouseListener`

See Also:

[JIDLMouseListener](#), [IDLmousePressed\(JIDLObjectI, MouseEvent\)](#)

mouseReleased(MouseEvent)

```
public final void mouseReleased(java.awt.event.MouseEvent e)
```

Internal use.

Called when the mouse is released.

If interested in this event, use `IDLmouseReleased`. This method should NOT be overridden by a child of `JIDLCanvas`.

Specified By:

`mouseReleased` in interface `MouseListener`

See Also:

[JIDLMouseListener](#), [IDLmouseReleased\(JIDLObjectI, MouseEvent\)](#)

paint(Graphics)

```
public void paint(java.awt.Graphics g)
```

Internal use. Paint the Canvas. (Do not override this method)

Overrides:

paint in class Canvas

removeIDLComponentListener(JIDLComponentListener)

```
public void  
removeIDLComponentListener(com.idl.javaidl.JIDLComponentListener  
listener)
```

Remove the specified JIDLComponentListener from a list of listeners that receive notification of Component events.

Note that registering/unregistering for events should happen in the initListeners method or AFTER the createObject method.

Parameters:

listener - the listener

See Also:

[JIDLComponentListener](#)

removeIDLKeyListener(JIDLKeyListener)

```
public void removeIDLKeyListener(com.idl.javaidl.JIDLKeyListener  
listener)
```

Removes the specified JIDLKeyListener from a list of listeners that receive notification of Key events.

Note that registering/unregistering for events should happen in the initListeners method or AFTER the createObject method.

Parameters:

listener - the listener

See Also:

[JIDLKeyListener](#)

removeIDLMouseListener(JIDLMouseListener)

```
public void  
removeIDLMouseListener(com.idl.javaidl.JIDLMouseListener listener)
```

Removes the specified JIDLMouseListener from a list of listeners that receive notification of Mouse events.

Note that registering/unregistering for events should happen in the `initListeners` method or AFTER the `createObject` method.

Parameters:

`listener` - the listener

See Also:

[JIDLMouseListener](#)

removeIDLMouseMotionListener(JIDLMouseMotionListener)

```
public void  
    removeIDLMouseMotionListener(com.idl.javaidl.JIDLMouseMotionListen  
er listener)
```

Removes the specified JIDLMouseMotionListener from a list of listeners that receive notification of MouseMotion events.

Note that registering/unregistering for events should happen in the `initListeners` method or AFTER the `createObject` method.

Parameters:

`listener` - the listener

See Also:

[JIDLMouseMotionListener](#)

removeIDLMouseWheelListener(JIDLMouseWheelListener)

```
public void  
    removeIDLMouseWheelListener(com.idl.javaidl.JIDLMouseWheelListen  
er listener)
```

Removes the specified JIDLMouseWheelListener to a list of listeners that receive notification of MouseWheel events.

Note that registering/unregistering for events should happen in the `initListeners` method or AFTER the `createObject` method.

Parameters:

`listener` - the listener

See Also:[JIDLMouseWheelListener](#)**removeIDLNotifyListener(JIDLNotifyListener)**

```
public void  
removeIDLNotifyListener(com.idl.javaidl.JIDLNotifyListener  
listener)
```

Removes the specified IDL notify listener so it no longer receives IDL notifications.

Note that registering/unregistering for events should happen in the `initListeners` method or AFTER the `createObject` method.

Specified By:

[removeIDLNotifyListener](#) in interface [JIDLObjectI](#)

Parameters:

`listener` - the listener

removeIDLOutputListener(JIDLOutputListener)

```
public void  
removeIDLOutputListener(com.idl.javaidl.JIDLOutputListener  
listener)
```

Removes the specified IDL output listener on this object.

Note that registering/unregistering for events should happen in the `initListeners` method or AFTER the `createObject` method.

Specified By:

[removeIDLOutputListener](#) in interface [JIDLObjectI](#)

Parameters:

`listener` - the listener

setCursor(String)

```
public void setCursor(java.lang.String idlCursor)
```

Set the JIDLCanvas cursor. Called automatically when the IDL cursor changes. This in turn calls `mapIDLCursorToJavaCursor` to map the IDL cursor name to a suitable Java cursor type.

Specified By:

setCursor in interface `JIDLCursorSupport`

Parameters:

idlCursor - A String representing the IDL cursor name.

See Also:

[mapIDLCursorToJavaCursor\(String\)](#)

setIDLVariable(String, Object)

```
public void setIDLVariable(java.lang.String sVar,  
    java.lang.Object obj)
```

Set/Create an IDL variable of the given name and value.

Note that in the case of arrays, the array will ALWAYS be convolved when passed between Java and IDL.

Specified By:

[setIDLVariable](#) in interface `JIDLObjectI`

Parameters:

sVar - the IDL variable name

obj - object to be passed to IDL. Should be an object of type `JIDLNumber`, `JIDLObject`, `JIDLString` or `JIDLArray`.

Throws:

[JIDLException](#) - If IDL encounters an error.

setProcessName(String)

```
public void setProcessName(java.lang.String process)
```

Set the process name that the object will be created in.

The process name may only be set before `createObject` is called. If called after the object has been created, this method call does nothing.

Specified By:

[setProcessName](#) in interface `JIDLObjectI`

Parameters:

`process` - Process name. Empty String means create in same process (in-process).

setProperty(String, Object, int)

```
public void setProperty(java.lang.String sProperty,
    java.lang.Object obj, int iPalFlag)
```

Call IDL `setProperty` method to set named property.

The `iPalFlag` parameter is a set of flags that are or-ed together. Currently this parameter is only used to specify whether a `JIDLArray` being passed in to IDL is convolved or not. For arrays `argpal` should be set to either `JIDLConst.PARMFLAG_CONVMAJORITY` or `JIDLConst.PARMFLAG_NO_CONVMAJORITY`.

Specified By:

`setProperty` in interface [JIDLObjectI](#)

Parameters:

`sProperty` - the property name

`obj` - object to be passed to IDL. Should be an object of type `JIDLNumber`, `JIDLObject`, `JIDLString` or `JIDLObject`.

`iPalFlag` - flag denoting whether the passed in parameter is convolved or not.

Note: `setProperty` does not allow `obj` to be modified by IDL

Throws:

[JIDLException](#) - If IDL encounters an error.

See Also:

[JIDLNumber](#), [JIDLObject](#), [JIDLString](#), [JIDLArray](#),
[JIDLConst.PARMFLAG_CONVMAJORITY](#),
[JIDLConst.PARMFLAG_NO_CONVMAJORITY](#)

toString()

```
public java.lang.String toString()
```

Returns a string representation of the object.

Overrides:

`toString` in class `Component`

update(Graphics)

```
public void update(java.awt.Graphics g)
```

Internal use. Update the Canvas. (Do not override this method)

Overrides:

update in class Canvas

JIDLChar

Declaration

```
public class JIDLChar implements JIDLNumber,  
    java.io.Serializable  
  
    java.lang.Object  
    |  
    +--com.idl.javaidl.JIDLChar
```

All Implemented Interfaces:

[JIDLNumber](#), [java.io.Serializable](#)

Description

The JIDLChar class wraps a char as a mutable object usable by the Java-IDL Export bridge.

Member Summary	
Constructors	
	JIDLChar(char value) Construct a wrapper object.
	JIDLChar(JIDLNumber value) Construct a wrapper object.
Methods	
boolean	booleanValue() Return the value of the wrapped primitive.
byte	byteValue() Return the value of the wrapped primitive
char	charValue() Return the value of the wrapped primitive
double	doubleValue() Return the value of the wrapped primitive
float	floatValue() Return the value of the wrapped primitive

Member Summary	
int	<code>intValue()</code> Return the value of the wrapped primitive
long	<code>longValue()</code> Return the value of the wrapped primitive
void	<code>setValue(char value)</code> Change the value of the wrapper object
void	<code>setValue(JIDLNumber value)</code> Change the value of the wrapper object
short	<code>shortValue()</code> Return the value of the wrapped primitive
java.lang.String	<code>toString()</code>

Inherited Member Summary
Methods inherited from class <code>Object</code>
<code>equals(Object)</code> , <code>getClass()</code> , <code>hashCode()</code> , <code>notify()</code> , <code>notifyAll()</code> , <code>wait(long, int)</code> , <code>wait(long, int)</code> , <code>wait(long, int)</code>

Constructors

JIDLChar(char)

```
public JIDLChar(char value)
```

Construct a wrapper object.

Parameters:

value - value to wrap for use in the export bridge

JIDLChar(JIDLNumber)

```
public JIDLChar(com.idl.javaidl.JIDLNumber value)
```

Construct a wrapper object.

Parameters:

value - JIDLNumber to wrap for use in the export bridge

Methods

booleanValue()

```
public boolean booleanValue()
```

Return the value of the wrapped primitive.

Specified By:

[booleanValue](#) in interface [JIDLNumber](#)

Returns:

true if non-zero, false otherwise

byteValue()

```
public byte byteValue()
```

Return the value of the wrapped primitive

Specified By:

[byteValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

charValue()

```
public char charValue()
```

Return the value of the wrapped primitive

Specified By:

[charValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

doubleValue()

```
public double doubleValue()
```

Return the value of the wrapped primitive

Specified By:

[doubleValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

floatValue()

```
public float floatValue()
```

Return the value of the wrapped primitive

Specified By:

[floatValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

intValue()

```
public int intValue()
```

Return the value of the wrapped primitive

Specified By:

[intValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

longValue()

```
public long longValue()
```

Return the value of the wrapped primitive

Specified By:

[longValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

setValue(char)

```
public void setValue(char value)
```

Change the value of the wrapper object

Parameters:

value - primitive value to wrap for use in the export bridge

setValue(JIDLNumber)

```
public void setValue(com.idl.javaidl.JIDLNumber value)
```

Change the value of the wrapper object

Specified By:

[setValue](#) in interface [JIDLNumber](#)

Parameters:

value - JIDLNumber to wrap for use in the export bridge

shortValue()

```
public short shortValue()
```

Return the value of the wrapped primitive

Specified By:

[shortValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

toString()

```
public java.lang.String toString()
```

Overrides:

[toString](#) in class [Object](#)

JIDLComponentListener

Declaration

```
public interface JIDLComponentListener
```

All Known Implementing Classes:

[JIDLCanvas](#)

Description

The listener interface for receiving component events (expose, resize) on a JIDLCanvas.

The class that is interested in handling these events implements this interface (and all the methods it contains). The listener object created from that class is then registered with the JIDLCanvas using the `addIDLComponentListener` method. The listener is unregistered with the `removeIDLComponentListener`.

Component events are provided for notification purposes; the JIDLCanvas automatically handles component redraws and resizes internally whether a program registers an additional JIDLComponentListener or not. The JIDLCanvas is itself a JIDLComponentListener and provides default behavior for expose and resize. For an expose event, the default behavior is for the JIDLCanvas to call the IDL program's `OnExpose` method. For a resize, the default is to call the IDL program's `OnResize` method.

Note that clients should not register to listen to JIDLCanvas ComponentEvents using a ComponentListener, preferring the JIDLComponentListener instead.

See Also:

```
java.awt.event.ComponentEvent,  
java.awt.event.ComponentListener
```

Member Summary	
Methods	
void	<code>IDLcomponentExposed(JIDLObjectI obj)</code> The IDL component (JIDLCanvas) has been exposed.
void	<code>IDLcomponentResized(JIDLObjectI obj, java.awt.event.ComponentEvent event)</code> The IDL component (JIDLCanvas) has been resized.

Methods

IDLcomponentExposed(JIDLObjectI)

```
public void IDLcomponentExposed(com.idl.javaidl.JIDLObjectI obj)
```

The IDL component (JIDLCanvas) has been exposed.

The default behavior of JIDLCanvas's default IDLcomponentExposed is to the IDL program's OnExpose method.

Parameters:

obj - The object that has been resized

IDLcomponentResized(JIDLObjectI, ComponentEvent)

```
public void IDLcomponentResized(com.idl.javaidl.JIDLObjectI obj, java.awt.event.ComponentEvent event)
```

The IDL component (JIDLCanvas) has been resized.

The default behavior of JIDLCanvas's default IDLcomponentResized is to call the IDL program's OnResize method.

Parameters:

obj - The object that has been resized

event - The Component event

JIDLConst

Declaration

```
public class JIDLConst  
  
java.lang.Object  
|  
+--com.idl.javaidl.JIDLConst
```

Description

Contains constants used by the Java-IDL wrapper classes.

Member Summary	
Fields	
static int	CONTROL_INPROC Control flag for determining object is to be created in-process
static int	CONTROL_OUTPROC Control flag for determining object is to be created out-of-process
static int	IDL_ABORT_NOT_OWNER Error code when an abort request is made, but the calling object does not have permission to request the abort.
static java.lang.String	IDL_ABORT_NOT_OWNER_MESSAGE Internal use.
static int	IDL_ABORTED Error code returned when IDL processing has aborted due to an abort request.
static int	IDL_BUSY Error code returned if IDL is called while processing another request.

Member Summary	
static int	IDL_NOTHING_TO_ABORT Error code when an abort request is made, but there is nothing to abort.
static java.lang.String	IDL_NOTHING_TO_ABORT_MESSAGE Internal use.
static int	PARMFLAG_CONST Parameter associated with this flag and passed to IDL is const (in-only).
static int	PARMFLAG_CONVMAJORITY Parameter associated with this flag and passed to IDL is an array whose majority will be convolved.
static int	PARMFLAG_IN_OUT Parameter associated with this flag and passed to IDL is in-out (mutable).
static int	PARMFLAG_NO_CONVMAJORITY Parameter associated with this flag and passed to IDL is an array whose majority will NOT be convolved.

Inherited Member Summary
Methods inherited from class <code>Object</code>
<code>equals(Object), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(long, int), wait(long, int), wait(long, int)</code>

Fields

CONTROL_INPROC

```
public static final int CONTROL_INPROC
```

Control flag for determining object is to be created in-process

CONTROL_OUTPROC

```
public static final int CONTROL_OUTPROC
```

Control flag for determining object is to be created out-of-process

IDL_ABORT_NOT_OWNER

```
public static final int IDL_ABORT_NOT_OWNER
```

Error code when an abort request is made, but the calling object does not have permission to request the abort.

Note that when this error occurs, a `JIDLException` is thrown to the calling client with this value as its error code.

See Also:

[JIDLException](#), [JIDLObjectI.abort\(\)](#)

IDL_ABORT_NOT_OWNER_MESSAGE

```
public static final java.lang.String IDL_ABORT_NOT_OWNER_MESSAGE
```

Internal use. Error message when an abort request is made, but the calling object does not have permission to request the abort.

IDL_ABORTED

```
public static final int IDL_ABORTED
```

Error code returned when IDL processing has aborted due to an abort request.

Note that when this error occurs, a `JIDLAbortedException` is thrown to the calling client with this value as its error code.

See Also:

[JIDLAbortedException](#), [JIDLException](#), [JIDLObjectI.abort\(\)](#)

IDL_BUSY

```
public static final int IDL_BUSY
```

Error code returned if IDL is called while processing another request.

Note that when this error occurs, a `JIDLBusyException` is thrown to the calling client with this value as its error code.

See Also:

[JIDLBusyException](#), [JIDLException](#), [JIDLObjectI.abort\(\)](#)

IDL_NOTHING_TO_ABORT

```
public static final int IDL_NOTHING_TO_ABORT
```

Error code when an abort request is made, but there is nothing to abort.

Note that when this error occurs, a [JIDLException](#) is thrown to the calling client with this value as its error code.

See Also:

[JIDLException](#), [JIDLObjectI.abort\(\)](#)

IDL_NOTHING_TO_ABORT_MESSAGE

```
public static final java.lang.String  
IDL_NOTHING_TO_ABORT_MESSAGE
```

Internal use. Error message when an abort request is made, but there is nothing to abort.

PARMFLAG_CONST

```
public static final int PARMFLAG_CONST
```

Parameter associated with this flag and passed to IDL is const (in-only). It is expected IDL will not change this parameter. Any changes that happened in IDL will be ignored.

See Also:

[PARMFLAG_IN_OUT](#)

PARMFLAG_CONVMAJORITY

```
public static final int PARMFLAG_CONVMAJORITY
```

Parameter associated with this flag and passed to IDL is an array whose majority will be convolved.

Note that if set, the array will be convolved when passed from Java to IDL, and convolved *again* in the in-out case, when passed back to Java.

See Also:

[PARMFLAG_NO_CONVMAJORITY](#)

PARMFLAG_IN_OUT

```
public static final int PARMFLAG_IN_OUT
```

Parameter associated with this flag and passed to IDL is in-out (mutable). It is expected IDL may change this parameter and on return from IDL the data will be copied back to the Java object.

See Also:

[PARMFLAG_CONST](#)

PARMFLAG_NO_CONVMAJORITY

```
public static final int PARMFLAG_NO_CONVMAJORITY
```

Parameter associated with this flag and passed to IDL is an array whose majority will NOT be convolved.

Note that for arrays of dimensions 2 through 8, this may be quicker than `PARMFLAG_CONVMAJORITY` because the array doesn't need to be re-ordered when passed between Java and IDL memory space.

See Also:

[PARMFLAG_CONVMAJORITY](#)

JIDLDouble

Declaration

```
public class JIDLDouble implements JIDLNumber,  
    java.io.Serializable  
  
    java.lang.Object  
    |  
    +---com.idl.javaidl.JIDLDouble
```

All Implemented Interfaces:

[JIDLNumber](#), [java.io.Serializable](#)

Description

The JIDLDouble class wraps a double as a mutable object usable by the Java-IDL Export bridge.

Member Summary	
Constructors	
	JIDLDouble(double value) Construct a wrapper object.
	JIDLDouble(JIDLNumber value) Construct a wrapper object.
Methods	
boolean	booleanValue() Return the value of the wrapped primitive.
byte	byteValue() Return the value of the wrapped primitive
char	charValue() Return the value of the wrapped primitive
double	doubleValue() Return the value of the wrapped primitive
float	floatValue() Return the value of the wrapped primitive

Member Summary	
int	<code>intValue()</code> Return the value of the wrapped primitive
long	<code>longValue()</code> Return the value of the wrapped primitive
void	<code>setValue(double value)</code> Change the value of the wrapper object
void	<code>setValue(JIDLNumber value)</code> Change the value of the wrapper object
short	<code>shortValue()</code> Return the value of the wrapped primitive
java.lang.String	<code>toString()</code>

Inherited Member Summary
Methods inherited from class Object
<code>equals(Object)</code> , <code>getClass()</code> , <code>hashCode()</code> , <code>notify()</code> , <code>notifyAll()</code> , <code>wait(long, int)</code> , <code>wait(long, int)</code> , <code>wait(long, int)</code>

Constructors

JIDLDouble(double)

```
public JIDLDouble(double value)
```

Construct a wrapper object.

Parameters:

value - value to wrap for use in the export bridge

JIDLDouble(JIDLNumber)

```
public JIDLDouble(com.idl.javaidl.JIDLNumber value)
```

Construct a wrapper object.

Parameters:

value - JIDLNumber to wrap for use in the export bridge

Methods

booleanValue()

```
public boolean booleanValue()
```

Return the value of the wrapped primitive.

Specified By:

[booleanValue](#) in interface [JIDLNumber](#)

Returns:

true if non-zero, false otherwise

byteValue()

```
public byte byteValue()
```

Return the value of the wrapped primitive

Specified By:

[byteValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

charValue()

```
public char charValue()
```

Return the value of the wrapped primitive

Specified By:

[charValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

doubleValue()

```
public double doubleValue()
```

Return the value of the wrapped primitive

Specified By:

[doubleValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

floatValue()

```
public float floatValue()
```

Return the value of the wrapped primitive

Specified By:

[floatValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

intValue()

```
public int intValue()
```

Return the value of the wrapped primitive

Specified By:

[intValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

longValue()

```
public long longValue()
```

Return the value of the wrapped primitive

Specified By:

[longValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

setValue(double)

```
public void setValue(double value)
```

Change the value of the wrapper object

Parameters:

value - primitive value to wrap for use in the export bridge

setValue(JIDLNumber)

```
public void setValue(com.idl.javaidl.JIDLNumber value)
```

Change the value of the wrapper object

Specified By:

[setValue](#) in interface [JIDLNumber](#)

Parameters:

value - JIDLNumber to wrap for use in the export bridge

shortValue()

```
public short shortValue()
```

Return the value of the wrapped primitive

Specified By:

[shortValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

toString()

```
public java.lang.String toString()
```

Overrides:

[toString](#) in class [Object](#)

JIDLException

Declaration

```
public class JIDLException extends java.lang.Error implements
java.io.Serializable

java.lang.Object
|
+--java.lang.Throwable
|
+--java.lang.Error
|
+--com.idl.javaidl.JIDLException
```

All Implemented Interfaces:

java.io.Serializable

Direct Known Subclasses:

[JIDLAbortedException](#), [JIDLBusyException](#)

Description

An unchecked exception thrown when a call to IDL encounters an error.

Member Summary	
Methods	
long	getErrorCode() Get the IDL error code associated with the IDL error.
java.lang.String	toString()

Inherited Member Summary
Methods inherited from class <code>Object</code>
<code>equals(Object)</code> , <code>getClass()</code> , <code>hashCode()</code> , <code>notify()</code> , <code>notifyAll()</code> , <code>wait(long, int)</code> , <code>wait(long, int)</code> , <code>wait(long, int)</code>
Methods inherited from class <code>Throwable</code>
<code>fillInStackTrace()</code> , <code>getCause()</code> , <code>getLocalizedMessage()</code> , <code>getMessage()</code> , <code>getStackTrace()</code> , <code>initCause(Throwable)</code> , <code>printStackTrace(PrintWriter)</code> , <code>printStackTrace(PrintWriter)</code> , <code>printStackTrace(PrintWriter)</code> , <code>setStackTrace(StackTraceElement[])</code>

Methods

getErrorCode()

```
public long getErrorCode()
```

Get the IDL error code associated with the IDL error.

toString()

```
public java.lang.String toString()
```

Overrides:

`toString` in class `Throwable`

JIDLFloat

Declaration

```
public class JIDLFloat implements JIDLNumber,  
    java.io.Serializable  
  
    java.lang.Object  
    |  
    +---com.idl.javaidl.JIDLFloat
```

All Implemented Interfaces:

[JIDLNumber](#), [java.io.Serializable](#)

Description

The JIDLFloat class wraps a float as a mutable object usable by the Java-IDL Export bridge.

Member Summary	
Constructors	
	JIDLFloat(float value) Construct a wrapper object.
	JIDLFloat(JIDLNumber value) Construct a wrapper object.
Methods	
boolean	booleanValue() Return the value of the wrapped primitive.
byte	byteValue() Return the value of the wrapped primitive
char	charValue() Return the value of the wrapped primitive
double	doubleValue() Return the value of the wrapped primitive

Member Summary	
float	<code>floatValue()</code> Return the value of the wrapped primitive
int	<code>intValue()</code> Return the value of the wrapped primitive
long	<code>longValue()</code> Return the value of the wrapped primitive
void	<code>setValue(float value)</code>
void	<code>setValue(JIDLNumber value)</code> Change the value of the wrapper object
short	<code>shortValue()</code> Return the value of the wrapped primitive
java.lang.String	<code>toString()</code> Return the value of the wrapped primitive

Inherited Member Summary
Methods inherited from class <code>Object</code>
<code>equals(Object)</code> , <code>getClass()</code> , <code>hashCode()</code> , <code>notify()</code> , <code>notifyAll()</code> , <code>wait(long, int)</code> , <code>wait(long, int)</code> , <code>wait(long, int)</code>

Constructors

JIDLFloat(float)

```
public JIDLFloat(float value)
```

Construct a wrapper object.

Parameters:

value - value to wrap for use in the export bridge

JIDLFloat(JIDLNumber)

```
public JIDLFloat(com.idl.javaidl.JIDLNumber value)
```

Construct a wrapper object.

Parameters:

`value` - JIDLNumber to wrap for use in the export bridge

Methods

booleanValue()

```
public boolean booleanValue()
```

Return the value of the wrapped primitive.

Specified By:

[booleanValue](#) in interface [JIDLNumber](#)

Returns:

true if non-zero, false otherwise

byteValue()

```
public byte byteValue()
```

Return the value of the wrapped primitive

Specified By:

[byteValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

charValue()

```
public char charValue()
```

Return the value of the wrapped primitive

Specified By:

[charValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

doubleValue()

```
public double doubleValue()
```

Return the value of the wrapped primitive

Specified By:

[doubleValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

floatValue()

```
public float floatValue()
```

Return the value of the wrapped primitive

Specified By:

[floatValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

intValue()

```
public int intValue()
```

Return the value of the wrapped primitive

Specified By:

[intValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

longValue()

```
public long longValue()
```

Return the value of the wrapped primitive

Specified By:

[longValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

setValue(float)

```
public void setValue(float value)
```

setValue(JIDLNumber)

```
public void setValue(com.idl.javaidl.JIDLNumber value)
```

Change the value of the wrapper object

Specified By:

[setValue](#) in interface [JIDLNumber](#)

Parameters:

value - primitive value to wrap for use in the export bridge

shortValue()

```
public short shortValue()
```

Return the value of the wrapped primitive

Specified By:

[shortValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

toString()

```
public java.lang.String toString()
```

Return the value of the wrapped primitive

Overrides:

[toString](#) in class [Object](#)

Returns:

value that is wrapped by this object

JIDLInteger

Declaration

```
public class JIDLInteger implements JIDLNumber,
    java.io.Serializable

    java.lang.Object
    |
    +--com.idl.javaidl.JIDLInteger
```

All Implemented Interfaces:

```
JIDLNumber, java.io.Serializable
```

Description

The JIDLInteger class wraps an int as a mutable object usable by the Java-IDL Export bridge.

Member Summary	
Constructors	
	JIDLInteger(int value) Construct a wrapper object.
	JIDLInteger(JIDLNumber value) Construct a wrapper object.
Methods	
boolean	booleanValue() Return the value of the wrapped primitive.
byte	byteValue() Return the value of the wrapped primitive
char	charValue() Return the value of the wrapped primitive
double	doubleValue() Return the value of the wrapped primitive

Member Summary	
float	<code>floatValue()</code> Return the value of the wrapped primitive
int	<code>intValue()</code> Return the value of the wrapped primitive
long	<code>longValue()</code> Return the value of the wrapped primitive
void	<code>setValue(int value)</code> Change the value of the wrapper object
void	<code>setValue(JIDLNumber value)</code> Change the value of the wrapper object
short	<code>shortValue()</code> Return the value of the wrapped primitive
java.lang.String	<code>toString()</code> Return the value of the wrapped primitive

Inherited Member Summary
Methods inherited from class <code>Object</code>
<code>equals(Object)</code> , <code>getClass()</code> , <code>hashCode()</code> , <code>notify()</code> , <code>notifyAll()</code> , <code>wait(long, int)</code> , <code>wait(long, int)</code> , <code>wait(long, int)</code>

Constructors

JIDLInteger(int)

```
public JIDLInteger(int value)
```

Construct a wrapper object.

Parameters:

value - value to wrap for use in the export bridge

JIDLInteger(JIDLNumber)

```
public JIDLInteger(com.idl.javaidl.JIDLNumber value)
```

Construct a wrapper object.

Parameters:

`value` - `JIDLNumber` to wrap for use in the export bridge

Methods

booleanValue()

```
public boolean booleanValue()
```

Return the value of the wrapped primitive.

Specified By:

[booleanValue](#) in interface [JIDLNumber](#)

Returns:

`true` if non-zero, `false` otherwise

byteValue()

```
public byte byteValue()
```

Return the value of the wrapped primitive

Specified By:

[byteValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

charValue()

```
public char charValue()
```

Return the value of the wrapped primitive

Specified By:

[charValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

doubleValue()

```
public double doubleValue()
```

Return the value of the wrapped primitive

Specified By:

[doubleValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

floatValue()

```
public float floatValue()
```

Return the value of the wrapped primitive

Specified By:

[floatValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

intValue()

```
public int intValue()
```

Return the value of the wrapped primitive

Specified By:

[intValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

longValue()

```
public long longValue()
```

Return the value of the wrapped primitive

Specified By:

[longValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

setValue(int)

```
public void setValue(int value)
```

Change the value of the wrapper object

Parameters:

value - primitive value to wrap for use in the export bridge

setValue(JIDLNumber)

```
public void setValue(com.idl.javaidl.JIDLNumber value)
```

Change the value of the wrapper object

Specified By:

[setValue](#) in interface [JIDLNumber](#)

Parameters:

value - JIDLNumber to wrap for use in the export bridge

shortValue()

```
public short shortValue()
```

Return the value of the wrapped primitive

Specified By:

[shortValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

toString()

```
public java.lang.String toString()
```

Return the value of the wrapped primitive

Overrides:

[toString](#) in class [Object](#)

Returns:

value that is wrapped by this object

JIDLKeyListener

Declaration

```
public interface JIDLKeyListener
```

All Known Implementing Classes:

[JIDLCanvas](#)

Description

The listener interface for receiving keyboard events (key pressed, key released) on a [JIDLCanvas](#).

The class that is interested in handling these events implements this interface (and all the methods it contains). The listener object created from that class is then registered with the [JIDLCanvas](#) using the `addIDLKeyListener` method. The listener is unregistered with the `removeIDLKeyListener`.

The [JIDLCanvas](#) automatically handles key events whether a program registers an additional [JIDLKeyListener](#) or not. The [JIDLCanvas](#) is itself a [JIDLKeyListener](#) and provides default behavior for press and release. For a key press or key release, the default behavior is for the [JIDLCanvas](#) to call the IDL program's `OnKeyboard` method.

Note that clients should not register to listen to [JIDLCanvas](#) `KeyEvents` using a `KeyListener`, preferring the [JIDLKeyListener](#) instead.

See Also:

[JIDLCanvas](#), `java.awt.event.KeyEvent`,
`java.awt.event.KeyListener`

Member Summary	
Methods	
void	<code>IDLkeyPressed(JIDLObjectI obj, java.awt.event.KeyEvent event, int x, int y)</code> A key press has occurred inside the JIDLCanvas.
void	<code>IDLkeyReleased(JIDLObjectI obj, java.awt.event.KeyEvent event, int x, int y)</code> A key release has occurred inside the JIDLCanvas.

Methods

IDLkeyPressed(JIDLObjectI, KeyEvent, int, int)

```
public void IDLkeyPressed(com.idl.javaidl.JIDLObjectI obj,
    java.awt.event.KeyEvent event, int x, int y)
```

A key press has occurred inside the JIDLCanvas.

The default behavior of JIDLCanvas's default implementation is to call the IDL program's OnKeyboard method.

Parameters:

`obj` - The JIDLCanvas in which the event occurred.

`event` - The key event

`x` - The x pixel location in the canvas where the event occurred

`y` - The y pixel location in the canvas where the event occurred

IDLkeyReleased(JIDLObjectI, KeyEvent, int, int)

```
public void IDLkeyReleased(com.idl.javaidl.JIDLObjectI obj,
    java.awt.event.KeyEvent event, int x, int y)
```

A key release has occurred inside the JIDLCanvas.

The default behavior of JIDLCanvas's default implementation is to call the IDL program's OnKeyboard method.

Parameters:

`obj` - The JIDLCanvas in which the event occurred.

event - The key event

x - The x pixel location in the canvas where the event occurred

y - The y pixel location in the canvas where the event occurred

JIDLLong

Declaration

```
public class JIDLLong implements JIDLNumber,  
    java.io.Serializable  
  
    java.lang.Object  
    |  
    +---com.idl.javaidl.JIDLLong
```

All Implemented Interfaces:

[JIDLNumber](#), [java.io.Serializable](#)

Description

The JIDLLong class wraps a long as a mutable object usable by the Java-IDL Export bridge.

Member Summary	
Constructors	
	JIDLLong(JIDLNumber value) Construct a wrapper object.
	JIDLLong(long value) Construct a wrapper object.
Methods	
boolean	booleanValue() Return the value of the wrapped primitive.
byte	byteValue() Return the value of the wrapped primitive
char	charValue() Return the value of the wrapped primitive
double	doubleValue() Return the value of the wrapped primitive

Member Summary	
float	<code>floatValue()</code> Return the value of the wrapped primitive
int	<code>intValue()</code> Return the value of the wrapped primitive
long	<code>longValue()</code> Return the value of the wrapped primitive
void	<code>setValue(JIDLNumber value)</code> Change the value of the wrapper object
void	<code>setValue(long value)</code> Change the value of the wrapper object
short	<code>shortValue()</code> Return the value of the wrapped primitive
<code>java.lang.String</code>	<code>toString()</code> Return the value of the wrapped primitive

Inherited Member Summary
Methods inherited from class <code>Object</code>
<code>equals(Object)</code> , <code>getClass()</code> , <code>hashCode()</code> , <code>notify()</code> , <code>notifyAll()</code> , <code>wait(long, int)</code> , <code>wait(long, int)</code> , <code>wait(long, int)</code>

Constructors

JIDLLong(JIDLNumber)

```
public JIDLLong(com.idl.javaidl.JIDLNumber value)
```

Construct a wrapper object.

Parameters:

`value` - `JIDLNumber` to wrap for use in the export bridge

JIDLLong(long)

```
public JIDLLong(long value)
```

Construct a wrapper object.

Parameters:

`value` - value to wrap for use in the export bridge

Methods

booleanValue()

```
public boolean booleanValue()
```

Return the value of the wrapped primitive.

Specified By:

[booleanValue](#) in interface [JIDLNumber](#)

Returns:

true if non-zero, false otherwise

byteValue()

```
public byte byteValue()
```

Return the value of the wrapped primitive

Specified By:

[byteValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

charValue()

```
public char charValue()
```

Return the value of the wrapped primitive

Specified By:

[charValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

doubleValue()

```
public double doubleValue()
```

Return the value of the wrapped primitive

Specified By:

[doubleValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

floatValue()

```
public float floatValue()
```

Return the value of the wrapped primitive

Specified By:

[floatValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

intValue()

```
public int intValue()
```

Return the value of the wrapped primitive

Specified By:

[intValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

longValue()

```
public long longValue()
```

Return the value of the wrapped primitive

Specified By:

[longValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

setValue(JIDLNumber)

```
public void setValue(com.idl.javaidl.JIDLNumber value)
```

Change the value of the wrapper object

Specified By:

[setValue](#) in interface [JIDLNumber](#)

Parameters:

value - JIDLNumber to wrap for use in the export bridge

setValue(long)

```
public void setValue(long value)
```

Change the value of the wrapper object

Parameters:

value - primitive value to wrap for use in the export bridge

shortValue()

```
public short shortValue()
```

Return the value of the wrapped primitive

Specified By:

[shortValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

toString()

```
public java.lang.String toString()
```

Return the value of the wrapped primitive

Overrides:

[toString](#) in class [Object](#)

Returns:

value that is wrapped by this object

JIDLMouseListener

Declaration

```
public interface JIDLMouseListener
```

All Known Implementing Classes:

[JIDLCanvas](#)

Description

The listener interface for receiving mouse events from IDL (press, release, enter, and exit) on a JIDLCanvas. A mouse event is generated when the mouse is pressed, released, the mouse cursor enters or leaves the JIDLCanvas component.

Note: Mouse moves and drags are tracked using JIDLMouseMotionListener.

The class that is interested in processing an IDL mouse event implements this interface (and all the methods it contains). The listener object created from that class is then registered with the JIDLCanvas using the addIDLMouseListener method. The listener is unregistered with the removeIDLMouseListener.

The JIDLCanvas automatically handles mouse events whether a program registers an additional JIDLMouseListener or not. The JIDLCanvas is itself a JIDLMouseListener and provides default behavior for the 4 events, as denoted in the specific methods below.

Note that clients should not register to listen to JIDLCanvas MouseEvents using a MouseListener, preferring the JIDLMouseListener instead.

See Also:

[JIDLCanvas](#), [JIDLMouseMotionListener](#), `java.awt.event.MouseEvent`, `java.awt.event.MouseListener`

Member Summary	
Methods	
void	<code>IDLmouseEntered(JIDLObjectI obj, java.awt.event.MouseEvent event)</code> The mouse has entered the JIDLCanvas.
void	<code>IDLmouseExited(JIDLObjectI obj, java.awt.event.MouseEvent event)</code> The mouse has exiting the JIDLCanvas.
void	<code>IDLmousePressed(JIDLObjectI obj, java.awt.event.MouseEvent event)</code> A mouse button was pressed inside the JIDLCanvas.
void	<code>IDLmouseReleased(JIDLObjectI obj, java.awt.event.MouseEvent event)</code> A mouse button was released inside the JIDLCanvas.

Methods

IDLmouseEntered(JIDLObjectI, MouseEvent)

```
public void IDLmouseEntered(com.idl.javaidl.JIDLObjectI obj,
    java.awt.event.MouseEvent event)
```

The mouse has entered the JIDLCanvas.

The default behavior of JIDLCanvas's default implementation is to call the IDL program's OnEnter method.

Parameters:

`obj` - The JIDLCanvas in which the event occurred.

`event` - The mouse event

IDLmouseExited(JIDLObjectI, MouseEvent)

```
public void IDLmouseExited(com.idl.javaidl.JIDLObjectI obj,
    java.awt.event.MouseEvent event)
```

The mouse has exiting the JIDLCanvas.

The default behavior of JIDLCanvas's default implementation is to call the IDL program's OnExit method.

Parameters:

obj - The JIDLCanvas in which the event occurred.

event - The mouse event

IDLmousePressed(JIDLObjectI, MouseEvent)

```
public void IDLmousePressed(com.idl.javaidl.JIDLObjectI obj,  
    java.awt.event.MouseEvent event)
```

A mouse button was pressed inside the JIDLCanvas.

The default behavior of JIDLCanvas's default implementation is to call the IDL program's OnMouseDown method.

Parameters:

obj - The JIDLCanvas in which the event occurred.

event - The mouse event

IDLmouseReleased(JIDLObjectI, MouseEvent)

```
public void IDLmouseReleased(com.idl.javaidl.JIDLObjectI obj,  
    java.awt.event.MouseEvent event)
```

A mouse button was released inside the JIDLCanvas.

The default behavior of JIDLCanvas's default implementation is to call the IDL program's OnMouseUp method.

Parameters:

obj - The JIDLCanvas in which the event occurred.

event - The mouse event

JIDLMouseMotionListener

Declaration

```
public interface JIDLMouseMotionListener
```

All Known Implementing Classes:

[JIDLCanvas](#)

Description

The listener interface for receiving mouse motion events from IDL (move and drag) on a JIDLCanvas. (Mouse presses, releases, enter and exits are tracked using JIDLMouseListener.)

The class that is interested in processing an IDL mouse motion event implements this interface (and all the methods it contains). The listener object created from that class is then registered with the JIDLCanvas using the `addIDLMouseMotionListener` method. The listener is unregistered with the `removeIDLMouseMotionListener`.

The JIDLCanvas automatically handles mouse motion events whether a program registers an additional JIDLMouseMotionListener or not. The JIDLCanvas is itself a JIDLMouseMotionListener and provides default behavior which is to call the IDL object's `OnMouseMotion` method.

Note that clients should not register to listen to JIDLCanvas mouse motion events using a `MouseListener`, preferring the `JIDLMouseMotionListener` instead.

See Also:

[JIDLCanvas](#), [JIDLMouseListener](#), `java.awt.event.MouseEvent`,
`java.awt.event.MouseMotionListener`

Member Summary	
Methods	
void	<code>IDLmouseDragged(JIDLObjectI obj, java.awt.event.MouseEvent event)</code> A mouse was dragged inside the JIDLCanvas.
void	<code>IDLmouseMoved(JIDLObjectI obj, java.awt.event.MouseEvent event)</code> A mouse was moved inside the JIDLCanvas.

Methods

IDLmouseDragged(JIDLObjectI, MouseEvent)

```
public void IDLmouseDragged(com.idl.javaidl.JIDLObjectI obj,
    java.awt.event.MouseEvent event)
```

A mouse was dragged inside the JIDLCanvas.

The default behavior of JIDLCanvas's default implementation is to call the IDL program's OnMouseMotion method.

Parameters:

`obj` - The JIDLCanvas in which the event occurred.

`event` - The mouse event

IDLmouseMoved(JIDLObjectI, MouseEvent)

```
public void IDLmouseMoved(com.idl.javaidl.JIDLObjectI obj,
    java.awt.event.MouseEvent event)
```

A mouse was moved inside the JIDLCanvas.

The default behavior of JIDLCanvas's default implementation is to call the IDL program's OnMouseMotion method.

Parameters:

`obj` - The JIDLCanvas in which the event occurred.

`event` - The mouse event

JIDLMouseWheelListener

Declaration

```
public interface JIDLMouseWheelListener
```

All Known Implementing Classes:

[JIDLCanvas](#)

Description

The listener interface for receiving mouse wheel events on a JIDLCanvas.

The class that is interested in processing an IDL mouse wheel event implements this interface. The listener object created from that class is then registered with the JIDLCanvas using the `addIDLMouseWheelListener` method. The listener is unregistered with the `removeIDLMouseWheelListener`.

The JIDLCanvas automatically handles mouse wheel events whether a program registers an additional JIDLMouseWheelListener or not. The JIDLCanvas is itself a JIDLMouseWheelListener and provides default behavior for the event.

Note that clients should not register to listen to JIDLCanvas MouseWheelEvents using a MouseWheelListener, preferring the JIDLMouseWheelListener instead.

Note

The Java convention for mouse wheel direction is the opposite of IDL's. This is transparent to IDL applications because when the MouseWheelEvent is passed to IDL, the sign is flipped.

See Also:

[JIDLCanvas](#), [JIDLMouseListener](#), `java.awt.event.MouseWheelEvent`, `java.awt.event.MouseWheelListener`

Member Summary	
Methods	
void	<code>IDLmouseWheelMoved(JIDLObjectI obj, java.awt.event.MouseWheelEvent event)</code> A mouse wheel was moved inside the JIDLCanvas.

Methods

IDLmouseWheelMoved(JIDLObjectI, MouseWheelEvent)

```
public void IDLmouseWheelMoved(com.idl.javaidl.JIDLObjectI obj,  
java.awt.event.MouseWheelEvent event)
```

A mouse wheel was moved inside the JIDLCanvas.

The default behavior of JIDLCanvas's default implementation is to call the IDL program's OnWheel method.

Parameters:

`obj` - The JIDLCanvas in which the event occurred.

`event` - The mouse wheel event

JIDLNotifyListener

Declaration

```
public interface JIDLNotifyListener
```

Description

The listener interface for receiving notify events from IDL.

IDL objects that sub-class `itComponent` may trigger a notification by calling `IDLitComponent::Notify`. Both drawable (`JIDLCanvas`) and non-drawable (`JIDLObject`) wrapper objects may be listened to. However by default, `JIDLObject` and `JIDLCanvas` objects do NOT listen to their output events.

The class that is interested in receiving IDL notify events of a particular object * implements this interface. The listener object created from that class is registered with the `JIDLObjectI` using the `addIDLNotifyListener` method. The listener is unregistered with the `removeIDLNotifyListener`.

See Also:

[JIDLCanvas](#), [JIDLObject](#), [JIDLObjectI](#)

Member Summary	
Methods	
void	<code>OnIDLNotify(JIDLObjectI obj, java.lang.String s1, java.lang.String s2)</code> An IDL notify has occurred.

Methods

OnIDLNotify(JIDLObjectI, String, String)

```
public void OnIDLNotify(com.idl.javaidl.JIDLObjectI obj,  
    java.lang.String s1, java.lang.String s2)
```

An IDL notify has occurred.

Parameters:

obj - The JIDLObjectI in which the event occurred.

s1 - The first string parameter sent via IdlIComponent:Notify

s2 - The second string parameter sent via IdlIComponent:Notify

JIDLNumber

Declaration

```
public interface JIDLNumber
```

All Known Implementing Classes:

[JIDLShort](#), [JIDLLong](#), [JIDLInteger](#), [JIDLFloat](#), [JIDLDouble](#), [JIDLChar](#), [JIDLByte](#), [JIDLBoolean](#)

Description

The JIDLNumber class wraps a primitive java number as a mutable object usable by the Java-IDL Export bridge.

Member Summary	
Methods	
boolean	booleanValue() Return the value of the wrapped primitive.
byte	byteValue() Return the value of the wrapped primitive.
char	charValue() Return the value of the wrapped primitive.
double	doubleValue() Return the value of the wrapped primitive.
float	floatValue() Return the value of the wrapped primitive.
int	intValue() Return the value of the wrapped primitive.

Member Summary	
long	<code>longValue()</code> Return the value of the wrapped primitive.
void	<code>setValue(JIDLNumber value)</code> Change the value of the wrapper object
short	<code>shortValue()</code> Return the value of the wrapped primitive.

Methods

booleanValue()

```
public boolean booleanValue()
```

Return the value of the wrapped primitive.

Returns:

true if non-zero, false otherwise

byteValue()

```
public byte byteValue()
```

Return the value of the wrapped primitive.

Returns:

value that is wrapped by this object

charValue()

```
public char charValue()
```

Return the value of the wrapped primitive.

Returns:

value that is wrapped by this object

doubleValue()

```
public double doubleValue()
```

Return the value of the wrapped primitive.

Returns:

value that is wrapped by this object

floatValue()

```
public float floatValue()
```

Return the value of the wrapped primitive.

Returns:

value that is wrapped by this object

intValue()

```
public int intValue()
```

Return the value of the wrapped primitive.

Returns:

value that is wrapped by this object

longValue()

```
public long longValue()
```

Return the value of the wrapped primitive.

Returns:

value that is wrapped by this object

setValue(JIDLNumber)

```
public void setValue(com.idl.javaidl.JIDLNumber value)
```

Change the value of the wrapper object

Parameters:

value - JIDLNumber to wrap for use in the export bridge

shortValue()

```
public short shortValue()
```

Return the value of the wrapped primitive.

Returns:

value that is wrapped by this object

JIDLObject

Declaration

```
public class JIDLObject implements JIDLObjectI,
    java.io.Serializable

    java.lang.Object
    |
    +--com.idl.javaidl.JIDLObject
```

All Implemented Interfaces:

JIDLObjectI, java.io.Serializable

Description

This class wraps an IDL object.

In many of the methods of this class, one or more flags are required to be specified for parameters being passed to or from the bridge. These flags follow the following guidelines:

For all types of parameters (subclasses of **JIDLNumber**, **JIDLString**, **JIDLObjectI** and **JIDLArray**), a flag should be set that determines whether the parameter is in-only (const) or in-out (we expect it to be changed by IDL). The constants that determine this are either **JIDLConst.PARMFLAG_CONST** or **JIDLConst.PARMFLAG_IN_OUT**.

For parameters that are arrays, a flag should be set that tells the bridge whether the array is to be convolved when passed to IDL. If the **PARM_IN_OUT** flag is set, this flag will also tell the bridge whether to convolve the array when it is copied back to Java. The constants that determine this are either **JIDLConst.PARMFLAG_CONVMAJORITY** or **JIDLConst.PARMFLAG_NO_CONVMAJORITY**.

For example, if the parameter in question is an array that is to be modified by IDL (in-out) and needs to be convolved when passed to and from IDL, we would set its argpal array member as follows:

```
argpal[2] = JIDLConst.PARMFLAG_IN_OUT | JIDLConst.PARMFLAG_CONV
MAJORITY;
```

Member Summary	
Methods	
void	<code>abort()</code> Requests that the IDL process containing the underlying IDL object abort its current activity.
void	<code>addIDLNotifyListener(JIDLNotifyListener listener)</code> Adds the specified IDL notify listener to receive IDL notification events on this object.
void	<code>addIDLOutputListener(JIDLOutputListener listener)</code> Adds the specified IDL output listener to receive IDL output events on this object.
java.lang.Object	<code>callFunction(java.lang.String sMethodName, int iPalFlag)</code> Call IDL function that accepts zero parameters.
java.lang.Object	<code>callFunction(java.lang.String sMethodName, int argc, java.lang.Object argv, int[] argpal, int iPalFlag)</code> Call IDL function.
void	<code>callProcedure(java.lang.String sMethodName)</code> Call IDL procedure that accepts zero parameters.
void	<code>callProcedure(java.lang.String sMethodName, int argc, java.lang.Object argv, int[] argpal)</code> Call IDL procedure.
void	<code>createObject()</code> Create the wrapped object by calling IDL's ::INIT method.
void	<code>createObject(int argc, java.lang.Object argv, int[] argpal)</code> Create the wrapped object by calling IDL's ::INIT method.

Member Summary	
void	<pre>createObject(int argc, java.lang.Object argv, int[] argpal, com.idl.javaidl.JIDLProcessInitializer initializer)</pre> <p>Create the wrapped object by calling IDL's ::INIT method.</p>
void	<pre>createObject(com.idl.javaidl.JIDLProcessInit ializer initializer)</pre> <p>Create the wrapped object by calling IDL's ::INIT method.</p>
void	<pre>destroyObject()</pre> <p>Destroys the underlying IDL object associated with the wrapper.</p>
void	<pre>executeString(java.lang.String sCmd)</pre> <p>Execute the given command string in IDL.</p>
java.lang.String	<pre>getClassName()</pre> <p>Get the class name of the object.</p>
long	<pre>getCookie()</pre> <p>Internal use.</p>
java.lang.String	<pre>getIdLObjectClassName()</pre> <p>Retrieves the IDL object class name of the underlying IDL object.</p>
java.lang.String	<pre>getIdLObjectVariableName()</pre> <p>When the underlying IDL object was created in the IDL process, it was assigned a variable name.</p>
java.lang.Object	<pre>getIdlVariable(java.lang.String sVar)</pre> <p>Given a variable name, return the IDL variable.</p>
java.lang.String	<pre>getObjVariableName()</pre> <p>Get the IDL Variable name of the given object</p>
java.lang.String	<pre>getProcessName()</pre> <p>Returns the name of the process that contains the underlying IDL object.</p>
java.lang.Object	<pre>getProperty(java.lang.String sProperty, int iPalFlag)</pre> <p>Call IDL getProperty method to get named property.</p>

Member Summary	
void	<code>initListeners()</code> Initialize listeners.
boolean	<code>isObjCreated()</code> Determine if object has been created successfully.
boolean	<code>isObjectCreated()</code> Determine if object has been created successfully.
boolean	<code>isObjectDisplayable()</code>
void	<code>removeIDLNotifyListener(JIDLNotifyListener listener)</code> Removes the specified IDL notify listener so it no longer receives IDL notifications.
void	<code>removeIDLOutputListener(JIDLOutputListener listener)</code> Removes the specified IDL output listener on this object.
void	<code>setIDLVariable(java.lang.String sVar, java.lang.Object obj)</code> Set/Create an IDL variable of the given name and value.
void	<code>setProcessName(java.lang.String process)</code> Set the process name that the object will be created in.
void	<code>setProperty(java.lang.String sProperty, java.lang.Object obj, int iPalFlag)</code> Call IDL setProperty method to set named property.
java.lang.String	<code>toString()</code> Returns a string representation of the object.

Inherited Member Summary
Methods inherited from class <code>Object</code>
<code>equals(Object)</code> , <code>getClass()</code> , <code>hashCode()</code> , <code>notify()</code> , <code>notifyAll()</code> , <code>wait(long, int)</code> , <code>wait(long, int)</code> , <code>wait(long, int)</code>

Methods

abort()

```
public void abort()
```

Requests that the IDL process containing the underlying IDL object abort its current activity.

This is only a request and IDL may take a long time before it actually stops.

The client can only Abort the current IDL activity if that wrapper object is the current “owner” of the underlying IDL.

Specified By:

[abort](#) in interface [JIDLObjectI](#)

Throws:

[JIDLException](#) - If IDL encounters an error.

See Also:

[JIDLAbortedException](#)

addIDLNotifyListener(JIDLNotifyListener)

```
public void  
addIDLNotifyListener(com.idl.javaidl.JIDLNotifyListener listener)
```

Adds the specified IDL notify listener to receive IDL notification events on this object.

Note that registering/unregistering for events should happen in the `initListeners` method or AFTER the `createObject` method.

Specified By:

[addIDLNotifyListener](#) in interface [JIDLObjectI](#)

Parameters:

`listener` - the listener

addIDLOutputListener(JIDLOutputListener)

```
public void  
addIDLOutputListener(com.idl.javaidl.JIDLOutputListener listener)
```

Adds the specified IDL output listener to receive IDL output events on this object.

Note that registering/unregistering for events should happen in the `initListeners` method or AFTER the `createObject` method.

Specified By:

[addIDLOutputListener](#) in interface [JIDLObjectI](#)

Parameters:

`listener` - the listener

callFunction(String, int)

```
public java.lang.Object callFunction(java.lang.String  
sMethodName, int iPalFlag)
```

Call IDL function that accepts zero parameters.

Parameters:

`sMethodName` - the function name

`iPalFlag` - a flag determining whether a returned array is convolved or not. If the returned value is not an array, this value is zero. See class description for more information.

Returns:

an Object of type `JIDLNumber`, `JIDLString`, `JIDLObject` or `JIDLArray`. The caller must know the type of the Object being returned and cast it to its proper type. May also return null.

Throws:

[JIDLException](#) - If IDL encounters an error.

See Also:

[JIDLObjectI.callFunction\(String, int, Object\[\], int\[\], int\)](#)

callFunction(String, int, Object[], int[], int)

```
public java.lang.Object callFunction(java.lang.String  
sMethodName, int argc, java.lang.Object[] argv, int[] argpal,  
int iPalFlag)
```

Call IDL function.

The `argpal` parameter is an array of flags created by OR-ing constants from class `JIDLConst`. Each array element corresponds to the equivalent parameter in `argv`.

Specified By:

[callFunction](#) in interface [JIDLObjectI](#)

Parameters:

sMethodName - the procedure name

argc - the number of parameters

argv - array of Objects to be passed to IDL. This array should be of length `argc` and should contain objects of type [JIDLNumber](#), [JIDLObject](#), [JIDLString](#) or [JIDLObject](#).

argpal - array of flags denoting whether each argv parameter passed to be bridge is 1) in-out vs constant; or 2) a convolved or non-convolved array This array should be of length `argc`.

iPalFlag - a flag determining whether a returned array if convolved or not. If the returned value is not an array, this value is zero.

Returns:

an Object of type [JIDLNumber](#), [JIDLString](#), [JIDLObjectI](#) or [JIDLArray](#). The caller must know the type of the Object being returned and cast it to its proper type.

Throws:

[JIDLException](#) - If IDL encounters an error.

See Also:

[JIDLNumber](#), [JIDLObject](#), [JIDLString](#), [JIDLArray](#),
[JIDLConst.PARMFLAG_CONST](#), [JIDLConst.PARMFLAG_IN_OUT](#),
[JIDLConst.PARMFLAG_CONVMAJORITY](#),
[JIDLConst.PARMFLAG_NO_CONVMAJORITY](#)

callProcedure(String)

```
public void callProcedure(java.lang.String sMethodName)
```

Call IDL procedure that accepts zero parameters.

Parameters:

sMethodName - the procedure name

Throws:

[JIDLException](#) - If IDL encounters an error.

See Also:

[callProcedure\(String, int, Object\[\], int\[\]\)](#)

callProcedure(String, int, Object[], int[])

```
public void callProcedure(java.lang.String sMethodName,
    int argc, java.lang.Object[] argv, int[] argpal)
```

Call IDL procedure.

The argpal parameter is an array of flags created by OR-ing constants from class JIDLConst. Each array element corresponds to the equivalent parameter in argv.

Specified By:

[callProcedure](#) in interface [JIDLObjectI](#)

Parameters:

sMethodName - the procedure name

argc - the number of parameters

argv - array of Objects to be passed to IDL. This array should be of length argc and should contain objects of type JIDLNumber, JIDLObject, JIDLString or JIDLObject.

argpal - array of flags denoting whether each argv parameter passed to be bridge is 1) in-out vs constant; or 2) a convolved or non-convolved array This array should be of length argc.

Throws:

[JIDLException](#) - If IDL encounters an error.

See Also:

[JIDLNumber](#), [JIDLObject](#), [JIDLString](#), [JIDLArray](#),
[JIDLConst.PARMFLAG_CONST](#), [JIDLConst.PARMFLAG_IN_OUT](#),
[JIDLConst.PARMFLAG_CONVMAJORITY](#),
[JIDLConst.PARMFLAG_NO_CONVMAJORITY](#)

createObject()

```
public void createObject()
```

Create the wrapped object by calling IDL's ::INIT method.

Used for ::INIT methods that take zero parameters. Assumes a default [JIDLProcessInitializer](#).

Throws:

[JIDLException](#) - If IDL encounters an error.

See Also:

[createObject\(int, Object\[\], int\[\]\)](#)

createObject(int, Object[], int[])

```
public void createObject (int argc, java.lang.Object[] argv,
int[] argpal)
```

Create the wrapped object by calling IDL's ::INIT method.

The argc, argv, argpal parameters are used to supply parameters to the underlying IDL object's ::Init method.

If the ::Init method does not have any parameters, the caller sets argc, argv, argpal to 0, null, null, respectively.

createObject does the following:

- Calls ::Init method in the IDL object
- Calls the superclass initListeners method to initialize any event handlers. The initListeners method has default behavior, which is different for graphical and non-graphical objects. If the default behavior is not desired, a sub-class to modify the listener initialization may override the initListeners method.

Specified By:

[createObject](#) in interface [JIDLObjectI](#)

Parameters:

argc - the number of parameters to be passed to INIT

argv - array of Objects to be passed to IDL. This array should be of length argc and should contain objects of type JIDLNumber, JIDLObject, JIDLString or JIDLArray.

argpal - array of flags denoting whether each argv parameter that is of type array should be convolved or not. For parameters that are not arrays, the value within the array will always be 0.

Throws:

[JIDLException](#) - If IDL encounters an error.

createObject(int, Object[], int[], JIDLProcessInitializer)

```
public void createObject(int argc, java.lang.Object[] argv,  
int[] argpal, com.idl.javaidl.JIDLProcessInitializer initializer)
```

Create the wrapped object by calling IDL's `::INIT` method.

The `argc`, `argv`, `argpal` parameters are used to supply parameters to the underlying IDL object's `::Init` method. The `initializer` parameter is used to supply IDL process initialization values.

If the `::Init` method does not have any parameters, the caller sets `argc`, `argv`, `argpal` to 0, null, null, respectively.

`createObject` does the following:

- Calls `::Init` method in the IDL object
- Calls the superclass `initListeners` method to initialize any event handlers. The `initListeners` method has default behavior, which is different for graphical and non-graphical objects. If the default behavior is not desired, a sub-class to modify the listener initialization may override the `initListeners` method.

Specified By:

`createObject` in interface [JIDLObjectI](#)

Parameters:

`argc` - the number of parameters to be passed to `INIT`

`argv` - array of Objects to be passed to IDL. This array should be of length `argc` and should contain objects of type `JIDLNumber`, `JIDLObject`, `JIDLString` or `JIDLArray`.

`argpal` - array of flags denoting whether each `argv` parameter that is of type array should be convolved or not. For parameters that are not arrays, the value within the array will always be 0.

`initializer` - a `JIDLProcessInitializer` object that specifies IDL process initialization parameters such as the licensing mode to be used. See [“IDL Licensing Modes”](#) on page 134 for details on the default licensing mechanism used when no `JIDLProcessInitializer` is specified.

Throws:

[JIDLException](#) - If IDL encounters an error.

createObject(JIDLProcessInitializer)

```
public void createObject(com.idl.javaidl.JIDLProcessInitializer  
initializer)
```

Create the wrapped object by calling IDL's ::INIT method.

Used for ::INIT methods that take zero parameters.

The initializer parameter is used to supply IDL process initialization values.

Parameters:

initializer - a JIDLProcessInitializer object that specifies IDL process initialization parameters such as the licensing mode to be used. See “[IDL Licensing Modes](#)” on page 134 for details on the default licensing mechanism used when no JIDLProcessInitializer is specified.

Throws:

[JIDLException](#) - If IDL encounters an error.

destroyObject()

```
public void destroyObject()
```

Destroys the underlying IDL object associated with the wrapper.

If the object being destroyed is the last object within an OPS process, the OPS process is also destroyed.

Note that this does not destroy the actual wrapper object. Because the wrapper object is a Java object, it follows all the Java reference counting/garbage collection schemes. Once all references to the wrapper object are released from Java code and once the Java Virtual Machine calls the garbage collector, the wrapper object may be deleted from memory.

Specified By:

[destroyObject](#) in interface [JIDLObjectI](#)

executeString(String)

```
public void executeString(java.lang.String sCmd)
```

Execute the given command string in IDL.

Specified By:

[executeString](#) in interface [JIDLObjectI](#)

Parameters:

sCmd - the single-line command to execute in IDL.

Throws:

[JIDLException](#) - If IDL encounters an error.

getClassName()

```
public java.lang.String getClassName()
```

Deprecated.

Replaced by [getIDLObjectClassName\(\)](#)

Get the class name of the object.

Returns:

class name ("" if object not created yet)

getCookie()

```
public long getCookie()
```

Internal use.

Specified By:

[getCookie](#) in interface [JIDLObjectI](#)

getIDLObjectClassName()

```
public java.lang.String getIDLObjectClassName()
```

Retrieves the IDL object class name of the underlying IDL object.

Specified By:

[getIDLObjectClassName](#) in interface [JIDLObjectI](#)

Returns:

the IDL object class name

getIDLObjectVariableName()

```
public java.lang.String getIDLObjectVariableName()
```

When the underlying IDL object was created in the IDL process, it was assigned a variable name. This method retrieves that name.

Specified By:

[getIDLObjectVariableName](#) in interface [JIDLObjectI](#)

Returns:

the variable name

getIDLVariable(String)

```
public java.lang.Object getIDLVariable(java.lang.String sVar)
```

Given a variable name, return the IDL variable.

Note that in the case of arrays, the array will ALWAYS be convolved when passed between Java and IDL.

Specified By:

[getIDLVariable](#) in interface [JIDLObjectI](#)

Parameters:

sVar - The IDL variable name

Returns:

an Object of type JIDLNumber, JIDLString, JIDLObject or JIDLArray. The caller must know the type of the Object being returned and cast it to its proper type. May also return null.

Throws:

[JIDLException](#) - If IDL encounters an error.

getObjVariableName()

```
public java.lang.String getObjVariableName()
```

Deprecated.

Replaced by [getIDLObjectVariableName\(\)](#)

Get the IDL Variable name of the given object

Returns:

a String representing the IDL Variable name

getProcessName()

```
public java.lang.String getProcessName()
```


Returns the name of the process that contains the underlying IDL object. For an in-process object, returns an empty string.

Specified By:

[getProcessName](#) in interface [JIDLObjectI](#)

Returns:

process name. Empty string if the process is in-process.

getProperty(String, int)

```
public java.lang.Object getProperty(java.lang.String  
sProperty, int iPalFlag)
```

Call IDL `getProperty` method to get named property.

Specified By:

[getProperty](#) in interface [JIDLObjectI](#)

Parameters:

`sProperty` - the property name

`iPalFlag` - a flag determining whether a returned array will be convolved or not. If the returned value is not is ignored.

Returns:

an Object of type `JIDLNumber`, `JIDLString`, `JIDLObject` or `JIDLArray`. The caller must know the type of the Object being returned and cast it to its proper type. May also return null.

Throws:

[JIDLException](#) - If IDL encounters an error.

See Also:

[JIDLNumber](#), [JIDLObjectI](#), [JIDLString](#), [JIDLArray](#),
[JIDLConst.PARMFLAG_CONVMAJORITY](#),
[JIDLConst.PARMFLAG_NO_CONVMAJORITY](#)

initListeners()

```
public void initListeners()
```

Initialize listeners.

This method is always called by `createObject`. The `JIDLObject` listens to no events, but this method may be overridden by sub-classes to initialize a different set of listeners (or none at all).

For example if a sub-class of `JIDLObject` wished to listen to IDL output events, it would need to implement `JIDLOutputListener` and register to listen for these events in `initListeners`, as follows:

```
public class newObject extends JIDLObject implements JIDLOutputLi
stener
{
    public void initListeners() {
        addIDLOutputListener(this);
    }
    void IDLoutput(JIDLObjectI obj, String s) {
        // do something with the output
    }
}
```

Specified By:

`initListeners` in interface [JIDLObjectI](#)

See Also:

[JIDLNotifyListener](#), [JIDLOutputListener](#)

isObjCreated()

```
public boolean isObjCreated()
```

Deprecated.

Replaced by [isObjectCreated\(\)](#)

Determine if object has been created successfully.

Returns:

true if object created successfully, or false if object not created or creation was unsuccessful.

isObjectCreated()

```
public boolean isObjectCreated()
```

Determine if object has been created successfully.

Specified By:

[isObjectCreated](#) in interface [JIDLObjectI](#)

Returns:

true if object created successfully, or false if object not created, destroyed, or creation was unsuccessful.

See Also:

[createObject\(\)](#)

isObjectDisplayable()

```
public boolean isObjectDisplayable()
```

Specified By:

[isObjectDisplayable](#) in interface [JIDLObjectI](#)

removeIDLNotifyListener(JIDLNotifyListener)

```
public void  
removeIDLNotifyListener(com.idl.javaidl.JIDLNotifyListener  
listener)
```

Removes the specified IDL notify listener so it no longer receives IDL notifications.

Note that registering/unregistering for events should happen in the `initListeners` method or AFTER the `createObject` method.

Specified By:

[removeIDLNotifyListener](#) in interface [JIDLObjectI](#)

Parameters:

`listener` - the listener

removeIDLOutputListener(JIDLOutputListener)

```
public void  
removeIDLOutputListener(com.idl.javaidl.JIDLOutputListener  
listener)
```

Removes the specified IDL output listener on this object.

Note that registering/unregistering for events should happen in the `initListeners` method or AFTER the `createObject` method.

Specified By:

[removeIDLOutputListener](#) in interface [JIDLObjectI](#)

Parameters:

listener - the listener

setIDLVariable(String, Object)

```
public void setIDLVariable(java.lang.String sVar,  
    java.lang.Object obj)
```

Set/Create an IDL variable of the given name and value.

Note that in the case of arrays, the array will ALWAYS be convolved when passed between Java and IDL.

Specified By:

[setIDLVariable](#) in interface [JIDLObjectI](#)

Parameters:

sVar - the IDL variable name

obj - object to be passed to IDL. Should be an object of type [JIDLNumber](#), [JIDLObject](#), [JIDLString](#) or [JIDLArray](#).

Throws:

[JIDLException](#) - If IDL encounters an error.

setProcessName(String)

```
public void setProcessName(java.lang.String process)
```

Set the process name that the object will be created in.

The process name may only be set before `createObject` is called. If called after the object has been created, this method call does nothing.

Specified By:

[setProcessName](#) in interface [JIDLObjectI](#)

Parameters:

process - Process name. Empty String means create in same process (in-process).

setProperty(String, Object, int)

```
public void setProperty(java.lang.String sProperty,  
java.lang.Object obj, int iPalFlag)
```

Call IDL setProperty method to set named property.

The iPalFlag parameter is a set of flags that are or-ed together. Currently this parameter is only used to specify whether a JIDLArray being passed in to IDL is convolved or not. For arrays argpal should be set to either JIDLConst.PARMFLAG_CONVMAJORITY or JIDLConst.PARMFLAG_NO_CONVMAJORITY.

Specified By:

[setProperty](#) in interface [JIDLObjectI](#)

Parameters:

sProperty - the property name

obj - object to be passed to IDL. Should be an object of type JIDLNumber, JIDLObject, JIDLString or JIDLObject.

iPalFlag - flag denoting whether the passed in parameter is convolved or not.
Note: setProperty does not allow obj to be modified by IDL

Throws:

[JIDLException](#) - If IDL encounters an error.

See Also:

[JIDLNumber](#), [JIDLObject](#), [JIDLString](#), [JIDLArray](#),
[JIDLConst.PARMFLAG_CONVMAJORITY](#),
[JIDLConst.PARMFLAG_NO_CONVMAJORITY](#)

toString()

```
public java.lang.String toString()
```

Returns a string representation of the object.

Overrides:

toString in class Object

JIDLObjectI

Declaration

```
public interface JIDLObjectI
```

All Known Implementing Classes:

[JIDLObject](#), [JIDLCanvas](#)

Description

The interface that wrapped IDL objects must implement. Both non-drawable and drawable IDL objects implement this interface.

In many of the methods of this class, one or more flags are required to be specified for parameters being passed to or from the bridge. These flags follow the following guidelines:

For all types of parameters (subclasses of `JIDLNumber`, `JIDLString`, `JIDLObjectI` and `JIDLArray`), a flag should be set that determines whether the parameter is in-only (const) or in-out (we expect it to be changed by IDL). The constants that determine this are either `JIDLConst.PARMFLAG_CONST` or `JIDLConst.PARMFLAG_IN_OUT`.

For parameters that are arrays, a flag should be set that tells the bridge whether the array is to be convolved when passed to IDL. If the `PARM_IN_OUT` flag is set, this flag will also tell the bridge whether to convolve the array when it is copied back to Java. The constants that determine this are either `JIDLConst.PARMFLAG_CONVMAJORITY` or `JIDLConst.PARMFLAG_NO_CONVMAJORITY`.

For example, if the parameter in question is an array that is to be modified by IDL (in-out) and needs to be convolved when passed to and from IDL, we would set its argpal array member as follows:

```
argpal[2] = JIDLConst.PARMFLAG_IN_OUT | JIDLConst.PARMFLAG_CONV  
MAJORITY;
```

Member Summary	
Methods	
void	<code>abort()</code> Requests that the IDL process containing the underlying IDL object abort its current activity.
void	<code>addIDLNotifyListener(JIDLNotifyListener listener)</code> Adds the specified IDL notify listener to receive IDL notification events on this object.
void	<code>addIDLOutputListener(JIDLOutputListener listener)</code> Adds the specified IDL output listener to receive IDL output events on this object.
java.lang.Object	<code>callFunction(java.lang.String sMethodName, int argc, java.lang.Object argv, int[] argpal, int iPalFlag)</code> Call IDL function.
void	<code>callProcedure(java.lang.String sMethodName, int argc, java.lang.Object argv, int[] argpal)</code> Call IDL procedure.
void	<code>createObject(int argc, java.lang.Object argv, int[] argpal, com.idl.idljava.JIDLProcessInitializer initializer)</code> Creates the underlying IDL object.
void	<code>destroyObject()</code> Destroys the underlying IDL object associated with the wrapper.
void	<code>executeString(java.lang.String sCmd)</code> Execute the given command string in IDL.
long	<code>getCookie()</code> Internal use.
java.lang.String	<code>getIDLObjectClassName()</code> Retrieves the IDL object class name of the underlying IDL object.

Member Summary	
java.lang.String	<code>getIDLObjectVariableName()</code> When the underlying IDL object was created in the IDL process, it was assigned a variable name.
java.lang.Object	<code>getIDLVariable(java.lang.String sVar)</code> Given a variable name, return the IDL variable.
java.lang.String	<code>getProcessName()</code> Returns the name of the process that contains the underlying IDL object.
java.lang.Object	<code>getProperty(java.lang.String sProperty, int iPalFlag)</code> Call IDL <code>getProperty</code> method to get named property.
void	<code>initListeners()</code> Initialize any listeners.
boolean	<code>isObjectCreated()</code> Determine if object has been created successfully.
boolean	<code>isObjectDisplayable()</code>
void	<code>removeIDLNotifyListener(JIDLNotifyListener listener)</code> Removes the specified IDL notify listener so it no longer receives IDL notifications.
void	<code>removeIDLOutputListener(JIDLOutputListener listener)</code> Removes the specified IDL output listener on this object.
void	<code>setIDLVariable(java.lang.String sVar, java.lang.Object obj)</code> Set/Create an IDL variable of the given name and value.
void	<code>setProcessName(java.lang.String process)</code> Set the process name that the object will be created in.
void	<code>setProperty(java.lang.String sProperty, java.lang.Object obj, int iPalFlag)</code> Call IDL <code>setProperty</code> method to set named property.

Methods

abort()

```
public void abort()
```

Requests that the IDL process containing the underlying IDL object abort its current activity.

This is only a request and IDL may take a long time before it actually stops.

The client can only Abort the current IDL activity if that wrapper object is the current “owner” of the underlying IDL.

Throws:

[JIDLExceptions](#) - If IDL encounters an error.

See Also:

[JIDLAbortedException](#)

addIDLNotifyListener(JIDLNotifyListener)

```
public void  
addIDLNotifyListener(com.idl.javaidl.JIDLNotifyListener listener)
```

Adds the specified IDL notify listener to receive IDL notification events on this object.

Note that registering/unregistering for events should happen in the `initListeners` method or AFTER the `createObject` method.

Parameters:

`listener` - the listener

addIDLOutputListener(JIDLOutputListener)

```
public void  
addIDLOutputListener(com.idl.javaidl.JIDLOutputListener listener)
```

Adds the specified IDL output listener to receive IDL output events on this object.

Note that registering/unregistering for events should happen in the `initListeners` method or AFTER the `createObject` method.

Parameters:

`listener` - the listener

callFunction(String, int, Object[], int[], int)

```
public java.lang.Object callFunction(java.lang.String
    sMethodName, int argc, java.lang.Object[] argv, int[] argpal,
    int iPalFlag)
```

Call IDL function.

The argpal parameter is an array of flags created by OR-ing constants from class JIDLConst. Each array element corresponds to the equivalent parameter in argv.

Parameters:

sMethodName - the procedure name

argc - the number of parameters

argv - array of Objects to be passed to IDL. This array should be of length argc and should contain objects of type JIDLNumber, JIDLObject, JIDLString or JIDLObject.

argpal - array of flags denoting whether each argv parameter passed to be bridge is 1) in-out vs constant; or 2) a convolved or non-convolved array. This array should be of length argc.

iPalFlag - a flag determining whether a returned array is convolved or not. If the returned value is not an array, this value is zero.

Returns:

an Object of type JIDLNumber, JIDLString, JIDLObjectI or JIDLArray. The caller must know the type of the Object being returned and cast it to its proper type.

Throws:

[JIDLException](#) - If IDL encounters an error.

See Also:

[JIDLNumber](#), [JIDLObject](#), [JIDLString](#), [JIDLArray](#),
[JIDLConst.PARMFLAG_CONST](#), [JIDLConst.PARMFLAG_IN_OUT](#),
[JIDLConst.PARMFLAG_CONVMAJORITY](#),
[JIDLConst.PARMFLAG_NO_CONVMAJORITY](#)

callProcedure(String, int, Object[], int[])

```
public void callProcedure(java.lang.String sMethodName,
    int argc, java.lang.Object[] argv, int[] argpal)
```

Call IDL procedure.

The `argpal` parameter is an array of flags created by OR-ing constants from class `JIDLConst`. Each array element corresponds to the equivalent parameter in `argv`.

Parameters:

`sMethodName` - the procedure name

`argc` - the number of parameters

`argv` - array of Objects to be passed to IDL. This array should be of length `argc` and should contain objects of type `JIDLNumber`, `JIDLObject`, `JIDLString` or `JIDLArray`.

`argpal` - array of flags denoting whether each `argv` parameter passed to be bridge is 1) in-out vs constant; or 2) a convolved or non-convolved array This array should be of length `argc`.

Throws:

[JIDLException](#) - If IDL encounters an error.

See Also:

[JIDLNumber](#), [JIDLObject](#), [JIDLString](#), [JIDLArray](#),
[JIDLConst.PARMFLAG_CONST](#), [JIDLConst.PARMFLAG_IN_OUT](#),
[JIDLConst.PARMFLAG_CONVMAJORITY](#),
[JIDLConst.PARMFLAG_NO_CONVMAJORITY](#)

createObject(int, Object[], int[], JIDLProcessInitializer)

```
public void createObject(int argc, java.lang.Object[] argv,
    int[] argpal, com.idl.javaidl.JIDLProcessInitializer initializer)
```

Creates the underlying IDL object. The `argc`, `argv`, `argpal` parameters are used to supply parameters to the underlying IDL object's `::Init` method. If the `::Init` method does not have any parameters, the caller sets `argc`, `argv`, and `argpal` to 0, null, and null, respectively. `createObject` does the following:

- Calls `::Init` method in the IDL object
- Calls the superclass `initListeners` method to initialize any event handlers. The `initListeners` method has default behavior, which is different for graphical and non-graphical objects. If the default behavior is not desired, a sub-class to modify the listener initialization may override the `initListeners` method.

Parameters:

`argc` - the number of parameters to be passed to INIT

`argv` - array of Objects to be passed to IDL. This array should be of length `argc` and should contain objects of type `JIDLNumber`, `JIDLObject`, `JIDLString` or `JIDLArray`.

`argpal` - array of flags denoting whether each `argv` parameter that is of type array should be convolved or not. For parameters that are not arrays, the value within the array will always be 0.

`initializer` - a `JIDLProcessInitializer` object that specifies IDL process initialization parameters such as the licensing mode to be used.

Throws:

[JIDLException](#) - If IDL encounters an error.

destroyObject()

```
public void destroyObject()
```

Destroys the underlying IDL object associated with the wrapper.

If the object being destroyed is the last object within an OPS process, the OPS process is also destroyed.

Note that this does not destroy the actual wrapper object. Because the wrapper object is a Java object, it follows all the Java reference counting/garbage collection schemes. Once all references to the wrapper object are released from Java code and once the Java Virtual Machine calls the garbage collector, the wrapper object may be deleted from memory.

executeString(String)

```
public void executeString(java.lang.String sCmd)
```

Execute the given command string in IDL.

Parameters:

`sCmd` - the single-line command to execute in IDL.

Throws:

[JIDLException](#) - If IDL encounters an error.

getCookie()

```
public long getCookie()
```

Internal use.

getIDLObjectClassName()

```
public java.lang.String getIDLObjectClassName()
```

Retrieves the IDL object class name of the underlying IDL object.

Returns:

the IDL object class name

getIDLObjectVariableName()

```
public java.lang.String getIDLObjectVariableName()
```

When the underlying IDL object was created in the IDL process, it was assigned a variable name. This method retrieves that name.

Returns:

the variable name

getIDLVariable(String)

```
public java.lang.Object getIDLVariable(java.lang.String sVar)
```

Given a variable name, return the IDL variable.

Note that in the case of arrays, the array will ALWAYS be convolved when passed between Java and IDL.

Parameters:

sVar - The IDL variable name

Returns:

an Object of type JIDLNumber, JIDLString, JIDLObject or JIDLArray. The caller must know the type of the Object being returned and cast it to its proper type. May also return null.

Throws:

[JIDLException](#) - If IDL encounters an error.

getProcessName()

```
public java.lang.String getProcessName()
```

Returns the name of the process that contains the underlying IDL object. For an in-process object, returns an empty string.

Returns:

process name. EMpty string if the process is in-process.

getProperty(String, int)

```
public java.lang.Object getProperty(java.lang.String  
sProperty, int iPalFlag)
```

Call IDL `getProperty` method to get named property.

Parameters:

`sProperty` - the property name

`iPalFlag` - a flag determining whether a returned array will be convolved or not. If the returned value is not is ignored.

Returns:

an Object of type `JIDLNumber`, `JIDLString`, `JIDLObject` or `JIDLArray`. The caller must know the type of the Object being returned and cast it to its proper type. May also return null.

Throws:

[JIDLException](#) - If IDL encounters an error.

See Also:

[JIDLNumber](#), [JIDLObjectI](#), [JIDLString](#), [JIDLArray](#),
[JIDLConst.PARMFLAG_CONVMAJORITY](#),
[JIDLConst.PARMFLAG_NO_CONVMAJORITY](#)

initListeners()

```
public void initListeners()
```

Initialize any listeners.

This method is always called by the `JIDLObject` and `JIDLCanvas` `createObject` methods.

The method may be overridden by sub-classes to initialize a different set of listeners (or none at all).

isObjectCreated()

```
public boolean isObjectCreated()
```

Determine if object has been created successfully.

Returns:

true if object created successfully, or false if object not created or creation was unsuccessful.

isObjectDisplayable()

```
public boolean isObjectDisplayable()
```

removeIDLNotifyListener(JIDLNotifyListener)

```
public void  
removeIDLNotifyListener(com.idl.javaidl.JIDLNotifyListener  
listener)
```

Removes the specified IDL notify listener so it no longer receives IDL notifications.

Note that registering/unregistering for events should happen in the `initListeners` method or AFTER the `createObject` method.

Parameters:

listener - the listener

removeIDLOutputListener(JIDLOutputListener)

```
public void  
removeIDLOutputListener(com.idl.javaidl.JIDLOutputListener  
listener)
```

Removes the specified IDL output listener on this object.

Note that registering/unregistering for events should happen in the `initListeners` method or AFTER the `createObject` method.

Parameters:

listener - the listener

setIDLVariable(String, Object)

```
public void setIDLVariable(java.lang.String sVar,  
java.lang.Object obj)
```

Set/Create an IDL variable of the given name and value.

Note that in the case of arrays, the array will ALWAYS be convolved when passed between Java and IDL.

Parameters:

sVar - the IDL variable name

obj - object to be passed to IDL. Should be an object of type JIDLNumber, JIDLObject, JIDLString or JIDLArray.

Throws:

[JIDLException](#) - If IDL encounters an error.

setProcessName(String)

```
public void setProcessName(java.lang.String process)
```

Set the process name that the object will be created in.

The process name may only be set before createObject is called. If called after the object has been created, this method call does nothing.

Parameters:

process - Process name. Empty String means create in same process (in-process).

setProperty(String, Object, int)

```
public void setProperty(java.lang.String sProperty,  
java.lang.Object obj, int iPalFlag)
```

Call IDL setProperty method to set named property.

The iPalFlag parameter is a set of flags that are or-ed together. Currently this parameter is only used to specify whether a JIDLArray being passed in to IDL is convolved or not. For arrays argpal should be set to either

JIDLConst.PARMFLAG_CONVMAJORITY or
JIDLConst.PARMFLAG_NO_CONVMAJORITY.

Parameters:

sProperty - the property name

obj - object to be passed to IDL. Should be an object of type JIDLNumber, JIDLObject, JIDLString or JIDLObject.

iPalFlag - flag denoting whether the passed in parameter is convolved or not.

Note: setProperty does not allow obj to be modified by IDL

Throws:

[JIDLException](#) - If IDL encounters an error.

See Also:

[JIDLNumber](#), [JIDLObject](#), [JIDLString](#), [JIDLArray](#),
[JIDLConst.PARMFLAG_CONVMAJORITY](#),
[JIDLConst.PARMFLAG_NO_CONVMAJORITY](#)

JIDLOutputListener

Declaration

```
public interface JIDLOutputListener
```

Description

The listener interface for receiving output events from IDL.

Both drawable (JIDLCanvas) and non-drawable (JIDLObject) wrapper objects may be listened to. However by default, JIDLObject and JIDLCanvas objects do NOT listen to their output events.

The class that is interested in receiving IDL output events on a particular object implements this interface. The listener object created from that class is registered with the JIDLObjectI using the addIDLOutputListener method. The listener is unregistered with the removeIDLOutputListener.

See Also:

[JIDLCanvas](#), [JIDLObject](#), [JIDLObjectI](#)

Member Summary	
Methods	
void	IDLoutput(JIDLObjectI obj, java.lang.String s) An IDL output has occurred

Methods

IDLoutput(JIDLObjectI, String)

```
public void IDLoutput(com.idl.javaidl.JIDLObjectI obj,  
    java.lang.String s)
```

An IDL output has occurred

Parameters:

obj - The JIDLObjectI in which the event occurred.

s - The output string

JIDLProcessInitializer

Declaration

```
public class JIDLProcessInitializer
{
    java.lang.Object
    |
    +--com.idl.javaidl.JIDLProcessInitializer
}
```

Description

When a client calls the createObject method of either the [JIDLCanvas](#) or [JIDLObject](#) class, the JIDLProcessInitializer object can be passed in to control IDL process creation. Currently, this object controls the licensing mode of the IDL Process. See “[IDL Licensing Modes](#)” on page 134 for details on the default licensing mechanism used when no JIDLProcessInitializer is specified.

Member Summary	
Fields	
static int	LICENSING_FULL The IDL process requires a full license.
static int	LICENSING_LICENSED_SAV The IDL process launches a SAVE file with an embedded license.
static int	LICENSING_RUNTIME The IDL process requires a runtime license.
static int	LICENSING_VM The IDL process runs in Virtual Machine mode.
Constructors	
	JIDLProcessInitializer() Construct a process initializer object.
	JIDLProcessInitializer(int) Construct a process initializer object.

Member Summary	
Methods	
int	<code>getLicenseMode()</code> Retrieve the current licensing mode.
void	<code>setLicenseMode(int)</code> Set the licensing mode.

Inherited Member Summary
Methods inherited from class <code>Object</code>
<code>equals(Object)</code> , <code>getClass()</code> , <code>hashCode()</code> , <code>notify()</code> , <code>notifyAll()</code> , <code>toString()</code> , <code>wait(long, int)</code> , <code>wait(long, int)</code> , <code>wait(long, int)</code>

Fields

LICENSING_FULL

```
public static final int LICENSING_FULL
```

If this flag is set, the Java application requires that a licensed copy of IDL be installed on the local machine. If IDL is installed but no license is available, the application will run in IDL Demo (7-minute) mode.

LICENSING_LICENSED_SAV

```
public static final int LICENSING_LICENSED_SAV
```

If this flag is set, the Java application looks for an embedded license in the save file being restored.

LICENSING_RUNTIME

```
public static final int LICENSING_RUNTIME
```

If this flag is set, the Java application looks for a runtime IDL license. If no runtime license is available, the application will run in Virtual Machine mode.

LICENSING_VM

```
public static final int LICENSING_VM
```

If this flag is set, the Java application will run in Virtual Machine mode.

Constructors

JIDLProcessInitializer()

```
public JIDLProcessInitializer()
```

JIDLProcessInitializer(int)

```
public JIDLProcessInitializer(int licenseMode)
```

Methods

getLicenseMode()

```
public int getLicenseMode()
```

Returns:

The current licensing mode.

setLicenseMode(int)

```
public void setLicenseMode(int licenseMode)
```

JIDLShort

Declaration

```
public class JIDLShort implements JIDLNumber,  
    java.io.Serializable  
  
    java.lang.Object  
    |  
    +--com.idl.javaidl.JIDLShort
```

All Implemented Interfaces:

[JIDLNumber](#), [java.io.Serializable](#)

Description

The JIDLShort class wraps a short as a mutable object usable by the Java-IDL Export bridge.

Member Summary	
Constructors	
	JIDLShort(JIDLNumber value) Construct a wrapper object.
	JIDLShort(short value) Construct a wrapper object.
Methods	
boolean	booleanValue() Return the value of the wrapped primitive.
byte	byteValue() Return the value of the wrapped primitive
char	charValue() Return the value of the wrapped primitive
double	doubleValue() Return the value of the wrapped primitive
float	floatValue() Return the value of the wrapped primitive

Member Summary	
int	<code>intValue()</code> Return the value of the wrapped primitive
long	<code>longValue()</code> Return the value of the wrapped primitive
void	<code>setValue(JIDLNumber value)</code> Change the value of the wrapper object
void	<code>setValue(short value)</code> Change the value of the wrapper object
short	<code>shortValue()</code> Return the value of the wrapped primitive
java.lang.String	<code>toString()</code>

Inherited Member Summary
Methods inherited from class Object
<code>equals(Object)</code> , <code>getClass()</code> , <code>hashCode()</code> , <code>notify()</code> , <code>notifyAll()</code> , <code>wait(long, int)</code> , <code>wait(long, int)</code> , <code>wait(long, int)</code>

Constructors

JIDLShort(JIDLNumber)

```
public JIDLShort (com.idl.javaidl.JIDLNumber value)
```

Construct a wrapper object.

Parameters:

value - JIDLNumber to wrap for use in the export bridge

JIDLShort(short)

```
public JIDLShort (short value)
```

Construct a wrapper object.

Parameters:

value - value to wrap for use in the export bridge

Methods

booleanValue()

```
public boolean booleanValue()
```

Return the value of the wrapped primitive.

Specified By:

[booleanValue](#) in interface [JIDLNumber](#)

Returns:

true if non-zero, false otherwise

byteValue()

```
public byte byteValue()
```

Return the value of the wrapped primitive

Specified By:

[byteValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

charValue()

```
public char charValue()
```

Return the value of the wrapped primitive

Specified By:

[charValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

doubleValue()

```
public double doubleValue()
```

Return the value of the wrapped primitive

Specified By:

[doubleValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

floatValue()

```
public float floatValue()
```

Return the value of the wrapped primitive

Specified By:

[floatValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

intValue()

```
public int intValue()
```

Return the value of the wrapped primitive

Specified By:

[intValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

longValue()

```
public long longValue()
```

Return the value of the wrapped primitive

Specified By:

[longValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

setValue(JIDLNumber)

```
public void setValue(com.idl.javaidl.JIDLNumber value)
```

Change the value of the wrapper object

Specified By:

[setValue](#) in interface [JIDLNumber](#)

Parameters:

`value` - JIDLNumber to wrap for use in the export bridge

setValue(short)

```
public void setValue(short value)
```

Change the value of the wrapper object

Parameters:

`value` - primitive value to wrap for use in the export bridge

shortValue()

```
public short shortValue()
```

Return the value of the wrapped primitive

Specified By:

[shortValue](#) in interface [JIDLNumber](#)

Returns:

value that is wrapped by this object

toString()

```
public java.lang.String toString()
```

Overrides:

[toString](#) in class [Object](#)

JIDLString

Declaration

```
public class JIDLString implements java.io.Serializable
    java.lang.Object
    |
    +---com.idl.javaidl.JIDLString
```

All Implemented Interfaces:

```
java.io.Serializable
```

Description

The JIDLString class wraps a String as a mutable object usable by the Java-IDL Export bridge.

Member Summary	
Constructors	
	<code>JIDLString(JIDLString value)</code> Construct a wrapper object.
	<code>JIDLString(java.lang.String value)</code> Construct a wrapper object.
Methods	
void	<code>setValue(JIDLString value)</code> Change the value of the wrapper object
void	<code>setValue(java.lang.String value)</code> Change the value of the wrapper object
java.lang.String	<code>stringValue()</code> Return the value of the wrapped primitive
java.lang.String	<code>toString()</code>

Inherited Member Summary
Methods inherited from class <code>Object</code>
<code>equals(Object), getClass(), hashCode(), notify(), notifyAll(), wait(long, int), wait(long, int), wait(long, int)</code>

Constructors

JIDLString(JIDLString)

```
public JIDLString(com.idl.javaidl.JIDLString value)
```

Construct a wrapper object.

Parameters:

value - value to wrap for use in the export bridge

JIDLString(String)

```
public JIDLString(java.lang.String value)
```

Construct a wrapper object.

Parameters:

value - value to wrap for use in the export bridge

Methods

setValue(JIDLString)

```
public void setValue(com.idl.javaidl.JIDLString value)
```

Change the value of the wrapper object

Parameters:

value - primitive value to wrap for use in the export bridge

setValue(String)

```
public void setValue(java.lang.String value)
```

Change the value of the wrapper object

Parameters:

value - primitive value to wrap for use in the export bridge

stringValue()

```
public java.lang.String stringValue()
```

Return the value of the wrapped primitive

Returns:

value that is wrapped by this object

toString()

```
public java.lang.String toString()
```

Overrides:

toString in class Object



Appendix B

COM Object Creation

The following topics in this appendix show how to create a custom IDL wrapper object (initialized with and without parameters) from several COM programming languages:

Sample IDL Object	472	C# Code Sample	479
Visual Basic .NET Code Sample	475	Visual Basic 6 Code Sample	481
C++ Client Code Sample	477		

Sample IDL Object

The COM CreateObject method creates an instance of an underlying IDL object and calls its Init method with any specified parameters (see [“CreateObject”](#) on page 194 for details). Through this object instance, you have access to the properties and methods of the object as well as the underlying IDL process.

The following samples rely upon an IDL object named `idlexfoo__define.pro` containing the following code:

```
; The Init method expects three parameters:
; a string, a 32-bit long, and an array which has
; 2 rows & 3 columns, containing 32-bit long values.
; The ::Init method can also be called without any parameters.

FUNCTION idlexfoo::Init, parmStr, parmVal, parmArr, _EXTRA=e

  IF (N_ELEMENTS(parmStr) EQ 1) THEN BEGIN
    IF ( SIZE(parmStr,/type) NE 7 ) THEN BEGIN
      PRINT, 'IDLexFoo::Init, parmStr is not a STRING'
      HELP, parmStr
      RETURN, 0
    ENDIF
  ENDIF

  IF (N_ELEMENTS(parmVal) EQ 1) THEN BEGIN
    IF ( (SIZE(parmVal,/type) NE 3) ) THEN BEGIN
      PRINT, 'IDLexFoo::Init, parmVal is not a LONG'
      HELP, parmVal
      RETURN, 0
    ENDIF
  ENDIF

  nElms = N_ELEMENTS(parmArr)
  IF (nElms GT 0) THEN BEGIN
    IF ( (nElms NE 6) OR (size(parmArr,/type) NE 3) ) THEN BEGIN
      PRINT, 'IDLexFoo::Init, parmArr is not a ARR(3,2) of
LONG)'
      HELP, parmArr
      RETURN, 0
    ENDIF
  ENDIF

  RETURN, 1
END
```



```
; Object definition.
PRO idlexfoo__define
    ; Create [col, row] 32-bit long array.
    initArr = LONARR(3,2)
    struct = {idlexfoo, $
        parmStr: '', $
        parmVal: 0L, $
        parmArr: initArr $
    }
END
```

Export the Sample IDL Object

You will need to create the necessary wrapper object files by using the Export Bridge Assistant to generate them. Once you have created the object definition file, `idlexfoo__define.pro`, complete the following steps:

1. Open the Export Bridge Assistant by entering `IDLEXBR_ASSISTANT` at the command line.
2. Select to create a COM export object by selecting **File** → **New Project** → **COM** and browse to select the `idlexfoo__define.pro` file. Click **Open** to load the file into the Export Assistant.

Note

Export Bridge Assistant details are available in [Chapter 7, “Using the Export Bridge Assistant”](#). Refer to that section if you need more information about the following steps.

3. The top-level project entry in the left tree panel is selected by default. There is no need to modify the default properties shown in the right-hand property panel, but you can enter different values if desired. There are no other parameters that need to be defined for this object.

Tree View Item	Parameter Configuration
IDL Export Bridge Project	Accept the default value or make changes as desired: <ul style="list-style-type: none"> • Output classname • Process name • Output directory
helloworldex	Drawable object equals False

Table B-4: Example Export Object Parameters

4. Save the project by selecting **File** → **Save project**. Accept the default name and location or make changes as desired.
5. Build the export object by selecting **Build** → **Build object**. The **Build log** panel shows the results of the build process. For a nondrawable object, .tlb and .dll files (named based on the object name) are created in the **Output directory**.
6. Register the .dll using `regsvr32 idlexfoo.dll`. See [“COM Registration Requirements”](#) on page 143 for details if needed.

See the language-specific section for information on how to create this object in your application:

- [“Visual Basic .NET Code Sample”](#) on page 475
- [“C++ Client Code Sample”](#) on page 477
- [“C# Code Sample”](#) on page 479
- [“Visual Basic 6 Code Sample”](#) on page 481

Visual Basic .NET Code Sample

Within Visual Studio .NET, select **Project** → **Add Reference....** This brings up a dialog. Select the **COM** tab, then **Browse**, and change the path to the `idlexfoo.dll`. This imports the object reference into the project.

Within the project that will use the wrapper object, include the following line at the top of the form:

```
Imports IDLexFooLib
```

Initiation Without Parameters in Visual Basic .NET

Use the following code to initialize the object with no parameters.

```
Private Sub Button1_Click(...)

    Dim oFoo As New IDLexFooClass()

    Try
        oFoo.CreateObject(0, 0, 0)
    Catch ex As Exception
        Debug.WriteLine(oFoo.GetLastError())
    Return
    End Try

    ' use object here...

End Sub
```

Initiation with Parameters in Visual Basic .NET

Use the following code to initialize the object with its three parameters (a string, a 32-bit long value, and an array that has two rows and three columns, containing 32-bit long values).

Inside the Public Class definition for the form and before any subroutines, you must add the following two lines:

```
Const PARMFLAG_CONST As Integer = &H1
Const PARMFLAG_CONV_MAJORITY As Integer = &H4000
```

Then create the object within your program:

```
Private Sub Button1_Click(...)

    Dim oFoo As New IDLexFooClass
```

```
Dim parmStr As String = "I am a string parameter"
Dim parmVal As Int32 = 24
Dim parmArr As Int32(,) = {{10, 11, 12}, {20, 21, 22}}

Dim argc As Int32 = 3
Dim argval As Object() = {parmStr, parmVal, parmArr}
Dim argpal As Int32() = {PARMFLAG_CONST, PARMFLAG_CONST, _
    (PARMFLAG_CONST + PARMFLAG_CONV_MAJORITY)}

Try
    oFoo.CreateObject(argc, argval, argpal)
Catch ex As Exception
    Debug.WriteLine(oFoo.GetLastError())
    Return
End Try

' use object here...

End Sub
```

C++ Client Code Sample

The C++ project must somewhere include the following line, in order to pull in the CoClass and Interface definitions for the wrapper object:

```
#import "IDLexFoo.tlb" no_namespace no_implementation \
    raw_interfaces_only named_guids
```

For details about the object parameters, see [“Sample IDL Object”](#) on page 472.

Initiation Without Parameters in C++

Use the following code to initialize the object with no parameters.

```
CComPtr<IIDLexFoo> spFoo;

if ( FAILED(spFoo.CoCreateInstance(__uuidof(IDLexFoo)) || !spFoo)
)
    return E_FAIL;

CComVariant vtNULL(0);
HRESULT hr = spFoo->CreateObject(0,vtNULL,vtNULL);
if ( FAILED(hr) )
{
    CComBSTR bstrErr;
    spFoo->GetLastError(&bstrErr);
    return E_FAIL;
}
```

Initiation with Parameters in C++

Use the following code to initialize the object with its three parameters (a string, a 32-bit long value, and an array that has two rows and three columns, containing 32-bit long values).

```
CComPtr<IIDLexFoo> spFoo;

if ( FAILED(spFoo.CoCreateInstance(__uuidof(IDLexFoo)) || !spFoo)
)
    return E_FAIL;

CComSafeArrayBound bound[2];
bound[0].SetLowerBound(0); bound[0].SetCount(2); // two rows
bound[1].SetLowerBound(0); bound[1].SetCount(3); // three cols

CComSafeArray<VARIANT> parmArr(bound,2);
```

```

long    ndx[2];
long lData[2][3] = { {10, 11, 12}, {20, 21, 22} };

for ( int i = 0; i < 2; i++ ) {          // row
    for ( int j = 0; j < 3; j++ ) { // col
        ndx[0] = i; ndx[1] = j;
        parmArr.MultiDimSetAt(ndx, CComVariant(lData[i][j]));
    }
}

CComBSTR    parmStr = "I am a string parameter";
CComVariant parmVal = (long)24;

CComSafeArray<VARIANT> argval(3);
CComSafeArray<long>    argpal(3);

argval[0] = parmStr;    argpal[0] = IDLBML_PARMFLAG_CONST;
argval[1] = parmVal;    argpal[1] = IDLBML_PARMFLAG_CONST;
argval[2] = parmArr;    argpal[2] =
    IDLBML_PARMFLAG_CONST | IDLBML_PARMFLAG_CONVMAJORITY;

long argc = 3;
CComVariant vargval = argval;
CComVariant vargpal = argpal;

HRESULT hr = spFoo->CreateObject(argc, vargval, vargpal);
if ( FAILED(hr) )
{
    CComBSTR bstrErr;
    spFoo->GetLastError(&bstrErr);
    return E_FAIL;
}

```

C# Code Sample

Within Visual Studio .NET, in the Solution Explorer window, underneath the project that will use the wrapper object, right-click on the **References** item, then select **Add Reference....** This brings up a dialog. Select the **COM** tab, then **Browse**, and change the path to the wrapper .dll. This imports the object reference into the project.

Then, within the project that will use the wrapper object, include the following line at the top, outside of the namespace for the class:

```
using IDLexFooLib;
```

Initiation Without Parameters in C#

Use the following code to initialize the object with no parameters.

```
private void button1_Click(...)
{
    IDLexFooClass oFoo = new IDLexFooClass();

    try {
        oFoo.CreateObject(0, 0, 0);
    }
    catch {
        Debug.WriteLine(oFoo.GetLastError());
        return;
    }

    // Use object here...
}
```

Initiation with Parameters in C#

Use the following code to initialize the object with its three parameters (a string, a 32-bit long value, and an array that has two rows and three columns, containing 32-bit long values).

```
private void button1_Click(...)
{
    const int PARMFLAG_CONST          = 0x0001;
    const int PARMFLAG_CONV_MAJORITY = 0x4000;

    IDLexFooClass oFoo = new IDLexFooClass();

    string parmStr = "I am a string parameter";
    int     parmVal = 24;
```

```
int[,] parmArr = {{10, 11, 12}, {20, 21, 22}};

int      argc = 3;
object[] argval = {parmStr, parmVal, parmArr};
int[]     argpal = {PARMFLAG_CONST, PARMFLAG_CONST,
    PARMFLAG_CONST | PARMFLAG_CONV_MAJORITY};

try {
    oFoo.CreateObject(argc, argval, argpal);
}
catch {
    Debug.WriteLine(oFoo.GetLastError());
    return;
}

// Use object here...
}
```


Visual Basic 6 Code Sample

Within Visual Basic 6, select **Project** → **Components**, then Browse for the .dll of the wrapper object in order to include the objects definition in the project.

For details about the object parameters, see “[Sample IDL Object](#)” on page 472.

Initiation Without Parameters in Visual Basic 6

Use the following code to initialize the object with no parameters.

```
Private Sub MyRoutine

    Dim oFoo As IDLexFoo
    Set oFoo = New IDLexFoo

    On Error GoTo ErrorHandler

    oFoo.CreateObject 0, 0, 0
    ' use object here...
    Return

ErrorHandler:
    If Not oFoo Is Nothing Then
        Debug.Print oFoo.GetLastError
    End If

End Sub
```

Initiation with Parameters in Visual Basic 6

Use the following code to initialize the object with its three parameters (a string, a 32-bit long value, and an array which has two rows and three columns, containing 32-bit long values).

```
Const PARMFLAG_CONST As Integer = &H1
Const PARMFLAG_CONV_MAJORITY As Integer = &H4000

Private Sub MyRoutine

    Dim oFoo As IDLexFoo

    Dim parmStr As String
    Dim parmVal As Long
    Dim parmArr(1, 2) As Long

    Dim argc As Long
```

```

Dim argv(2) As Variant
Dim argpal(2) As Long

parmStr = "I am a string parameter"
parmVal = 24
parmArr(0, 0) = 10: parmArr(0, 1) = 11: parmArr(0, 2) = 12
parmArr(1, 0) = 20: parmArr(1, 1) = 21: parmArr(1, 2) = 22

argc = 3
argv(0) = parmStr: argpal(0) = PARMFLAG_CONST
argv(1) = parmVal: argpal(1) = PARMFLAG_CONST
argv(2) = parmArr: argpal(2) = PARMFLAG_CONST + _
    PARMFLAG_CONV_MAJORITY

Set oFoo = New IDLexFoo

On Error GoTo ErrorHandler

oFoo.CreateObject argc, argv, argpal
    ' use object here...
Return

ErrorHandler:
    If Not oFoo Is Nothing Then
        Debug.Print oFoo.GetLastError
    End If

End Sub

```



Appendix C

Java Object Creation

The following topics in this appendix show how to create a custom IDL wrapper object (initialized with and without parameters) in Java:

Sample IDL Object	484	Java Object Initiation with Parameters ...	489
Java Object Initiation Without Parameters	487		

Sample IDL Object

The Java `createObject` method creates an instance of an underlying IDL object and calls its `Init` method with any specified parameters (see “[createObject](#)” on page 220 for details). Through this object instance, you have access to the properties and methods of the object as well as the underlying IDL process.

The following samples rely upon an IDL object contained in file named `idlexfoo__define.pro`. This file must be in the IDL path and needs to contain the following code:

```
; The Init method expects three parameters:
; a string, a 32-bit long, and an array which has
; 2 rows & 3 columns, containing 32-bit long values.
; The ::Init method can also be called without any parameters.

FUNCTION idlexfoo::Init, parmStr, parmVal, parmArr, _EXTRA=e

  IF (N_ELEMENTS(parmStr) EQ 1) THEN BEGIN
    IF ( SIZE(parmStr,/type) NE 7 ) THEN BEGIN
      PRINT, 'IDLexFoo::Init, parmStr is not a STRING'
      HELP, parmStr
      RETURN, 0
    ENDIF
  ENDIF

  IF (N_ELEMENTS(parmVal) EQ 1) THEN BEGIN
    IF ( (SIZE(parmVal,/type) NE 3) ) THEN BEGIN
      PRINT, 'IDLexFoo::Init, parmVal is not a LONG'
      HELP, parmVal
      RETURN, 0
    ENDIF
  ENDIF

  nElms = N_ELEMENTS(parmArr)
  IF (nElms GT 0) THEN BEGIN
    IF ( (nElms NE 6) OR (size(parmArr,/type) NE 3) ) THEN BEGIN
      PRINT, 'IDLexFoo::Init, parmArr is not a ARR(3,2) ' $
        + 'of LONG)'
      HELP, parmArr
      RETURN, 0
    ENDIF
  ENDIF

  RETURN, 1
END
```

```
; Object definition.
PRO idlexfoo__define
    ; Create [col, row] 32-bit long array.
    initArr = LONARR(3,2)
    struct = {idlexfoo, $
        parmStr: '', $
        parmVal: 0L, $
        parmArr: initArr $
    }
END
```

Export the Sample IDL Object

You will need to create the necessary wrapper object files by using the Export Bridge Assistant to generate them. Once you have created the object definition file, `idlexfoo__define.pro`, complete the following steps:

1. Open the Export Bridge Assistant by entering `IDLEXBR_ASSISTANT` at the command line.
2. Select to create a Java export object by selecting **File** → **New Project** → **Java** and browse to select the `idlexfoo__define.pro` file. Click **Open** to load the file into the Export Assistant.

Note

Export Bridge Assistant details are available in [Chapter 7, “Using the Export Bridge Assistant”](#). Refer to that section if you need more information about the following steps.

3. The top-level project entry in the left tree panel is selected by default. There is no need to modify the default properties shown in the right-hand property panel, but you can enter different values if desired. There are no other parameters that need to be defined for this object.

Tree View Item	Parameter Configuration
IDL Export Bridge Project	Accept the default value or make changes as desired: <ul style="list-style-type: none"> • Output classname • Process name • Output directory • Package name
idlexfoo	Drawable object equals False

Table C-5: Example Export Object Parameters

4. Save the project by selecting **File** → **Save project**. Accept the default name and location or make changes as desired.
5. Build the export object by selecting **Build** → **Build object**. The **Build log** panel shows the results of the build process. A subdirectory, named `idlexfoo` (based on the object name), contains the `.java` and `.class` files, and is located in the **Output directory**.

See the following for information on how to create this object in your application:

- [“Java Object Initiation Without Parameters”](#) on page 487
- [“Java Object Initiation with Parameters”](#) on page 489

Note on Running the Java Examples

Examples in this appendix provide Windows-style compile `javac` (compile) and `java` (run) commands. If you are running on a platform other than Windows, use your platform’s path and directory separators and see [“Java Requirements”](#) on page 143 for information about the `bridge_setup` file, which sets additional information.

Java Object Initiation Without Parameters

To initialize an instance of the newly created wrapper object (based on the IDL object described in “[Sample IDL Object](#)” on page 484) using [createObject](#), complete the following steps:

1. Create a Java file named `idlexfoo_example.java` and save it in the **Export directory** created by the Assistant. Include the following lines of code in the file:

```
// Reference the default package generated by the Assistant.
package idlexfoo;

// Reference the javaidl export bridge classes.
import com.idl.javaidl.*;

//Create main class, subclassing from object created by
//Bridge Assistant. You can either subclass or create a
//member variable of the object.
public class idlexfoo_example extends idlexfoo
implements JIDLOutputListener
{
    //Create a variable referencing the exported object
    private idlexfoo fooObj;

    // Constructor.
    public idlexfoo_example() {

        // Create the wrapper object
        fooObj = new idlexfoo();

        // Add output listener to access IDL output.
        fooObj.addIDLOutputListener(this);

        // Create the underlying IDL object and call
        // its ::Init method with parameters
        fooObj.createObject( );
        fooObj.executeString("PRINT, 'Created object'");
    }

    // Implement JIDLOutputListener
    public void IDLoutput(JIDLObjectI obj, String sMessage) {
        System.out.println("IDL: "+sMessage);
    }

    //Instantiate a member of the class.
    public static void main(String[] argv) {
```

```
        idlexfoo_example exampleObj =  
            new idlexfoo_example();  
    }  
}
```

2. Open the Windows Command window by selecting **Start** → **Run** and enter `cmd` in the textbox.
3. Use the `cd` command to change to the directory containing the `idlexfoo` directory.
4. Reference the classpath of `javaidl.b.jar` in the compile statement. Enter the following two commands (as single lines) to compile and execute the program, replacing `<IDL_DIR>` with the IDL installation directory:

```
javac -classpath  
    ".;IDL_DIR\resource\bridges\export\java\javaidl.b.jar"  
    idlexfoo\idlexfoo_example.java  
java -classpath  
    ".;IDL_DIR\resource\bridges\export\java\javaidl.b.jar"  
    idlexfoo.idlexfoo_example
```

Tip

See [“Note on Running the Java Examples”](#) on page 486 for information on non-Windows-style compile and execution commands.

After compiling and running the project, the output message will appear in the command window.

Java Object Initiation with Parameters

Use the following code to initialize the newly created Java wrapper object (based on the IDL object described in “[Sample IDL Object](#)” on page 484) with its three parameters:

- A string
- A 32-bit long value
- An array that has two rows and three columns, containing 32-bit long values

See [createObject](#) for more information about object parameters. See [Appendix A, “IDL Java Object API”](#) for information on JIDL* objects.

1. Create a Java file named `idlexfoo_example.java` and save it in the **Export directory** created by the Assistant. Include the following lines of code in the file:

```
// Reference the default package generated by the Assistant.
package idlexfoo;

// Reference the javaidl export bridge classes.
import com.idl.javaidl.*;

//Create main class, subclassing from object created by
//Bridge Assistant. You can either subclass or create a
//member variable of the object.
public class idlexfoo_example extends idlexfoo
implements JIDLOutputListener
{
    //Create a variable referencing the exported object
    private idlexfoo fooObj;

    // Constructor.
    public idlexfoo_example() {

        // These are the parameters we want to pass to
        // the ::Init method
        String str = "I am a string parameter";
        int var = 24;
        int[][] array = {{10, 11, 12}, {20, 21, 22}};

        // Wrap the Java types using Export Bridge data types
        JIDLString parmStr = new JIDLString(str);
        JIDLInteger parmVar = new JIDLInteger(var);
        JIDLArray parmArray = new JIDLArray(array);
```

```

// Create the wrapper object
fooObj = new idlexfoo();

// Set up parameters to pass to createObject
final int ARGC = 3;
Object[] argv = new Object[ARGC];
int[] argp = new int[ARGC];

// NOTE: JIDLConst.PARMFLAG_CONST indicates
// "in-only" parameter
argv[0] = parmStr;
argp[0] = JIDLConst.PARMFLAG_CONST; //
argv[1] = parmVar;
argp[1] = JIDLConst.PARMFLAG_CONST;
argv[2] = parmArray;
argp[2] = JIDLConst.PARMFLAG_CONST;

// Add output listener to access IDL output.
fooObj.addIDLOutputListener(this);

// Create the underlying IDL object and call
// its ::Init method with parameters
fooObj.createObject(ARGC, argv, argp);
fooObj.executeString("PRINT, 'Created object'");
}

// implement JIDLOutputListener
public void IDLoutput(JIDLObjectI obj, String sMessage) {
    System.out.println("IDL: "+sMessage);
}

//Instantiate a member of the class.
public static void main(String[] argv) {
    idlexfoo_example exampleObj =
        new idlexfoo_example();
}
}

```

2. Open the Windows Command window by selecting **Start** → **Run** and enter `cmd` in the textbox.
3. Use the `cd` command to change to the directory containing the `idlexfoo` directory.

4. Reference the classpath of `javaidl.jar` in the compile statement. Enter the following two commands (as single lines) to compile and execute the program, replacing `IDL_DIR` with the IDL installation directory:

```
javac -classpath
    ".;IDL_DIR\resource\bridges\export\java\javaidl.jar"
    idlexfoo\idlexfoo_example.java
java -classpath
    ".;IDL_DIR\resource\bridges\export\java\javaidl.jar"
    idlexfoo.idlexfoo_example
```

Tip

See [“Note on Running the Java Examples”](#) on page 486 for information on non-Windows-style compile and execution commands.

After compiling and running the project, the output message will appear in the command window.



Appendix D

Multidimensional Array Storage and Access

This appendix discusses the following topics.

Overview	494	Storage and Access in COM and IDL ...	496
Why Storage and Access Matter	495	2D Array Examples	498

Overview

This appendix is designed to explain how multidimensional arrays are stored and accessed, with specific relevance to marshaling arrays between COM clients and IDL.

Please note that if you use the Convert Majority property in the Export Bridge Assistant on exported property or method parameters (described in [“Converting Array Majority”](#) on page 165), you do not have to worry about the information or examples in this appendix. For more information, see [Table 7-8](#) in [“Property Information”](#) on page 171.

A linear, one-dimensional (1D) vector is a contiguous list of items in memory. There is no room for misinterpreting what order the items are stored and accessed. However, moving beyond 1D can introduce contradictory definitions and connotations, depending on the source consulted and the programming language in question.

Accordingly, we will stay away from words of strong and conflicting meaning, such as “column majority” and “row majority.” (You can read [“Columns, Rows, and Array Majority”](#) (Chapter 15, *Application Programming*) for more information on those terms.) What matters more than vocabulary is how multidimensional arrays are stored in physical memory (linear memory) and how they are accessed. For brevity’s sake, we will use two-dimensional arrays (2D) to illustrate storage, and focus on Visual Basic, C++, Win32 APIs, and IDL pro code for how the arrays are accessed.

Note

Java has the same issues as COM with multidimensional array storage and access. You can assume that this appendix addresses both external languages, although it names only COM.

Why Storage and Access Matter

Clients that need to pass an array to IDL need to understand the memory layouts of the arrays in order to know if they should convert arrays from one format to the other. Simply trying to understand which format is “row” and which is “column” major is not enough because the definitions of those terms can differ in context.

Understanding these distinctions are critical when programming in Visual Basic and C++ as each language natively stores arrays differently. However, using the Win32 Safearray APIs, either directly or indirectly through the ATL wrapper classes, allows C++ code to create safe arrays in the same order as Visual Basic. However, C++ has the flexibility to create safe arrays ordered differently, which is useful for testing.

In summary:

- SAFEARRAYs and IDL arrays are stored differently and must be converted to be used by each other
- Multidimensional SAFEARRAYs are stored as “column major” in linear memory (i.e., a column is stored contiguously in memory)
- IDL stores multidimensional arrays as “scanline major” (i.e., stores each scanline contiguously in memory)
- All the Win32 APIs and ATL safe array wrapper classes access SAFEARRAYs in column major
- Visual Basic accesses SAFEARRAYs as “column major”
- Native C++ arrays are stored and accessed as “row major”

Storage and Access in COM and IDL

There is a difference between storage and access. Storage focuses on the way a multidimensional array of items gets arranged in linear memory. Since all memory is linear memory, it is paramount to understand how arrays are arranged in linear memory. Access is the way a language allows interaction with a multidimensional array.

Since we are creating and reading arrays from a computer language, we must understand the language's perspective on the array and how to access it.

Arrays in COM

In order to move an array around within the COM world, it must be described by a `SAFEARRAY` descriptor whose dimensions are defined by `SAFEARRAYBOUND` descriptors.

SAFEARRAY Descriptors

The `SAFEARRAY` descriptor has the following definition:

```
typedef struct SAFEARRAY
{
    USHORT cDims;
    USHORT fFeatures;
    ULONG cbElements;
    ULONG cLocks;
    PVOID pvData;
    SAFEARRAYBOUND rgsabound[ 1 ];
} SAFEARRAY;
```

This structure describes different aspects of the safe array, such as total number of dimensions, `cDims`, flags indicating if the array is fixed and cannot be resized, `fFeatures`, if there are any locks on the array, `cLocks`, and then a pointer to the actual array data itself, `pvData`.

Usually, the `SAFEARRAY` descriptor is wrapped by the OLE Automation data type Variant, and the Variant itself is passed around as the data type in method calls. Either way, an array must be wrapped by a `SAFEARRAY` before it can be marshaled.

SAFEARRAYBOUND Descriptors

A `SAFEARRAY` can have an unlimited number of dimensions, whose dimension count is stored in `cDims`. For each dimension, there must an element of type

SAFEARRAYBOUND, which stores the lower bound and number of elements in the dimension, as given by the structure:

```
typedef struct SAFEARRAYBOUND
{
    ULONG cElements;
    LONG lLbound;
} SAFEARRAYBOUND;
```

The SAFEARRAY descriptor member `rgsabound[]` is an array of SAFEARRAYBOUND elements. (Visual Basic lets you define an element range such as “10 to 20” or “-10 to 10” such that the `lLbound` item on the dimension is not zero, but 10 and -10, respectively. For all of our examples, we assume the lower bound is zero.)

Note that in COM, items are frequently in reverse order than what you would expect, which is the case with the SAFEARRAY descriptor’s `rgsabound[]` member array. You must specify the dimensions in reverse order. For example, if you are constructing an array of 3 rows by 5 columns (3x5), the first SAFEARRAYBOUND array item would have its `cElements` member set to 5, and the second item `rgsabound[]` array item would have its `cElements` member set to 3.

However, you rarely set `rgsabound[]` yourself. All the Win32 API calls to create safe arrays set these values for you, from information specified in the expected order (i.e., 3 and 5). Do be aware that if you look in memory at the actual SAFEARRAY descriptor data, you will see the `rgsabound[]` member array in reverse order.

Arrays in IDL

IDL arrays are stored in “scanline majority,” meaning that each scanline is contiguous in memory. Additionally, the dimensions are listed backwards from standard computer-science notation.

For example, if you want to create an array of bytes with 5 columns and 3 rows, you use the following code:

```
myarr = BYTARR(5,3)
```

Simply put: SAFEARRAYs and IDL arrays are arranged differently in linear memory. Thus, when you create an array in the COM world that you want to give to IDL, you must “convert the majority.” For how to do so in three programming languages, see [“2D Array Examples”](#) on page 498.

2D Array Examples

Let's create a 2D array that has 3 rows by 5 columns (3x5). Since the ultimate goal is to give the array to IDL for processing, let's pretend it is an "image." We will set the first row to all red, the second row to all green, and the third row to all blue. Here's the conceptual layout of our array

```
rrrrr
ggggg
bbbbbb
```

We will see shortly that even though the conceptual 2D layout is the above, the actual layout in linear memory is quite different between SAFEARRAYs and IDL.

Note

In the examples below, the "red" value is really the ASCII character 'r', "green" is the ASCII character 'g', and so on. We use this scheme so when you look at the actual memory, you'll see the letters "rgb", which makes for easy reading. It is much less confusing than using the cardinal numbers 1, 2, 3, when you are also talking about ordinal numbering involving 1, 2, 3.

Note

These examples illustrate how different languages store data. You should not need to include such code in your applications to make them work; the wrapper does the conversion for you.

Visual Basic

Here is how to create the RGB array (matrix) in Visual Basic. This example, by default, creates a valid SAFEARRAY that is compliant with the information above, and stored within a Variant when passed as a parameter in a method call (not shown).

```
Const RED As Byte = 114
Const GREEN As Byte = 103
Const BLUE As Byte = 98
` This creates an array with dimension indices 0..2 & 0..4
` inclusive:
` i.e., it creates a 3x5 array; with "lower bounds" set to 0.
Dim m(2, 4) As Byte
For I = 0 To 4
  m(0, I) = RED
  m(1, I) = GREEN
  m(2, I) = BLUE
Next I
```

Resulting linear memory:

```
rgbrgbrgbrgbrgb
```

Resulting SAFEARRAY.rgsabounds:

```
[0,5], [0,3]
```

Note the reversed order!

C++ Using ATL SAFEARRAY Wrapper Objects

This example uses the ATL Safearray wrapper objects: CComSafeArrayBound and CComSafeArray, which simply wraps the calls to the native Win32 Safearray API calls.

```
CComSafeArrayBound bound[2];
bound[0].SetCount(3); // 3 rows
bound[1].SetCount(5); // 5 columns
CComSafeArray<byte> matx(bound,2);
long ndx[2];
for ( int i = 0; i < 5; i++ )
{
    ndx[0] = 0; ndx[1] = i;
    matx.MultiDimSetAt(ndx, 'r');
    ndx[0] = 1; ndx[1] = i;
    matx.MultiDimSetAt(ndx, 'g');
    ndx[0] = 2; ndx[1] = i;
    matx.MultiDimSetAt(ndx, 'b');
}
```

Resulting linear memory:

```
rgbrgbrgbrgbrgb
```

Resulting SAFEARRAY.rgsabounds:

```
[0,5], [0,3]
```

Observe that when the CComSafeArrayBound array is created, it is initialized in the conceptually correct order (i.e., specifying the “3 rows” by “5 columns”). But, if you look at the actual SAFEARRAY.rgsabounds[] element in memory, you see that they were reversed when the array was created.

C++ Using SAFEARRAY API Calls and Creating Different Memory Layout

C++ has the flexibility to create SAFEARRAYs in many different ways. By calling the SAFEARRAY API calls directly and judiciously, you can create a SAFEARRAY

with data in a different order than what is normally expected. IDL and traditional SAFEARRAY data ordering are different. This example puts the data into the SAFEARRAY in the same order as IDL expects it. In other words, it puts the data in the *opposite* order that is used for SAFEARRAYs when you use the API calls to set individual data elements.

But first, we must step back and see how the C++ language stores multidimensional arrays. If you have the following declaration:

```
byte data[3][5] = {
    'r','r','r','r','r',
    'g','g','g','g','g',
    'b','b','b','b','b' };
```

the resulting linear memory looks like this:

```
rrrrrrgggggbbbbbb
```

This is the same order that IDL expects. However, C++ accesses the memory in the opposite way that IDL would access the same data. For example, if you wanted to set the k^{th} element of the first row (0-indexed), here's how the two languages compare:

C++:

```
data[0][k] = value;
```

IDL:

```
Data[k,0] = value
```

However, the resulting linear memory layout is the same.

This example creates the 2D RGB array in C++ using the SAFEARRAY API calls and arranging memory in the same layout as IDL.

```
// First, create the linear memory in the format: rrrrrgggggbbbbbb
byte data[3][5];
for ( int i = 0; i < 5; i++ )
{
    data[0][i] = 'r';
    data[1][i] = 'g';
    data[2][i] = 'b';
}
SAFEARRAYBOUND sab[2];
sab[0].lLbound = 0;
sab[0].cElements = 3; // 3 rows
sab[1].lLbound = 0;
sab[1].cElements = 5; // 5 columns
SAFEARRAY* psa = SafeArrayCreateEx(VT_UI1, 2, sab, NULL);
// By copying the source data into the safearray data area,
// we can create the data in a different order. Since the
```

```
// source data is in the same order as IDL expects, this creates
// a SAFEARRAY with a non-standard ordering.
//
memcpy(psa->pvData, data, sizeof(data));
```

Resulting linear memory:

```
rrrrrgggggbbbb
```

Resulting `SAFEARRAY.rgsabounds`:

```
[0,5], [0,3]
```

The consumer of this array needs some indication that the order is different than standard `SAFEARRAY`s and that it would not need to be converted before passing off to IDL.

Here is how to create the 2D RGB array in IDL pro code:

```
arr = BYTARR(5, 3)
for i=0,4 do begin
    arr[i,0] = 114B
    arr[i,1] = 103B
    arr[i,2] = 98B
endfor
```

Resulting linear memory:

```
rrrrrgggggbbbb
```

Calling `help, arr` gives the following information:

```
ARR BYTE = Array[5, 3]
```




Index

A

abort method

COM connector, [193](#)

Java connector, [219](#)

ActiveX controls

class ID, [52](#)

destroying, [60](#)

example IDL code, [61](#), [65](#)

IDL object wrapper, [191](#)

IDLcomActiveX object references, [55](#)

inserting into IDL widget hierarchy, [18](#), [53](#)

method calls, [55](#)

naming scheme, [52](#)

overview, [16](#)

program ID, [52](#)

properties, [56](#)

registering, [50](#)

skills required, [19](#)

using in IDL, [50](#)

widget events, [57](#)

WIDGET_ACTIVEX, [18](#)

ActiveXCal.pro, [61](#)

ActiveXExcel.pro, [65](#)

allprops.pro, [89](#)

arraydemo.pro, [99](#)

arrays

converting majority in Export Bridge, [165](#)

multidimensional storage and access, [494](#)

passing

by reference, [40](#)

by value, [40](#)

See also multidimensional arrays

arrays_example.java, [256](#)

array2d.java, [99](#)

B

- bridge_setup script, [144](#)
- bridge_version.pro, [95](#)
- bridges
 - definition, [10](#)
 - Export
 - about, [12](#)
 - Export Bridge Assistant, [148](#)
 - supported data types, [166](#)
 - Import, [11](#)
- by reference array passing, [40](#)
- by value array passing, [40](#)

C

classes

Java

- data members, [89](#)
- methods, [87](#)
- names, [84](#)
- path, [75](#)
- properties, [89](#)
- static, [85](#)

COM

- Program ID, [156](#)

COM connector

- about, [246](#)
- debugging, [213](#)
- error handling, [211](#)
- event handling, [208](#)
- examples, [249](#)
- methods
 - Abort, [193](#)
 - CreateObject, [194](#)
 - CreateObjectEx, [196](#)
 - DestroyObject, [199](#)
 - ExecuteString, [200](#)
 - GetIDLObjectClassName, [201](#)
 - GetIDLObjectVariableName, [202](#)
 - GetIDLVariable, [203](#)

- GetLastError, [204](#)
- GetProcessName, [205](#)
- SetIDLVariable, [206](#)
- SetIProcessName, [207](#)

- reference, [189](#)

- using, [247](#)

COM export bridge

- about wrapper objects, [191](#)
- methods, [192](#)
- reference, [189](#)

COM objects

- array passing by reference, [40](#)
- class ID, [24](#)
- creating IDLcomIDDispatch objects, [28](#)
- data type mapping, [44](#)
- data types, [30](#)
- definition, [16](#)
- destroying, [43](#)
- example IDL code, [46](#)
- exposing as IDLcomIDDispatch objects, [18](#)
- in IDL, [22](#)
- method calls, [29](#)
- Microsoft Object Viewer, [26](#)
- optional method arguments, [30](#)
- overview, [16](#)
- program ID, [25](#)
- properties, [37](#)
- See also* ActiveX
- See also* IDLcomIDDispatch objects
- skills required, [19](#)

com.idl.javaidl

- import statement, [144](#)
- package, [312](#)

- com_export_arrays_doc.txt, [251](#)

- com_export_commandline_doc.txt, [252](#)

- com_export_grwindow_doc.txt, [280](#)

- com_export_hello_doc.txt, [250](#)

- com_export_helloex_doc.txt, [275](#)

- com_export_itwinmanip_doc.txt, [283](#)

- com_export_triwindow_doc.txt, [288](#)

- configuring the IDL-Java bridge, [75](#)

- connecting
 - to Java objects, [72](#)
- connector object. *See* Java connector object or COM connector object
- copyrights, [2](#)
- createObject method
 - COM connector, [194](#)
 - Java connector, [220](#)
- createObjectEx method
 - COM connector, [196](#)
- creating
 - IDL object in COM, [194](#), [196](#)
 - IDL object in Java, [220](#)
 - Java object in IDL, [84](#)

D

- data types
 - IDL and Java, [80](#)
 - IDL-Java bridge conversion, [82](#)
 - Java and IDL, [78](#)
 - supported by Export Bridge, [166](#)
- destroyObject method
 - COM connector, [199](#)
 - Java connector, [223](#)
- drawable objects, [264](#)

E

- environment variables
 - IDL_PREFER_64, [144](#)
- errors
 - handling
 - COM wrapper objects, [211](#)
 - IDL-Java bridge, [96](#)
 - Java wrapper objects, [242](#)
 - Java exceptions, [96](#)
- examples
 - ActiveX
 - ActiveXCal.pro, [61](#)

- ActiveXExcel.pro, [65](#)
- including controls, [65](#)
- bridges
 - See also* examples
 - COM.
 - Java.
 - export_grwindow_doc__define.pro, [277](#), [299](#)
 - export_itwinmanip_doc__define.pro, [281](#), [304](#)
 - helloworld__define.pro, [182](#)
 - helloworldex__define.pro, [272](#), [294](#)
 - IDispatchDemo.pro, [46](#)
 - idlgrwindowexample__define.pro, [266](#), [284](#)
 - idlitdirectwindowexample__define.pro, [266](#), [284](#)
 - idlitwindowexample__define.pro, [266](#), [284](#)
- COM
 - export
 - com_export_arrays_doc.txt, [251](#)
 - com_export_commandline_doc.txt, [252](#)
 - com_export_grwindow_doc.txt, [280](#)
 - com_export_hello_doc.txt, [250](#)
 - com_export_helloex_doc.txt, [275](#)
 - com_export_itwinmanip_doc.txt, [283](#)
 - com_export_tr>window_doc.txt, [288](#)
- import
 - ActiveXCal.pro, [61](#)
 - ActiveXExcel.pro, [65](#)
 - IDispatchDemo.pro, [46](#), [46](#)
- Java
 - export
 - arrays_example.java, [256](#)
 - export_grwindow_doc_example.java, [302](#)
 - export_itwinmanip_delete.java, [306](#)
 - export_itwinmanip_doc_example.java, [306](#)
 - hello_example.java, [254](#)

- helloworldex_example.java, 296
- JIDLCommandLine.java, 258
- import
 - allprops.pro, 89
 - array2d.java, 99
 - arraydemo.pro, 99
 - bridge_version.pro, 95
 - exception.pro, 97
 - GreyBandsImage.java, 104
 - helloJava.java, 92
 - hellojava.pro, 84
 - hellojava2.pro, 92
 - javaprops.pro, 85
 - jbexamples.jar, 108
 - publicmembers.pro, 89
 - showexcept.pro, 97
 - showgreyimage.pro, 104
 - urlread.pro, 102
 - URLReader.java, 102
- using COM objects, 46
- wrapper objects
 - COM, 182
 - Java, 184
- exception.pro, 97
- executeString method
 - COM connector, 200
 - Java connector, 224
- Export Bridge
 - IDL object requirements, 261
 - Java setup script, 144
 - overview, 12
 - programming limitations, 263
- Export Bridge Assistant
 - building wrapper objects, 161
- examples
 - COM, 182, 270
 - Java, 184, 292
- exporting wrapper objects
 - bridge information, 167
 - converting array majority, 165
 - skipped information, 178
- source object
 - method information, 173
 - modification, 181
 - object information, 170
 - parameter information, 176
 - property information, 171
 - states, 162
 - superclasses, 180
 - specifying information, 164
 - supported data types, 166
- interface
 - logs panel, 154
 - menu bar, 151
 - property sheet view, 154
 - toolbar, 152
- logs
 - build, 155
 - change, 154
 - export, 155
- output destinations, 141
- projects
 - bridge information, 167
 - opening, 157
 - saving, 157
 - updating, 158
- running
 - in different IDL modes, 141
- supported platforms and compilers, 140
- Update dialog, 159
- using, 150
- export_grwindow_doc__define.pro, 277, 299
- export_grwindow_doc_example.java, 302
- export_itwinmanip_delete.java, 306
- export_itwinmanip_doc__define.pro, 281, 304
- export_itwinmanip_doc_example.java, 306
- exporting
 - IDL objects to COM, 148
 - IDL objects to Java, 148
- exporting drawable objects
 - examples, 266
 - requirements, 264

F

file

IDL-Java, [75](#)

G

getIDLObjectClassName method

COM connector, [201](#)

Java connector, [225](#)

getIDLObjectVariableName method

COM connector, [202](#)

Java connector, [226](#)

getIDLVariable method

COM connector, [203](#)

Java connector, [227](#)

GetLastError method

COM connector, [204](#)

getProcessName method

COM connector, [205](#)

Java connector, [228](#)

GreyBandsImage.java, [104](#)

H

handling Java exceptions, [96](#)

hello_example.java, [254](#)

helloJava.java, [92](#)

hellojava.pro, [84](#)

hellojava2.pro, [92](#)

helloworld__define.pro, [182](#)

helloworldex__define.pro, [272](#), [294](#)

helloworldex_example.java, [296](#)

I

IDispatchDemo.pro, [46](#), [46](#), [46](#)

IDL Java Package, [312](#)

IDL_PREFER_64 environment variable, [144](#)

IDLcomActiveX object

see ActiveX controls

IDLcomIDispatch objects

creating, [28](#)

destroying, [43](#)

method calls, [29](#)

naming scheme, [24](#)

overview, [18](#), [22](#)

idlgrwindowexample__define.pro, [266](#), [284](#)

idlitdirectwindowexample__define.pro, [266](#), [284](#)

idlitwindowexample__define.pro, [266](#), [284](#)

IDL-Java bridge. *See* Java Import Bridge

Import Bridge overview, [11](#)

initListeners method, [220](#), [239](#)

isObjectCreated method

Java connector, [229](#)

J

Java connector

about, [246](#)

debugging, [244](#)

error handling, [242](#)

event handling, [232](#)

examples, [253](#)

methods

abort, [219](#)

createObject, [220](#)

destroyObject, [223](#)

executeString, [224](#)

getIDLObjectClassName, [225](#)

getIDLObjectVariableName, [226](#)

getIDLVariable, [227](#)

getProcessName, [228](#)

isObjectCreated, [229](#)

setIDLVariable, [230](#)

setIProcessName, [231](#)

reference, [215](#)

using, [247](#)

Java Export Bridge

about wrapper objects, [217](#)

- methods, [218](#)
- reference, [215](#)
- runtime environment (JRE) requirements, [140](#)
- Java Import Bridge
 - class name in IDL, [84](#)
 - classes
 - data members, [89](#)
 - methods, [87](#)
 - names, [84](#)
 - path, [75](#)
 - properties, [89](#)
 - static, [85](#)
 - configuration, [75](#)
 - converting data types with IDL, [82](#)
 - creating IDL-Java bridge objects, [84](#)
 - destroying objects, [91](#)
 - IDL data types, [78](#)
 - Java data types, [80](#)
 - Native Interface (JNI), [73](#)
 - objects, [72](#)
 - runtime environment (JRE) requirements, [72](#)
 - session object, [94](#)
 - static
 - classes, [85](#)
 - data members, [85](#)
 - methods, [85](#)
 - version, [94](#)
- javaprops.pro, [85](#)
- jbexamples.jar, [108](#)
- JIDL (IDL Java) package
 - classes
 - JIDLArray, [317](#)
 - JIDLBoolean, [321](#)
 - JIDLByte, [328](#)
 - JIDLCanvas, [333](#)
 - JIDLChar, [376](#)
 - JIDLConst, [383](#)
 - JIDLDouble, [388](#)
 - JIDLFloat, [395](#)
 - JIDLInteger, [400](#)
 - JIDLLong, [409](#)
 - JIDLObject, [428](#)
 - JIDLProcessInitializer, [460](#)
 - JIDLShort, [463](#)
 - JIDLString, [468](#)
 - errors
 - JIDLAbortedException, [315](#)
 - JIDLBusyException, [326](#)
 - JIDLException, [393](#)
 - interfaces
 - JIDLComponentListener, [381](#)
 - JIDLKeyListener, [406](#)
 - JIDLMouseListener, [415](#)
 - JIDLMouseMotionListener, [418](#)
 - JIDLNotifyListener, [422](#)
 - JIDLNumber, [424](#)
 - JIDLObjectI, [446](#)
 - JIDLOutputListener, [458](#)
- JIDL Package class summary, [312](#)
- JIDLAbortedException, [315](#)
- JIDLArray, [317](#)
- JIDLBoolean, [321](#)
- JIDLBusyException, [326](#)
- JIDLByte, [328](#)
- JIDLCanvas, [333](#)
- JIDLChar, [376](#)
- JIDLCommandLine.java, [258](#)
- JIDLComponentListener, [381](#)
- JIDLConst, [383](#)
- JIDLDouble, [388](#)
- JIDLException, [393](#)
- JIDLFloat, [395](#)
- JIDLInteger, [400](#)
- JIDLKeyListener, [406](#)
- JIDLLong, [409](#)
- JIDLMouseListener, [415](#)
- JIDLMouseMotionListener, [418](#)
- JIDLNotifyListener, [422](#)
- JIDLNumber, [424](#)
- JIDLObject, [428](#)
- JIDLObjectI, [446](#)
- JIDLOutputListener, [458](#)
- JIDLProcessInitializer, [460](#)

JIDLShort, [463](#)
 JIDLString, [468](#)

L

legalities, [2](#)
 licensing
 exported COM objects, [134](#)
 exported Java objects, [134](#)
 Java export bridge, [221](#)

M

method calls
 ActiveX controls, [55](#)
 COM objects, [29](#)
 Microsoft Object Viewer, [26](#)
 multidimensional arrays
 2D examples, [498](#)
 storage and access, [494](#)
 COM, [496](#)
 IDL, [497](#)

O

object properties
 COM, [37](#)
 Object Viewer, [26](#)
 objects
 IDL-Java bridge session
 exceptions, [96](#)
 parameters, [94](#)
 Java classes
 IDL-Java bridge, [72](#)
 path, [75](#)
 OLE/COM Object Viewer, [26](#), [32](#), [52](#)

P

package
 com.idl.javaidl, [312](#)
 ProgID, [156](#)
 Program ID, [156](#)
 properties
 ActiveX controls, [56](#)
 COM objects, [37](#)
 publicmembers.pro, [89](#)

S

session object
 IDL-Java bridge exceptions, [96](#)
 IDL-Java bridge parameters, [94](#)
 setIDLVariable method
 COM connector, [206](#)
 Java connector, [230](#)
 setProcessName method
 COM connector, [207](#)
 Java connector, [231](#)
 showexcept.pro, [97](#)
 showgreyimage.pro, [104](#)

T

trademarks, [2](#)

U

urlread.pro, [102](#)
 URLReader.java, [102](#)

V

Virtual Machine
 Java (JRE) requirements, [72](#), [140](#)

W

widget events

 ActiveX, [57](#)

WIDGET_ACTIVEX function

 using, [18](#)

widgets

 WIDGET_ACTIVEX function

 using, [18](#)

wrapper objects

 about, [127](#)

 building in the Export Bridge Assistant, [161](#)

 converting array majority, [165](#)

 debugging

 COM, [213](#)

 Java, [244](#)

error handling

 COM, [211](#)

 Java, [242](#)

event handling

 COM, [208](#)

 Java, [232](#)

examples

 COM, [182](#), [249](#), [270](#)

 Java, [184](#), [253](#), [292](#)

exporting, [162](#)

exporting drawable objects, [264](#)

supported data types, [166](#)