IBM ILOG Views

Foundation V5.3

User's Manual

June 2009

Copyright notice

Trademarks

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

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

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

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

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

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

Notices

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

Table of Contents

About This Manual

This *User's Manual* explains how to use the C++ API and grammar that are detailed in the IBM ILOG Views *Foundation Reference Manual*.

What You Need to Know

This manual assumes that you are familiar with the PC or UNIX environment in which you are going to use IBM® ILOG® Views, including its particular windowing system. Since IBM ILOG Views is written for C++ developers, the documentation also assumes that you can write C++ code and that you are familiar with your C++ development environment so as to manipulate files and directories, use a text editor, and compile and run C++ programs.

Manual Organization

This manual provides conceptual and hands-on information for developing applications that incorporate IBM® ILOG® Views Foundation. It describes the fundamentals that underlie IBM® ILOG® Views graphic objects and shows how to create and use graphic objects.

This manual contains the following chapters:

- Chapter 1, *Introducing IBM ILOG Views Foundation* provides an overview of IBM ILOG Views Foundation.
- Chapter 2, *Graphic Objects* describes the concept of a graphic object and explains the use of the many classes derived from the `IlvGraphic` class.
- Chapter 3, *Graphic Resources* describes the resource and palette classes that define the appearance of graphic objects and text.
- Chapter 4, *Graphic Formats* describes the vectorial and bitmap formats available with IBM ILOG Views.
- Chapter 5, *Image Processing Filters* shows the subclasses of `IlvBitmapFilter` that allow you to process your bitmap images in various ways, such as combining two images with a selection of filters.
- Chapter 6, *The Display System* provides information on `IlvDisplay`, IBM ILOG Views' basic class for connection to the display system.
- Chapter 8, *Drawing Ports* describes the `IlvPort` base class.
- Chapter 7, *Views* explains the concept of a view or visual display area as used in IBM ILOG Views.
- Chapter 9, *Containers* explains how to use containers to provide efficient display and behavior of graphic objects in applications.
- Chapter 10, *Dynamic Modules* contains information about creating and loading a dynamic library or DLL.
- Chapter 11, *Events* contains information on the classes that implement event loops.
- Chapter 12, *IlvNamedProperty: The Persistent Properties Class* explains how to associate application-dependent data with IBM ILOG Views objects.
- Chapter 13, *Printing in IBM ILOG Views* explains how to use the IBM ILOG Views printing framework to define printer, document, and paper formats and other printing controls.
- Chapter 14, *IBM ILOG Script Programming* explains how to use IBM ILOG Script, the IBM ILOG Views high-level scripting language.
- Chapter 15, *Internationalization* explains how to develop localized language versions of IBM ILOG Views applications.

The appendixes provide auxiliary and reference information as follows:

- Appendix A, *Packaging IBM ILOG Views Applications* describes the `ilv2data` tool for packaging your applications with IBM ILOG Views.
- Appendix B, *Using IBM ILOG Views on Microsoft Windows* discusses requirements and give tips on interfacing IBM ILOG Views with Microsoft Windows.

- Appendix C, *Using IBM ILOG Views on X Window Systems* discusses requirements and give tips on interfacing IBM ILOG Views with the X Window system.
- Appendix D, *Portability Limitations* discusses the system-dependent aspects of IBM ILOG Views Foundation that may limit portability across multiple platforms.
- Appendix E, *Error Messages* lists the error messages that ILOG Views Foundation may generate and discusses possible causes and workarounds.
- Appendix F, *IBM ILOG Script 2.0 Language Reference* is a reference for the syntax of IBM ILOG Script.

Notation

Typographic Conventions

The following typographic conventions apply throughout this manual:

- Code extracts and file names are written in a `"code"` typeface.
- Entries to be made by the user, such as in dialog boxes, are written in a `"code"` typeface.
- Command variables to be supplied by the user are written in *italics*.
- Some words in *italics*, when seen for the first time, may be found in the glossary.

Naming Conventions

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

- The names of types, classes, functions, and macros defined in the ILOG Views Foundation library begin with `Ilv`, for example `IlvGraphic`.
- The names of types and macros not specific to IBM ILOG Views begin with `Il`, for example `IlBoolean`.
- The names of classes as well as global functions are written as concatenated words with each initial letter capitalized.

  class `IlvDrawingView`;
- The names of virtual and regular methods begin with a lowercase letter; the names of static methods start with an uppercase letter. For example:

  `virtual IlvClassInfo*` **`getClassInfo`**`() const;`

  `static IlvClassInfo*` **`ClassInfo*`**`() const;`

A Note on Examples

The documentation offers examples and explanations of how to use IBM ILOG Views effectively. Moreover, some examples are extracted from the source code delivered with IBM ILOG Views, which is in the `samples` directory, just below the directory where IBM ILOG Views is installed.

Further Reading

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

- Lippman, Stanley B. *C++ Primer*, 3rd ed. Reading, MA: Addison-Wesley, 1998.
- Stroustrup, Bjarne. *The C++ Programming Language*, 3rd ed. Reading, MA: Addison-Wesley, 1997.
- Stroustrup, Bjarne. *The Design and Evolution of C++*. Reading, MA: Addison-Wesley, 1994.
- ISO/IEC 14882:1998 *Programming Languages - C++* and
  ISO/IEC 14882-1998 *Information Technology - Programming Languages - C++*

  The ISO/ANSI C++ Standard. Available in online and printed forms from the American National Standards Institute (`http://www.ansi.org`).

The following books provide good advice on several graphics-related issues:

- Foley, James D., Andries van Dam, Steven K. Feiner, and John F. Hughes, *Computer Graphics: Principles and Practice*, 2nd ed. Reading, MA: Addison-Wesley, 1996.
- *Graphics Gems.*
  Vol. I: Glassner, Andrew S. (ed.), 1990. Reissue 1993.
  Vol. II: Arvo, James (ed.), 1991. Reissue 1994.
  Vol. III: Kirk, David (ed.), 1992, 1994.
  Vol. IV: Heckbert, Paul S. (ed.), 1994
  Vol. V: Paeth, Alan W. (ed.), 1995.
  Boston: Academic Press.
- Murray, James D. and William van Ryper. *Encyclopedia of Graphics File Formats*, 2nd ed. Sebastopol, CA: O'Reilly and Associates, 1996.
- Nye, Adrian.
  Vol. 1 *Xlib Programming Manual*, 3rd ed., 1992.
  Vol. 2 *Xlib Reference Manual*, 3rd ed., 1992
  O'Reilly & Associates.
- O'Rourke, Joseph. *Computational Geometry in C*, 2nd ed. Cambridge University Press, 1998.

- Rogers, David F. and J. Alan Adams. *Mathematical Elements for Computer Graphics*. McGraw-Hill Publishing Co., 1990.
- Young, Douglas A. *The X Window System: Programming and Applications with Xt, OSF/Motif*, 2nd ed. Prentice Hall, 1994.

C H A P T E R

1

Introducing IBM ILOG Views Foundation

IBM® ILOG® Views Foundation is the base IBM ILOG Views package, providing the core features for developers creating graphical user interfaces (GUIs) and interactive two-dimensional graphics for applications running in UNIX and PC environments.

We include here:

- The *Application Programming Interface (API)* introducing the set of C++ libraries for designing your graphic interface.
- *Using IBM ILOG Views* which provides an orientation to the basic concepts of views and graphic objects.

Application Programming Interface (API)

IBM® ILOG® Views is organized as a set of C++ class libraries accompanied by several auxiliary editing tools to help you design your interfaces.

Libraries

The IBM ILOG Views libraries provide the API needed to implement the programmable portions of your applications. As a true object-oriented C++ library, IBM ILOG Views emphasizes code reuse through inheritance. Each derived class specializes its base class,

adding to or modifying inherited structure and behavior. This means that if a particular class does not have a feature you are looking for, you should also check its base class to see if the feature is inherited. When deriving your own classes, you can use the existing class features and write only the new code you need, thus reducing development and maintenance costs.

The IBM ILOG Views API is written in C++, a superset of C, from which you can call your C routines if necessary. Because C++ is flexible and resource efficient, it has become the most widely used and preferred object-oriented language.

The object-oriented capabilities provided by C++ allow code reuse and thus saves coding time. With a class hierarchy, a library of C++ classes is more flexible, extensible, and dependable than a procedure-oriented library.

Object-oriented programming suits graphics-oriented applications particularly well, because graphic objects often have similar operations performed on them. For example, a button is a specialized type of rectangle, and can thus inherit all the features of a rectangle without recoding. This hierarchical nature provides for easier, timesaving development and maintenance procedures.

Object-oriented code allows you to extend or specialize IBM ILOG Views objects for your own application (or library) without knowing the details of the IBM ILOG Views implementation. Similarly, your customers can specialize your objects without knowing details of your implementation. In addition, you can create your own library for your applications on top of the IBM ILOG Views library by creating subclasses.

Class Hierarchy

The organization of IBM ILOG Views class hierarchy makes it easy for you to find what you need. For example, using the classes in the diagram below, you can easily create sophisticated interfaces with minimum coding.

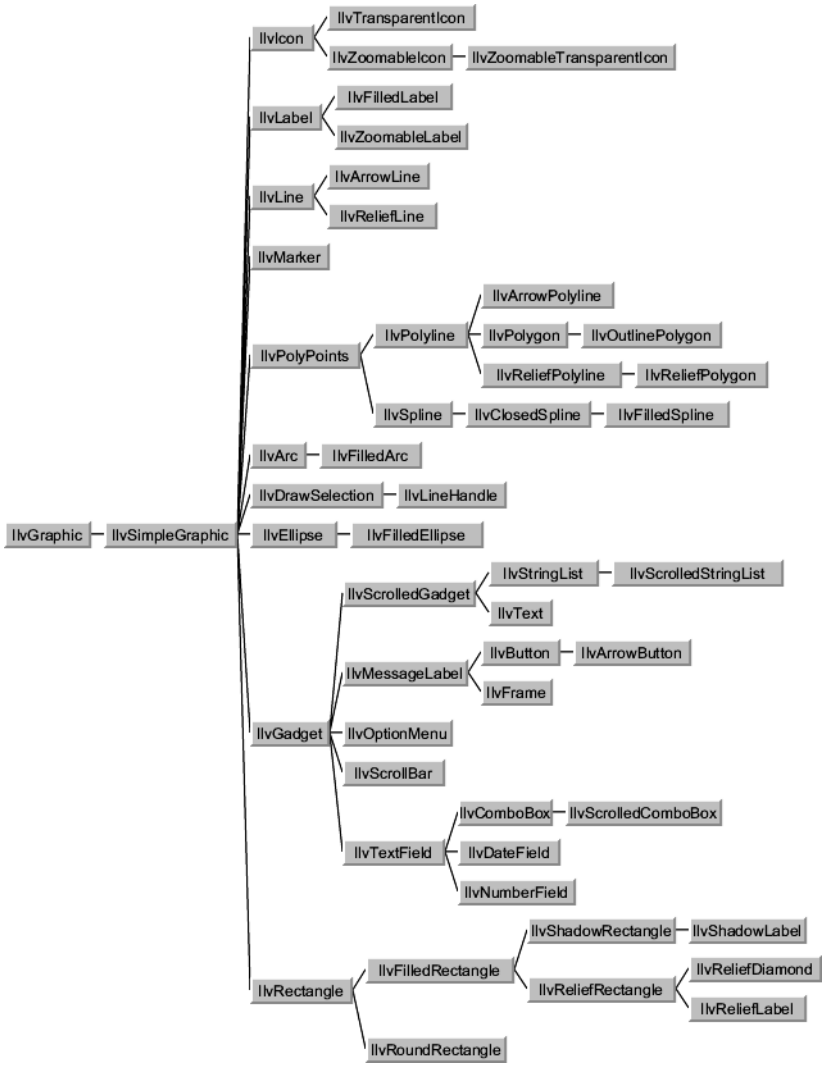


Figure 1.1 *Partial Class Hierarchy of IBM ILOG Views Graphic Objects*

Using IBM ILOG Views

You use IBM® ILOG® Views Foundation to create graphical user interfaces (GUIs) and interactive two-dimensional graphics for applications running in UNIX and PC environments.

Here are introductions to some basic terms and concepts related to IBM ILOG Views. Later chapters cover in detail the classes for implementing them in the API.

- *Windows and Views* defines views, windows, and related terminology.
- *Containers: Controlling the View* discusses the role of containers in your use of IBM ILOG Views. It also discusses the difference between a container and a manager.
- *Introducing Graphic Objects* provides an orientation to displaying and transforming graphic objects in IBM ILOG Views.
- *Drawing Attributes and Palettes* discusses the rich variety of colors, fonts, and other IBM ILOG Views resources that affect the appearance of graphic objects.
- *Basic Drawing Types* relates the drawing attributes to lines, regions, and strings: the fundamental types of drawings in IBM ILOG Views.

Windows and Views

In IBM® ILOG® Views, a view is an object to which basic services can be added, and which is associated with the window of the underlying display system, such as X Window™ in UNIX. Drawing frequently takes place in the view, which displays an image of the objects or a subset of them. This image can be geometrically transformed by moving, zooming, or rotating without affecting the objects themselves.

What is a View?

Your initial efforts when you write an IBM® ILOG® Views program will be focused on creating and combining *views* where people can display and possibly interact with your program.

A view is a visual place holder—a rectangular area on your screen—where elements of your IBM ILOG Views application are displayed.

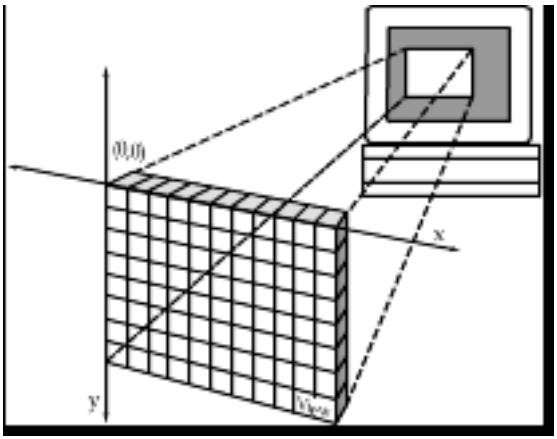


Figure 1.2 *A View*

Each view is distinguished by its:

- **location** (x,y coordinates that you define),
- **size** (height and width that you define),
- **visibility** (a view might be present but not visible).

You create visible elements of your IBM ILOG Views application by combining views and their contents.

Looking at a View Window

Here is a simplified drawing of an IBM® ILOG® Views window:

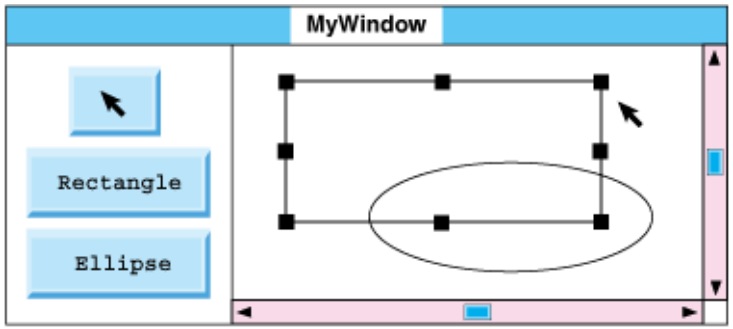


Figure 1.3 *An IBM ILOG Views Window*

This window contains buttons enabling you to draw rectangles and ellipses, and an arrow button to select an existing object and either move or resize it. It has a set of scroll bars for moving different parts of a larger working view into the display area.

This window is composed of four different IBM ILOG Views views:

- *Top-Level View: Top Window*
- *Scroll View*
- *Tools View*
- *Working View*

Top-Level View: Top Window

The purpose of the top window is generally to hold various kinds of lower-level views. You rarely draw directly into a top window, but most often into one of the lower-level views that it holds.

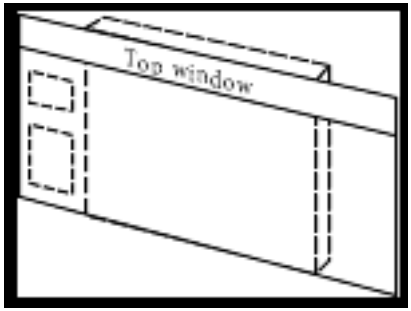


Figure 1.4 *Top Window*

Views of this kind are the only views that:

- Include a title bar.
- Can be associated with a system menu that allows the user of your programs to intervene concerning such things as resizing or iconifying your window.

You can associate as many lower-level views as you need with the top window. The top window holds not only the title of the window but also its current size. None of the views held within the top window can extend beyond the rectangular perimeter of the top window. A corollary of this situation is that a top window can never be held within any other kind of lower-level view.

Scroll View

The scroll view is a lower-level view.The sole purpose of the scroll view is to contain a pair of scroll bars, allowing you to scroll the lower-level drawing view that is contained within the scroll view.

Figure 1.5 *Scroll View*

> ***Note:*** *This type of window is provided with the Gadgets package of IBM ILOG Views. There is a native implementation in the Foundation package, for Microsoft Windows and Motif ports only.*

Tools View

The tools view is a lower-level view that contains drawing and selection command buttons. This kind of view can store and display graphic objects as well as coordinate actions that the user can perform on these objects.

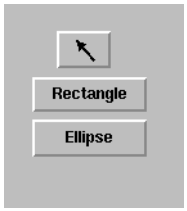


Figure 1.5 *Tools View*

Working View

The working view is the lowest-level view. The working view is larger than the part that you see at any particular moment. In the following figure, the white rectangle in the middle of the large gray rectangle represents what you see. For example, there's an ellipse in the top-right-hand corner of the view that is not currently visible. To see it, the user has to use the scroll bars in the higher-level scroll view.

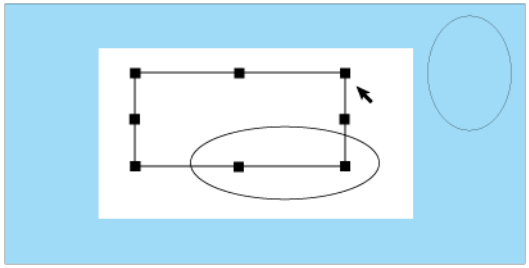


Figure 1.6 *Working View*

You can control the working area of change by *clipping*. While a clipping region is active, only changes to that region are displayed.

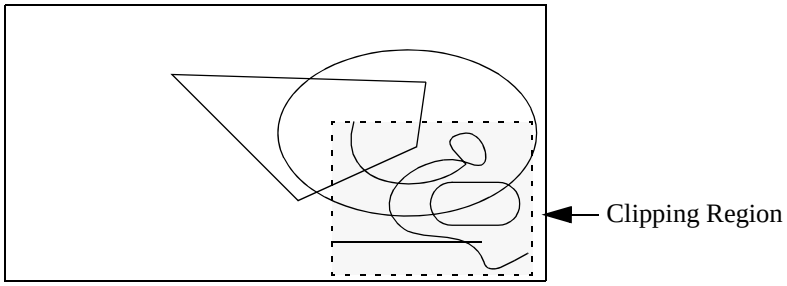


Figure 1.7 *Clipped Area*

Containers: Controlling the View

Containers coordinate the storage and display of graphic objects.

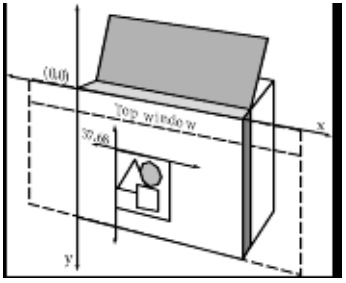


Figure 1.8 *A Container*

Fundamentally, a container is a view, with predefined callbacks to handle the automatic refresh of the graphic objects it stores, as well as the system events and user events that occur in that view.

Each graphic object stored in a container is unique and can only be displayed by that container. In short, a container:

- is essentially another kind of view where you can collect any number of graphic objects.
- automatically manages all drawing operations within the view.
- can associate interactors to its objects to give them particular behaviors.
- lets you access objects by their names.
- can use a transformer to move, zoom, and rotate when drawing objects.
- associates single actions with events received by the view.

Containers versus Managers

IBM ILOG Views groups objects in one of two basic types of storage data structures:

- Containers
- Managers

A view is associated with a set of graphic objects stored in a container or manager.

A container stores a certain number of graphic objects, and it is associated with a view, which displays the objects stored in the container. Each object can be associated with a specific behavior, and accelerators—which are keyboard events that immediately call a predefined function—can be attached to the container itself. Containers are part of the functionality of the Foundation package.

A manager is another type of data structure that provides layers, multiple-view, fast redraw, persistency, and editing functionality. For details on managers see the Managers documentation.

> ***Note:*** *For efficient drawing of numerous objects, multiple views, and layers, you should use the managers instead of containers.*

Introducing Graphic Objects

Using a two-dimensional vector graphic engine, IBM® ILOG® Views provides drawing ports (memory, screen, and dumpfile) as well as a large set of drawing primitives to create basic geometric forms. You can draw basic geometric shapes such as arcs, curves, rectangles, labels, and so on. You can draw on the screen, in memory, or generate dump files

such as PostScript. You can create black-and-white and color images. The graphic engine builds on these primitives to define graphic objects.

Displaying Graphic Objects

A graphic object is an image that users can view on their screen. When you display a graphic object, you associate the coordinates of the graphic object with the coordinate system of a particular container.

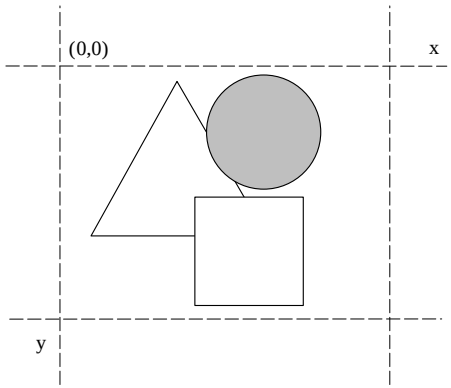


Figure 1.9 *Geometric Properties of Graphic Objects*

Geometric properties define the shape and placement of your graphic object. Every graphic object has an x value, a y value, and dimensions (that is, width and height). The x and y values indicate the upper-left corner of the graphic object bounding box, which is the smallest rectangle that entirely contains the area covered by this object.

You define the exact shapes of graphic objects in your IBM ILOG Views–based programs and then make them concrete using various drawing member functions. Other member functions provide you with information about your graphic objects, and let you carry out geometric tests concerning the shapes that you are using. For example, you can check whether a point with given coordinates lies inside a certain shape.

Interactors

IBM ILOG Views makes a clear separation between graphic objects and behaviors, thus allowing you to apply a particular behavior to an object.

In IBM ILOG Views, a predefined behavior is called an “interactor.” An interactor can be applied to any graphic object to give it a particular behavior, thus defining its functionality.

For example, by applying a “button” interactor to an object, that object, which before only had a visual aspect, now takes on the behavior of a button—that is, when you click it, it blinks.

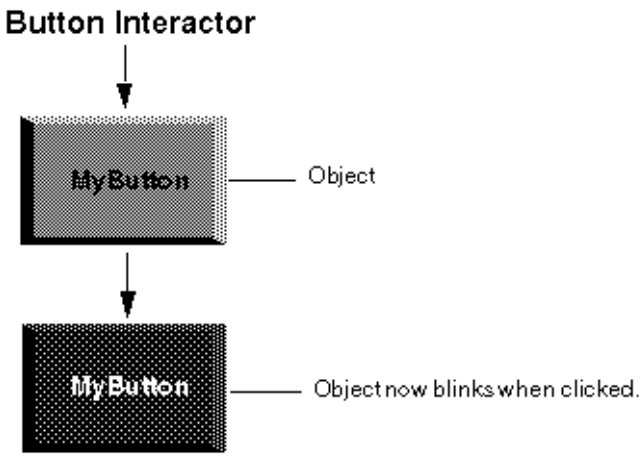


Figure 1.10 *Object/Interactor Concept*

The benefits of separating objects from behavior is that you can apply a certain type of behavior to any type of graphic object. For example, you can apply button behavior to a bitmap (such as an icon) with only a single line of code.

Furthermore, it is quite easy to extend behavior by subclassing the interactor classes provided with IBM ILOG Views.

Drawing Attributes and Palettes

With IBM® ILOG® Views you have at your disposal a large selection of fill and line patterns, colors, and font attributes to apply to graphic objects and text.

Because these resources are grouped in palettes and are shared among any number of objects, you can easily make global changes while minimizing memory consumption.

Color

In a simple drawing situation, such as the creation of a rectangle, the actual drawing is carried out with what is referred to as the *foreground* color, and the area "behind" the drawing is referred to as the *background* color.

- **Foreground Color** The foreground color is used for drawing dots, arcs, lines, polylines, and so on. It is also used to display character strings and to fill areas such as polygons and arcs.

- **Background Color** The background color is used as a second color when filling with patterns and drawing character strings.

Line Style and Width

Besides ordinary lines, referred to as “solid,” you can draw straight lines and curves composed of dots or dashes. This is the line style. Line width refers to the thickness of the lines in a drawing.

The line style and width define the exact visible aspect of all line drawings as well as line-type drawings including polyline and spline.

Patterns

A pattern refers to the design used to fill surfaces. In IBM ILOG Views, there are two types of pattern, distinguished by the number of colors that can be applied.

Monochrome Pattern

The word “pattern” designates a monochrome (or two-color) design. IBM ILOG Views offers sixteen ready-to-use patterns.

Here is an example of a pattern:

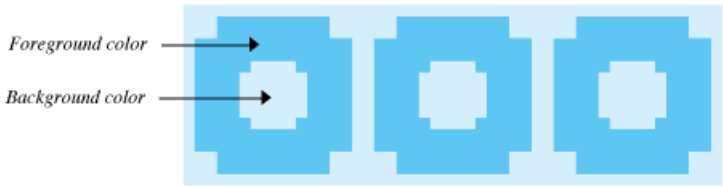


Figure 1.11 *A Pattern*

This particular pattern could be obtained by using a *mask* composed of a 16x16 array of bits, as you see here:

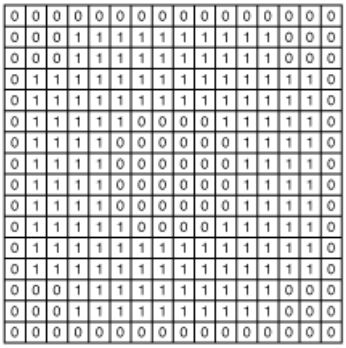


Figure 1.12 *A Pattern Mask*

Inside a pattern mask, we speak of the 1 bits as the *foreground*, and the 0 bits as the *background*. In other words, the pattern resource invokes the color resource.

Color Pattern

Whereas an ordinary pattern is two-dimensional, a color pattern incorporates a third dimension, *depth*, to deal with color.

Instead of having simply a `1` or a `0` at each location in the array, you insert a number that indicates the color to be used at that point in the color pattern. The default value for the color pattern is zero (`0`) indicating that no color pattern is to be used.

Font

Fonts are used with character strings, that is, when drawing text.

Basic Drawing Types

Basically, there are three kinds of drawings in IBM® ILOG® Views: lines, regions, and strings, with the attributes applied depending on the individual drawing needs and capabilities.

Lines

This category includes straight lines, curves and open-ended sets of connected straight lines or curves. Attributes are applied to lines as follows:

- **Color** Straight lines or curves are drawn with the current foreground color.
- **Line Style** The current line style (such as solid, dots or dashes) determines how the straight lines or curves are drawn.

- **Line Width** An unsigned integer indicates the current line width.
- **Pattern** Using monochrome (two-color), lines are drawn with the current pattern, but this is noticeable only with thick lines.
- **Color Pattern** Using color, lines are drawn with the current color pattern. Here again, this effect is noticeable only for thick lines.

Regions

This term designates closed sets of connected lines or curves. Attributes are applied to regions as follows:

- **Color** The closed curve around a region retains its original color; that is, the current foreground color.
- **Pattern** Regions are filled with the current fill pattern or fill mask pattern.
- **Color Pattern** Regions are filled with the current color pattern.
- **Fill Style** This value determines if patterns are to be handled as masks, monochrome patterns or color patterns.
- **Fill Rule** Determines the fill strategy for self-intersecting polygons. For details see *Fill Rule* on page 81.
- **Arc Mode** Determines how an arc is closed for filling. For details see *Arc Mode* on page 82.

Strings

Attributes are applied to strings as follows:

- **Color** The actual characters are printed with the foreground color.
- **Font** The font in which the string is printed.

C H A P T E R

2

Graphic Objects

IBM® ILOG® Views provides a hierarchy of classes that let you create various high-level graphic objects. The starting points for these objects are the classes `IlvGraphic` and `IlvSimpleGraphic`.

- *IlvGraphic: The Graphic Objects Class* is the foundation class for IBM ILOG Views graphic objects.
- *The IlvSimpleGraphic Class* is a fundamental class that inherits from `IlvGraphic`. It allows you to assign graphic resources and apply transformations to your graphic objects.
- *Predefined Graphic Objects* illustrates the numerous graphic objects provided by IBM ILOG Views for producing standard geometric forms such as arcs, rectangles, and so forth.
- *Composite Graphic Objects* allow you to optimize object usage by grouping for various purposes.
- *Other Base Classes* describe the additional graphic classes used primarily with other IBM ILOG Views packages.
- *Creating a New Graphic Object Class* illustrates in detail how to create a new, customized graphic object in IBM ILOG Views.

IlvGraphic: The Graphic Objects Class

IBM® ILOG® Views graphic objects inherit attributes from the abstract base class `IlvGraphic`. This class allows an IBM ILOG Views graphic object to draw itself at a given destination port, and, if desired, with a transformation of its coordinates determined by an associated object of the `IlvTransformer` class.

`IlvGraphic` has member functions that allow you to set and change geometric dimensions. A handful of member functions are given to set and get user properties that can be associated with an object for application-specific purposes. The `IlvGraphic` class does not actually implement these member functions. They are declared as virtual member functions, and are defined to do various operations in the classes that inherit `IlvGraphic` attributes. Though the member functions to manipulate geometric shapes and graphic attributes are present, they do nothing.

Member Functions

`IlvGraphic` member functions can be presented in several groups:

- **Geometric properties** These member functions handle location, size, and drawing properties, which include the `IlvGraphic::draw` method used to draw the graphic object. The virtual `IlvGraphic::draw` method should be defined conjointly with the method `IlvGraphic::boundingBox`, which defines the smallest rectangle that entirely contains all the area covered by the graphic object.
- **Graphic properties** Use these member functions to change the visible aspect of the objects, that is, their color or pattern. You do so by means of member functions that indicate graphic properties of graphic objects and modify the palette bound to these graphic objects. The following example shows how to set the background of any graphic object:

```
IlvButton* mybutton = new IlvButton(display,
                                    IlvPoint(20,20),
                                    "Quit");
IlvColor* color = display->getColor("gold");
if (color) mybutton->setBackground(color);
```

- **Named properties** Named properties handle the persistence of properties associated with graphic objects (see Chapter 12).
- **User properties** `IlvGraphic` objects can be associated with a set of source code user properties. User properties are a set of *key-value* pairs, where the key is an `refcppfoundation:IlSymbol` object and the value may be any kind of information value. User properties are not persistent.

  These member functions provide you with a simple way to connect your graphic objects with information that comes from your application. You can keep track of the graphic

part of your application by storing the pointers to objects you create, and connect the graphic side to the application by means of user properties, for example:

```
IlInt index = 10;
IlSymbol* key = IlGetSymbol("objectIndex");
mybutton->addProperty(key, (IlAny)index);
```

Some member functions provide tag management. Tags are markers that you can apply to graphic objects to identify them. You can then use various IBM ILOG Views functions to manipulate only the tagged objects.

- **Gadget properties** Gadget properties handle the sensitivity of objects to events, the callbacks to be called when the object is activated, the client data stored with objects, and the object interactor associated with the object class. For details on using callbacks, see the section *Callbacks*.
- **Focus chain properties** The *focus* indicates the object on the screen that is receiving any keyboard events. The *focus chain* is the order of the objects on the screen that receive the focus. The focus moves to the next object in the focus chain when, typically, the Tab key is pressed, or to the previous object when shift-Tab is pressed.
- **Class information** Subtypes of the class `IlvGraphic` can handle information at the class level. This means that all instances of a given class can share the same information. For example, `IlvGraphic::className` allows you to get the class name and `IlvGraphic::isSubtypeOf` returns `IlTrue` if the target `IlvGraphic` object is a subclass of the given class argument. This example shows how to use class information member functions:

```
IlvButton* button = new IlvButton(display,
                                  IlvPoint(10,10),
                                  "sample");
// Get the IlvClassInfo object associated with the button class.
IlvClassInfo* classInfo = button->getClassInfo();

// Get the name of the IlvGraphic class and print it: "IlvButton"
const char* name = classInfo->getClassName();
IlvPrint(name);

// Get the name of the super class and print it: "IlvMessageLabel"
name = classInfo->getSuperClass()->getClassName();
IlvPrint(name);

IlBoolean isSubtype =
   classInfo->isSubtypeOf(IlvSimpleGraphic::ClassInfo());
name = isSubtype ? "It's a subtype" : "error";
IlvPrint(name);
```

- **Class properties** Static member functions and their non-static equivalents let you handle properties at the class level, that is, these properties are defined for every instance of the class. In some methods, the `IlBoolean` parameter allows you to operate iteratively on each superclass of the object until a match is found. Here are the member functions that deal with class properties:

```
void AddProperty(const IlSymbol* key, IlAny value);
void RemoveProperty(const IlSymbol* key);
void ReplaceProperty(const IlSymbol* key, IlAny value);
void GetProperty(const IlSymbol* key,
                 IlBoolean checkSuperClass = IlFalse);
const IlvClassInfo* HasProperty(const IlSymbol* key,
                                IlBoolean checkSuperClass = IlFalse);
void addClassProperty(const IlSymbol* key, IlAny value);
IlBoolean removeClassProperty(const IlSymbol* key);
IlBoolean replaceClassProperty(const IlSymbol* key,
                                IlAny value);
IlAny getClassProperty(const IlSymbol* key,
                       IlBoolean checkSupCl = IlFalse) const;
const IlvClassInfo* hasClassProperty(const IlSymbol* key,
                                     IlBoolean checkSupCl = IlFalse) const;
```

  Here is an example that shows how to use class properties:

  Let us imagine a map where graphic instances are shown with toggle-like sensitive behavior (IBM ILOG Views provides specific objects called *interactors* that allow you to associate a behavior with graphic objects). Sometimes we may want these elements to be insensitive. Instead of scanning the object list to set the sensitivity to `IlFalse`, we use a class-level property in this way:

  Let `myClass` be a subclass of `IlvGraphic`; and let `myInteractor`, a subclass of `IlvToggleInteractor`, be an interactor attached to `myClass`.

```
// Add the class-level property
myClass* obj = new myClass(display);
obj->addClassProperty(IlGetSymbol("sensitive"),
                      (IlAny)IlTrue);
```

  In the applicative code, the application tests whether sensitivity must be inverted. There is another way to add a property to a class object, using a static member function, since both statements are equivalent:

```
if (anyValue == IlTrue)
{
   myClass::AddProperty(IlGetSymbol("sensitive"),
                        (IlAny)IlFalse);
}
```

In the implementation file of the `myInteractor` class, we redefine the parent class method `IlvInteractor::handleEvent` to add our specific behavior, that is freezing the sensitivity when a certain condition is present:

```
IlBoolean
myInteractor::handleEvent(IlvGraphic* object,
                          IlvEvent& event,
                          IlvContainer* cont,
                          IlvTransformer* transf)
{
   // gets the sensitivity state
   IlSymbol* symbol = IlGetSymbol("sensitive");
   if (object->hasClassProperty(symbol))
   {
      if (!object->getClassProperty(symbol))
         return IlFalse;
   }
   return IlvViewToggleInteractor::handleEvent(object,
                                               event,
                                               cont,
                                               transf);
}
```

- **Input/Output properties** These member functions let you read and write your object descriptions from and to special kinds of streams, known as `IlvInputFile` and `IlvOutputFile`, which handle the reading and writing of objects from C++ streams.

  IBM ILOG Views offers a basic implementation of these classes, which were designed so that you can easily add your specific information. Therefore, when you need to save and load application-dependent data, you should create your own subtypes of these two classes.

  - **Writing Graphic Objects**

    The `IlvOutputFile` class writes the complete description of a series of objects in an output stream. You can use this class in the following way:

    ```
    // Open a file output stream
    fstream outstream("image.ilv", ios::out | ios::trunc);

    // Initialize the number of objects and their array of pointers
    const IlvUInt n = 10;
    IlvGraphic* outObjects[n];
    for (IlvUInt i=0; i<n; i++)
       outObjects[i] = new IlvRectangle(display,
                                        IlvRect(0, 0, 200, 100));

    // Create the IlvOutputFile
    IlvOutputFile outfile(outstream);

    // Write the objects and get in outTotalCount the number
    // of objects actually stored
    IlvUInt outTotalCount = 0;
    outfile.saveObjects(n, outObjects, outTotalCount);
    ```

  - **Reading Graphic Objects**

The `IlvInputFile` class is the main class for reading objects from a stream. The following code shows how to read `IlvGraphic` objects from an input stream:

```
// Open a file input stream
fstream instream("image.ilv", ios::in);

// Create the IlvInputFile
IlvInputFile infile(instream);

// Get the number of created objects and their array of pointers
IlUInt InTotalCount = 0;
IlvGraphic* const* inObjects = infile.readObjects(display,
                                                  InTotalCount);
```

Callbacks

When an object is designated by the user to perform an action, the user may need to call a specific function, a *callback*, that the user has defined. These functions are usually called by the `handleEvent` method of the object.

There are two ways you can set a specific callback to be called when an action is triggered:

- Register the callback as a pointer to a user-defined function.

  This function must be an `IlvGraphicCallback` type.

  The type `IlvGraphicCallback` is defined in the file `<ilviews/graphic.h>`:

```
#include <ilviews/graphic.h>
typedef void (* IlvGraphicCallback)(IlvGraphic* obj, IlAny arg);
```

  The first argument (`obj`) is the graphic object that called the callback and the second argument (`arg`) is the *user data*. The user data could be defined when setting the callback to a particular gadget. If no data is defined, the parameter is the graphic object's client data that you can set with the `IlvGraphic::setClientData` method.

- Register a callback name that is in turn associated with a function to be called by the graphic object container. The association between a callback function and its name must be unique for a particular container.

Registering Callbacks

The methods used to register a callback in a container are:

```
#include <ilviews/contain.h>

void   registerCallback(const char* callbackName,
                        IlvGraphicCallback callback);
void   unregisterCallback(const char* callbackName);
IlvGraphicCallback getCallback(const IlSymbol* callbackName) const;
```

Callback Types

An object can define several *callback types.* Each callback type corresponds to a specific action. For example, you can find, in every gadget, a “Focus In” callback type that refers to the callbacks to be invoked when the gadget receives the keyboard focus.

Each callback type stores a list of callbacks to be invoked when the related event occurs. The class `IlvGraphic` contains generic methods that allow you to add or remove a callback for a specific callback type:

```
#include <ilviews/graphic.h>

void addCallback(const IlSymbol*   callbackType,
                 IlvGraphicCallback callback);
void addCallback(const IlSymbol* callbackType,
                 const IlSymbol* callbackName);
void addCallback(const IlSymbol*   callbackType,
                 IlvGraphicCallback callback,
                 IlAny data);
void addCallback(const IlSymbol* callbackType,
                 const IlSymbol* callbackName,
                 IlAny data);
void removeCallback(const IlSymbol*   callbackType,
                    IlvGraphicCallback callback);
void removeCallback(const IlSymbol* callbackType,
                    const IlSymbol* callbackName);
```

The argument `data` that you can pass when adding a callback is sent to the callback. It corresponds to the argument named `arg` in the `IlvGraphicCallback` definition.

The Main Callback

The Main callback type can be used to perform the main action of an object that is designated to perform several actions. For example, it could be used when a button object is activated, or when you double-click on an item in a string list.

Some useful methods help you set the Main callback of an object:

```
#include <ilviews/graphic.h>

IlvGraphicCallback getCallback() const;
IlSymbol* getCallbackName() const;
void   setCallback(IlvGraphicCallback callback);
void   addCallback(IlvGraphicCallback callback);
void   setCallback(IlvGraphicCallback callback, IlAny data);
void   addCallback(IlvGraphicCallback callback, IlAny data);
void   setCallback(const IlSymbol* callbackName);
void   setCallback(const IlSymbol* callbackName, IlAny data);
void   setCallbackName(const IlSymbol* callbackName);
```

The IlvSimpleGraphic Class

`IlvSimpleGraphic` is a fundamental class inherited from `IlvGraphic`. `IlvSimpleGraphic` implements all functionality of the `IlvGraphic` class and adds to each instance an `IlvPalette` resource used to draw the object. This class lets you carry out operations that access and change graphic properties such as colors, fonts, and patterns that are gathered in an `IlvPalette` instance associated with the graphic object. It also allows you to apply geometric transformations to objects.

`IlvSimpleGraphic` objects contain their own `IlvPalette` object. This means that a graphical object is simultaneously a geometric shape and a set of attributes to display this shape. Thus, from this class you can create new objects needed for your application. Some member functions will be needed, some not. The IBM® ILOG® Views object library contains a large number of such objects, and offers a wide range of solutions for almost all kinds of problems.

Member Functions

The `IlvSimpleGraphic` class includes member functions that allow you to access palette attributes. Every `IlvSimpleGraphic` object has an `IlvPalette` object, which can be shared among objects. Therefore, when you ask an `IlvSimpleGraphic` object to change a graphic property such as its foreground, the following operations are performed:

1. The `IlvDisplay::getPalette` function is used to search for a new `IlvPalette` for the new foreground.
2. The member function `IlvResource::lock` is called for the new palette to increment its reference count.
3. The graphic object's old palette is called.
4. The member function `IlvResource::unLock` is called for the old palette.
5. The new palette is registered as the current palette of the object.

These operations guarantee the sharing of the `IlvPalette`. Users are encouraged to use the same mechanism in the case of `IlvPalette` objects. This is why member functions that can manipulate resources by changing the graphic attributes, such as `IlvGraphic::setForeground`, are defined as virtual functions.

Graphic Attributes

The `IlvSimpleGraphic` constructor needs the `IlvPalette` object from which it is to obtain resources. The `palette` parameter can be set to a specific value or left unspecified, by which it takes the value `0`. When the palette is set to `0`, the default palette of the display object is used. This palette is the one returned by the member function

`IlvDisplay::defaultPalette`. The `palette` parameter is locked when the object is created and unlocked when it is deleted.

Predefined Graphic Objects

This section presents basic classes, all subclasses of `IlvSimpleGraphic`, that provide you with predefined graphic objects.

IlvArc

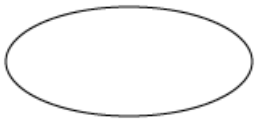
An `IlvArc` object appears as an outlined arc of an ellipse.

IlvFilledArc

An `IlvFilledArc` object appears as a filled arc.

IlvEllipse

An `IlvEllipse` object appears as an outlined ellipse.



IlvFilledEllipse

An `IlvFilledEllipse` object appears as a filled ellipse.

IlvIcon

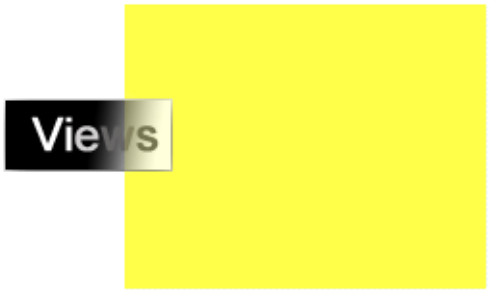
An `IlvIcon` object appears as an image.

IlvZoomableIcon

An `IlvZoomableIcon` object is a kind of `IlvIcon` object that can be zoomed in or reshaped.

IlvTransparentIcon

An `IlvTransparentIcon` object appears as an image that can have transparent areas.



IlvZoomableTransparentIcon

An `IlvZoomableTransparentIcon` object is a version of the `IlvZoomableIcon` object that leaves the background of the image (the `0` bits) unchanged.

IlvLabel

An `IlvLabel` object appears as a single line of text. It cannot be zoomed in nor reshaped.

```
This object is an IlvLabel instance
```

IlvFilledLabel

An `IlvFilledLabel` object appears as a single line of text, drawn on a filled rectangle that exactly fits the bounding box of the text.

```
This is an IlvFilledLabel instance
```

IlvListLabel

An `IlvListLabel` object appears as a vertical list of strings, so that it looks like a series of `IlvLabels`.

```
First element in an IlvListLabel
Second element
Third and final element
```

IlvZoomableLabel

An `IlvZoomableLabel` object acts just like a regular `IlvLabel` object, but any transformation can be applied to it, including zooming.

IlvLine

An `IlvLine` object appears as a straight line between two given points.

IlvArrowLine

An `IlvArrowLine` object appears as a straight line between two given points, with a small arrow head drawn on the line trajectory.

IlvReliefLine

An `IlvReliefLine` object appears as a line with a three-dimensional look. How the `IlvReliefLine` looks depends on the thickness of the line.

IlvMarker

An `IlvMarker` object is drawn as a specific graphic symbol at a given location.

IlvZoomableMarker

An `IlvZoomableMarker` object is a version of the `IlvMarker` object that can be zoomed as follows:

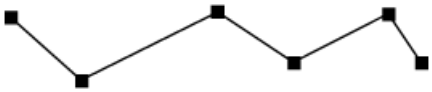
- For zooming out, the current size is reduced to fit the transformed bounding box.
- For zooming in, the current size stays fixed to that specified by the `IlvMarker::setSize` method.

IlvPolyPoints

`IlvPolyPoints` is an abstract class from which is derived every class having shapes composed of several point coordinates.

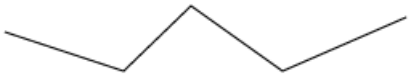
IlvPolySelection

The `IlvPolySelection` class is used to fill squares on all the points of an object of type `IlvPolyPoints`.



IlvPolyline

An `IlvPolyline` object appears as connected segments.



IlvArrowPolyline

An `IlvArrowPolyline` object appears as a polyline and adds one or more arrows to the various lines.

IlvPolygon

An `IlvPolygon` object appears as a filled polygon.

IlvOutlinePolygon

An `IlvOutlinePolygon` object appears as an outlined and filled polygon.

IlvRectangle

An `IlvRectangle` object appears as an outlined rectangle.

> ***Note:*** *Rectangles can be rotated only at 90, 180, 270 and 360 degrees. If you need to rotate a rectangle at other angles, use a polygon instead.*

IlvFilledRectangle

An `IlvFilledRectangle` object appears as a solid rectangle.

IlvRoundRectangle

An `IlvRoundRectangle` object appears as an outlined, round-cornered rectangle.

IlvFilledRoundRectangle

An `IlvFilledRoundRectangle` object appears as a filled, round-cornered rectangle.

IlvShadowRectangle

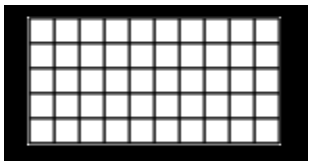
An `IlvShadowRectangle` object appears as a shadowed `IlvFilledRectangle` object.

IlvShadowLabel

An `IlvShadowLabel` object appears as an `IlvShadowRectangle` containing a text string that is clipped by the containing rectangle.

IlvGridRectangle

An `IlvGridRectangle` object appears as a rectangular grid.



IlvReliefRectangle

An `IlvReliefRectangle` object appears as a filled rectangle in relief.

IlvReliefLabel

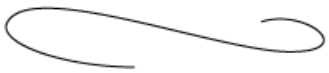
An `IlvReliefLabel` object appears as a relief rectangle that holds a single line of text.

IlvReliefDiamond

An `IlvReliefDiamond` object appears as a filled diamond in relief.

IlvSpline

An `IlvSpline` object appears as an outline Bézier spline.



IlvClosedSpline

An `IlvClosedSpline` object appears as a closed Bézier spline.

IlvFilledSpline

An `IlvFilledSpline` object appears as a filled Bézier spline.

Composite Graphic Objects

Composite graphic object classes and subclasses provide member functions that allow you to reference instances of `IlvGraphic` subtyped objects. You can use these references for:

- Controlling polygon fill. See *Filling Polygons: IlvGraphicPath* on page 55.
- Grouping objects. See *Grouping Objects: IlvGraphicSet* on page 56.
- Modifying properties of one object without duplicating or modifying the object itself. See *Referencing Objects: IlvGraphicHandle* on page 57.

  For example, in an electronic schema displaying a thousand transistors, you would consume much less memory by creating one transistor image and a thousand handle objects to reference it than by creating a thousand separate images.

Filling Polygons: IlvGraphicPath

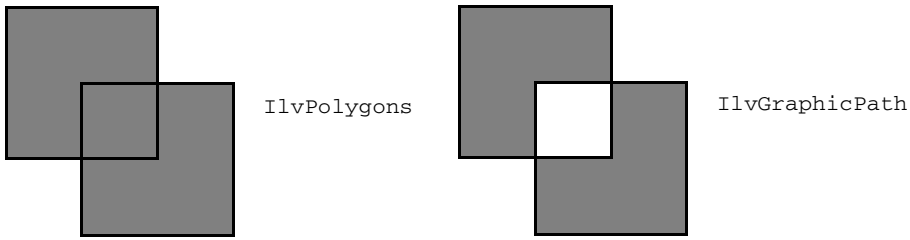
An `IlvGraphicPath` object is a collection of *polypoint* objects, that is, each object consists of a series of points. The polypoint objects are drawn differently depending on the value of the draw rule attribute of the object:

- `IlvStrokeOnly`: Polylines.

- `IlvFillOnly`: Filled polygons.
- `IlvStrokeAndFill`: Both of the above, that is, filled polygons with an outline.

The palette defined by the `IlvSimpleGraphic` superclass is used to draw the outline of the polygons. `IlvGraphicPath` defines a second palette (`backgroundPalette`) to fill them.

While the resources (graphic attributes such as color) used to draw the polypoints for both the `IlvGraphicPath` and `IlvPolygon` functions are the same, the ways in which the shapes affect each other are different. Each polypoint has an influence on the rendering of the other polypoints (not applicable in `IlvStrokeOnly` mode). For example, depending on the position of its points, a polypoint may appear either as an ordinary polygon or as a hole in another polygon:



`IlvGraphicPath` also allows user-specific actions when the polypoints are drawn. This is done by attaching a data structure to the `IlvGraphicPath`.

Note that the bounding box of `IlvGraphicPath` does not take into consideration the bounding box of the graphic object displayed along the path. The `IlvGraphic` is only known to the stepping data structure. However, the `IlvGraphicPath` provides a member function which allows you to extend the bounding box of the graphic path by the given value:

```
void setBBoxExtent(IlUInt extent);
```
Usually you would do the following:

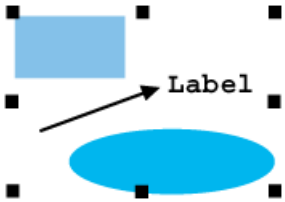
```
grpath->setPathDrawingData(new IlvPathDrawingData(step, obj));
grpath->setBBoxExtent(bboxExtension);
```

where `bboxExtension` is computed from the geometry of `obj` and the way it will be displayed (whether rotations are involved or not).

The diagonal of the object's bounding box is a reasonable value for `bboxExtension`.

Grouping Objects: IlvGraphicSet

An `IlvGraphicSet` object organizes a set of `IlvGraphic` objects.



It implements all geometric and graphic member functions by means of calls to the member functions of the objects that it contains (for example, the `draw` method of `IlvGraphicSet` calls the `draw` methods of the objects contained in the graphic set).

Referencing Objects: IlvGraphicHandle

An `IlvGraphicHandle` object is used to reference an `IlvGraphic` object. The `IlvGraphicHandle` object is called the *handle* object (or simply *handle*), and the `IlvGraphic` object is called the *referenced* object.

Referencing Objects

This relationship enables you to use the handle object to indirectly access the referenced object. Also, the same referenced object can be shared among several handles. Through handle objects, you can thus reproduce a complex graphic object many times by simply creating new handles that all reference the same original object. Since handles consume much less memory than creating new images, using handle objects is very economical.

Owning Objects

You can decide to make a handle object the *owner* of the unique referenced object with which it is associated. If you do this, you should no longer access the referenced object directly, but only through its handle.

When a handle owns its referenced object, a delete operation on the handle removes both the handle and the referenced object. On the other hand, when there is no ownership relationship between the handle and its referenced object, a delete operation removes only the handle, leaving the other graphic object intact.

IlvTransformedGraphic

You can use the specialized graphic handle subclass, `IlvTransformedGraphic`, to display the same object several times with different geometric transformations applied to it.

The `IlvTransformedGraphic` class is derived from the basic *handle* class, `IlvGraphicHandle`. An object of the `IlvTransformedGraphic` type is a kind of handle that is associated with a certain referenced object of the `IlvGraphic` class. An `IlvTransformedGraphic` instance is drawn by applying a graphic transformation to its referenced object.

Object geometry is disturbed by various transformations because of rounding errors. To avoid this, the object can be associated with `IlvTransformedGraphic`.

IlvFixedSizeGraphic

You can use the specialized graphic handle subclass, `IlvFixedSizeGraphic`, to always display an object at the same size. For example, suppose there is a map displayed with an `IlvButton` object used for quitting the map viewer. When the map is zoomed and unzoomed, the button, of course, should stay the same size. To accomplish this, you can reference the button with the specialized `IlvGraphicHandle` object `IlvFixedSizeGraphic`.

The `IlvFixedSizeGraphic` class is derived, like `IlvTransformedGraphic`, from the `IlvGraphicHandle` class. An `IlvFixedSizeGraphic` object is a kind of handle, which is associated with a certain referenced object of the `IlvGraphic` class. An `IlvFixedSizeGraphic` instance is drawn by applying a graphic transformation to its referenced object so that the referenced object can never change its visible size.

Whatever the transformation applied, the object keeps the same dimensions and a constant offset relative to a reference point. These values are internally computed by the IBM ILOG Views libraries or are specified by the user.

IlvGraphicInstance

You can use the specialized graphic handle subclass, `IlvGraphicInstance`, to encapsulate an object with a graphic resource modification.

The `IlvGraphicInstance` references another graphic object so that it can be drawn with another palette attribute. You can use an optional `IlvTransformer` to apply a geometric transformation to this object.

Other Base Classes

Some subclasses of `IlvSimpleGraphic` form base classes for more complex graphic objects.

IlvGauge

Gauges are graphic objects that provide a representation of a certain value contained between a minimum and a maximum value. `IlvGauge` is the main abstract class from which all gauge objects derive.

IlvScale

`IlvScale` is an abstract class from which are derived all instances of scale object classes. It manages the basic required information concerning scales.

IlvGadget

The `IlvGadget` class is the base class for all the IBM ILOG Views Gadgets package classes. It implements all the basic functionalities of gadgets by providing the necessary parameters to create a graphic object with a palette allowing shadowing management.

For details on gadgets, see the Gadgets documentation.

IlvGroupGraphic

`IlvGroupGraphic` is a graphic object class used to display and manipulate a set of graphic objects as a group. This class is used in the IBM ILOG Views Prototypes package.

For details, see the Prototypes documentation.

IlvMapxx

Some `IlvMapxx` classes are subclasses of `IlvSimpleGraphic`, providing various graphic services for the IBM ILOG Views Maps package, such as scales (`IlvMapScale`, `IlvMapDefaultScaleBar`, and `IlvMapDefaultNeedle`) .

For details on all mapping classes, see the Maps documentation.

Creating a New Graphic Object Class

Here is an example of how you can subtype graphic objects, creating new graphic object classes.

The Example: ShadowEllipse

In this example a new graphic object, `ShadowEllipse`, will be created, which inherits from `IlvSimpleGraphic`:

The `ShadowEllipse` object is a normal `IlvEllipse` object with a drop shadow underneath.

This example shows how you can design such an object from scratch by implementing a subtype of the `IlvSimpleGraphic` class, which is the procedure most commonly used. We

will show how to implement member functions that deal with geometric properties and drawing, how to manipulate the object's palette, and how to make this object persistent.

Stepping through the Example

The example is developed in:

- *Basic Steps to Subtype a Graphic Object*
- *Redefining IlvGraphic Member Functions*
- *Creating the Header File*
- *Implementing the Object Functions*
- *Updating the Palettes*
- *Saving and Loading the Object Description*

Basic Steps to Subtype a Graphic Object

To create derived classes of the `IlvGraphic` class, you:

1. Create a header file that declares the new class and the necessary overloaded member functions. Not every member function needs to be overloaded.
2. Add the `DeclareTypeInfo();` statement in the class definition.

   This creates the necessary fields and member function declarations for input/output operations and class hierarchy information.
3. Add the `DeclareIOConstructors(ShadowEllipse);` statement in the class declaration, which declares two additional constructors:

- The following constructor initializes a new `ShadowEllipse` graphic object, which is a copy of `source`:

```
ShadowEllipse(const ShadowEllipse& source);
```

- The following constructor initializes a new `ShadowEllipse` graphic object from the parameters read in the `inputfile`:

```
ShadowEllipse(IlvInputFile& inputfile,
              IlvPalette*   palette = 0);
```

4. Create an implementation file (usually `class.cpp`) to implement the member functions that you need. Outside the body of a function, add a call to the two following macros:

- `IlvRegisterClass`, which updates the class hierarchy information.
- `IlvPredefinedIOMembers`, which is used to define the member functions `copy` and `read`.

Redefining IlvGraphic Member Functions

The following member functions of `IlvGraphic` must always be redefined (these are the member functions that make `IlvGraphic` an abstract class):

```
virtual void draw(IlvPort* dst,
                  const IlvTransformer* t = 0,
                  const IlvRegion* clip = 0) const;
virtual void boundingBox(IlvRect& bbox,
                         const IlvTransformer* t = 0) const;
virtual void applyTransform(const IlvTransformer* t);
virtual void write(IlvOutputFile&) const;
```

The other member functions—such as `IlvGraphic::move`, `IlvGraphic::resize`, `IlvGraphic::rotate`, and `IlvGraphic::contains`—have a default implementation from the `IlvGraphic` class. That is, `IlvGraphic::resize` is implemented by means of a call to your `applyTransform` function, and so on.

If the new class has a parent that defines some of these member functions, you can simply inherit the functions from this parent.

Creating the Header File

For this example, you create a header file that declares the new class and the necessary overloaded member functions.

The header file `shadellp.h` contains the following lines:

```
#define DefaultShadowThickness 4

class  ShadowEllipse
: public IlvSimpleGraphic {
public:
    ShadowEllipse(IlvDisplay*    display,
                  const IlvRect& rect,
                  IlUShort      thickness = DefaultShadowThickness,
                  IlvPalette*    palette   = 0)
    : IlvSimpleGraphic(display, palette),
      _rect(rect), _thickness(thickness)
    {
        _invertedPalette = 0;
        computeInvertedPalette();
    }
    ~ShadowEllipse();

    virtual void draw(IlvPort*, const IlvTransformer* t = 0,
                      const IlvRegion* clip = 0) const;
    virtual IlBoolean contains(const IlvPoint& p,
                               const IlvPoint& tp,
                               const IlvTransformer* t) const;
    virtual void boundingBox(IlvRect& rect,
                             const IlvTransformer* t = 0) const;
    virtual void applyTransform(const IlvTransformer* t);
    IlUShort getThickness() const
```

```
        { return  _thickness; }
    void      setThickness(IlUShort thickness)
        { _thickness = thickness; }

    virtual void setBackground(IlvColor* c);
    virtual void setForeground(IlvColor* c);
    virtual void setMode(IlvDrawMode m);
    virtual void setPalette(IlvPalette* p);

    DeclareTypeInfo();
    DeclareIOConstructors(ShadowEllipse);
protected:
    IlvRect     _rect;
    IlUShort   _thickness;
    IlvPalette* _invertedPalette;
    void computeInvertedPalette();
};
```

This object, like a few others in the standard IBM ILOG Views library, makes use of two different `IlvPalette` objects. This is a common practice when you want your object to be very efficient in terms of drawing time, since you do not need to create dummy palette objects when drawing the ellipse itself and its shadow.

The `ShadowEllipse` class defines the member functions `draw`, `contains`, and `boundingBox`. It also defines the necessary palette-management related member functions to update both the standard palette object (the one stored in the `IlvSimpleGraphic` class) and the new one, `_invertedPalette`.

No input/output member functions are declared in this synopsis. In fact, they are declared by the `DeclareTypeInfo` macro that declares them as external. These member functions are `read`, `write`, and `copy`. They have no default implementation, and you must provide a version of these for each of your subclasses of the `IlvGraphic` class. There is a second version of this macro, called `DeclareTypeInfoRO`, that does not declare the member function `write`, if you know this object type will never be saved.

Implementing the Object Functions

For this example, you created a header file that declares the new class and the necessary overloaded member functions.

This section explains the code for the functions implemented in the file `shadellp.cpp`.

computeInvertedPalette Member Function

```
void
ShadowEllipse::computeInvertedPalette()
{
    IlvPalette* newPalette = getDisplay()->getInvertedPalette(getPalette());
    newPalette->lock();
    if (_invertedPalette)
        _invertedPalette->unLock();
    _invertedPalette = newPalette;
}
```

The member function `computeInvertedPalette` computes the inverted palette from the one we get by means of the call to the member function `getPalette`. We create this inverted palette, unlock the previous one (if there was one), and lock the new one.

This function is called whenever the original palette is modified (by overloading the appropriate member functions), and when the object is originally created.

Creating this second palette may seem strange. In the member function `draw`, the second palette is used only when calling two `IlvDisplay` drawing member functions. Another method could have been to call the member function `IlvPalette::invert` before we call these member functions and then bring the palette to its original state by another call to `IlvPalette::invert`. Developing with IBM ILOG Views shows that this is not an efficient way of manipulating objects. Palette management is one of the very efficient tasks performed by IBM ILOG Views and you should not hesitate to use palette management when needed.

Destructor

```
ShadowEllipse::~ShadowEllipse()
{
    _invertedPalette->unLock();
}
```

In the destructor, we need to release the inverted palette to the display, so it can be deleted if not used by any other object.

draw Member Function

The member function `draw` fills the two ellipses, then draws the topmost ellipse border. The global bounding rectangle (`_rect`) actually covers both ellipses.

Now we can show the member function `draw`. It demonstrates that drawing an object is merely a call to some of the primitive member functions of the `IlvDisplay` class:

```
void
ShadowEllipse::draw(IlvPort* dst, const IlvTransformer* t,
                    const IlvRegion* clip) const
{
    // Transform the bounding rectangle ___________________________________
    IlvRect rect = _rect;
    if (t)
        t->apply(rect);
```

```
    // Store both the display and palette ____________________________
    IlvPalette* palette = getPalette();

    // Find a correct value for thickness ____________________________
    IlUShort thickness = _thickness;
    if ((rect.w() <= thickness) || (rect.h() <= thickness))
        thickness = IlMin(rect.w(), rect.h());

    // Compute actual shadow rectangle _______________________________
    rect.grow(-thickness, -thickness);
    IlvRect shadowRect = rect;
    shadowRect.translate(thickness, thickness);

#if defined(USE_2_PALETTES)
    // Set the clipping region for both palettes _____________________
    if (clip) {
        palette->setClip(clip);
        _invertedPalette->setClip(clip);
    }
   // Fill shadow Ellipse ____________________________________________
    dst->fillArc(palette, shadowRect, 0., 360.);

    // Fill inverted Ellipse _________________________________________
    dst->fillArc(_invertedPalette, rect, 0., 360.);

    // Draw ellipse __________________________________________________
    dst->drawArc(palette, rect, 0., 360.);
    if (clip) {
        palette->setClip();
        _invertedPalette->setClip();
    }
#else  /* !USE_2_PALETTES */
    // Set the clipping region for both palettes _____________________
    if (clip)
        palette->setClip(clip);

    // Fill shadow ellipse ___________________________________________
    dst->fillArc(palette, shadowRect, 0., 360.);

    // Compute inverted palette and fill inverted ellipse ____________
    palette->invert();
    dst->fillArc(palette, rect, 0., 360.);
    palette->invert();
 // Draw elliptic border __________________________________________
    dst->drawArc(palette, rect, 0., 360.);

    // Set the clipping region for both palettes _____________________
    if (clip)
        palette->setClip();
#endif /* !USE_2_PALETTES */
}
```

We do not need the transformer `t` to perform our drawing work, because we want the thickness to be the same whatever the transformation is.

The `clip` parameter can be used when complex drawing is to take place (which is not the case here). You must reset the clipping region of all affected palettes to an empty region before you return from this function.

boundingBox Member Function

The member function `boundingBox` simply transforms the global bounding rectangle.

```
void
ShadowEllipse::boundingBox(IlvRect& rect,
                           const IlvTransformer* t) const
{
    rect = _rect;
    if (t)
        t->apply(rect);
}
```

Note: *The bounding box must contain the complete drawing to avoid erasing errors.*

contains Member Function

The member function `contains` returns `IlTrue` if the point is inside one of the two ellipses. All the coordinates are given in terms of the view's coordinate system.

```
static IlBoolean
IsPointInEllipse(const IlvPoint& p, const IlvRect& bbox)
{
    if (!bbox.contains(p))
        return IlFalse;
    IlUInt rx = bbox.w() / 2,
           ry = bbox.h() / 2;
    IlUInt dx = (p.x() - bbox.centerx()) * (p.x() - bbox.centerx()),
           dy = (p.y() - bbox.centery()) * (p.y() - bbox.centery()),
    rrx = rx*rx,
    rry = ry*ry;
    return (rrx * dy + rry * dx <= rrx * rry) ? IlTrue : IlFalse;
}
IlBoolean
ShadowEllipse::contains(const IlvPoint&, const IlvPoint& tp,
                           const IlvTransformer* t) const
{
    IlvRect rect = _rect;
    if (t)
        t->apply(rect);
    if ((rect.w() <= _thickness) || (rect.h() <= _thickness))
        return IsPointInEllipse(tp, rect);
    else {
        rect.grow(-_thickness, -_thickness);
        IlvRect shadowRect = rect;
        shadowRect.translate(_thickness, _thickness);
        return (IlBoolean)(IsPointInEllipse(tp, rect) ||
                           IsPointInEllipse(tp, shadowRect));
    }
}
```

You can see that `contains` calls the static function `IsPointInEllipse` which tests whether the point parameter is inside the ellipse defined by the rectangle parameter.

applyTransform Member Function

The applyTransform member function applies the transformer to the rectangle of the graphic.

```
void ShadowEllipse::applyTransform(const IlvTransformer* t)
{
 if (t)
   t->apply(_rect);
}
```

Updating the Palettes

To make sure that both palettes are updated when modifications are applied to the original one, we need to overload the following member functions.

```
void
ShadowEllipse::setBackground(IlvColor* color)
{
    IlvSimpleGraphic::setBackground(color);
    computeInvertedPalette();
}

// --------------------------------------------------------------------------
void
ShadowEllipse::setForeground(IlvColor* color)
{
    IlvSimpleGraphic::setForeground(color);
    computeInvertedPalette();
}

// --------------------------------------------------------------------------
void
ShadowEllipse::setMode(IlvDrawMode mode)
{
    getPalette()->setMode(mode);
    _invertedPalette->setMode(mode);
}

// --------------------------------------------------------------------------
void
ShadowEllipse::setPalette(IlvPalette* palette)
{
    IlvSimpleGraphic::setPalette(palette);
    computeInvertedPalette();
}
```

Saving and Loading the Object Description

Now come the input/output member functions that we have declared in the class synopsis using the `DeclareTypeInfo` macro.

copy and read Member Functions

Another macro can be used to define the member functions `copy` and `read`:

```
IlvPredefinedIOMembers(IlvShadowEllipse);
```

This macro must be used in the implementation file, outside any function definition block, just like `IlvRegisterClass`.

It is equivalent to:

```
IlvGraphic*
ShadowEllipse::read(IlvInputFile& input, IlvPalette* palette)
{
    return new ShadowEllipse(input, palette);
}
IlvGraphic*
ShadowEllipse::copy() const
{
    return new ShadowEllipse(*this);
}
```

The static member function `read` calls the class reading constructor and returns the new instance. The macro `DeclareIOConstructors` declares the reading and copying constructors in the header file. The definition of these constructors must be written in this way in the implementation file:

```
ShadowEllipse::ShadowEllipse(IlvInputFile& f,
                             IlvPalette* pal)
: IlvSimpleGraphic(f, pal),
  _rect(),
  _thickness(0)
{
    int thickness;
    f.getStream() >> _rect >> thickness;
    _thickness = (IlvDim)thickness;
    _invertedPalette = 0;
    computeInvertedPalette();
}
```

The above constructor calls the superclass reading constructor, which reads the superclass-specific information from the stream object. Then the subclass can read its own information.

The member function `copy` creates a copy of the `IlvShadowEllipse` class, calling the class copy constructor:

```
ShadowEllipse::ShadowEllipse(const ShadowEllipse& source)
: IlvSimpleGraphic(source),
  _rect(source._rect),
  _thickness(source._thickness)
{
    _invertedPalette = source._invertedPalette;
    _invertedPalette->lock();
}
```

write Member Function

The member function `write` writes the dimensions of the rectangle and the thickness of the shadow to the given `ostream` output stream:

```
void
ShadowEllipse::write(IlvOutputFile& f) const
{
    f.getStream() << _rect << IlvSpc() << (int)_thickness;
}
```

This `write` method is special because the `IlvSimpleGraphic` superclass has no information to write. It is better to call the superclass `write` method to be consistent with the `read` method, if the superclass has information to write. Here is an example of a usual `write` method:

```
void
IlvRoundRectangle::write(IlvOutputFile& os) const
{
    IlvRectangle::write(os);
    os.getStream() << IlvSpc() << _radius;
}
```

IlvRegisterClass Macro

```
IlvRegisterClass(ShadowEllipse, IlvSimpleGraphic);
```

Outside of the body of any function, we have to register the class `IlvShadowEllipse` as a subclass of the class `IlvSimpleGraphic`.

C H A P T E R

3

Graphic Resources

The classes that implement graphic resources are `IlvResource` and its subclasses. There are five basic kinds of graphic drawing resources (subclasses of `IlvResource`) in IBM® ILOG® Views: *color*, *line style*, *pattern*, *color pattern*, and *font*, each supported by its corresponding class. Another subclass of `IlvResource`, the `IlvPalette` class, manages a group of resources. Additionally, `IlvQuantizer` is the abstract base class of all color conversion classes.

- *IlvResource: The Resource Object Base Class*
- *IlvColor: The Color Class*
- *IlvLineStyle: The Line Style Class*
- *IlvPattern and IlvColorPattern: The Pattern Classes*
- *IlvFont: The Font Class*
- *IlvCursor: The Cursor Class*
- *Other Drawing Parameters* describes additional settings that you control via the palette.
- *IlvPalette: Drawing Using a Group of Resources*
- *IlvQuantizer: The Image Color Quantization Class*

IlvResource: The Resource Object Base Class

All of the drawing member functions of the `IlvPort` class take a parameter of the `IlvPalette` type, which is a subclass of `IlvResource`.

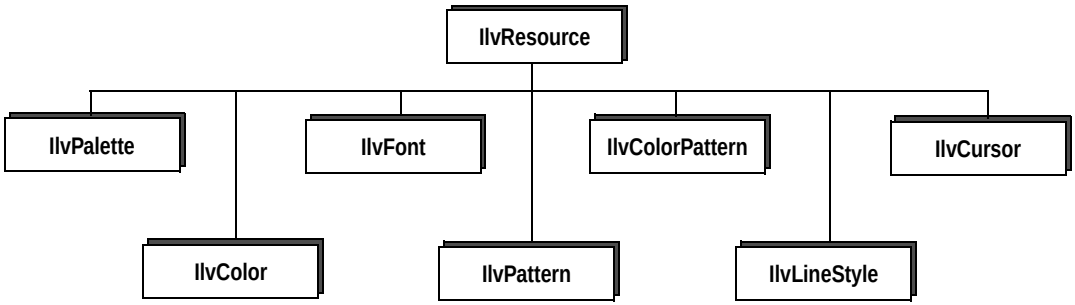


Figure 3.0 *IlvResource Hierarchy*

Resources are described further in the topics:

- *Predefined Graphic Resources*
- *Named Resources*
- *Resource Creation and Destruction: lock and unLock*

Predefined Graphic Resources

The following list summarizes the `IlvDisplay` member functions that produce the predefined graphic resources:

```
IlvColor*     defaultBackground() const;
IlvColor*     defaultForeground() const;
IlvFont*      defaultFont() const;
IlvLineStyle* defaultLineStyle() const;
IlvPattern*   defaultPattern() const;
IlvCursor*    defaultCursor() const;
```

- The foreground color default value in a palette is the color returned by `IlvDisplay::defaultForeground`.
- The background color default value in a palette is the color returned by `IlvDisplay::defaultBackground`.
- The default font value in a palette is the font returned by `IlvDisplay::defaultFont`.
- The default line style value in a palette is the line style returned by `IlvDisplay::defaultLineStyle`.

- The default fill pattern value in a palette is the pattern returned by `IlvDisplay::defaultPattern`.
- The default cursor value in a palette is the pattern returned by `IlvDisplay::defaultCursor`.

The graphic resource classes are subclasses of `IlvResource`. For details on these subclasses see:

- *IlvColor: The Color Class*
- *IlvLineStyle: The Line Style Class*
- *IlvPattern and IlvColorPattern: The Pattern Classes*
- *IlvFont: The Font Class*
- *IlvCursor: The Cursor Class*

In IBM ILOG Views, the drawing resources are assembled into an object of the `IlvPalette` class, also a subclass of `IlvResource`. For more details on palettes see:

- *IlvPalette: Drawing Using a Group of Resources*

Named Resources

You can assign specific names to resources by using the `IlvResource` member functions:

```
void setName(const char* name);
const char* getName() const;
```

> ***Note:*** `IlvFont` *and* `IlvColor` *make private use of their name field, so these classes restrict the use of* `IlvResource::setName`*.* `IlvFont` *disables the use of* `IlvResource::setName`*, and* `IlvColor` *only allows the renaming of mutable colors. Non-mutable colors either have a predefined name or get a default name based on their RGB values.*

Resource Creation and Destruction: lock and unLock

Because the creation of graphic resources is generally memory intensive on most graphic systems, IBM® ILOG® Views implements a caching mechanism to minimize graphic resource allocation.

Resource objects are maintained by the `IlvDisplay` instances of your application. They should normally not be created and destroyed using the operators `new` and `delete`. Instead, IBM ILOG Views provides the following member functions:

- `IlvDisplay` methods `getXXX`, where `XXX` stands for a resource class name without the `Ilv` prefix (for instance, `IlvDisplay::getColor`, `IlvDisplay::getFont`, and so on).
- Methods `IlvResource::lock` and `IlvResource::unLock` respectively increment and decrement the internal reference count of the resource. When this count reaches zero, the resource is deleted.

Locking and Unlocking Procedures

Graphic resources should be used in the following manner:

1. Request your `IlvDisplay` instance to allocate a resource for you. If this resource already exists in the system (for instance, the color you query for is already in use in a palette somewhere), no further allocation is done, and the existing resource is returned.
2. Inform IBM ILOG Views that this resource must be kept safe by calling `IlvResource::lock`, then use that resource.
3. Use `IlvResource::unLock` to let IBM ILOG Views know that you have finished using the resource.

Resource management is closely concerned with the ways in which you lock and unlock your resources. Whenever you need a specific resource in one of your persistent objects, you should use this mechanism to make sure that it will stay safe within your `IlvDisplay` instance. If your application needs more than one instance of `IlvDisplay`, you have to create resources within each environment, since resources cannot be shared between the different `IlvDisplay` contexts.

During the lifetime of a resource, the number of calls to `IlvResource::lock` must exactly match the number of calls to `IlvResource::unLock`. If there are more calls to `lock`, the resource remains allocated even if it is no longer in use, and therefore limits your application's requirements. If there are more calls to `unlock`, the application may crash because of a memory error.

Rules for Locking and Unlocking

You should follow these rules for locking and unlocking graphic resources:

- Once you get a resource, lock it, use it, and unlock it when you are done.
- You should not unlock a resource that you have not locked yourself, unless you are sure that your operation is correct.
- You should never use a resource after you have unlocked it, just like you should never use a pointer after you have freed it; `IlvResource::unLock` potentially means `delete`.
- There are times when you do not need to lock and unlock a resource. For instance, if you get the foreground color of an object and pass it to another object that will lock it. In such cases, locking and unlocking the resource is not necessary, but it does no damage either.

IlvColor: The Color Class

Color is described in the topics:

- *Color Models*
- *Using the IlvColor Class*
- *Converting between Color Models*
- *Computing Shadow Colors*

Color Models

The description of a color, in IBM® ILOG® Views, is stored in an instance of `IlvColor`.

RGB

Colors can be handled in IBM ILOG Views using the familiar RGB (red/green/blue) system. In this system, a color is entirely defined through its three component values: `red`, `green` and `blue`. These values are stored as unsigned 16-bit numbers. For example, *black* is defined with its three components set to zero, and *white* has its three components set to 65535.

HSV

Alternatively, you can use the HSV (hue/saturation/value of luminosity) model, shown here:

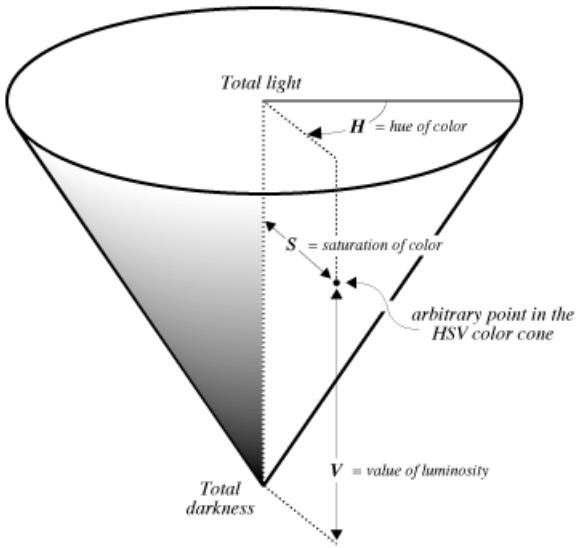


Figure 3.1 *Hue, Saturation, and Value of Luminosity (HSV) Model*

The preceding figure illustrates a mathematical model based upon three parameters: `H` (hue), `S` (saturation) and `V` (value of luminosity). Following are their possible values:

- **H** The hue parameter, `H`, is an angle from 0 to 360°. For fixed values of the `S` and `V` parameters (somewhere towards their upper limits), varying the angle `H` through a full circle would take you through the entire spectrum of colors.
- **S** For a given `H` parameter, varying the `S` value changes the vividness of the color. At the vertical axis of the cone, where `S` is zero, there is a total absence of any chromatic intensity, which means that there is a shade of *gray*. On the outer surface of the cone, where `S` has the value 1.0, the intensity is maximal, which means that colors are as vivid as possible.
- **V** The third parameter, `V`, determines the quantity of light in which the colors are bathed, in other words, the *brightness* of the spectrum obtained by varying the angle `H` through a full circle. The bottom point of the cone, where `V` is zero, represents the color *black*. As we ascend, the spectrum of the hue circle becomes increasingly brighter. The top of the vertical axis, where `V` has the value 1.0, represents the color white.

Using the IlvColor Class

In IBM ILOG Views, the `IlvColor` class lets you manipulate both the RGB and the HSV color models. You obtain a color by requesting your `IlvDisplay` object to get it for you.

Colors are generally stored in a color table—sometimes referred to as a lookup table—that is maintained internally by your `IlvDisplay` object. You can obtain the index of a color object (a long unsigned integer) by means of the member function `IlvColor::getIndex`. You can then use this number for remapping processes involving color bitmaps when you want to map pixel values to color objects.

Color Name

Colors always have a name. IBM ILOG Views has a predefined set of color names. The naming mechanism comes from the X Window color-naming scheme. Each name in this set is associated with a specific RGB triplet. If a color is not specified by a predefined name but by an RGB value, this color gets a default name of the following form: `"#RRRRGGGGBBBB"`, where each of the red, green and blue values is represented by four hexadecimal digits.

Only mutable colors can be renamed, however. The name of a static color cannot be modified. It either belongs to the set of predefined color names or is computed from the RGB values defining the color.

New Colors

`IlvColor` does not have a public constructor; you must get the colors from the display. Several `IlvDisplay::getColor` member functions enable you to obtain a new color, specifying either RGB values, HSV values, or a name. You can also indicate if the color must be mutable or not. If a problem arises making it impossible to create the desired color, these member functions return `0`.

The `IlvDisplay` class provides two member functions returning internal resources that are often used as callback values for unspecified colors. These functions are:

```
IlvColor* defaultForeground();
IlvColor* defaultBackground();
```

Usually, these foreground and background colors are black and gray respectively. They can easily be set to whatever colors you like by means of your display system resource mechanism.

Mutable Colors

An IBM ILOG Views color can be either *static* (impossible to modify after its creation) or *mutable*. In the latter case, you can use modifier member functions that are set to dynamically modify a color, even when there are drawings with this color on the screen.

Mutable colors can be renamed using the method `IlvResource::setName`. Unlike static colors, mutable colors are not transparently shared. The method `IlvDisplay::getColor` always creates a new object if the parameter `mutable` is `IlTrue`. Mutable colors are more costly, in terms of internal resource management.

Converting between Color Models

Two global functions are available to convert color values from the RGB system to HSV, and the reverse.

- `IlvRGBtoHSV`
- `IlvHSVtoRGB`

Computing Shadow Colors

Use the `IlvComputeReliefColors` global function to compute the colors that make a shadow effect.

IlvLineStyle: The Line Style Class

You can create your own line styles by requesting `IlvLineStyle` resource objects from your `IlvDisplay` object and specifying the way dashes are to be drawn.

A line style is an array of unsigned characters returned by the member function `IlvLineStyle::getDashes`. The length of the array is returned by the `IlvLineStyle::getCount` member function. This array must not be modified or deleted by the user.

Starting with the "pen down," IBM® ILOG® Views draws the number of foreground-colored pixels that is indicated by the first element of the `IlvLineStyle::getDashes` array. Then, the second element indicates the number of pixels to be skipped before the drawing starts up again until the array is completely read. Then, the loop begins again. The `IlvLineStyle::getOffset` member function returns the number of pixels to be skipped before the loop restarts.

New Line Styles

You can create your own line styles by requesting `IlvLineStyle` resource objects from your `IlvDisplay` object and specifying the way dashes are to be drawn.

Line styles can be named. To obtain a new line style, use the following member functions of the class `IlvDisplay`:

```
IlvLineStyle* getLineStyle(IlUShort           count,
                           const unsigned char* dashes,
                           IlUShort           offset = 0);
IlvLineStyle* getLineStyle(const char* name)    const;
```

The class `IlvDisplay` provides a set of predefined line styles that you can obtain using their name:

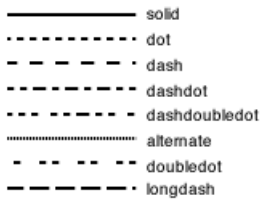


Figure 3.2 *Line Styles*

IlvPattern and IlvColorPattern: The Pattern Classes

A pattern can be of any size and either monochrome, defined with only one bit per pixel, or colored. For details see:

- *Monochrome Patterns*
- *Colored Patterns*

Monochrome Patterns

Monochrome patterns are handled by the `IlvPattern` class. Two constructors are provided that you can use depending on the data available:

```
IlvPattern(IlvDisplay*    display,
           IlvDim         w,
           IlvDim         h,
           unsigned char* data);
IlvPattern(IlvBitmap*     bitmap);
```

The first constructor initializes a new `IlvPattern` object with a pattern `w` pixels wide and `h` pixels high, filled in by the data stored in the `data` array of bit values. The pixel values are packed into 16-bit words from left to right in a most-significant-bits-first manner, and each scan line, stored from top to bottom, must be padded to 16 bits.

The second constructor initializes a new `IlvPattern` object from the given `bitmap` monochrome image.

To obtain a previously defined pattern, use the member function `IlvDisplay::getPattern`.

Other patterns are predefined within IBM® ILOG® Views, which you can access by name.

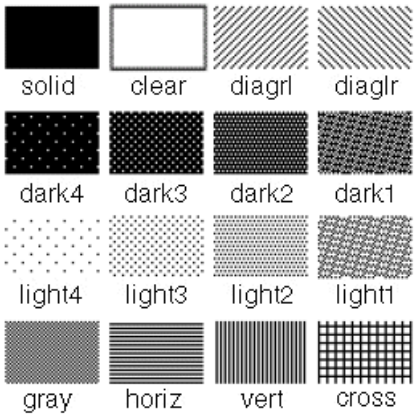


Figure 3.3 *Predefined Pattern Resources*

Colored Patterns

Patterns can also be colored and are represented by instances of the `IlvColorPattern` class.

IlvFont: The Font Class

A text string is drawn with specific spacing values as in the following illustration:

Figure 3.4 *Spacing Values*

You can get the parameters of an `IlvFont` object by using the member functions `IlvFont::getFamily`, `IlvFont::getSize`, `IlvFont::getStyle`, and `IlvFont::getFoundry`.

The member functions `IlvFont::ascent`, `IlvFont::descent`, and `IlvFont::height` return the font metrics.

You can also get the metrics of a specific string by means of calls to the member functions `IlvFont::stringWidth`, `IlvFont::stringHeight`, and `IlvFont::sizes`.

The member function `IlvFont::isFixed` returns `IlTrue` if the font object has a fixed width for all characters (which is not the case in the above figure).

You can also obtain the width of the narrowest and widest characters of this font by means of the width values returned by the two member functions `IlvFont::maxWidth` and `IlvFont::minWidth`. When both return the same value, `IlvFont::isFixed` returns `IlTrue`.

Additional details of the font class are given in:

- *New Fonts*
- *Font Names*

New Fonts

`IlvFont` does not have a public constructor. New fonts must be obtained from the display using one of the two member functions `IlvDisplay::getFont`. You can specify a font name or a set of font characteristics:

- Family
- Size
- Style
- Foundry

Font Names

All fonts have a name. When a font is not created using a valid font name but a set of values— family, size, style and foundry—IBM ILOG Views computes from these values a name of the form:

```
"%family-size-style-foundry"
```

where:

- `family` is the string specified as the parameter family.
- `size` is the ASCII representation of the parameter size.
- `style` is a combination of the letters `B`, `I` and `U`, standing respectively for bold, italic and underlined (upper and lower case do not matter). This field can be empty, in which case the normal style is assumed.
- `foundry` is an optional string. It often identifies the company that designed the font. This field is seldom specified. When it is ignored, the trailing '`-`' can be omitted too.

Fonts cannot be renamed.

Following are examples of syntactically well-formed IBM ILOG Views font names (which are not necessarily valid font names in the sense that they may not exist on all platforms):

- `"%helvetica-12-"`
- `"%time-12-BU"`
- `"%courier-14-i-adobe"`
- `"%terminal-11--bitstream"`

IlvCursor: The Cursor Class

The IBM® ILOG® Views cursor is an icon appearing on the screen that follows every mouse movement. Cursors are maintained in IBM ILOG Views by means of the `IlvCursor` class. Other cursors are predefined, which you can access by name.

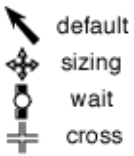


Figure 3.5 *Predefined Cursors and their Names*

Other Drawing Parameters

The following attributes affect drawing operations and are used in the `IlvPalette` class: *Line Width*, *Fill Style*, *Fill Rule*, *Arc Mode*, *Draw Mode*, *Alpha Value*, and *Anti-Aliasing Mode*.

These attributes are represented by C++ type definitions. Although they cannot be called "resources," as they are not represented by subclasses of `IlvResource`, they operate in conjunction with the graphic resources to define the drawing attributes of IBM® ILOG® Views drawings.

Line Width

The line width is an unsigned short integer. Zero (`0`) is a valid value, producing a line whose width is such that it can be drawn as thin and rapidly as possible.

Fill Style

The most complex of the graphic resources is that of patterns, where there are the simple masking patterns of the monochrome (two-color) domain and the rich pixel patterns of color. This aspect is referred to as the *fill style*.

The fill style indicates the way in which patterns are used to fill shapes. There are three possible cases represented in the `IlvFillStyle` enumeration type. A monochrome pattern is used when the fill mode is `IlvFillPattern` or `IlvFillMaskPattern`. Its value is an instance of `IlvPattern`, whether created by the user or returned by specific member functions of the `IlvDisplay` class. Color pattern refers to what is used to fill shapes when the fill mode is `IlvFillColorPattern`.

IlvFillPattern

With `IlvFillPattern` a shape is filled by being copied with the chosen pattern. In an IBM ILOG Views object, there is a pattern property, which refers to an object of the `IlvPattern` class. To fill a shape with a given pattern:

- Each "`0`" pixel in the relevant `IlvPattern` object produces a colored pixel with the current background color
- Each "`1`" pixel in the `IlvPattern` object produces a colored pixel with the current foreground color.

This is the default value of the fill-style property of an `IlvPalette` object.

IlvFillMaskPattern

`IlvFillMaskPattern` is similar to the `IlvFillPattern` style, except that "`0`" pixels in the relevant `IlvPattern` object have no effect upon the corresponding pixels in the destination port. That is, the drawing masks its destination.

IlvFillColorPattern

In the case of `IlvFillColorPattern`, the pattern used to fill a shape is indicated, not by the pattern property of the `IlvPalette` object, but rather by its colored pattern property. It is used when you wish to fill a region with a full-color pattern; that is, an actual object of the `IlvColorPattern` class. The pattern property plays no role in the case of this filling mode.

Fill Rule

This attribute indicates how self-intersecting polygons are filled, since there is an ambiguity concerning what is meant by "fill" in the case of such surfaces.

The fill rule indicates which points are to be considered as inside a filled polygon, depending on the count of crossing segments that define the shape of the area to be filled. The `IlvFillRule` provides two possibilities:

- `IlvEvenOddRule` According to this rule, in the case of the complex polygon shown below, the central area of the star is not considered to lie inside the polygon, and therefore, is not filled. This is the default value.
- `IlvWindingRule` According to this rule, the central area of the star is considered to lie inside the polygon, and therefore, is filled.

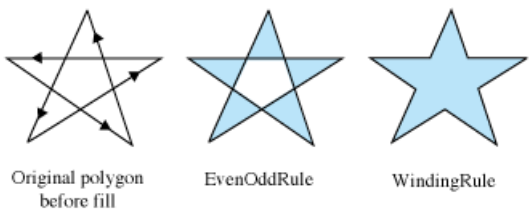


Figure 3.6 `IlvFillRule`

Arc Mode

The arc mode indicates the way to close arcs in order to fill them, that is, the way in which filled arcs are to be drawn: either by radii that form a wedge-shaped “pie” or by a simple “chord” line segment. There are two possible cases that are handled by the `IlvArcMode` enumeration type.

- `IlvArcPie` The arc is closed by adding two lines, from the center of the complete circle to the start and end points of the arc. This is the default mode.
- `IlvArcChord` The arc is closed by adding a line from the start point to the end point.

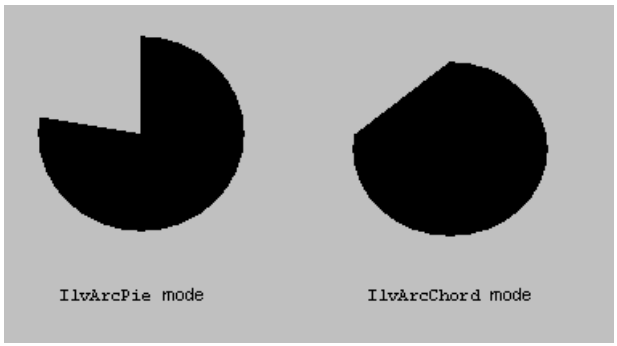


Figure 3.7 *Arc Modes*

Draw Mode

The draw mode specifies the operation to be performed on pixels when they are sent to the destination port. The operation is the one that affects the destination pixel value when the source pixel value is to be drawn at that place. The draw mode has several possible values, which are handled by the `IlvDrawMode` enumeration type. Except for the `IlvModeXor` value, used in temporary drawings, these types produce no significant graphic result when drawing in color.

- `IlvModeSet` The resulting pixel is a copy of the source pixel.
- `IlvModeOr` The resulting pixel is the result of an `OR` operation on the source and destination pixels.
- `IlvModeAnd` The resulting pixel is the result of an `AND` operation on the source and destination pixels.
- `IlvModeXor` The resulting pixel is the result of an `XOR` (exclusive or) operation on the source and destination pixels. This mode can be used a second time to delete a drawing.
- `IlvModeNot` The resulting pixel is the result of the `NOT` operation on the destination pixel. The source pixel value is not used.
- `IlvModeInvert` The resulting pixel is the result of a `NOT` operation on the source pixel.
- `IlvModeNotOr` The resulting pixel is the result of a `NOT OR` operation upon the source and destination pixels.
- `IlvModeNotAnd` The resulting pixel is the result of a `NOT AND` operation upon the source and destination pixels.
- `IlvModeNotXor` The resulting pixel is the result of an `NOT XOR` operation upon the source and destination pixels. When you draw the same object twice with `IlvModeNotXor` set, the drawing disappears.

Alpha Value

The alpha value indicates the amount of transparency the drawing will be given. A value of 0 means that the drawing will be completely transparent, that is, nothing will be drawn. A value of `IlvFullIntensity` means that the drawing will be opaque.

Drawing involves the use of two objects:

- An `IlvPort` object. This is the port where the drawing will be done. See Chapter 8, *Drawing Ports* for details.
- An `IlvPalette` object. This is a set of graphic resources that will be used to draw. See IlvPalete for details.

It is possible to control transparency at both levels: For example, you can set an alpha value on the port (using `IlvPort::setAlpha`) and also on the palette you are going to draw with (`IlvPalette::setAlpha`). In this case, the resulting drawing will use the composition of the two alpha values.

> ***Note:*** *This attribute is currently being supported only on Windows platforms using GDI+.* *See the section Using GDI+ Features with IBM ILOG Views on page 268 for details.*

Anti-Aliasing Mode

The anti-aliasing mode indicates whether smooth lines are drawn using anti-aliasing. The possible values for this mode are:

- `IlvDefaultAntialiasingMode` The anti-aliasing mode is not explicitly specified. It will be inherited using a default value.
- `IlvNoAntialiasingMode` No anti-aliasing will be used to draw.
- `IlvUseAntialiasingMode` Drawings will be done using anti-aliasing.

The anti-aliasing mode can be specified at different levels:

- `IlvDisplay` To set the default anti-aliasing mode of the display (`IlvDisplay::setAntialiasingMode`).
- `IlvPort` To set the anti-aliasing mode of a whole port (`IlvPort::setAntialiasingMode`).
- `IlvPalette` To set the anti-aliasing mode of a palette (`IlvPalette::setAntialiasingMode`).

The following rules are applied to determine if the final drawing will use anti-aliasing or not:

- For the palette:
  - If the anti-aliasing mode of the palette has been set (using the member function `IlvPalette::setAntialiasingMode`) then this mode is used.
  - Otherwise, the palette has the `IlvDefaultAntialiasingMode`, and the mode of the port in which the drawing is done is used.
- For the port:
  - If the anti-aliasing mode of the port has been set (using the member function `IlvPort::setAntialiasingMode`) then this mode is used.
  - Otherwise, the port has the `IlvDefaultAntialiasingMode`, and the mode of the display is used.
- For the display, the default anti-aliasing mode is `IlvNoAntialiasingMode`. This setting can be changed by either:

- Using the member function `IlvDisplay::setAntialiasingMode`.
- Setting the resource `Antialiasing` to `true`.
- Setting the environment variable `IlvAntialiasing` to `true`.

> ***Note:*** *This attribute is currently being supported only on Windows platforms using GDI+. See the section Using GDI+ Features with IBM ILOG Views on page 268 for details.*

IlvPalette: Drawing Using a Group of Resources

In IBM® ILOG® Views, the drawing resources are assembled into an object of the `IlvPalette` class. The only way to draw anything is to use an `IlvPalette` object. Most predefined graphic objects handle one `IlvPalette` (but sometimes more) to draw themselves.

Unlike other `IlvResource` subclasses, `IlvPalette` has public constructors. However, the standard way of creating a palette is still to get it from the `IlvDisplay::getPalette` methods. The public constructors must only be used to create palettes that cannot be shared.

An `IlvPalette` can be shared or not. Named palettes are a subset of shared palettes.

For additional details, see the sections:

- *Locking and Unlocking Resources*
- *Clipping Area*
- *Creating a Non-shared Palette*
- *Creating a Shared Palette*
- *Naming Palettes*

Locking and Unlocking Resources

An `IlvPalette` locks all the resources it contains and unlocks them when they are no longer used (palette destruction or replacement of a resource).

Clipping Area

The following methods are used to change the clipping area used to draw with the palette.

```
void setClip(const IlvRect* = 0) const;
void setClip(const IlvRegion*) const;
```

If you use a drawing method, the drawing will appear only in the clipping area; other areas will not be modified.

That is why you must clip the drawing when you write the `draw` method of an `IlvGraphic` subclass. The `draw` method gives you a clipping region as a parameter. You must set this clip on all the palettes that you are using to do the drawing. Once you have finished drawing, you must reset the clip of each palette to its previous clip, since the palettes are shared. This can be done using the `IlvPushClip` class.

Here is an example:

```
void MyGraphic::draw(IlvPort* dst,
                     const IlvTransformer* t,
                     const IlvRegion* clip) const
{
     IlvPalette* myPalette = getPalette();
     IlvPushClip (*myPalette, clip);
     IlvPoint p1(10, 10), p2(50, 50);
     // Do my drawings
     dst->drawLine(myPalette, p1, p2);
}
```

Creating a Non-shared Palette

You can create a palette that cannot be shared by using the public constructors of `IlvPalette`:

```
IlvPalette(IlvDisplay* display);
IlvPalette(IlvPalette* palette);
IlvPalette(IlvDisplay* display,
           IlvColor*   background,
           IlvColor*   foreground,
           IlvFont*    font,
           IlvPattern* pattern);
```

The first constructor creates a default palette, the second creates a copy of the palette given as its argument, and the third creates the palette with its characteristics passed as arguments. Once you have a new palette, you can use its member functions to set its internal resources (doing so with shared palettes is not recommended).

You can use this technique on the rare occasions when you do not want the palette to be shared at all or when you want to have total control on the way the palette is shared. You then have the responsibility of deleting it explicitly when it is no longer needed. Note that a palette built this way still uses shared resources (colors, fonts, and so on).

Creating a Shared Palette

Each `IlvDisplay` instance maintains a list of shared palettes. When you need a new palette, you must ask the display to supply it. This class provides a method `IlvDisplay::getPalette` that lets you specify the internal resources of the palette. The

other member function `IlvDisplay::getPalette(const char* name)` is discussed in the next section on named palettes.

If a palette matching your requirements already exists in the list, this palette is returned. If no such palette is found in the list of shared palettes, a new palette is created, added to the list and returned. The member function `IlvDisplay::getPalette` does not lock the returned palette. You can set some resource parameters of this function to `NULL`. The display then uses the corresponding default resources.

The use of shared palettes is very common and sufficient for most applications. However, you must keep in mind that these palettes are indeed shared and that modifying one of them is very likely to have undesirable side effects. Most of the time, palettes are used to control the way graphic objects (that is, subclasses of `IlvGraphic`) are drawn.

You should not try to modify a palette itself, but instead use the graphic object member functions to modify its graphic properties. The graphic object then gets another palette for the display and keeps the old palette unchanged, in case it is used somewhere else in the application.

The following code shows the right and wrong ways of using palettes:

```
// To set the foreground color of IlvGraphic* graphic
IlvColor* color = graphic->getDisplay()->getColor("blue");

// The following line will affect all objects sharing the palette
graphic->getPalette()->setForeground(color);  // Wrong way

// The following line will give another palette to the graphic object
// and will not affect objects pointing to the previous palette
graphic->setForeground(color);      // Right way
```

Naming Palettes

As with most other resources, palettes can be named using their `IlvResource::setName` member function. This function overwrites any existing name, so before naming a palette, you should check if the palette already has a name.

You can use the member function `IlvDisplay::getPalette(const char* name)` to retrieve a shared palette by its name.

The name of a palette is saved when graphic objects are written to an output stream. When this data is read as an input stream, the display first tries to find an existing palette with the same name. If none is found, the display tries to load the palette the usual way (that is, it looks for an existing palette matching the description, and if none exists, it creates a new palette and names it).

IlvQuantizer: The Image Color Quantization Class

`IlvQuantizer` is the abstract base class of all color conversion classes. It is used to convert true color images to indexed images of given numbers of colors. It defines basic functionality common to all IBM® ILOG® Views quantizers such as dithering.

Subclasses must redefine the `IlvQuantizer::computeColorMap` method to return an appropriate `IlvColorMap`.

It has two main subclass categories:

- The first category uses a fixed colormap.
- The second one computes a colormap from the input image.

Currently IBM ILOG Views has four predefined quantizers:

- The `IlvFixedQuantizer` remaps true color images to indexed ones according to a user specified colormap.
- The `IlvQuickQuantizer` specializes the `IlvFixedQuantizer` with a predefined colormap distributed in the color cube with 3 bits for the red component, 3 bits for the green component and 2 bits for the blue component, leading to a 256 color map well distributed in the color cube.
- The `IlvNetscapeQuantizer` specializes the `IlvFixedQuantizer` with a predefined colormap known as the Netscape colormap. This colormap has 216 entries. Images generated with this colormap are guaranteed not to dither in the Netscape web browser.
- The `IlvWUQuantizer` computes a colormap from the input image using the Wu algorithm. This algorithm generates very accurate colormaps even with a low number of colors (see the quantize sample). It is, however, slower than the others.

Other methods for quantization not implemented in IBM ILOG Views are Neural Nets and Octrees.

Sample code:

```
IlvWUQuantizer quantizer;
// bdata is an instance of an IlvRGBBitmapData
IlvIndexedBitmapData* idata = quantizer.quantizer(bdata, 64);
```

C H A P T E R

4

Graphic Formats

IBM® ILOG® Views is mainly a tool for manipulating vectorial entities, that is, shapes made of lines and curves that can easily be manipulated to change their visual aspect in terms of geometric characteristics. But IBM ILOG Views also has the capability of manipulating raster or bitmap images.

- *Graphic Formats Supported*
- *Bitmaps* describes the characteristics of bitmap images
- *IlvBitmap: The Bitmap Image Class*
- *IlvBitmapData: The Portable Bitmap Data Management Class*

Graphic Formats Supported

IBM® ILOG® Views can work with the following vectorial and bitmap formats:

- Vectorial:
  - **DXF** (input and output)
  - **DCW** (input)
  - **WMF** (Microsoft Windows only, output)

- **PostScript** (output)

◆ Bitmap (input and output):

- **BMP** - the standard Microsoft Windows bitmap format.
- **JPG** or **JPEG** - one of the most common formats used, especially for photos
- **PNG**
- **SGI RGB** - mainly used on the SGI Platform
- **TIFF** - Tagged Image Format File
- **PPM** - mainly used on UNIX platforms
- **WBMP** - used on WAP devices

For details on using bitmaps in IBM ILOG Views, see the section *Bitmaps* on page 90 and the `IlvBitmap` class.

Bitmaps

IBM® ILOG® Views supports bitmap (also called raster) images. Bitmaps have the following characteristics.

Color Bitmaps

If your display system has no true-color capabilities, each pixel value represents a color index. To find the exact color that will be displayed for this pixel, the system lookup table is consulted. If your display system has true-color capabilities, each pixel of the bitmap stores its complete color information.

Black-and-White Bitmaps

Images can also be monochrome. In this case, there is only one bit per pixel. The drawing of these one-bit-deep bitmaps takes place by setting the "`1`" pixels to the foreground color of the palette given to the `IlvDisplay` instance and the "`0`" pixels to the background color of the palette. When displayed as a transparent bitmap, the "`0`" pixels leave the destination port unchanged.

Transparent Bitmaps and Masks

A colored bitmap can be associated with a mask. A mask is a monochrome bitmap that indicates which of the pixels in the actual source image will be displayed. The pixels in the bitmap that correspond to "`0`" bits in the mask are not displayed, achieving the effect of a transparent bitmap. A transparent bitmap is a color bitmap that has transparent parts.

IlvBitmap: The Bitmap Image Class

Raster or bitmap images are represented by instances of the `IlvBitmap` class. For details on using `IlvBitmap`, see:

- *Bitmap-Related Member Functions*
- *Bitmap Formats*
- *Loading Bitmaps: Streamers*
- *Loading Transparent Bitmaps*

Bitmap-Related Member Functions

Special member functions dealing with bitmaps can be found in the `IlvDisplay` class.

Bitmaps are very often shared between different objects. For example, the same bitmap can be used as a fill pattern and an image on its own. Therefore, we need a management of bitmap resources, which is accomplished by a locking/unlocking policy.

Bitmap management is closely concerned with the ways in which you lock or unlock your bitmaps. Whenever you need a specific bitmap in one of your persistent objects, use the following mechanism to make sure that it will stay safe within your `IlvDisplay` instance:

```
void lock();
```

This member function of the class `IlvBitmap` ensures that your bitmap will never be modified or destroyed before every object that needs it tells IBM ILOG Views to do so. Basically, this function increments a reference count initially set to `0`.

```
void unLock();
```

This member function unlocks your bitmap; that is, it decrements the reference count of the bitmap and deletes it when this count becomes `0`. The creation/deletion mechanism of an `IlvBitmap` object by the `new` and `delete` C++ operators must be reserved for bitmap objects used in a temporary way and which are not shared.

Bitmap Formats

IBM ILOG Views allows you to create `IlvBitmap` objects from files or streams containing images in various formats. These formats are:

- **BMP** (all subtypes, RLE and RGB encoded, Indexed and True Color). This format is very common on Microsoft Windows platforms. Not compressed.

- Portable Network Graphics (**PNG**). This format is becoming more common. It allows transparent areas or colors, has indexed and high resolution true color subtypes.

  > ***Note:*** *This format is the patent-free replacement for GIF. See* `http://www.libpng.org/pub/png/`.

- Joint Photographic Experts Group (**JPEG**) This format is widely used for photographic images. It is "lossy," meaning that original information is missing in a JPEG image. This format allows important compression factors.
- Portable Pixmap (**PPM**, **PGM**, **PBM**) This format is very common on UNIX platforms. It is uncompressed and generates huge files.
- WAP Bitmap (**WBMP**) This format is used on WAP devices, such as mobile phones. It is a monochrome format.

Loading Bitmaps: Streamers

Each of the bitmap formats is associated with a streamer object (class `IlvBitmapStreamer`).

Streamers can be registered at compile time or at run time. Registering a streamer at compile time consists of including the header file for this format:

Table 4.1 *Header Files for Bitmap Formats*

Bitmap Format	Header File
JPEG	`ilviews/bitmaps/jpg.h`
PNG	`ilviews/bitmaps/png.h`
BMP	`ilviews/bitmaps/bmp.h`
PPM	`ilviews/bitmaps/ppm.h`
SGI RGB	`ilviews/bitmaps/rgb.h`
TIFF	`ilviews/bitmaps/tiff.h`
WBMP	`ilviews/bitmaps/wbmp.h`

You then just use the following call to load an image into a bitmap:

```
IlvBitmap* IlvDisplay::readBitmap(const char* filename);
```

Image type is recognized using a file signature, and the correct streamer is called automatically by `IlvDisplay::readBitmap`.

All the bitmap streamers are dynamic modules. It means that the reader or writer is dynamically loaded when necessary. Thus, you need only to request IBM ILOG Views to read or write an image, and it will do it for all known formats.

Streamers are modules and can be loaded at run time if an unknown (or unregistered) file format is being loaded. The corresponding module (if any) will be loaded and the streamer registered. This works only on platforms where modules are supported.

Additional formats are always registered and do not need modules:

- XPM
- XBM

Loading Transparent Bitmaps

An `IlvTransparentIcon` object appears as a bitmap. Pixels in the source bitmap with a zero value do not affect the destination port when the drawing is performed. Usually the transparent region of the bitmap icon lets the background pattern show through. This process works only for monochrome bitmaps or colored bitmaps that have either a transparency mask or a transparent color index.

IBM ILOG Views is able to load transparent bitmaps from the following file formats :

- XPM

  The transparent areas match the areas defined as “none” in the bitmap description file. If this information is omitted, the bitmap is not loaded as a transparent icon.

- PNG

  IBM ILOG Views uses transparency information form the PNG stream to create transparent areas in the bitmap.

IlvBitmapData: The Portable Bitmap Data Management Class

`IlvBitmapData` and its associated classes provide portable bitmap data management. For details see:

- *The IlvBitmapData Class*
- *The IlvIndexedBitmapData Class*
- *The IlvRGBBitmapData Class*
- *The IlvBWBitmapData Class*

The IlvBitmapData Class

Raster images in display systems such as X11 or Windows are generally represented using very system-dependent representations. These representations are deeply dependent on the display system configuration, forcing you to write display depth dependent code. The `IlvBitmapData` class allows you to describe raster images using a common portable API. `IlvBitmapData` is the base class of three subclasses that allow management of indexed images, true color images with alpha channel, and black and white images commonly used for masking and clipping. Bitmap data are managed like resources and can be locked/unlocked. This class is generally not used directly. The `IlvBitmapData` class manages memory, reference counting, and generic access to pixels of the image. It also provides access to basic image processing methods such as stretching. IBM® ILOG® Views also provides functionality for converting true color images to indexed images, a process known as quantizing (see *IlvQuantizer: The Image Color Quantization Class* on page 88). You also have access to the full SVG specification filters, allowing for very advanced image processing features (see Chapter 5).

The IlvIndexedBitmapData Class

The `IlvIndexedBitmapData` class is dedicated to indexed color images, where raster data is described as indexes to a color map (8-bit values, meaning that you can have only 256 colors in an indexed bitmap data).

Creating an Indexed Bitmap Data of Dimensions 256 * 256 Pixels

The first step is to create a colormap. We create a colormap with 256 entries. We then fill this colormap with grayscale values. Each component is described using 8 bits for each component.

```
IlvColorMap* cmap = new IlvColorMap(256);
for (IlUInt idx = 0; idx < 256; ++idx) {
   // sets the red, green and blue components for a entry
   cmap->setEntry(idx, idx, idx, idx);
}
```

We then create an indexed bitmap data of desired size:

```
IlvIndexedBitmapData* idata = new IlvIndexedBitmapData(256, 256, cmap);
```

We then fill the bitmap data with a gradient of indexes:

```
for (IlUInt h = 0; h < 256; ++h)
   for (IlUInt w = 0; w < 256; ++w)
      idata->setPixel(w, h, h);
```

To be able to display this bitmap data on the screen, you have to create an `IlvBitmap` from it.

```
IlvBitmap* bitmap = new IlvBitmap(display, idata);
```

You can then use the `IlvIcon` class to create a graphic object from this.

The IlvRGBBitmapData Class

The `IlvRGBBitmapData` class is dedicated to true color images, where raster data is a direct representation of the colors of the pixels.

Creating a True Color Bitmap Data of Dimensions 256 * 256 Pixels and Filling it With a Gradient

```
IlvRGBBitmapData* bdata = new IlvRGBBitmapData(256, 256);
for (IlUInt h = 0; h < 256; ++h)
  for (IlUInt w = 0; w < 256; ++w)
    bdata->fastSetRGBPixel(w, h, w, h, w);
```

As with `IlvIndexedBitmapData`, you can then create an `IlvBitmap` and display it using IBM ILOG Views standard methods.

When on an 8-bit color display, the IBM® ILOG® Views library automatically converts this true color image to an indexed image using an algorithm yielding a very high quality image.

The internal representation for true color bitmap data is an array of width * height entries. Each entry is a 4-byte quadruplet describing a pixel as follows:

- First byte is the alpha component.
- Second byte is the red component.
- Third byte is the green component.
- Fourth byte is the blue component.

The array can be described top-bottom or bottom-top, so you have a line access method using `IlvBitmapData::getRowStartData`.

You can use various methods to access raster data:

- `IlvBitmapData::getData` returns a pointer to the raw raster data.
- `IlvRGBBitmapData::getRGBPixel` allows you to retrieve a given pixel.
- `IlvRGBBitmapData::getRGBPixels` allows you to retrieve the RGB representation of a given rectangle.
- `IlvRGBBitmapData::fill` allows you to fill a rectangle with a given color.
- `IlvRGBBitmapData::copy` allows you to copy a rectangle of a bitmap data to a given position in another bitmap data.
- `IlvRGBBitmapData::blend` allows you to smoothly blend a bitmap data into another using a blend factor.

- `IlvRGBBitmapData::alphaCompose` uses the alpha channel to compose two bitmap data.
- `IlvRGBBitmapData::tile` allows you to tile a bitmap data into another.
- `IlvRGBBitmapData::stretch` allows you to stretch a portion of a bitmap data into another.
- `IlvRGBBitmapData::stretchSmooth` allows you to stretch a portion of a bitmap data into another using high quality resampling methods.

You can also independently access the color and alpha values for a given pixel.

The IlvBWBitmapData Class

This class is dedicated to black and white images, where only two values are possible for a given pixel: on or off.

C H A P T E R

5

Image Processing Filters

This chapter presents the various image processing classes that IBM® ILOG® Views provides.

These classes are all related to the SVG filters (for a complete description of the features, see `http://www.w3.org/TR/2000/CR-SVG-20001102/filters.html`).

IlvBitmapFilter: The Image Processing Class

`IlvBitmapFilter` is the base class of all image processing classes in IBM® ILOG® Views. It defines the interface for image processing classes using a single method:

```
IlvBitmapFilter::apply
```

This method accepts an array of `IlvBitmapData` and returns another `IlvBitmapData`.

The `ilvbmpflt` library from the IBM ILOG Views foundation package defines many subclasses of `IlvBitmapFilter`; most of them are implementations of the W3Cs SVG filters specification. In the following sections, you will find a list of the image processing classes and their features.

- *The IlvBlendFilter Class*
- *The IlvColorMatrixFilter Class*

- *The IlvComponentTransferFilter Class*
- *The IlvComposeFilter Class*
- *The IlvConvolutionFilter Class*
- *The IlvDisplaceFilter Class*
- *The IlvFloodFilter Class*
- *The IlvGaussianBlurFilter Class*
- *The IlvImageFilter Class*
- *The IlvLightingFilter Class*
- *The IlvLightSource Class*
- *The IlvMergeFilter Class*
- *The IlvMorphologyFilter Class*
- *The IlvOffsetFilter Class*
- *The IlvTileFilter Class*
- *The IlvTurbulenceFilter Class*
- *The IlvFilterFlow Class*
- *Using IlvFilteredGraphic to Apply Filter Flows to Graphic Objects*

The IlvBlendFilter Class

The `IlvBlendFilter` class lets you blend two images A and B using various modes.

The blend modes define the following formulas:

- Normal Blend Mode: `cr = (1 - qa) * cb + ca`
- Multiply Blend Mode: `cr = (1 - qa) * cb + (1 - qb) * ca + ca * cb`
- Screen Blend Mode: `cr = cb + ca - ca * cb`
- Darken Blend Mode: `cr = Min((1 - qa) * cb + ca, (1 - qb) * ca + cb)`
- Lighten Blend Mode: `cr = Max((1 - qa) * cb + ca, (1 - qb) * ca + cb)`

where:

`cr`	Result color (RGB) - premultiplied
`qa`	Opacity value at a given pixel for image A
`qb`	Opacity value at a given pixel for image B

`ca`	Color (RGB) at a given pixel for image A (premultiplied)
`cb`	Color (RGB) at a given pixel for image B (premultiplied)

For all blend modes, the resulting opacity `qr` is computed as follows:

```
qr = 1 - (1 - qa) * (1 - qb)
```

The IlvColorMatrixFilter Class

The `IlvColorMatrixFilter` class lets you apply a matrix transformation on the RGBA components of an input image.

The matrix is given as 5*4 row major order coefficients.

```
| R' |     | a00 a01 a02 a03 a04 |   | R |
| G' |     | a10 a11 a12 a13 a14 |   | G |
| B' |  =  | a20 a21 a22 a23 a24 | * | B |
| A' |     | a30 a31 a32 a33 a34 |   | A |
| 1  |     |  0   0   0   0   1  |   | 1 |
```

This class has three subclasses with specific coefficients.

The IlvSaturationFilter Class

`IlvSaturationFilter` computes the transformation matrix from the formula:

```
| R' |     |0.213+0.787s  0.715-0.715s  0.072-0.072s 0  0 |   | R |
| G' |     |0.213-0.213s  0.715+0.285s  0.072-0.072s 0  0 |   | G |
| B' |  =  |0.213-0.213s  0.715-0.715s  0.072+0.928s 0  0 | * | B |
| A' |     |          0            0             0  1  0 |   | A |
| 1  |     |          0            0             0  0  1 |   | 1 |
```

where `s` is the saturation factor.

The IlvHueRotateFilter Class

`IlvHueRotateFilter` computes the transformation matrix from the formula:

```
| R' |     | a00  a01  a02  0  0 |   | R |
| G' |     | a10  a11  a12  0  0 |   | G |
| B' |  =  | a20  a21  a22  0  0 | * | B |
| A' |     | 0    0    0    1  0 |   | A |
| 1  |     | 0    0    0    0  1 |   | 1 |
```

where the terms `a00`, `a01`, and so on, are calculated as follows:

```
 | a01 a01 a02 |    [+0.213 +0.715 +0.072]
| a10 a11 a12 | =  [+0.213 +0.715 +0.072] +
| a20 a21 a22 |    [+0.213 +0.715 +0.072]

                        [+0.787 -0.715 -0.072]
cos(hueRotate value) *  [-0.212 +0.285 -0.072] +
                        [-0.213 -0.715 +0.928]

                        [-0.213 -0.715 +0.928]
sin(hueRotate value) *  [+0.143 +0.140 -0.283]
                        [-0.787 +0.715 +0.072]
```

where `value` is the angle of rotation for the hue.

Thus, the upper-left term of the hue matrix turns out to be:

```
.213 + cos(hueRotate value)*.787 - sin(hueRotate value)*.213
```

The IlvLuminanceToAlphaFilter Class

`IlvLuminanceToAlphaFilter` computes the transformation matrix from the formula:

```
| R' |     |      0        0        0  0  0 |   | R |
| G' |     |      0        0        0  0  0 |   | G |
| B' |  =  |      0        0        0  0  0 | * | B |
| A' |     | 0.2125   0.7154   0.0721  0  0 |   | A |
| 1  |     |      0        0        0  0  1 |   | 1 |
```

This filter converts color images to grayscale images.

The IlvComponentTransferFilter Class

The `IlvComponentTransferFilter` class lets you perform component-wise remapping on images as follows:

```
R' = feFuncR( R )
G' = feFuncG( G )
B' = feFuncB( B )
A' = feFuncA( A )
```

feFuncR, feFuncG, feFuncB, and feFuncA define the transfer functions for each component.

It allows operations such as brightness adjustment, contrast adjustment, color balance, or thresholding.

Five predefined transfer functions are defined:

- identity: `C' = C`
- table: the function is defined by linear interpolation into a lookup table by attribute values, which provides a list of `n+1` values (that is, `v0` to `vn`) in order to identify `n` interpolation ranges. Interpolations use the following formula:

  ```
  k/N <= C < (k+1)/N => C' = vk + (C - k/N)*N * (vk+1 - vk)
  ```

- discrete: the function is defined by the step function defined by attribute values, which provides a list of `n` values (that is, `v0` to `vn-1`) in order to identify a step function consisting of `n` steps. The step function is defined by the following formula:

  ```
  k/N <= C < (k+1)/N => C' = vk
  ```

- linear: the function is defined by the following linear equation:

  ```
  C' = slope * C + intercept
  ```

  where `slope` and `intercept` are user specified.

- gamma: the function is defined by the following exponential function:

  ```
  C' = amplitude * pow(C, exponent) + offset
  ```

  where `amplitude`, `exponent`, and `offset` are user specified.

Transfer functions are classes on their own and can be redefined (see `IlvTransferFunction`, `IlvIdentityTransfer`, `IlvLinearTransfer`, `IlvTableTransfer`, `IlvDiscreteTransfer`, and `IlvGammaTransfer`).

The IlvComposeFilter Class

The `IlvComposeFilter` class lets you perform the combination of the two input images pixel-wise in image space using one of the Porter-Duff compositing operations: `over`, `in`, `atop`, `out`, `xor`. Additionally, a component-wise arithmetic operation (with the result clamped between [0..1]) can be applied. You have a choice of these six operators for the compositing, shown in Table 5.1 on page 102.

The resulting color is given by the formula:

```
Cresult = Fa * Ca + Fb * Cb
```

where:

- `Fa` and `Fb` depend on the operator as shown in Table 5.1 on page 102.

- `Ca` is the color from the first image, and `Cb` is the color from the second image.
- In the table, `Aa` is the alpha value from the first image and `Ab` is the alpha value from the second image.

Table 5.1 *Compositing Operators*

Operator	Operation
`over`	`Fa = 1, Fb = 1 - Aa`
`in`	`Fa = Ab, Fb = 0`
`out`	`Fa = 1 - Ab, Fb = 0`
`atop`	`Fa = Ab, Fb = 1 - Aa`
`xor`	`1 - Ab, Fb = a - Aa`
arithmetic	`Cresult = k1 * Ca * Cb + k2 * Ca + k3 * Ca * Cb + k4`

The IlvConvolutionFilter Class

The `IlvConvolutionFilter` class lets you apply a matrix convolution filter effect. A convolution combines pixels in the input image with neighboring pixels to produce a resulting image. A wide variety of imaging operations can be achieved through convolutions, including blurring, edge detection, sharpening, embossing and beveling.

A matrix convolution is based on an n-by-m matrix (the convolution kernel) which describes how a given pixel value in the input image is combined with its neighboring pixel values to produce a resulting pixel value. Each result pixel is determined by applying the kernel matrix to the corresponding source pixel and its neighboring pixels.

To illustrate, suppose you have an input image which is 5 pixels by 5 pixels, whose color values are as follows:

```
  0  20  40 235 235
100 120 140 235 235
200 220 240 235 235
225 225 255 255 255
225 225 255 255 255
```

and you define a 3-by-3 convolution kernel as follows:

```
1 2 3
4 5 6
```

```
7 8 9
```

Let us focus on the pixel at the second row and second column of the image (the source pixel value is `120`). Then the resulting pixel value will be:

```
(1*  0 + 2* 20 + 3* 40 +

4*100 + 5*120 + 6*140 +

7*200 + 8*220 + 9*240) / (1+2+3+4+5+6+7+8+9)
```

You can specify a divisor (a number by which the result of the matrix convolution is divided) and a bias (a number by which the result of the matrix convolution is added).

The kernel is described by the `IlvBitmapDataKernel` class.

The IlvDisplaceFilter Class

The `IlvDisplaceFilter` class lets you displace pixels from an image using pixel values of another image.

This is the transformation to be performed:

```
P'(x,y) <- P( x + scale * ((XC(x,y) - .5), y + scale * (YC(x,y) -
.5))
```

where:

- `P(x,y)` is the input image.
- `P'(x,y)` is the destination.
- `XC(x,y)` and `YC(x,y)` are the component values of the displacement map. They can be chosen from any of the color components of the image map (an example is that the Red component displaces in X while the Alpha component displaces in Y.
- `scale` is a user-specified scaling value.

The IlvFloodFilter Class

The `IlvFloodFilter` class lets you fill an image with a given color.

The IlvGaussianBlurFilter Class

The `IlvGaussianBlurFilter` class lets you apply a Gaussian blur effect to an image. The Gaussian blur kernel is an approximation of the normalized convolution:

```
H(x) = exp(-x^2/ (2*s^2)) / sqrt(2* pi*s^2)
```

where `s` is a user-specified deviation.

This filter uses the `IlvConvolutionFilter` internally.

The IlvImageFilter Class

The `IlvImageFilter` class lets you load an image from a string describing the image name.

The IlvLightingFilter Class

The `IlvLightingFilter` class lets you light an image using the alpha channel as a bump map. Several types of lights can be specified (see later).

This class is an abstract class and has two usable subclasses.

The IlvDiffuseLightingFilter Class

For the `IlvDiffuseLightingFilter` class, the resulting image is an RGBA opaque image based on the light color with alpha = 1.0 everywhere. The lighting calculation follows the standard diffuse component of the Phong lighting model. The resulting image depends on the light color, light position, and surface geometry of the input bump map.

The light map produced by this filter primitive can be combined with a texture image using the multiply term of the arithmetic `IlvComposeFilter` compositing method. Multiple light sources can be simulated by adding several of these light maps together before applying it to the texture image.

The resulting RGBA image is computed as follows:

```
Dr = kd * N.L * Lr
Dg = kd * N.L * Lg
Db = kd * N.L * Lb
Da = 1.0
```

where:

`kd`	Diffuse lighting constant
`N`	Surface normal unit vector, a function of x and y (see below)
`L`	Unit vector pointing from surface to light, a function of x and y in the point and spot light cases
`Lr, Lg, Lb`	RGB components of light, a function of x and y in the spot light case

`N` is a function of x and y and depends on the surface gradient as follows:

The surface described by the input alpha image `Ain (x,y)` is:

```
Z (x,y) = surfaceScale * Ain (x,y)
```

Surface normal is calculated using the Sobel gradient 3x3 filter:

```
Nx (x,y)= - surfaceScale * 1/4*(( I(x+1,y-1) + 2*I(x+1,y) + I(x+1,y+1))
                                - (I(x-1,y-1) + 2*I(x-1,y) + I(x-1,y+1)))

Ny (x,y)= - surfaceScale * 1/4*(( I(x-1,y+1) + 2*I(x,y+1) + I(x+1,y+1))
                                - (I(x-1,y-1) + 2*I(x,y-1) + I(x+1,y-1)))

Nz (x,y) = 1.0

N = (Nx, Ny, Nz) / Norm((Nx,Ny,Nz))
```

See `IlvLightSource` for a further description of `L`, the unit vector from the image sample to the light.

The IlvSpecularLightingFilter Class

For `IlvSpecularLightingFilter`, the resulting image is an RGBA image based on the light color. The lighting calculation follows the standard specular component of the Phong lighting model. The resulting image depends on the light color, light position, and surface geometry of the input bump map. The result of the lighting calculation is added. The filter primitive assumes that the viewer is at infinity in the z direction (that is, the unit vector in the eye direction is (0,0,1) everywhere).

This filter primitive produces an image that contains the specular reflection part of the lighting calculation. Such a map is intended to be combined with a texture using the add term of the arithmetic `IlvComposeFilter` method. Multiple light sources can be simulated by adding several of these light maps before applying it to the texture image.

The resulting RGBA image is computed as follows:

```
Sr = ks * pow(N.H, specularExponent) * Lr
Sg = ks * pow(N.H, specularExponent) * Lg
Sb = ks * pow(N.H, specularExponent) * Lb
Sa = max(Sr, Sg, Sb)
```

where:

`ks`	Specular lighting constant
`N`	Surface normal unit vector, a function of x and y (see below)
`H`	"Halfway" unit vector between eye unit vector and light unit vector
`Lr, Lg, Lb`	RGB components of light

See `IlvDiffuseLightingFilter` for definitions of `N` and (`Lr, Lg, Lb`).

The definition of `H` reflects our assumption of the constant eye vector `E = (0,0,1)`:

```
H = (L + E) / Norm(L+E)
```

where `L` is the light unit vector.

The IlvLightSource Class

The `IlvLightSource` class lets you model lights. It has three usable subclasses

The IlvDistantLight Class

`IlvDistantLight` models an infinite light source using an azimuth and an elevation:

```
Lx = cos(azimuth)*cos(elevation)
Ly = sin(azimuth)*cos(elevation)
Lz = sin(elevation)
```

The IlvPointLight Class

`IlvPointLight` models an positional light using three coordinates `Lightx`, `Lighty`, and `Lightz`.

The IlvSpotLight Class

`IlvSpotLight` models a positional spot light using three coordinates `Lightx`, `Lighty`, and `Lightz`.

```
Lx = Lightx - x
Ly = Lighty - y
Lz = Lightz - Z(x,y)
L = (Lx, Ly, Lz) / Norm(Lx, Ly, Lz)
```

where:

`Lightx`, `Lighty`, and `Lightz`	The input light position

`Lr, Lg, Lb`	The light color vector, is a function of position in the spot light case only:
	`Lr = Lightr*pow((-L.S),specularExponent)`
	`Lg = Lightg*pow((-L.S),specularExponent)`
	`Lb = Lightb*pow((-L.S),specularExponent)`

Given `S` as the unit vector pointing from the light to the point (`pointsAtX`, `pointsAtY`, `pointsAtZ`) in the x-y plane:

```
Sx = pointsAtX - Lightx
Sy = pointsAtY - Lighty
Sz = pointsAtZ - Lightz
S = (Sx, Sy, Sz) / Norm(Sx, Sy, Sz)
```

The IlvMergeFilter Class

The `IlvMergeFilter` class lets you composite input image layers on top of each other using the `over` operator.

Many effects produce a number of intermediate layers in order to create the final output image. This filter allows us to collapse them into a single image. Although this could be done by using n-1 `IlvComposeFilter` filters, it is more convenient to have this common operation available in this form, and offers the implementation some additional flexibility.

The IlvMorphologyFilter Class

The `IlvMorphologyFilter` class lets you perform "fattening" or "thinning" of artwork. It is particularly useful for fattening or thinning an alpha channel.

The dilation (or erosion) kernel is a rectangle with a width of `2*x-radius+1` and a height of `2*y-radius+1` where `radius` is a user-specified value. In dilation, the output pixel is the individual component-wise maximum of the corresponding R,G,B,A values in the input image kernel rectangle. In erosion, the output pixel is the individual component-wise minimum of the corresponding R,G, B, A values in the input image kernel rectangle.

The IlvOffsetFilter Class

The `IlvOffsetFilter` class lets you offset an image by given x and y values. This is important for effects such as drop shadows.

The IlvTileFilter Class

The `IlvTileFilter` class lets you create a target image with a repeated, tiled pattern.

The IlvTurbulenceFilter Class

The `IlvTurbulenceFilter` lets you create an image using the Perlin turbulence function. It allows the synthesis of artificial textures such as clouds or marble.

> ***Note:*** *For a detailed description the of the Perlin turbulence function, see "Texturing and Modeling", Ebert et al, AP Professional, 1994.*

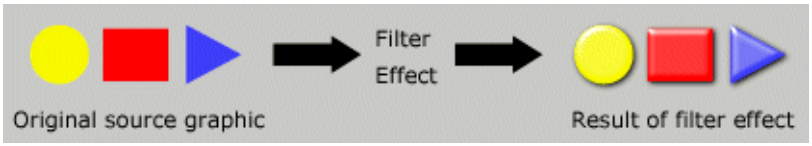
The resulting image will have maximal size in the image space.

It is possible to create bandwidth-limited noise by synthesizing only one octave.

You can choose whether fractal noise or turbulence is created and the number of repetitions (octaves) of the noise generation functions to use.

The IlvFilterFlow Class

The `IlvFilterFlow` class lets you chain `IlvBitmapFilter` instances using names as inputs and outputs. An example of such a flow is the creation of a drop shadow effect. You use the alpha channel of an image as the input for a Gaussian blur; you then offset the blurred image and merge this with the source image.



The `IlvFilterFlow` class can be created programmatically but it is much more convenient to use the XML representation of filter flows, in a similar way to the definition of filters in SVG.

An example of such a flow is given by the following XML file (see `$ILVHOME/data/filters` for many predefined XML filter flows):

```
<?xml version="1.0"?>
<filters>
  <filter id="DropShadow2" x="-10" y="-10" width="125" height="125">
    <desc>Applies a drop shadow effect</desc>
    <feGaussianBlur in="SourceAlpha" stdDeviation="3"/>
    <feOffset dx="2" dy="2" result="offsetBlur"/>
    <feComposite in="SourceGraphic" in2="offsetBlur" operator="over"/>
  </filter>
</filters>
```

Here is a line-by-line description:

```
<?xml version="1.0"?>
```

The definition of IBM ILOG Views filters follows XML conventions.

```
<filters>
```

Opening element for filters (the file can contain any number of filters)

```
<filter id="DropShadow2" x="-10" y="-10" width="125" height="125">
```

Opening element for a filter name `DropShadow2`.

Some filters extend the source image by some pixels in all dimensions, so we must specify the additional extension of the filter.

This filter grows the source image by 10 pixels at the left and top and extends the width and height by 25 pixels.

```
<desc>Applies a drop shadow effect</desc>
```

This tag contains a description of the filter.

```
<feGaussianBlur in="SourceAlpha" stdDeviation="3"/>
```

The first atomic filter to use in this flow is an `IlvGaussianBlurFilter` with a deviation of 3 in both directions.

Two predefined names are defined by the filter flow:

- `SourceAlpha`: contains only the Alpha values of the input image.
- `SourceGraphic`: contains the input image.

Here we only need to apply the blur on the alpha component of the image.

```
<feOffset dx="2" dy="2" result="offsetBlur"/>
```

The second atomic filter is an `IlvOffsetFilter` with displacement 2.

When not specified, the input from a filter is the output of the previous filter, so it does not need to be specified. The result will be an image stored with the name `offsetBlur`.

```
<feComposite in="SourceGraphic" in2="offsetBlur" operator="over"/>
```

The third and last atomic filter is an `IlvComposeFilter` that will compose the offset blurred image with the input image using the `over` operator.

```
</filter>
```

Closes the filter flow description.

```
</filters>
```

Closes the filters enumeration.

Such an `IlvFilterFlow` can be created by using the following lines. We suppose that the file containing the filter flow is stored on the disk under the name `standard.xml`:

```
IlIUrlStream input("standard.xml");
IlvFilterFlow* flow = new IlvFilterFlow(input, "DropShadow2");
```

Using IlvFilteredGraphic to Apply Filter Flows to Graphic Objects

The IBM® ILOG® Views foundation package provides a simple way to apply filter flows to graphic objects: the `IlvFilteredGraphic` class.

This class encapsulates a graphic object and internally computes an `IlvBitmapData` from the `draw` method of the object. It then applies a given filter flow to this `IlvBitmapData` and draws the result on the screen. It is then very easy to add image processing effects to vectorial objects.

Sample code:

```
IlvZoomableLabel* embosssource = new IlvZoomableLabel(display,
```

```
                                                    IlvPoint(100, 100),
                                                    "Views");

    embosssource->setForeground(display->getColor((IlvIntensity)(5 * 255),
                                                  (IlvIntensity)(5 * 255),
                                                  (IlvIntensity)(56 * 255)));
    embosssource->setFont(display->getFont("%Helvetica-75-"));
    IlvFilteredGraphic* emboss = new IlvFilteredGraphic(display,
                                                        embosssource,
                                                        "standard.xml#DropShadow",
                                                        IlTrue);
```

Many predefined filter flows are provided in the IBM ILOG Views distribution in the `$ILVHOME/data/filters` directory. You can use them interactively with the IBM ILOG Views Studio application.

C H A P T E R

6

The Display System

The `IlvDisplay` class is a fundamental class in the IBM® ILOG® Views library. It handles every aspect of the connection with the display system. The topics are:

- *IlvDisplay: The Display System Class*
- *Connecting to the Display Server*
- *Display System Resources*
- *Home*
- *The Display Path*

IlvDisplay: The Display System Class

To develop a graphic application using IBM® ILOG® Views, you use a set of `IlvDisplay` member functions or IBM ILOG Views *primitives*:

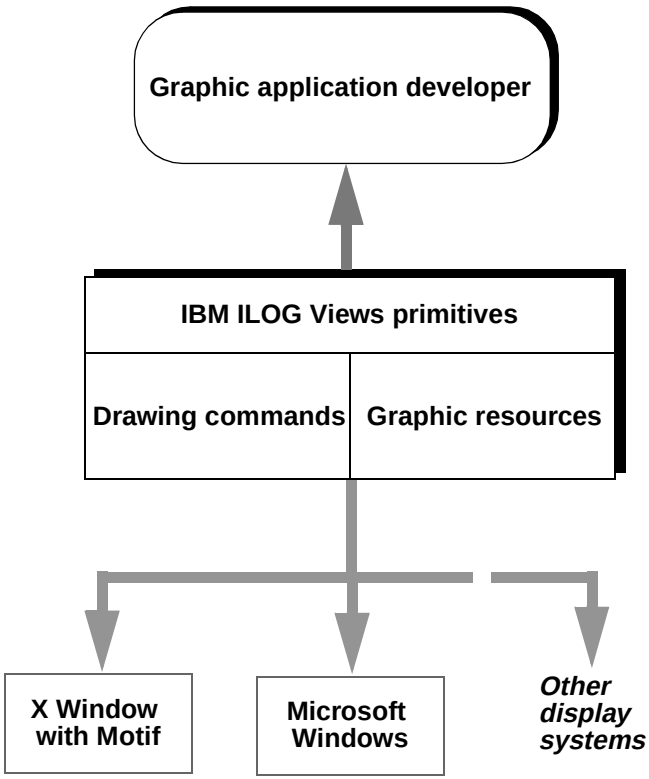


Figure 6.0 *IlvDisplay Drawing Member Functions: The Primitives*

The class `IlvDisplay` enables you to communicate transparently with a display system such as X Window or Microsoft Windows.

Two basic tasks are drawing commands and graphic resource handling:

- **Drawing Commands** Drawing commands handle the basic geometric classes for such entities as points, rectangles, regions (list of rectangles), curves, and strings.

  There are more than twenty drawing member functions of this kind (see the `IlvPort` class for details). Drawing operations produce their results inside a region—either in memory or on the screen—that is defined as an instance of the `IlvPort` class.

- **Graphic Resources** Some `IlvDisplay` drawing member functions deal with graphic resources, such as colors, line styles, patterns, and fonts. These resources are objects that inherit the features of a class called `IlvResource`. They are created by means of various `IlvDisplay` member functions. Specific resources are grouped together into objects of the `IlvPalette` class for drawing purposes.

Connecting to the Display Server

To initialize an IBM® ILOG® Views session, you have to create an instance of the `IlvDisplay` class. This involves:

- *Opening a Connection and Checking the Display*
- *Closing a Connection and Ending a Session*

Opening a Connection and Checking the Display

The member function `IlvDisplay::isBad` returns a Boolean value that tells you whether the `IlvDisplay` object has been successfully created, as shown in the following code. The reasons for a display failure vary from one display system to another.

```
IlvDisplay* display = new IlvDisplay("AppName",
                                     "DisplayName",
                                     argc,
                                     argv);
if (display->isBad()) {
   delete display;
   IlvFatalError("Could not create display");
   IlvExit(-1);
}

const char* dirName  = "./localDirectory/subDirectory";
const char* fileName = "foo.txt";
display->prependToPath(dirName);
// Now, if a file such as
//    "./localDirectory/subDirectory/foo.txt"
// or
//    ".\localDirectory\subDirectory\foo.txt"
// exists, we should be able to find it.
const char* filePath = display->findInPath(fileName);
if (filePath)
    IlvPrint("File %s found at %s", fileName, filePath);
else
    IlvWarning("File %s not found", fileName);
```

For more information about IBM ILOG Views error messages, see Appendix E, *Error Messages*.

Closing a Connection and Ending a Session

To close a connection to the display server, just destroy the `IlvDisplay` object. The destructor, `IlvDisplay::~IlvDisplay`, immediately frees all graphic resources used by the display.

If the `IlvDisplay` was created on the basis of an existing link to a display system, `delete` does not destroy this link. Except in the rare cases of multidisplay applications, destroying the `IlvDisplay` means the end of the session, since you cannot do much without a display.

You must call `IlvExit` to end the session properly. It frees memory allocated by IBM ILOG Views. This is especially important with Microsoft Windows, where this memory is not automatically freed by the system.

```
delete display;
IlvExit(0);
```

Display System Resources

A display system resource is an association of *two* strings: a `name` and a `value`. Display system resources are very convenient for building customizable applications.

> ***Note:*** *"Display system" resources are not to be confused with "graphic" resources, which are described in Chapter 3, Graphic Resources.*

The resource name-value pair can be specified as follows in specific sections of a resource file (the `.Xdefaults` file on UNIX or the `.INI` file on PCs, for example):

```
[MyApplication]
view.background=green
label.txt=This is my contents
```

The value associated with the resource name can be modified by the end user at run time.

> ***Note:*** *Resource files have an* `[IlogViews]` *section that is common to all IBM ILOG Views applications.*

More details on display system resources are in the topics:

- *The getResource Method*
- *How Display System Resources are Stored*
- *Default Display System Resources*
- *Environment Variables and Resource Names*
- *Display System Resources on Windows*

The getResource Method

The application uses the `IlvDisplay::getResource` method to retrieve a resource value from the display system.

```
const char* res = display->getResource("resourceName", default);
```

The method `IlvDisplay::getResource` returns a value string associated with the application name of the current IBM ILOG Views session, which is specified in the `IlvDisplay` constructor, and the string representing the resource name. If no resource matches the given string, then this member function returns the default value provided in the optional `default` string parameter. The only type returned by `IlvDisplay::getResource` is `const char*`. It is up to the application programmer to convert the string to another data type. The place in memory where the result of an `IlvDisplay::getResource` call is stored gets reused each time you call the function. Thus, the previous result is overwritten. If you want to save your result, you must recopy it right away.

How Display System Resources are Stored

The way resources are stored within your display system configuration files is system-dependent.

- With Microsoft Windows, you must add the following line to the `VIEWS.INI` file or the application-dependent `.INI` file. (The `VIEWS.INI` file can be found in the Windows directory.)

  ```
  [AppName]
  myDialogTitle=Load file
  ```

- On the X Window system, you must pass the following line to the resource manager:

  ```
  AppName*myDialogTitle: Load file
  ```

  You can use the `xrdb` program or include it into a file read by X clients (the `.Xdefaults` file or the file specified by your `XENVIRONMENT` variable).

Default Display System Resources

When an instance of the `IlvDisplay` class is created, the default display system resources are initialized using the system resources mechanism:

Table 6.1 *IlvDisplay Default Resources*

IlvDisplay Method	System Resource Name	Default Value
`IlvDisplay::defaultForeground`	`foreground`	`black`
`IlvDisplay::defaultBackground`	`background`	`gray`
`IlvDisplay::defaultFont`	`font`	system-dependent
`IlvDisplay::defaultNormalFont`	`normalfont`	system-dependent
`IlvDisplay::defaultBoldFont`	`boldfont`	system-dependent
`IlvDisplay::defaultItalicFont`	`italicfont`	system-dependent
`IlvDisplay::defaultLargeFont`	`largefont`	system-dependent

Environment Variables and Resource Names

Default UNIX and PC environment variables have precedence over the resource name specified in resource files (namely `.Xdefault`, and `.INI`). The following table gives a list of environment variables with their associated resource names.

Table 6.2 *Environment Variables and Resource Names*

Environment Variable Name	Resource Name
`ILVHOME`	`home`. For details see the section *Home*.
`ILVLANG`	`lang`. For more information, see the section *The IlvMessageDatabase Class*.
`ILVDB`	`messageDB`. For more information, see the section *The IlvMessageDatabase Class*.
`ILVLOOK`	`look`. The `look` resource takes either one of the following values: `motif`, `windows`, or `win95`. For more information, refer to the Gadgets documentation.

Display System Resources on Windows

On the Microsoft Windows environment, you can define Windows-specific resources in addition to the standard resources. These resources are listed below:

- `[TTY]` If this resource is set to `TRUE`, a message window is created to which all IBM ILOG messages are sent. The default value is `FALSE`.
- `[TTYw]`, `[TTYh]`, `[TTYx]`, `[TTYy]` These resources specify the size and position of the message window, provided that `TTY` is set to `TRUE`. The default values are `TTYw=200`, `TTYh=100`, `TTYx=screen_width-TTYw`, `TTYy=screen_height-TTYh`.
- `[UseRightButton]` If this resource is set to `TRUE` and your mouse has two buttons, the `IlvEvent` generated by the IBM ILOG Views library holds the `IlvRightButton` value. Otherwise, it contains the `IlvMiddleButton` value. The default value is `FALSE`.

  For more information, see the member functions `IlvDisplay::isRightButtonValueUsed` and `IlvDisplay::useRightButtonValue`.
- `[SolidColors]` If this resource is set to `TRUE`, the VGA system palette is used. The default value is `FALSE` if the number of colors available on the system is greater than 16; otherwise the default value is `TRUE`. This avoids dithered images on low-color graphical systems.
- `[Warnings]` If this resource is set to `TRUE`, the warning messages are displayed. The default value is `FALSE`.

The member function `IlvDisplay::getResource` searches for resource definitions in several files, which are listed below in a decreasing order of priority:

1. `EXECDIR\APP.INI`
2. `EXECDIR\PROG.INI`
3. `EXECDIR\VIEWS.INI`
4. `WINDIR\APP.INI`
5. `WINDIR\PROG.INI`
6. `WINDIR\VIEWS.INI`

`EXECDIR` is the directory containing the executable program, and `WINDIR` is the directory where Microsoft Windows is installed. `APP` represents the name of the your application; this string is the one you provide to the `IlvDisplay` constructor. `PROG` is the base name of the executable file, its complete name being `PROG.EXE`.

Home

Most IBM® ILOG® Views applications might need to load predefined data files. For data files to be transparently loaded, libraries need a way to locate these data files on the disk. This is done by getting the value of the `ILVHOME` environment variable. If this variable is undefined, IBM ILOG Views tries to retrieve the value of the display system resource `home`. Generally, this value is set to the directory where IBM ILOG Views was installed (the one containing the subdirectories `include`, `lib`, `data`, and so on).

> ***Note:*** *The old* `IlvHome` *display system resource is maintained for compatibility reasons but is deprecated.*

There are two global functions that force the setting of `home`:

- `IlvGetDefaultHome`
- `IlvSetDefaultHome`

Use these functions if you want to provide a reasonable default value for a specific application without asking the user to set the environment variable `ILVHOME` or the resource `home`.

The value of `home` is used to compute the default value of the display path, described in the next section.

The Display Path

The access to files is greatly simplified by the *display path* mechanism. If the path name provided in a call to the functions that open and read files is relative, the function searches for the file name in the directories specified in the display path.

The member functions of `IlvDisplay` can be used to check and manipulate the display path, as well as to check whether a file name exists in any of the directories specified in the display path.

For details on the display path mechanism, see:

- *Setting the Display Path*
- *The Path Resource*
- *The ILVPATH Environment Variable*
- *Querying or Modifying the Display Path*
- *Example: Add a Directory to the Display Path*

Setting the Display Path

The display path is a string that contains multiple directory path names, separated by the regular system path separator (‘:’ under *UNIX*, and ‘;’ for *DOS*).

Initially (when the `IlvDisplay` instance is created), the display path is set to the concatenation of three distinct elements as follows (using UNIX notation for path):
`<.:user path:system path:>`

- The first section contains only the current directory (noted ‘.’).
- The second section, `user path`, is composed of the contents of the display resource `path` followed by the contents of the environment variable `ILVPATH`.
- The third section, `system path`, contains subdirectories of `IlvHome`.

In short, the `IlvPath` initial value is (assuming `ILVHOME` is defined):

```
.:<path resource>:$ILVPATH:$ILVHOME/data:$ILVHOME/data/icon:$ILVHOME
data/images
```

The Path Resource

- On X Window, the `path` resource will be, for example:

  ```
  AppName*path: /usr/local/views/ilv
  ```

- On Microsoft Windows, the `path` resource can be in the `VIEWS.INI` or application-dependent file `.INI`:

  ```
  [AppName]
  path=C:\USER\DATA\ILV
  ```

The ILVPATH Environment Variable

The `ILVPATH` environment variable can be set by the user before the application is launched.

- With UNIX, this setting can be defined by the lines:

  ```
  $ ILVPATH=/usr/home/user/ilvimages:/usr/home/user/ilvpanels
  $ export ILVPATH
  ```

- In a Microsoft Windows command prompt window, this setting could be:

  ```
  C:\> SET ILVPATH=C:\USER\DATA\ILV;C:\USER\DATA\IMAGES
  ```

Querying or Modifying the Display Path

The class `IlvDisplay` provides member functions to manipulate the display path. These are `IlvDisplay::getPath`, `IlvDisplay::setPath`, `IlvDisplay::appendToPath`, and `IlvDisplay::prependToPath`.

These methods allow the user to get, set, and modify the *user path* (that is, the second section of the display path). The structure of the display path remains the same, that is:
`<.:user path:system path.>`

The method `IlvDisplay::findInPath` is used to:

- Check whether a file is in the display path.
- Get its absolute path name.

Example: Add a Directory to the Display Path

The following example shows how to add a directory to the display path and check whether a file is in the display path.

```
IlvDisplay* display = new IlvDisplay("AppName",
                                     "DisplayName",
                                     argc,
                                     argv);
if (display->isBad()) {
   delete display;
   IlvFatalError("Could not create display");
   IlvExit(-1);
}

const char* dirName  = "./localDirectory/subDirectory";
const char* fileName = "foo.txt";
display->prependToPath(dirName);
// Now, if a file such as
//    "./localDirectory/subDirectory/foo.txt"
// or
//    ".\localDirectory\subDirectory\foo.txt"
// exists, we should be able to find it.
const char* filePath = display->findInPath(fileName);
if (filePath)
    IlvPrint("File %s found at %s", fileName, filePath);
else
    IlvWarning("File %s not found", fileName);
```

C H A P T E R

7

Views

Chapter 8, *Drawing Ports* explained the drawing port concept managed by the `IlvPort` class. The views hierarchy deals with another `IlvPort` subclass, `IlvAbstractView` and its derived subclasses. `IlvView` is a major subclass, representing the actual place on the screen where drawing occurs.

- *View Hierarchies: Two Perspectives* discusses ways you can look at the components that make up a view. The approaches are a:
  - *Window-Oriented View Hierarchy*
  - *Class-Oriented View Hierarchy*
- *IlvAbstractView: The Base Class*
- *IlvView: The Drawing Class*
- *IlvView Subclasses*
- *The IlvScrollView Class*

View Hierarchies: Two Perspectives

There are two ways of looking at how views are constructed:

- *Window-oriented*. From the viewpoint of somebody sitting in front of a screen and using IBM ILOG Views, different kinds of views are assembled into various **windows** having different appearances. In fact they are created one after the other.
- *Class-oriented*. From a C++ viewpoint, the `IlvView` and related **classes** enable you to create the different kinds of views that together create windows.

Window-Oriented View Hierarchy

Here is a schematic representation of the window-oriented view hierarchy, with the corresponding classes also shown.

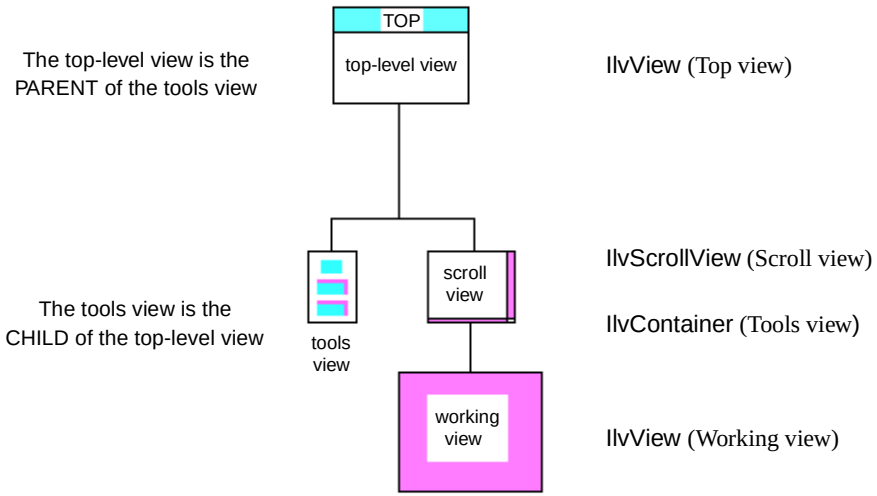


Figure 7.1 *View Hierarchy*

Parent-Child Relationships

The terms “parent” and “child” designate the relationship between pairs of views, taking into account which one contains the other. In the figure, the top-level view is the parent of both the tools and the scroll views, and the latter is the parent of the working view. Inversely, the scroll view is a child of the top-level view, and so on.

C++ classes and subclasses also represent parent-child relationships. Note that there is no one-to-one correspondence between the window-oriented versus class-oriented view hierarchies. The C++ class hierarchy is a different way of looking at how a view is constructed. For the complete class-oriented diagram, see the *Class-Oriented View Hierarchy* on page 123.

Class-Oriented View Hierarchy

In the *Window-Oriented View Hierarchy* on page 122 we looked briefly at four different kinds of views, which are instances of the following classes (or subclasses):

- `IlvView` for the top-level view (the top window) and the working view.
- `IlvContainer` for the tools view.
- `IlvScrollView` for the scroll view.

The following diagram shows these and other classes that derive from `IlvAbstractView`.

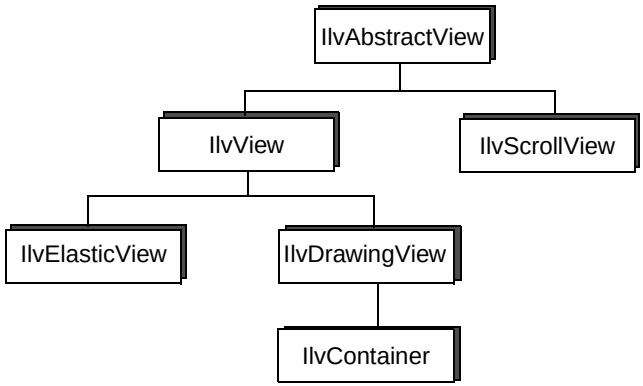


Figure 7.2 *The IlvAbstractView Base Class*

These views classes give rise to actual windows or views that are displayed on your screen. When you instantiate one of the derived subclasses of `IlvAbstractView`, the object you obtain is referred to as a *view*. A window on the screen is, in fact, an associated set of one or several views.

For More Details See

- *IlvAbstractView: The Base Class*
- *IlvView: The Drawing Class*
- *The IlvScrollView Class*
- *IlvView Subclasses* for more details on `IlvElasticView`, `IlvDrawingView`, and `IlvContainer`.

IlvAbstractView: The Base Class

`IlvAbstractView` is an abstract class (of which instances can only be made from subtyped classes). This class has functions for handling basic properties of a window, such as size, visibility, color, and so on. The `IlvAbstractView` object encapsulates the real interface object of your display system (that is, a system view, sometimes referred to as a widget). This interface object is platform-dependent and may be accessed by the following function:

```
IlvSystemView getSystemView() const;
```

where `IlvSystemView` is the basic type of the display system widget.

IlvView: The Drawing Class

The class `IlvView` is one of the descendants of `IlvAbstractView` (and thus of `IlvPort`) in the *Class-Oriented View Hierarchy* on page 123.

The `IlvView` subclass is a major class, since it represents the actual place on the screen where drawing occurs. An instance of `IlvView` can also contain zones that are sensitive to mouse clicks.

The `IlvView` class and its subclasses provide objects that are used to draw things on the screen. They may be top-level windows or children of a previously created parent view.

Two constructors are specifically used to create a new top-window in your display instance:

```
IlvView(IlvDisplay*    display,
        const char*    name,
        const char*    title,
        const IlvRect& size,
        IlBoolean      visible = IlTrue);

IlvView(IlvDisplay*    display,
        const char*    name,
        const char*    title,
        const IlvRect& size,
        IlUInt         properties,
        IlBoolean      visible       = IlTrue,
        IlvSystemView  transientFor  = 0);
```

The second constructor allows you to specify the top-window aspect that deals with borders, banners, and handles. You can provide a valid system view value to the `transientFor` parameter. If you do, your new `IlvView` object will be transient for that system view. This has the interesting effect that when the system view is iconified, your view is implicitly iconified as well.

Some member functions of the `IlvView` class are specific and meaningful only if the view is a top-window.

The other constructors are:

```
IlvView(IlvAbstractView* parent,
        const IlvRect&   size,
        IlBoolean       visible = IlTrue);

IlvView(IlvDisplay*     display,
        IlvSystemView   parent,
        const IlvRect&  size,
        IlBoolean      visible = IlTrue);
```

The `parent` parameter is either an `IlvAbstractView` or an existing `IlvSystemView`.

The last constructor of `IlvView` is used to let IBM® ILOG® Views take control over an existing `IlvSystemView` such as one created by another application.

```
IlvView(IlvDisplay*   display,
        IlvSystemView existingWindow);
```

You will use this constructor when you want to extend a native application (written with the Microsoft Windows SDK or MFC or, on UNIX, X Window or Motif code) with IBM ILOG Views graphic capabilities.

You will probably create your `IlvDisplay` from an existing connection (see *Connecting to the Display Server*). Because the windows hierarchy is likely to have been set up already, your `IlvView` objects will have to take control of existing windows.

IlvView Subclasses

The subclasses of `IlvView`, completing that portion of the *Class-Oriented View Hierarchy*, are as follows:

The IlvElasticView Class

The `IlvElasticView` class offers the same capabilities as `IlvView` except that, when instances of this class are resized, they resize their children in an elastic manner. This is a special `IlvView` class available for a view containing other views for which you want automatic resizing.

The IlvDrawingView Class

Another subclass of `IlvView` is `IlvDrawingView`. The `IlvDrawingView` class has predefined member functions for handling incoming events such as expose and resize events.

The IlvContainer Class

The `IlvContainer` class is the first class in the *Class-Oriented View Hierarchy* that coordinates the storage and display of graphic objects. Numerous specialized subclasses are described in *Containers*.

The IlvScrollView Class

A special type of view in the *Class-Oriented View Hierarchy* is managed by an instance of the `IlvScrollView` class. To operate properly, this class needs a widget toolkit linked with your application.

> ***Note:*** `IlvScrollView` *has an implementation on Microsoft Windows and on top of Motif only (the ports that have native controls). If you have the IBM ILOG Views Gadgets package, the class* `IlvScrolledView` *provides similar services.*

An `IlvScrollView` must contain a single child view, which is usually bigger than its parent. The `IlvScrollView` takes care of all automatic scrolling operations. It does so by means of scroll bars, located on the right and bottom of the scrolling view. You can manipulate the scroll bars to display new areas of the subview.

You can handle a non-IBM® ILOG® Views window object within an `IlvScrollView` object. This can be any of your system windows, such as one created by another application.

C H A P T E R

8

Drawing Ports

A drawing port, defined by the `IlvPort` class, is an area where the user will be drawing. It may be to any output device such as the screen or a printer. Details are discussed in the topics:

- *IlvPort: The Drawing Port Class*
- *Derived Classes of IlvPort*
- *The IlvSystemPort Class*
- *The IlvPSDevice Class*

IlvPort: The Drawing Port Class

The class `IlvPort` defines a drawing port. The `IlvPort` class has the necessary member functions to draw any shape to a specific dump device such as a printer. These member functions are:

- The virtual member function `IlvPort::initDevice`, called by the `init` function, initializes the dump device and writes its result to the file `filename`.
- The virtual member function `IlvPort::isBad` returns `IlTrue` if the dumping device is not valid. This return value indicates an initialization problem.

- The virtual member function `IlvPort::end` closes the dump device and does all the necessary cleaning.
- The virtual member function `IlvPort::send` lets you send any character string to an output device so you can send information to the device.
- The virtual member function `IlvPort::newPage` produces an output page and prepares the dump device for a new page. It returns `IlFalse` if there was an error. In this case, you should stop producing output data.
- The virtual member function `IlvPort::setTransformer` lets you apply an additional transformer—that is, any geometric transformation— to the coordinates with which you "feed" the drawing functions.

Derived Classes of IlvPort

The illustration below shows some of the predefined classes that derive from `IlvPort`:

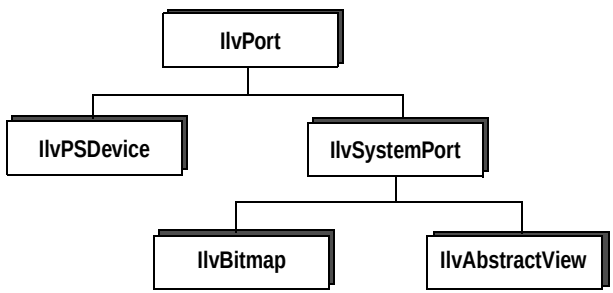


Figure 8.1 *The IlvPort Base Class*

The class `IlvPort` defines a drawing port in one of two ways, either:

- Physically as a screen or bitmap, via *The IlvSystemPort Class*.
- In a dedicated zone as a file or a printer.

The other two subclasses are:

- `IlvBitmap`, providing bitmap support as described in *Bitmaps*.
- The subclass `IlvAbstractView` is the base class for views. This subject is treated at length in Chapter 7, *Views*. See especially *IlvAbstractView: The Base Class*.

The IlvSystemPort Class

The class `IlvSystemPort` defines a rectangular area in which the user can draw. It can be either a real place or a virtual place. In the first case, the user draws directly into a region of the screen of the workstation. In the second case, the user draws into a bitmap in memory. The classes `IlvBitmap` and `IlvAbstractView` are derived from `IlvSystemPort` to accommodate these two possibilities. The `IlvBitmap` class is described later in this chapter. The `IlvAbstractView` class and its subclasses are described in the next chapter.

The IlvPSDevice Class

To redirect the drawing operations to a dumping device, such as a printer or a plotter, IBM® ILOG® Views calls the member functions of the subclass that implement all the drawing operations. These member functions have to be overloaded to define the drawing operations needed in the various implementations of dump devices.

So that you can immediately “print what you see,” IBM ILOG Views provides you with a predefined class, `IlvPSDevice`.

The `IlvPSDevice` class lets you print any region of a view to a text file that can be immediately printed on a *PostScript* printer. Furthermore, all the drawing member functions are implemented to create the *PostScript* code corresponding to the expected result.

C H A P T E R

9

Containers

A container is an instance of the `IlvContainer` class, which is a special kind of view that can store and display graphic objects. The subject is covered in depth in the following sections:

- *IlvContainer: The Graphic Placeholder Class*
- *Displaying Containers*
- *Managing Events: Accelerators*
- *Managing Events: Object Interactors*
- *Creating Objects with Complex Behavior*

IlvContainer: The Graphic Placeholder Class

`IlvContainer` is the graphic placeholder class. Its member functions for handling objects within containers are described in:

- *General-Purpose Member Functions*
- *Applying Functions to Objects*
- *Tagged Objects*

- *Object Properties*

General-Purpose Member Functions

Some member functions of the `IlvContainer` class, for example `IlvContainer::addObject` and `IlvContainer::removeObject`, enable you to store and remove objects within containers. (See the IBM ILOG Views *Reference Manual* for more details.) The `IlvContainer` object stores graphic objects in a list.

Applying Functions to Objects

Additional member functions are used to apply user-defined functions to the container objects. These are:

- `IlvContainer::applyToObjects` Apply a user-defined function to each object in a `IlvContainer`.
- `IlvContainer::applyToObject` Apply a user-defined function to the specified graphic object only.

Tagged Objects

The `IlvContainer` class provides member functions to manage graphic objects that have been tagged by the user. A tag is a kind of marker represented by an object of the `IlSymbol` class and which can be associated with several graphic objects:

- `IlvContainer::applyToTaggedObjects` is similar to the method `IlvContainer::applyToObjects` but is used with tagged objects.
- `IlvContainer::getTaggedObjects` returns an array of pointers to objects stored in the container which are tagged with the specified tag.
- `IlvContainer::removeTaggedObjects` removes from the container all the objects tagged with the specified tag.

IlSymbol Class

IBM ILOG Views sometimes needs string constants to manipulate specific entities, such as tags. To do this, there is a generic manner of handling unique strings within a given application. The strings are called *symbols*, and are managed by the `IlSymbol` class.

The `IlGetSymbol` global function lets you create new symbols or access symbols already created.

Object Properties

Several member functions of the `IlvContainer` class let you manage container object properties, such as the functions `IlvContainer::getObject`,

`IlvContainer::setObjectName`, and `IlvContainer::setVisible`. For example, it is possible to:

- Access a graphic object according to different criteria: its name or its index identifier, since the container stores objects in a list.
- Interchange two objects with respect to their order in the container list (`IlvContainer::swap` method).
- Request an object's state of visibility or to change the state. (Visibility refers to whether the object is visible on the screen or not.)
- Ask by means of the static method `IlvContainer::GetContainer` where a graphic object is stored, that is, in which container. You cannot store an object in more than one container.

> ***Note:*** *Since the member function* `IlvContainer::GetContainer` *is static, it is not necessary to apply it to an existing instance of* `IlvContainer`*. You can use this method from anywhere, with the notation:* `IlvContainer::GetContainer(myobject);`

Displaying Containers

The member functions for displaying containers are described in:

- *Drawing Member Functions*
- *Geometric Transformations*
- *Managing Double Buffering*
- *Reading Objects from Disk*

Drawing Member Functions

Although the member functions `IlvContainer::draw` and `IlvContainer::reDraw` are inherited from the parent `IlvDrawingView` class, they interact in a specific way with containers. Following is a list of these specific member functions:

- `IlvContainer::draw` is used to draw all `IlvGraphic` objects stored in the `IlvContainer` object. When you add an object to a container, calling this method is sufficient. Two virtual member functions allow the user to draw at any destination port with any transformer, and with a clipping region specified.
- `IlvContainer::reDraw` If you need to refresh your working area (for example, when an object is translated), use this method, which erases the specified clipping region before calling the `IlvContainer::draw` method.

- `IlvContainer::reDrawObj` When applied to a graphic object, redraws the object's bounding box.
- `IlvContainer::bufferedDraw` Performs a temporary drawing in a hidden pixmap, then displays the pixmap on the screen at once. It is different from double buffering in that the operation is localized to a rectangle, a region, or an object, and lasts only as long as the drawing operation.

  One function draws the area included in the specified `IlvRect` object and another draws the area included in the specified `IlvRegion` object. Both of them draw in the coordinate system of the container. The third form draws the specified `IlvGraphic` object in the coordinate system of the object.

Geometric Transformations

Several member functions deal with geometric transformations applied to the container view, that is, they handle the `IlvTransformer` object associated with the view.

- `IlvContainer::getTransformer` Returns the transformer associated with the container view. If `0` is returned, this container has no transformer, that is, there is identity between the object and its display.
- `IlvContainer::setTransformer` Sets the specified transformer parameter.
- `IlvContainer::addTransformer` Sets up the current transformer with the one given as a parameter, and sets the resulting transformer as the new current one.
- `IlvContainer::translateView` and `IlvContainer::zoomView` Sets up the current transformer with, respectively, a translation transformer and a zooming transformer.
- `IlvContainer::fitToContents` Resizes the container view so that its bounding box exactly fits around all the objects that have the visibility attribute set to `IlTrue`. The top-left coordinates of the view remain at the same location. This method is typically used when the container reads a set of `IlvGraphic` objects from a file, without knowing their positions beforehand:

```
IlvRect size(0, 0, 300, 300);
IlvContainer* cont = new IlvContainer(display, "Cont", "My Window",
                                      size, IlTrue, IlFalse);
cont->readFile("myfile.ilv");
cont->fitToContents();
```

- `IlvContainer::fitTransformerToContents` Computes a new transformer so that all objects that have the visibility attribute set to `IlTrue` can be seen in the view. The container view size does not change. A call to `IlvDrawingView::reDraw` is issued if the `IlBoolean` argument is set to `IlTrue`. This method is typically used when the user wants to have a global look at a map after zooming it several times:

```
static void
ShowAllMap(IlvContainer* container)
{
    container->fitTransformerToContents(IlTrue);
}
```

Managing Double Buffering

The double buffering mode enables animated displays or displays with numerous objects to appear without flickering. This mode is handled by the following member functions:

- `IlvContainer::setDoubleBuffering` Indicates whether or not the container should use double buffering.
- `IlvContainer::isDoubleBuffering` Informs as to whether double buffering is or is not in use.

Reading Objects from Disk

Two member functions exist for reading objects from disk:

- `IlvContainer::readFile` Reads from a file whose name is specified as a parameter.
- `IlvContainer::read` Reads from the input stream specified as a parameter.

Both these member functions return the result of their reading: `IlTrue` if successful, `IlFalse` if an error occurred.

> ***Note:*** *Containers do not have any* `write` *member functions to save their contents as do managers. For details on managers, see the Managers documentation.*

Managing Events: Accelerators

An accelerator manages a single user event that occurs in the container to which the accelerator is attached. An accelerator is the direct connection from this single user event to a function call. You can declare a given function to be called if an appropriate event occurs. If the appropriate event occurs, the accelerator triggers a visible response from the container to which it is attached.

You can also install an instance of the `IlvContainerAccelerator` class in an accelerator list. This instance contains the description of the event to be watched for, and a member function of this class is called when this event occurs.

For more details, see:

- *Member Functions*
- *Implementing Accelerators: IlvContainerAccelerator*
- *Predefined Container Accelerators*

Member Functions

Several member functions deal with accelerators:

- `IlvContainer::addAccelerator` Installs a new accelerator in the container. In the example below, the function `Quit` is triggered when one event matches the keyboard event "release Q key."

```
static void
Quit(IlvContainer* cont, IlvEvent&, IlAny)
{
   IlvDisplay* d = cont->getDisplay();
   delete d;
   IlvExit(0);
}IlvRect size(0, 0, 300, 300);
IlvContainer* cont = new IlvContainer(display, "Cont", "My Window",
                                      size, IlTrue, IlFalse);
cont->addAccelerator(Quit, IlvKeyUp, 'Q', 0);
```

- `IlvContainer::removeAccelerator` Removes the association between the event description given in the argument and the action previously set by `IlvContainer::addAccelerator`.
- `IlvContainer::getAccelerator` Queries a container for a particular accelerator action and user argument.

Implementing Accelerators: IlvContainerAccelerator

If you need to add parameters to a callback function, you can implement accelerators by means of class subtyping, through the `IlvContainerAccelerator` class.

The member functions dealing with this class are the following:

- `IlvContainer::addAccelerator` Installs an `IlvContainerAccelerator` object in the container. The previous example can thus be written in the following way:

```
IlvContainerAccelerator* acc =
   new IlvContainerAccelerator(Quit, IlvKeyUp, 'Q', 0);
cont->addAccelerator(acc);
```

- `IlvContainer::removeAccelerator` Removes the given accelerator argument from the container list. The accelerator is not deleted.
- `IlvContainer::getAccelerator` Returns a pointer to the `IlvContainerAccelerator` instance that matches the given event argument, or `0` if no matching accelerator exists.

Predefined Container Accelerators

IBM ILOG Views provides a number of predefined accelerators to allow your programs to easily manipulate the visible aspect of the objects that are stored in a container:

Table 9.1 *Predefined Container Accelerators*

Event Type	Key or Button	Action
`IlvKeyDown`	i	Sets the transformer to identity.
`IlvKeyDown`	<Right>	Moves the view to the left.
`IlvKeyDown`	<Left>	Moves the view to the right.
`IlvKeyDown`	<Down>	Moves the view to the top (decreasing x)
`IlvKeyDown`	<Up>	Moves the view to the bottom. (increasing x)
`IlvKeyDown`	Z	Zooms into the view.
`IlvKeyDown`	U	Zooms out from the view.
`IlvKeyDown`	R	Rotates the view 90 degrees counterclockwise.
`IlvKeyDown`	f	Computes a new transformer so that every object can be seen.

Managing Events: Object Interactors

An object interactor filters user events for the graphic object to which it is attached. If an appropriate series of events occurs, the object interactor triggers a visible response from that graphic object. This response is called the object behavior.

IBM® ILOG® Views provides a comprehensive set of predefined object interactors. If you find yourself needing a very specific functionality not already predefined in IBM ILOG Views, you can subtype one of the interactor classes and replace its member function `handleEvent` with the functionality you need.

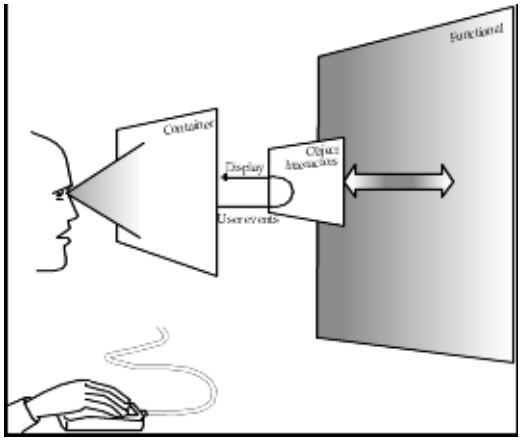


Figure 9.1 *Attaching an Interactor to an Object*

The class `IlvInteractor` lets you associate a behavior with an object.

> ***Note:*** *These object interactors are not intended for creating editors. The IBM ILOG Views Manager interactors that are associated with the whole view instead of individual objects are used to create interactive editors.*

For more details on object interactors, see:

- *Using Object Interactors*
- *Predefined Object Interactors*
- *Example: Linking an Interactor and an Accelerator*

Using Object Interactors

The member functions that deal with object interactors are the following:

- `IlvGraphic::getInteractor` Returns the `IlvInteractor` instance associated with the `IlvGraphic` object given as an argument.
- `IlvGraphic::setInteractor` Associates the `IlvInteractor` object given as an argument with the given `IlvGraphic` object.

In the following example, a predefined IBM ILOG Views interactor is associated with an `IlvLabel` graphic. It is an instance of the `IlvMoveInteractor` class which lets the user move an object by pressing the left mouse button when pointing to this object.

```
IlvRect size(0, 0, 300, 300);
IlvContainer* cont = new IlvContainer(display, "Cont", "My Window",
                                      size, IlTrue, IlFalse);
IlvLabel* label = new IlvLabel(display, IlvPoint(100,100),
                               "Hello world!");
cont->addObject(label);
label->setInteractor(IlvInteractor::Get("Move"));
```

The static member function `IlvInteractor::Get` returns the unique instance of the object interactor whose name is “Move.” You usually do not create an interactor by calling its constructor directly, but by using this static member function. This is because most object interactors can be shared by numerous graphic objects at the same time.

Registering a New IlvInteractor Subclass

If you subtype the `IlvInteractor` class, you have to register your subclass in order to use the static member function `IlvInteractor::Get`. Below is that part of the header file where the `IlvInteractor` class in `MyInteractor` class is subtyped:

```
class MyInteractor
: public IlvInteractor {
public:
    IlBoolean handleEvent(IlvGraphic*     obj,
                          IlvEvent&       event,
                          IlvTransformer* t);
    ...
     DeclareInteractorTypeInfo(MyInteractor);
};
```

Note that we have added the line that calls the macro `DeclareInteractorTypeInfo` that allows for class persistence, as well as registration. This macro forces you to define a constructor that expects a reference to an `IlvInputFile`, a copy constructor that expects a reference to a `MyInteractor`, and a `write` member function that is needed to save the interactor instance. Of course the constructor and the write must match, just as in the case of the class `IlvGraphic`.

In the situation where your interactor has no extra information to save, you would have used the macro `DeclareInteractorTypeInfoRO` that does not force you to define a `write` member function.

In the example there is not any extra information to save, but for the completeness of the example, we pretend to have a dummy integer value to save and read:

```
MyInteractor::MyInteractor(IlvInputFile& file)
: IlvInteractor(file)
{
    IlInt i;
    file.getStream() >> i; // Read a (dummy) integer value
}

IlvInteractor*
MyInteractor::write(IlvOutputFile& file)
{
    file.getStream() << (IlInt)0;
}
```

If you had used `DeclareInteractorTypeInfoRO`, the constructor would have been empty, and write would not have been defined.

In the source file, outside the body of any other function, you must write the two following instructions:

- `IlvPredefinedInteractorIOMembers(MyInteractor)`

  `IlvPredefinedInteractorIOMembers` is a macro that generates the proxy function for you that will call your constructor from input file and define the `copy` member function.

- `IlvRegisterInteractorClass(MyInteractor, IlvInteractor);`

  `IlvRegisterInteractorClass` is a macro that registers the class `MyInteractor` as a new available interactor class. The second parameter must be the name of the parent class.

Predefined Object Interactors

Several classes help you program predefined object interactors:

- `IlvButtonInteractor` This class can be attached to any graphic object to make it behave like a standard interface button.
- `IlvRepeatButtonInteractor` This class is a subtype of the class `IlvButtonInteractor`. It handles an automatic repeat action of the button, as if the user were pressing and releasing the mouse button at a given speed.
- `IlvToggleInteractor` This class is a subtype of the class `IlvButtonInteractor`. This subclass inverts the object (calls its member function `invert`) to which this interactor is associated when the user presses then releases the mouse button over it.
- `IlvMoveInteractor` Lets the user move an object by clicking on it and dragging the pointing device to another place.

- `IlvReshapeInteractor` Lets the user reshape an object by creating a rectangle with the right mouse button. The rectangle becomes the object's new bounding box.
- `IlvMoveReshapeInteractor` Combines the two interactors `IlvMoveInteractor` and `IlvReshapeInteractor`.
- `IlvDragDropInteractor` Provides a way to drag and drop an object from a container to another view. It lets you click an object and move a copy of the object around, even outside the container itself.

Example: Linking an Interactor and an Accelerator

In the following example, we create the window below and the two enclosed drawings:

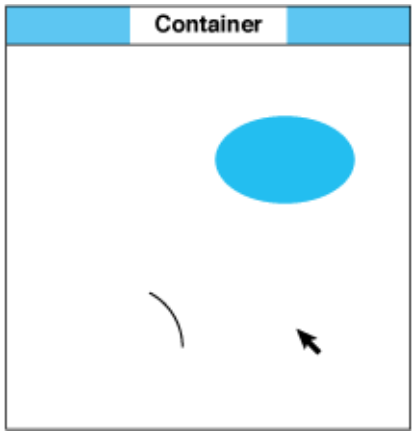


Figure 9.2 *Container with Two Objects*

In other words, we create a container as a top window, then we add two objects to the container: a gray ellipse and an arc. However, before actually placing these objects in the container, we create a reshaping interactor, enabling us to use the mouse to change the shape of a graphic object. We actually use this single interactor in two places:

- Immediately after adding the ellipse to the container, we associate the reshaping interactor with this object.
- Immediately after adding the arc to the container, we associate the same reshaping interactor with it.

We can now click either of the two graphic objects, and the reshaping interactor lets us change the shape of the selected object by dragging the mouse.

Furthermore, after creating the two objects and the unique interactor, we create an accelerator for the container, so we can get printed information about an object by double-clicking it with the left mouse button:

- If we double-click the ellipse, we get the following message:
  `Object is an IlvFilledEllipse`
- If we double-click the arc, we get the following message:
  `Object is an IlvArc`

We write a three-argument function called `PrintType`, which has the predefined type `IlvContainerAction`, to produce these messages. The name of this function is given to the container as the first argument of the call to the member function `IlvContainer::addAccelerator`.

Following is the entire demonstration program. To keep the example short, we do not provide any way to exit from the program:

```
#include <ilviews/contain/contain.h>
#include <ilviews/graphics/ellipse.h>
#include <ilviews/graphics/arc.h>
#include <ilviews/graphics/inter.h>

static void PrintType(IlvContainer*, IlvEvent&, IlAny);

int
main(int argc, char* argv[])
{
    IlvDisplay* display = new IlvDisplay("Demo", "", argc, argv);
    if (!display || display->isBad()) {
        IlvFatalError("Couldn't open display");
        delete display;
        IlvExit(-1);
    }
    IlvContainer* container =
        new IlvContainer(display, "Demo", "Demo",
                         IlvRect(0, 0, 200, 200),
                         IlTrue, IlFalse);
    IlvInteractor* reshape = IlvInteractor::Get("Reshape");
    IlvGraphic* object =
        new IlvFilledEllipse(display, IlvRect(150, 50, 40, 20));

    container->addObject(object);
    object->setInteractor(reshape);
    container->addObject(object =
        new IlvArc(display, IlvRect(10, 150, 40, 40), 0., 60.));
    object->setInteractor(reshape);
    container->addAccelerator(PrintType,
                              IlvDoubleClick,
                              IlvLeftButton);
    container->show();
    IlvMainLoop();
    return 0;
}

static void
PrintType(IlvContainer* view, IlvEvent& event, IlAny)
{
    IlvGraphic* object =
        view->contains(IlvPoint(event.x(), event.y()));
    if (object)
        IlvPrint("Object is an `%s'\n" , object->className());
}
```

Analyzing the Example

This section explains the code used for the above example.

```
static void PrintType(IlvContainer*, IlvEvent&, IlAny);
```

The user-defined `PrintType` function is called by an accelerator, which is launched when the user double-clicks a graphic object with the left mouse button. The signature of the `PrintType` function corresponds to the type that is called `IlvContainerAction`.

```
IlvContainer* container = new IlvContainer(display, "Demo", "Demo",
                                    IlvRect(0, 0, 200, 20), IlTrue, IlFalse);
```

A container is created as the top-level view.

```
IlvInteractor* reshape = IlvInteractor::Get("Reshape");
```

A reshaping interactor is located. This was registered automatically when you included `<ilviews/graphics/inter.h>`.

```
IlvGraphic* object = new IlvFilledEllipse(display, IlvRect(150,50, 40,20));
container->addObject(object);
```

A filled ellipse is created and added to the container.

```
object->setInteractor(reshape);
```

The reshaping interactor is associated with the filled ellipse.

```
container->addObject(object =
                      new IlvArc(display, IlvRect(10, 150, 40, 40), 0., 60.));
```

An arc is added to the container.

```
object->setInteractor(reshape);
```

The reshaping interactor is associated with the arc.

```
container->addAccelerator(PrintType, IlvDoubleClick, IlvLeftButton);
```

An accelerator is added to the container. This accelerator applies the user-defined function named `PrintType` whenever the user double-clicks an object with the left mouse button.

```
static void
PrintType(IlvContainer* view, IlvEvent& ev, IlAny)
{
    IlvGraphic* object = view->contains(IlvPoint(ev.x(), ev.y()));
    if (object)
        IlvPrint("Object is a '%s'\n", object->className());
}
```

This is the actual implementation of the user-defined `PrintType` function. We access the object located under the mouse pointer by calling the member function `IlvContainer::contains`. We use the function `IlvPrint` to ensure the portability of this example.

Creating Objects with Complex Behavior

The behavior handled by a container and its interactors is usually more sophisticated than just a series of simple actions.

Creating such an object is developed in the topics:

- *Example: Creating a Slider*
- *Associating a Behavior with Your Device*
- *Building and Extending your Device*

> ***Note:*** *Remember that you can also use the Prototypes package of IBM ILOG Views to create your own specialized objects.*

Example: Creating a Slider

Let us suppose that you have written a C++ program to drive a multimedia system and you are going to use IBM ILOG Views to create the graphical user interface for your software. You would like to incorporate the following kind of device into your GUI:

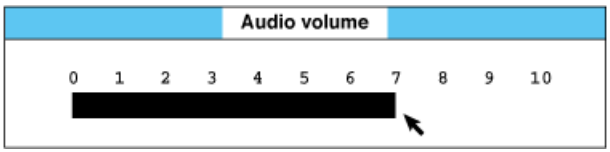


Figure 9.3 *Creating a Visual Device: the Slider*

You want your user to be able to change the width of the black bar by placing the cursor anywhere on the bar and dragging the mouse either to the left or to the right. And, as soon as the width of the bar is changed, a function in your C++ software changes the volume of the audio system accordingly. There are two different aspects to the behavior of the slider:

- **Visible Elements** From a purely graphic point of view, the appearance of the device is modified when the user intervenes with the mouse. The black rectangle becomes either narrower or wider, depending on the direction in which the cursor is dragged.

  There is thus a modification of the shape of a geometric form. In this example, it is a black rectangle that changes its width, but you could imagine many other kinds of meters, gauges, and dials that could have this same basic behavior. For example, you might have an elliptically-shaped object that becomes either thinner or fatter depending on the direction in which the cursor is dragged.

- **Functional Elements** The real purpose of the slider, of course, is to bring about changes in a domain that is quite remote from the on-screen graphics environment; namely, the audio domain. This is the *operational behavior* of the slider.

  To handle the operational behavior of the slider, we can imagine a function called `SliderValue` that returns the current value of the slider, and another function, `SliderChange`, that returns the positive or negative value by which the setting of the slider has just been changed. With these two values, you could establish links to the audio sections of your software, so that the volume is changed accordingly.

Associating a Behavior with Your Device

In the context of IBM ILOG Views, we can create a special object—a behavioral object or *interactor*—that encapsulates the particular behavior described in the section *Example: Creating a Slider*. In other words, it is an instance of the basic class `IlvInteractor`. Once you have a behavioral object, you can associate it with various concrete things—a window, a view, a black rectangle, and a few numbers—in order to build an actual slider on the screen.

In the case of the slider, we would probably begin by using `IlvInteractor` to derive a class called `IlvGaugeInteractor` with member functions such as `SliderValue` and `SliderChange` for the operational behavior that we described above.

The member function `handleEvent` of this `IlvGaugeInteractor` class would be written in such a way that it changes the size of the rectangle (or whatever other shape is used) that indicates the current value of the slider.

Since the application domain in which we want to use our slider is rather special, we would probably then derive a subclass of `IlvGaugeInteractor`, called `AudioSlider`, with special-purpose member functions for setting the values of audio parameters in a multimedia system. So, we would then have the following class hierarchy:

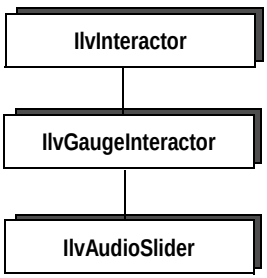
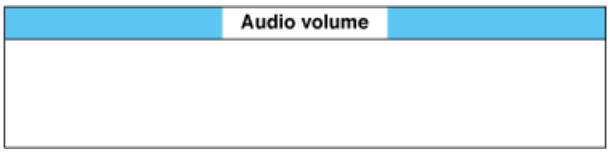


Figure 9.4 *IlvAudioSlider Hierarchy*

Building and Extending your Device

The example that follows illustrates a typical manner in which an open-ended product such as IBM ILOG Views can be extended:

1. Connect your application to a display server.
2. Create an empty top window where elements of your application can be organized and displayed.



The following code creates a container, which is the top window located in `(0, 0)`, with a size of `100` units wide and `35` units high.

```
IlvRect viewsize(0, 0, 100, 35);
IlvContainer* topview = new IlvContainer(display, "Audio volume",
                                         "Audio volume", viewsize);
```

The container coordinates the storage and the display of your graphic objects.

Since you intend to draw certain objects inside this window, and then cause one of these objects to behave in a specific manner by associating an interactor with it, you are obliged to make use of a container.

3. Place graphic objects in the container.

   The following code places two independent objects inside the container:

```
container->addObject(scale);
container->addObject(bar);
```

4. Create functional behavior.

   In this final step, you give the bar the kind of audio-volume slider behavior that is normally associated with an audio slider. Recall that this behavior was encapsulated in a class called `AudioSlider`. So, the behavior of the audio-volume slider can be given by means of the following lines:

```
IlvInteractor* inter = new AudioSlider();
bar->setInteractor(inter);
```

where the class `AudioSlider` is defined as:

```
class Audioslider
: IlvGaugeInteractor {
public:
    .../...
    virtual void doIt(IlvGauge* gauge)
    {
       setVolume(gauge->getValue());
    }
};
```

C H A P T E R

10

Dynamic Modules

A dynamic module is composed of a set of object files that are contained in a shared library (also called a *dynamic link library* or DLL). IBM® ILOG® Views is able to load dynamically, on the fly and at run time, a dynamic module that lets you define new classes, and therefore provide more functionality to a running program.

Typically, dynamic modules are used when files are read. For example, if a data file contains a reference to an `IlvGraphic` subclass that the application reading the file does not expect, IBM ILOG Views generates an error message and stops reading the file immediately. With dynamic modules, IBM ILOG Views can load the code that defines this class and make it available in a dynamic way.

For details on using dynamic modules, see:

- *IlvModule: The Dynamic Module Class*
- *Building a Dynamic Module*
- *Loading a Dynamic Module*
- *An Example: Dynamic Access*

IlvModule: The Dynamic Module Class

Every dynamic module is an instance of a subclass of the `IlvModule` base class, defined in the header file `<ilviews/base/modules.h>`.

Every dynamic module must define one and only one subclass. The constructor of this class is called when IBM® ILOG® Views loads the module, making it possible to perform all the static initializations that the module requests (graphic classes registration, for example).

Once you have declared (in a public header file) and defined your module class, you have to enable IBM ILOG Views to load it. To do so, add a call, outside the body of any function, to the macro `ILVINITIALIZEMODULE`, providing the class name of the module that you have defined as its only parameter.

Dynamic Module Code Skeleton

The skeleton of an IBM ILOG Views dynamic module is the following:

```
#include <ilviews/base/modules.h>

// Pre-initialization code goes here
// This is, typically, the declaration of global variables or
// static data members.

class MyModule
: public IlvModule {
public:
    MyModule(void*)
    {
       // Initialization code goes here
    }
};

ILVINITIALIZEMODULE(MyModule);
```

The parameter that is provided to the constructor of the module class makes it possible for you to send application-dependent data to your module initialization. This is required if your module needs external data to initialize properly. We will clarify this point later in this chapter (see the section *Explicit Mode*).

> ***Note:*** *Most of the time, the complete initialization of a module has to be split into two parts: the declaration of variables, outside the code of any function, and their initialization, which may require function calls, that must appear in the module constructor.*

Building a Dynamic Module

Dynamic modules are system-dependent. This section explains how to compile and install a dynamic module properly, or in other words how to create a shared library that will load correctly, depending on the system you are using.

UNIX Systems

If you are working on UNIX, use the following generic syntax:

```
<CCC> -c -O -I$ILVHOME/include moduleSrc.cpp
<MAKESHLIB> -o module.<SHEXT> moduleSrc.o [other object files...]
```

The table below lists the various options available for the different ports of IBM® ILOG® Views. Note that you always have to specify the extension of the module file name.

Table 10.1 *Compiling Options of Dynamic Modules for UNIX Systems*

Port Name	CCC	MAKESHLIB	SHEXT
alpha_5.1_6.5	cxx -x cxx	/usr/lib/cmplrs/cc/ld -shared	so
hp32_11_3.73	aCC +DAportable -mt - AA -z +Z	aCC -b -n -mt -AA -Wl,+s	sl
hp64_11_3.73	aCC +DAportable -mt - AA -z +D A2.0W	aCC -b -n -mt -AA +DA2.0W	sl
x86_sles10.0_4.1 x86_RHEL4.0_3.4 x86-64_RHEL4.0_3.4	g++ -fPIC	g++ -shared	so
rs6000_5.1_6.0 power32_aix5.2_7.0	xlC -qrtti=all	xlC -qmkshrobj=1024	so
ultrasparc32_8_6.2 ultrasparc32_10_11	CC -KPIC	CC -G -h	so
ultrasparc64_8_6.2 ultrasparc64_10_11	CC -KPIC -mt -xtarget=ultra -x arch=v9	CC -xtarget=ultra -xarch=v9 -G -h	so

If you are a:

- **Linux or Solaris user** Make sure that the `LD_LIBRARY_PATH` variable contains a path to the modules to be loaded. On Linux, the executable must be linked with the library `libdl.so`.

Windows Systems

On Microsoft Windows, a dynamic module is in fact a DLL.

This functionality is only available in the format `dll_mda`. This is due to the fact that IBM ILOG Views stores the registered classes in a global variable that would be re-created locally by every module if we tried to link the "client" (that is, the application that loads the module) statically.

All you need to do to build your module is add the flag `/DILVDLL` when compiling your object files. For an optimized code, the complete set of compiler flags for the module files should be the following:.

Table 10.2 *Compiling Options of Dynamic Modules for Windows Systems*

Port Name	Compiler Flags
`x86_.net2003_7.1`	`CL /Gs /Ot /Ox /O2 /c /DWIN32 /MD /W3 /G3 /DILVDLL /I<ILVHOME>/include moduleSrc.cpp`
`x86_.net2005_8.0` and `x86_.net2008_9.0`	`CL /Gs /O1 /c /DWIN32 /MD /W3 /DILVDLL /I<ILVHOME>/include moduleSrc.cpp`
`x64_.net2008_9.0`	`CL /Gs /O1 /c /DWIN64 /MD /W3 /DILVDLL /I<ILVHOME>/include moduleSrc.cpp`

Use the following lines to link your module:

```
LINK /SUBSYSTEM:WINDOWS /DLL <ILVHOME>/lib/[PLATFORM]/dll_md/<lib>.lib\
<systemLibs> -OUT:<moduleName>.dll moduleSrc.obj [objectfiles...]
```

where `[PLATFORM]` can take one of the following values `x86_.net2003_7.1`, `x86_.net2005_8.0`, `x86_.net2008_9.0`, `x64_.net2008_9.0`.

Note that for `x64_.net2008_9.0`, you must also add the following linker option:

```
/MACHINE:X64
```

As mentioned, the application that you want to be module-enabled must be linked with the IBM ILOG Views DLLs as well. Of course, as always when using DLLs, you have to make sure that your system path contains the access path to the directory where you want to store your modules.

Versioning Note

IBM ILOG Views dynamic modules have no versioning mechanism. You must make sure that the installed dynamic modules are binary-compatible with the application that loads them. This means that a module that was built with IBM ILOG Views version X.Y will run with all the X.Y.Z versions, but will have to be recompiled if loaded by an application that was developed with IBM ILOG Views X.W.

Loading a Dynamic Module

Basically, a dynamic module can be loaded in two different ways, either implicitly or explicitly.

- *Implicit Mode*: The implicit mode is transparent for the end user, which means that new classes are loaded by the application when they are referred to by their name.
- *Explicit Mode*: In explicit mode, you specify precisely which module is to be loaded and where it can be found.

Implicit Mode

In implicit mode, IBM® ILOG® Views loads new classes when they appear in a data file and when a class name is not registered in the loading application.

For this mode to work properly, you must create a *module definition file*. For IBM ILOG Views to know automatically which classes are defined and which dynamic module they are in, you must create a *module definition file* (with the `.imd` extension) and place it in the same directory as your shared library. This file must have the same name as the module, except that its extension must be `imd`, instead of `so`, `sl`, or `dll`. Creating this file allows IBM ILOG Views to know which classes are defined without having to open and read all the dynamic modules it finds.

Let us suppose that you have created the module `myModule.dll` on Microsoft Windows. You have to create a file called `myModule.imd` in the same directory as `myModule.dll` to enable IBM ILOG Views to call the classes defined in this file automatically.

The contents of this file is ASCII text that will be read by IBM ILOG Views at start-up to identify which classes are defined and in which module they are located. The file must always begin with the following lines:

```
<?xml version="1.0"?>
<module>
```

and end with the closing tag:

```
</module>
```

Within the `<module>` block, you must have any number of groups of the form:

```
<class name = "NewClass" rootClass = "RootClass"/>
```

where `RootClass` is the name of the topmost class in the class hierarchy (`IlvGraphic`, for example), and `NewClass` is the name of the class you created. If you have defined more than one class in your module, just add other `<class>` tags.

If the classes that are defined in your module derive from more than one root base class, you can add as many new blocks as there are root base classes.

When an IBM ILOG Views application starts, the module path is read to search for possible modules (files that have the proper extension). If both a module and an associated `imd` file are found, the description file is read and the information it contains is stored for future implicit class loading.

Explicit Mode

You can use the explicit module loading mode when your application knows which modules to load and where they can be found.

Let us suppose that you have created a module called `myModule.dll` on Microsoft Windows in `C:\ilog\Views\Modules`. If this directory is in your PATH variable, you can directly load your module (and therefore perform all the appropriate initialization) by calling the static member function `Load`:

```
IlvModule* myModule = IlvModule::Load("myModule", myParameter);
```

IBM® ILOG® Views will attempt to open this module for you, sending the parameter `myParameter` to the module constructor, which you can use to send specific information to specific modules. When loading a module in implicit mode, this parameter is always set to 0.

The first parameter lets you specify a name for a module. It is sometimes easier to manipulate module names instead of pointers to `IlvModule`.

If the loading fails, `Load` returns 0.

The dynamic library is unloaded when the module instance that is returned is destroyed.

An Example: Dynamic Access

The purpose of this example is to make a graphic class dynamically accessible from a running application.

Let us suppose that we have a class `CrossedRectangle`, deriving from `IlvFilledRectangle`, that is displayed as an outlined rectangle with a cross in it. The example is developed in:

- *Writing the Sample Module Definition File*
- *Implementing the New Class*
- *Loading and Registration of the Example*
- *Registration Macros*
- *Adding the Sample Class to a Dynamic Module*

Writing the Sample Module Definition File

First, you have to make sure that IBM ILOG Views properly loads the code of this class when a data file refers to its name, thus allowing the corresponding module to be implicitly loaded.

To that end, we have to write a module definition file, which in our example will be very simple since we only have one class.

Here is the content of an appropriate module definition file for this module. Since we have only one class, whose root base class is `IlvGraphic`, the definition file is as follows:

```
<?xml version = "1.0"?>
<module name="correct" version="1.0">
<class name = "CrossedRectangle" rootClass = "IlvGraphic"/>
</module>
```

We could have added many other classes to this module definition file, even classes that do not inherit from `IlvGraphic`. You can add classes to the same module in an incremental way.

Implementing the New Class

Once the module definition file is written, we have to do some more work to implement this new class. The specifications are quite simple:

- Create a filled rectangle that displays a cross inside a frame.
- Make sure that this new class is persistent.
- Add this class to a dynamic module.

Although you are probably already familiar with the first two steps, we provide the corresponding code below. The last point represents the most difficult part. We must need to know how we are going to proceed to register our class properly. Unfortunately, the macro `IlvRegisterClass` does not work in a dynamic loading context, or at least not in a portable way. IBM ILOG Views provides a few macros very close to the ones that you are already familiar with for solving this problem. We will discuss them after you see the code for the `CrossedRectangle` class, which is given below.

```
#include <ilviews/graphics/rectangl.h>

class CrossedRectangle
: public IlvFilledRectangle {
public:
    MyRectangle(IlvDisplay* display,
                const IlvRect& size, IlvPalette* pal=0)
    : IlvFilledRectangle(display, drawrect, palette)
    {}
    virtual void draw(IlvPort* dst, const IlvTransformer* t = 0,
                      const IlvRegion* clip = 0) const;
    DeclareTypeInfoRO();
```

```
    DeclareIOConstructors(MyRectangle);
};

// Copy constructor
CrossedRectangle::CrossedRectangle(const CrossedRectangle& source)
: IlvFilledRectangle(source)
{}

// Read constructor
CrossedRectangle::CrossedRectangle(IlvInputFile& is,
                                   IlvPalette* pal)
: IlvFilledRectangle(is, pal)
{}

void
CrossedRectangle::draw(IlvPort* dst, const IlvTransformer* t,
                       const IlvRegion* clip) const
{
    if (clip)
        _palette->setClip(clip);
    IlvRect r = _drawrect;
    if (t)
        t->apply(r);
    dst->drawRectangle(_palette, r);
    dst->drawLine(_palette, r.upperLeft(), r.lowerRight()));
    dst->drawLine(_palette, r.upperRight(), r.lowerLeft());
    if (clip)
        _palette->setClip();
}

IlvPredefinedIOMembers(CrossedRectangle)
```

The `draw` method is straightforward.

You can see that, as required, this class contains a copy constructor and persistence-related methods. (It has no `write` method since we do not have any information to save, and so we used the `DeclareTypeInfoRO` macro in the class declaration).

We have already addressed two of the three points for implementing the new class.

The usual way to register this class with the IBM ILOG Views persistence mechanism would be to use the well-known statement:

```
IlvRegisterClass(CrossedRectangle, IlvFilledRectangle);
```

which appears outside the body of any function.

If an application links with that code, it will be able to manipulate, save, and read instances of the class `CrossedRectangle`. Remember, however, that you want to make the `CrossedRectangle` class known to existing applications that were not aware of it when they were developed so that they can read data files generated by applications in which this class is defined. To do so, you have to plug this class into a dynamic module, and you cannot use the preceding macro for this purpose.

Loading and Registration of the Example

It is important to understand what happens when a module is loaded and what registration actually does:

- When a module is loaded, its constructor is called.
- Registration is both the declaration of class-level variables that store class-level attributes and function calls that actually update these variables.

With the dynamic modules feature, IBM ILOG Views provides an alternate form of the `IlvRegisterXXXClass` macros, which are used in many places (in `IlvGraphic` and `IlvNamedProperty` subclasses, for example). This set of macros separates the declaration part of the registration from its definition.

The name of the macro used for the **declarative** part of the registration is similar to `IlvRegisterXXXClass`, except that `IlvRegister` is replaced by `IlvPreRegister`. The second parameter of `IlvRegisterXXXClass` is dropped. This macro call must appear outside the body of any function (it declares only class-level variables).

The name of the macro used for the **definition** part of the registration is similar to `IlvRegisterXXXClass`, except that `IlvRegister` is replaced by `IlvPostRegister`. The second parameter of `IlvRegisterXXXClass` remains. Thus, macro calls must appear inside the body of a function that must be called to actually perform the proper registration (it does call code to register new classes).

In our example, to register the graphic class, we have to use both the macros `IlvPreRegisterClass` (outside the body of any function) and `IlvPostRegisterClass` (inside the body of a function).

Registration Macros

Below is a list of the macros that you must use for registering most of the IBM ILOG Views classes that are persistent (`c` indicates the class name that is being registered, and `s` indicates its parent class name):

Table 10.3 *Registration Macros in Dynamic Modules*

Class Name	Static Registration Macros	Registration Macros in Dynamic Modules
`IlvGraphic`	`IlvRegisterClass(c, s);`	`IlvPreRegisterClass(c);` `IlvPostRegisterClass(c, s);`
`IlvNamedProperty`	`IlvRegisterPropertyClass(c, s);`	`IlvPreRegisterPropertyClass(c);` `IlvPostRegisterPropertyClass(c, s);`
`IlvView`	`IlvRegisterViewClass(c, s);`	`IlvPreRegisterViewClass(c);` `IlvPostRegisterViewClass(c, s);`

Table 10.3 *Registration Macros in Dynamic Modules (Continued)*

Class Name	Static Registration Macros	Registration Macros in Dynamic Modules
IlvGadgetItem	IlvRegisterGadgetItemClass(c, s);	IlvPreRegisterGadgetItemClass(c); IlvPostRegisterGadgetItemClass(c, s); For details on gadgets, refer to the Gadgets documentation.
IlvNotebookPage	IlvRegisterNotebookPageClass(c, s);	IlvPreRegisterNotebookPageClass(c); IlvPostRegisterNotebookPageClass(c, s);
IlvSmartSet	IlvRegisterSmartSetClass(c, s);	IlvPreRegisterSmartSetClass(c); IlvPostRegisterSmartSetClass(c, s);
IlvGroup	IlvRegisterGroupClass(c, s);	IlvPreRegisterGroupClass(c); IlvPostRegisterGroupClass(c, s);
IlvGroupNode	IlvRegisterGroupNodeClass(c, s);	IlvPreRegisterGroupNodeClass(c); IlvPostRegisterGroupNodeClass(c, s);
IlvUserAccessor	IlvRegisterUserAccessorClass(c, s);	IlvPreRegisterUserAccessorClass(c); IlvPostRegisterUserAccessorClass(c, s);

Because we are dealing with a subclass of `IlvGraphic`, the `IlvPreRegisterClass` and `IlvPostRegisterClass` macros are all we need to complete the source code of our module.

Adding the Sample Class to a Dynamic Module

Here is the code that we have to add to the definition of the `CrossedRectangle` class to make the final source code compilable as an IBM ILOG Views dynamic module:

```
#include <ilviews/modules.h>

IlvPreRegisterClass(CrossedRectangle);

class MyModule
: public IlvModule
{
public:
    MyModule(void*)
    {
        IlvPostRegisterClass(CrossedRectangle, IlvFilledRectangle);
    }
};

ILVINITIALIZEMODULE(MyModule);
```

Note that you can add this code to a `#if defined()/#else/#endif` precompiler block, with the regular

```
IlvRegisterClass(CrossedRectangle, IlvFilledRectangle);
```

in the `#else` part of `if`. You will then be able to compile your code as a regular static object file or as a dynamic module:

```
#if defined(MAKE_A_MODULE)
#include <ilviews/modules.h>

IlvPreRegisterClass(CrossedRectangle);

class MyModule
: public IlvModule
{
public:
    MyModule(void*)
    {
        IlvPostRegisterClass(CrossedRectangle, IlvFilledRectangle);
    }
};

ILVINITIALIZEMODULE(MyModule);
#else  /* DONT_MAKE_A_MODULE */
IlvRegisterClass(CrossedRectangle, IlvFilledRectangle);
#endif /* DONT_MAKE_A_MODULE */
```

C H A P T E R

11

Events

This chapter contains information about events and event loops. You will see how to manipulate timers, how to add external sources of data, and how to customize the event loop. Refer to the sections:

- *IlvEvent: The Event Handler Class*
- *The IlvTimer Class*
- *External Input Sources (UNIX only)*
- *Idle Procedures*
- *Low-level Event Handling*

IlvEvent: The Event Handler Class

Mouse and keyboard events are handled by the `IlvEvent` class.

Recording and Playing Back Event Sequences: IlvEventPlayer

Using `IlvEventPlayer`, IBM® ILOG® Views can record sequences of events that occur in the views that it controls. These event sequences can be saved or read from a data file, and played back at any speed.

Functions Handling Event Recording

A set of global functions exists that let you start recording events and playing an event sequence back:

- `IlvCurrentEventPlayer`
- `IlvRecordingEvents`

The IlvTimer Class

IBM® ILOG® Views has an internal mechanism for implementing *timers*. The internal mechanism is hidden and system-dependent. It is based on the `IlvTimer` class.

The purpose of a timer is to call a function repeatedly, once every given time period. If you want one of your functions to be called in this way, create an `IlvTimer` instance and call its member function `IlvTimer::run`. The timer object calls its member function `IlvTimer::doIt` each time this period expires. Timers are based on the timeout mechanisms of the display system.

The timer automatically repeats the call to `IlvTimer::doIt` after every period if not specified to run only once. Before calling `IlvTimer::doIt`, the event loop disables the timer. After returning from `IlvTimer::doIt`, if it is not a run-once timer and if it is still disabled (it can be enabled by a call to `IlvTimer::run` from within the timer's callback), the timer is enabled again. This mechanism shows that a timer is not active during its own callback, even if the callback contains a local event loop. This is only true when the timer is triggered by the event loop, not when the application explicitly calls the `IlvTimer::doIt` method. The application is responsible for deleting the timers it has created.

> ***Note:*** *If the function called by the timer takes too much time to execute compared to the periodicity of the timer, the periodicity may not be respected.*

The `IlvTimer` class can be used in two different cases:

- The first case supposes that you have a user-defined function which must match the `IlvTimerProc` type:

```
typedef void (* IlvTimerProc)(IlvTimer* timer,IlAny    userarg);
```

  In this case, you simply instantiate an `IlvTimer` object, specifying this function and an argument it can use.

- The second case uses a derived subclass of `IlvTimer` with overloading of the member function `IlvTimer::doIt`.

External Input Sources (UNIX only)

On UNIX platforms, IBM® ILOG® Views allows the application to add new sources of input using file descriptors. These alternate input sources can be registered and unregistered with the `IlvEventLoop` methods `IlvEventLoop::addInput`, `IlvEventLoop::addOutput`, `IlvEventLoop::removeInput`, and `IlvEventLoop::removeOutput`. For backward compatibility, the old functions `IlvRegisterInput`, `IlvRegisterOutput`, `IlvUnRegisterInput` and `IlvUnRegisterOutput` are still supported; they are equivalent to:

```
IlvEventLoop::getEventLoop()->[add|remove][Input|Output]()
```

IBM ILOG Views does not read any data from these input sources but rather monitors them and notifies the application when the file descriptor has received input or is ready for writing. When this happens, IBM ILOG Views calls the application callback routine associated with the given source of input; this callback routine is then responsible for reading (or writing) data from (or to) the file descriptor. It is also the responsibility of the application to open the file descriptors before adding them as new input sources in IBM ILOG Views and to close them after removing them.

Here is an example of a short IBM ILOG Views program reading from the standard input and copying it one word per line to the standard output.

```
#include <strstream.h>
#include <string.h>
#include <ilviews/view.h>

static void MyInputCallback(int, IlAny) {
   char buffer[1048];
   cin >> buffer;
   cout << buffer << endl;
   if (!strcasecmp(buffer, "quit"))
       exit(0);
}

int main(int, char*[]) {
   IlvEventLoop::getEventLoop()->addInput(0 /*stdin*/,
                                    MyInputCallback, 0, 0);
   IlvMainLoop();
}
```

Idle Procedures

An idle procedure is a function provided by the application and called by the event loop at times when the application would otherwise be idle, waiting for events. Idle procedures must perform short computations; if an idle procedure is too long, it can affect the interactive response of the application.

Idle procedures are useful to perform tasks that do not need to be done before other tasks can continue. Their immediate completion should not be crucial to the application. For instance, idle procedures can be used to create hidden dialog boxes before they are requested by user actions.

When an idle procedure returns `IlTrue`, it is automatically removed and will not be called again. If it returns `IlFalse`, it is called each time the application is idle, until it returns `IlTrue` or it is explicitly removed by the application.

To register and unregister idle procedures, the application uses the `IlvEventLoop` methods `IlvEventLoop::addIdleProc` and `IlvEventLoop::removeIdleProc`. The returned value of `IlvEventLoop::addIdleProc` is an ID that can be used for explicitly removing an idle procedure by calling `IlvEventLoop::removeIdleProc`. Generally, idle procedures do not need to be removed; they return `IlTrue` instead.

Low-level Event Handling

The most common way for an application to handle events is to call `IlvMainLoop` after the application is initialized. `IlvMainLoop` is simply an infinite loop that gets the next incoming event and dispatches it to the appropriate component. However, some applications may need to define their own event loop. To do this, IBM® ILOG® Views provides the following functions or methods:

- `IlvDisplay` methods for defining event loops are:
  - `IlvDisplay::hasEvents`
  - `IlvDisplay::readAndDispatchEvents`
  - `IlvDisplay::waitAndDispatchEvents`
- `IlvEventLoop` methods for defining event loops are:
  - `IlvEventLoop::pendingInput`
  - `IlvEventLoop::processInput`
  - `IlvEventLoop::nextEvent`
  - `IlvEventLoop::dispatchEvent`

Main Loop Definition: An Example

Here is a list of constructions that are equivalent to `IlvMainLoop`:

```
while (1)
    display->waitAndDispatchEvents();

while (1)
```

```
   IlvEventLoop::getEventLoop()->processInput(IlvInputAll);
```

Windows platforms only:

```
MSG msg;
while (IlvEventLoop::getEventLoop()->nextEvent(&msg))
    IlvEventLoop::getEventLoop()->dispatchEvent(&msg);

MSG msg; // obsolete version
while (IlvNextEvent(&msg))
    IlvDispatchEvent(&msg);
```

UNIX platforms only:

```
XEvent xev;
while (1) {
    IlvEventLoop::getEventLoop()->nextEvent(&xev);
    IlvEventLoop::getEventLoop()->dispatchEvent(&xev);
}

XEvent xev; // obsolete version
while (1) {
    IlvNextEvent(&xev);
    IlvDispatchEvent(&xev);
}
```

UNIX platforms using only `libmviews` (as opposed to `libxviews`):

```
XtAppMainLoop(IlvApplicationContext());

XEvent xev;
while (1) {
    XtAppNextEvent(IlvApplicationContext(), &xev);
    XtDispatchEvent(&xev);
}
```

C H A P T E R

12

IlvNamedProperty: The Persistent Properties Class

The class `IlvNamedProperty` is used to associate application-dependent information with IBM® ILOG® Views objects. This information, called a named property, is stored in a subclass of `IlvNamedProperty` that you define. Unlike a user property, a named property is copied with your object and made persistent so that its content is preserved when data files are saved or read.

Using and extending named properties are described in the following sections:

- *Associating Named Properties with Objects*
- *Extension of Named Properties*

Associating Named Properties with Objects

As with user properties, you associate a named property with a graphic object using an `IlSymbol`. An `IlSymbol` is attached to one and only one subclass of `IlvNamedProperty` to ensure that the type of the retrieved property is correct.

To handle named properties, you can use the following three member functions of the class `IlvGraphic`:

```
IlvNamedProperty* getNamedProperty(const IlSymbol*) const;
```

```
IlvNamedProperty* setNamedProperty(IlvNamedProperty*);
IlvNamedProperty* removeNamedProperty(IlSymbol*);
```

You can see that an `IlSymbol` is all you need to indicate which named property you are dealing with.

A Predefined Named Property: the Tooltip

An example of a predefined named property is the *tooltip*, a small text window that pops up when the pointing device enters a panel control element (such as a gadget) and stays for a moment. The class `IlvToolTip` is defined in the header file `<ilviews/graphics/tooltip.h>`.

To set a tooltip for a graphic object, retrieve it, and then remove it, write the following:

```
obj->setNamedProperty(new IlvToolTip("Text"));
...
IlvToolTip* toolTip = IlvToolTip::GetToolTip(obj);
...
delete obj->removeNamedProperty(toolTip->getSymbol());
```

Named properties may be transferred from object to object. To remove a named property from the list of named properties without deleting it, use the `IlvGraphic::removeNamedProperty` member function. To delete a named property and thus clear the memory it occupies, you must call `delete` explicitly.

Because a subclass of `IlvNamedProperty` and the symbol (that is, the pointer to `IlSymbol`) that refers to it are tightly coupled, most of the time this symbol is a static data member of the property class. Note, however, that this is not mandatory. Using a static data member of the property class makes it possible for you to retrieve a named property from an object using a symbol that you may not know, but which is directly accessible from the class.

In the above example, you can see that the symbol used by the tooltip property never appears. To retrieve the named property, we simply get the property symbol using the member function `IlvNamedProperty::getSymbol`.

Note that named properties are copied and saved with the object, and they are deleted when you delete the object. Named properties behave like additional data members of existing classes, with the possibility of defining a powerful API to access the data at the level of the named property class.

Extension of Named Properties

Creating your own named property class is a straightforward, three-step procedure:

1. Create a subclass of `IlvNamedProperty`.
2. Choose the symbol that will be used to access this property.

We remind you that in IBM ILOG Views all symbols whose names begin with the characters "`_ilv`" are reserved for internal use.

3. Define how this class will be persistent and register this class with IBM ILOG Views.

Using a very simple example—the storage of two values—the following section explains how to create a named property that you can associate with graphic objects. You can, however, build named properties with member functions to handle more elaborate data members, or have named properties that store pointers to existing classes. Using named properties, you can link application data with a high level of complexity to light graphic objects, with minimum coding and without altering the API of your application classes.

Example: Creating a Named Property

To illustrate, we are going to create a named property that holds both an integer and a character string and make it easily accessible and persistent.

The general development tasks are:

- *Declaring the Named Property: Header File*
- *Defining the Symbol for Accessing the Named Property*
- *Defining the Constructor for the Named Property*
- *Defining the setString Member Function*
- *Defining the Persistence and Copy Constructors*
- *Defining the write Member Function*
- *Providing an Entry Point to the Read and Copy Constructors*
- *Registering the Class*
- *Using the New Named Property*

Declaring the Named Property: Header File

The named property that we are going to create must be a subclass of `IlvNamedProperty`. The named property stores an integer and a string. Its complete header file is given below:

```
#include <ilviews/base/graphic.h>

class MyProperty
: public IlvNamedProperty
{
public:
    MyProperty(int   integer,
               char* string);
    virtual ~MyProperty();

    int         getInteger() const      { return _integer; }
    void        setInteger(int integer) { _integer = integer; }
```

```
    const char* getString()  const { return _string; }
    void        setString(const char* string);

    static IlSymbol* GetSymbol();

    DeclarePropertyInfo();
    DeclarePropertyIOConstructors(MyProperty);

private:
    int   _integer;
    char* _string;
    static IlSymbol* _Symbol;
};
```

In addition to the two data members `_integer` and `_string` (and their accessors), we will focus on the `_Symbol` static data member and on the two macro calls that appear in the declaration part of the class, namely `DeclarePropertyInfo` and `DeclarePropertyIOConstructors`.

Note that the destructor of the class is virtual like the one of the base class `IlvNamedProperty`.

Defining the Symbol for Accessing the Named Property

First we are going to define the property symbol that will be used to access the class. A simple way to define this symbol is to make it a static data member of your property class, `_Symbol`, and provide it with a public accessor, `GetSymbol`. By doing this, any application will be able to retrieve an instance of `MyProperty` without having to know which symbol is used to associate it with an object.

Therefore, the public and static accessor `GetSymbol` is defined to return the appropriate `IlSymbol` and create it if necessary.

In the code below, an extract of the implementation file, we define both the accessor to the property symbol and the static data member, which we initialize to `0`. Note that the symbol is created the first time it is queried in `MyProperty::GetSymbol`.

```
IlSymbol*
MyProperty::GetSymbol()
{
    if (!_Symbol)
       _Symbol = IlGetSymbol("MyPropertySymbol");
    return _Symbol;
}

IlSymbol* MyProperty::_Symbol = 0;
```

Defining the Constructor for the Named Property

Let’s now examine the constructor and the destructor. All we need to do is call the constructor of the parent class, `IlvNamedProperty`, and initialize our data members:

```
MyProperty::MyProperty(int   integer,
                       char* string)
```

```
: IlvNamedProperty(GetSymbol()),
  _integer(integer),
  _string(0)
{
    setString(string);
}

MyProperty::~MyProperty()
{
    if (_string)
        delete [] _string;
}
```

The first time a property of the `MyProperty` type is created, the static member function `GetSymbol` is called to set the static data member `_Symbol` to a valid value.

The `string` parameter is copied by the `setString` member function that will check whether the data member `_string` is valid. This is why we initialize this data member to `0` in the initializers of the constructor. This parameter is destroyed in the destructor, if it was valid.

Defining the setString Member Function

Below is the definition of the `setString` member function that copies and stores the string:

```
void
MyProperty::setString(const char* string)
{
    if (_string)
        delete [] _string;
    _string = string
                ? strcpy(new char [strlen(string)+1], string)
                : 0;
}
```

This code is quite simple. If a valid string—a non-null string— was stored, it is destroyed. If the parameter is valid—non-null—a copy of this string is made and stored. If the parameter is not valid, the data member is simply reset to `0`.

At this stage, our class can store and retrieve both an integer and a character-string value.

Defining the Persistence and Copy Constructors

For our named property to be complete, we have to add the class-level information and persistence-related member functions. The easiest way to do this is to use the following two macros in the body of the class declaration:

- `DeclarePropertyInfo` declares the class information data members for the `MyProperty` class. These members are used to retrieve information, such as the class name and its hierarchy. It also declares the member functions that are required to implement persistence for this class.
- `DeclarePropertyIOConstructors` declares the constructors required for persistence and copy.

These macros make it very easy to add copy and persistence functionality to your class.

Once the macros have been declared, all we have to do to add copy and persistence features to our class is to define a copy constructor and a constructor that take an `IlvInputFile` reference as its parameter:

```
MyProperty::MyProperty(const MyProperty& source)
: IlvNamedProperty(GetSymbol()),
  _integer(source._integer),
  _string(0)
{
    setString(source._string);
}

MyProperty::MyProperty(IlvInputFile& i, IlSymbol* s)
: IlvNamedProperty(GetSymbol()),
  _integer(0),
  _string(0)
{
    // 's' should be equal to GetSymbol()
    i.getStream() >> _integer >> IlvQuotedString();
    setString(IlvQuotedString().Buffer);
}
```

The first constructor initializes a new instance of `MyProperty` with a copy of its source parameter.

The second constructor reads the provided input stream to initialize its instance with what is read.

Defining the write Member Function

Now that we are able to read a new instance of the class, we can save it. To do so, we have to define a `write` member function, which is implicitly declared by the macro `DeclarePropertyInfo`.

```
void
MyProperty::write(IlvOutputFile& o) const
{
    o.getStream() << _integer << IlvSpc() << IlvQuotedString(_string);
}
```

The saving order must be the same as the reading order.

Note that you may define a named property that does not have any additional information to save. In this case, you would use the `DeclarePropertyInfoRO` macro instead of `DeclarePropertyInfo` in the class declaration and drop the `write` member function that would be useless.

Providing an Entry Point to the Read and Copy Constructors

To provide IBM ILOG Views with an entry point to the read and copy constructors, you must add another macro to the implementation file, outside the body of any function, as follows:

```
IlvPredefinedPropertyIOMembers(MyProperty)
```

Calling this macro actually creates a `read` static member function that invokes the read constructor. It also defines a `copy` member function that calls the copy constructor.

Registering the Class

The final step to have our named property up and running in our application is to register the `MyProperty` class with IBM ILOG Views as follows:

```
IlvRegisterPropertyClass(MyProperty, IlvNamedProperty);
```

Calling the macro `IlvRegisterPropertyClass` registers the class `MyProperty` with the IBM ILOG Views persistence mechanism.

Using the New Named Property

You can now use this new named property as an extension of any graphic object with which it would be associated:

```
IlvGraphic* myObject = ...;
myObject->setNamedProperty(new MyProperty(12, "Some text"));
...
MyProperty* property =
          (MyProperty*)(myObject->getNamedProperty(MyProperty::GetSymbol());
if (property && (property->getInteger() == someValue))
    doSomething();
```

You have extended your graphic object's API in a persistent manner, without subclassing the base class.

C H A P T E R

13

Printing in IBM ILOG Views

IBM® ILOG® Views provides a framework for printing. This framework consists of the following classes:

- *The IlvPrintableDocument Class* describes a document, that is, a list of printable objects associated with page layouts.
- *The IlvPrintable Class* describes a printable object, various subclasses deal with printing containers, manager views, text, and so forth.
- *The IlvPrintableLayout Class* describes the page layout of the document using predefined areas such as background, foreground, header, and footer. Predefined layouts allow printing on one or multiple pages, or identity layouts.
- *The IlvPrinter Class* describes the printer and its physical characteristics such as paper format, margins, color or grayscale capabilities, and page orientation.
- *The IlvPrintUnit Class* describes a printing unit and allows conversion of various units such as Pica, Centimeter, Inches, and Points.
- *The IlvPaperFormat Class* describes physical paper formats such as A4, Letter, and so forth.
- The *Dialogs* section describes the user interface dialogs provided by IBM ILOG Views for choosing printer and printer characteristics. A print preview dialog is also available in the Gadgets package.

The IlvPrintableDocument Class

The `IlvPrintableDocument` class manages a list of printable objects. It uses iterators to sequence through the printable objects. It provides a default layout, but each printable can specify its own layout.

Multiple copies of the document can be printed using two modes:

- The whole document is printed n times.
- Each page is printed n times, then the next page is printed.

Iterators

Iterators are instances of the inner class `IlvPrintableDocument::Iterator`. The most used are returned by the following `IlvPrintableDocument` methods:

- `IlvPrintableDocument::begin() const;`
- `IlvPrintableDocument::end() const;` iterators are then used like any other variable.

Example

```
IlvPrintableDocument document;
// add some printables to the document
document.append(new IlvPrintableContainer(container);
.....
// the iterate through the printables
IlvPrintableDocument::Iterator begin = document.begin();
IlvPrintableDocument::Iterator end = document.end();

for (IlvPrintableDocument::Iterator iter = document.begin();
     iter != end;
     ++iter) {
   // do something with the printable.
   IlvPrintable* printable = iter.getPrintable();

}
```

The IlvPrintable Class

`IlvPrintable` is an abstract class that provides a base for describing objects that can be printed. Its is associated with a printable job that contains the printing parameters for a given job.

A printable can be described by subclassing the following methods:

- `public virtual IlvRect getBBox(IlvPrintableJob const& job) const = 0;`
- `protected virtual IlBoolean internalPrint(IlvPrintableJob const& job) const = 0;`

A few subclasses of `IlvPrintable` are available:

- `IlvPrintableContainer` encapsulates an instance of `IlvContainer`.

```
// declares a printable using the given region of a container.
// if the rectangle is null then the whole container is printed.
IlvPrintableContainer* printcont = new IlvPrintableContainer(container,
                                                             &rect);
```

- `IlvPrintableText` allows printing of text. An alignment parameter can be specified.

```
// declares a printable using a simple text.
IlvPrintableText* printtext = new IlvPrintableText
                                        (display->defaultPalette(),
                                         "This is a text",
                                          IlvCenter);
```

- `IlvPrintableFormattedText` allows printing of text with various predefined attributes. Each Conversion specification is introduced by the character `%`. The following attributes are defined:

`%p`	The index of the page will be printed.
`%P`	The total number of pages will be printed.
`%N`	The document name will be printed.
`%y`	The year will be printed.
`%M`	The month will be printed numerically.
`%d`	The day of the month will be printed.
`%h`	The hour of the day (0-24) will be printed.
`%H`	The hour will be printed.
`%m`	The minute will be printed.
`%s`	The second will be printed.
`%AM`	The AM/PM indicator will be printed in upper case.
`%am`	The am/pm indicator will be printed in lower case.

You may still print `%p` codes by replacing them with `%\p`.

```
// declares a printable formated text
IlvPrintableFormattedText* printftext = new
   IlvPrintableFormattedText(display->defaultPalette(),
                             "%N : (Page %p/%P - %d/%M/%y - %h:%m:s)");
```

- `IlvPrintableGraphic` encapsulates an instance of `IlvGraphic`. Any `IlvGraphic` object can be printed.

```
// declares a printable graphic
IlvGraphic* ellipse = new IlvFilledEllipse(display,
                                           IlvRect(0, 0, 100, 50));
IlvPrintableGraphic* printgraphic = new IlvPrintableGraphic(ellipse);
```

- `IlvPrintableFrame` encapsulates a simple rectangle.

```
// declares a printable frame
IlvPrintableFrame* printframe = new IlvPrintableFrame
                                        (display->defaultPalette());
```

- `IlvPrintableManager`, `IlvPrintableMgrView`, and `IlvPrintableManagerLayer` (available only with the manager package) allow a whole manager, a manager view, or a manager layer, respectively, to be printed.
- `IlvPrintableComposite` allows you to define a printable as a composition of printables.

The IlvPrintableLayout Class

`IlvPrintableLayout` is an abstract class that is the base class for describing page layouts. It defines a usable area by specifying left, right, top, bottom, and gutter margins.

It defines five subareas within the usable area and associates them with printables:

- The main area that will be used to print the main printable.

  You can choose to stretch the printable to the usable area or to keep the printable aspect ratio.
- The header area that will be used to print a header printable.
- The footer area that will be used to print a footer printable.
- The background area that will be printed behind the main area.
- The foreground area that will be printed in front of the main area.

The dimensions of the header and footer areas can be specified.

Predefined layouts are:

- `IlvPrintableLayoutOnePage` lays out the printable on one page. This layout uses a single page to render the printable.
- `IlvPrintableLayoutMultiplePages` lays out the printable on an array of pages. The dimension of the pages matrix is user specified.

This layout defines a virtual page that spans through the multiple pages. The header area is defined at the top of the virtual page, the footer area at the bottom of the virtual page.

- `IlvPrintableLayoutIdentity` allows the printed document to be the same size as the printable.

  This layout inherits from `IlvPrintableLayoutMultiplePages` and uses whatever number of pages are necessary.

- `IlvPrintableLayoutFixedSize` allows you to choose the printed document size.

  This layout inherits from `IlvPrintableLayoutMultiplePages` and uses whatever number of pages are necessary.

The IlvPrinter Class

The `IlvPrinter` class describes a physical printer with characteristics such as paper size, paper orientation, and physical margins. It encapsulates an instance of `IlvPort`.

This is an abstract class and it has two predefined subclasses:

- `IlvPSPrinter` that allows you to print to a PostScript file.

```
// creating a PostScript printer
IlvPSPrinter* psprinter = new IlvPSPrinter(display);
psprinter->setPaperFormat(*IlvPaperFormat::Get("A3"));
psprinter->setOrientation(IlvPrinter::Landscape);
psprinter->setDocumentName("viewsprint.ps");
```

- `IlvWindowsPrinter` that allows printing to a printer connected to a Windows computer (this class is only available on Windows).

  Some characteristics are dependent on the printer and cannot be set, such as paper size or margins.

```
// creating a Windows printer
IlvWindowsPrinter* wprinter = new IlvWindowsPrinter(display);
```

The IlvPrintUnit Class

The `IlvPrintUnit` class allows you to describe a dimensional unit. Units of various types can be converted.

Four commonly used units have been defined:

- `IlvPrintPointUnit` represents units in centimeters. This is the reference unit.
- `IlvPrintCMUnit` represents units in centimeters.

- `IlvPrintInchUnit` represents units in inches.
- `IlvPrintPicaUnit` represents units in centimeters.

This class is mainly useful when using `IlvPSPrinter`.

Converting units:

```
IlvPrintCMUnit oneMeter(100.0);
IlvPrintInchUnit oneMeterInInches(oneMeter);
IlvDim result = oneMeterInInches.getUnits();
```

The IlvPaperFormat Class

The `IlvPaperFormat` class describes paper formats. Paper formats can be registered and queried by name.

A number of commonly used paper formats have been preregistered. Dimensions must be given in PostScript points.

> ***Note:*** *On the Windows platform, when using* `IlvWindowsPrinter`*, the printer driver is responsible for the paper sizes, so this class is used only with* `IlvPSPrinter`*.*

Retrieving a paper format:

```
IlvPaperFormat* letterformat = IlvPaperFormat::Get("Letter");
```

Creating a new paper format:

```
IlvPrintCMUnit width(100.0);
IlvPrintCMUnit height(100.0);
IlvPaperFormat::Register("MyFormat", width.getPoints(), height.getPoints());
```

Predefined paper formats are shown in Table 13.1:

Table 13.1 *Predefined Paper Formats*

Name	Width (in points)	Height (in points)
`A0`	2380	3368
`A1`	1684	2380
`A2`	1190	1684
`A3`	842	1190
`A4`	595	842
`A5`	421	595

Table 13.1 *Predefined Paper Formats (Continued)*

Name	Width (in points)	Height (in points)
A6	297	421
B4	709	1003
B5	516	729
C5	459	649
Quarto	610	780
Folio	612	936
Statement	396	612
Monarch	279	540
Executive	540	720
Ledger	1224	792
Tabloid	792	1224
Legal	612	1008
Letter	612	792

Dialogs

The Gadgets package comes with predefined dialogs for previewing the print job or selecting the PostScript printer capabilities.

The `IlvPostScriptPrinterDialog` class allows you to select various PostScript printer capabilities such as:

- Output Filename
- Orientation
- Color Mode
- Paper Format
- Collate Mode
- Number of copies
- Margins

Usage example: (see Figure 13.1).

```
IlvPostScriptPrinterDialog psdialog(display);
   psdialog.get();
   IlvPrinter::Orientation orientation = psdialog.getOrientation();
   IlBoolean collate = psdialog.isCollateOn();
```

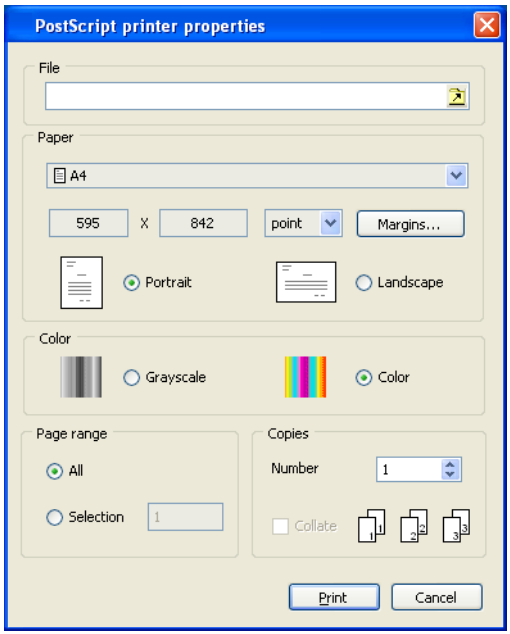


Figure 13.1 *PS Printer Example*

The `IlvPrinterPreviewDialog` class allows you to preview a printing job (see Figure 13.2). It supports various modes such as:

- One page preview
- Two page preview
- Tiled preview

A custom zooming factor can be specified.

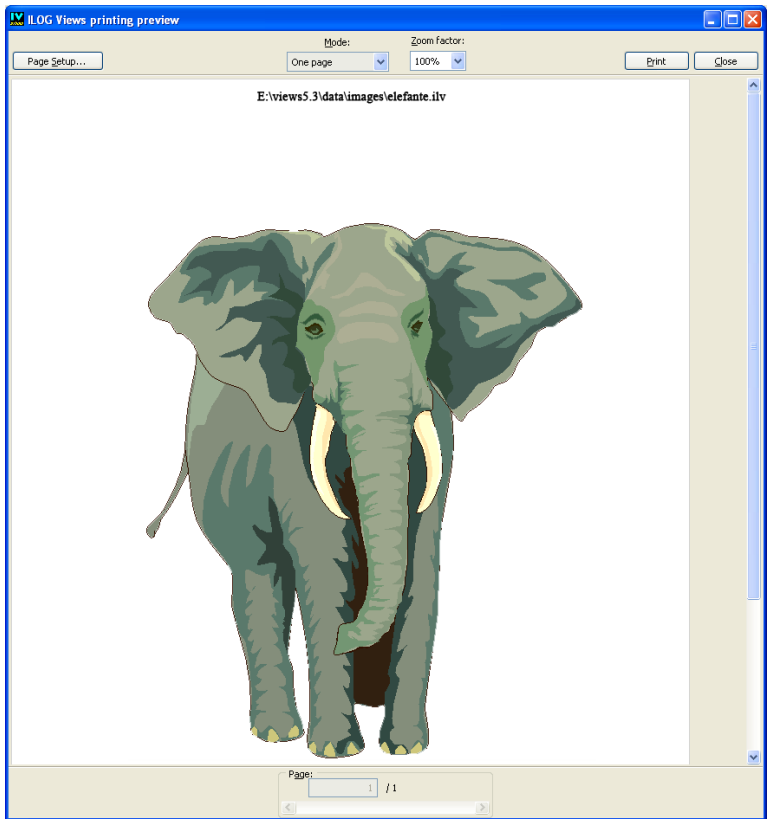


Figure 13.2 *Print Preview Example*

C H A P T E R

14

IBM ILOG Script Programming

This chapter is a programming guide to IBM ILOG Script for IBM® ILOG® Views. It covers the following topics:

- An introduction to *IBM ILOG Script for IBM ILOG Views*
- *Making IBM ILOG Views Applications Scriptable*
- *Binding IBM ILOG Views Objects*
- *Loading IBM ILOG Script Modules*
- *Using IBM ILOG Script Callbacks*
- *Handling Panel Events*
- *Creating IBM ILOG Views Objects at Run Time*
- *Common Properties of IBM ILOG Views Objects*
- *Using Resources in IBM ILOG Script for IBM ILOG Views*
- The topic concludes with *Guidelines for Developing Scriptable Applications* and reference tables providing *Resource Names* that you can use in your scripting.

Details on the syntax of IBM ILOG Script are in Appendix F, *IBM ILOG Script 2.0 Language Reference*.

IBM ILOG Script for IBM ILOG Views

IBM ILOG Script for IBM® ILOG® Views is an object-oriented scripting language for developing high-performance graphic applications.

IBM ILOG Script for IBM ILOG Views is an enhanced version of IBM ILOG Script, an IBM ILOG implementation of the JavaScript™ scripting language that lets you access most of the IBM ILOG Views powerful graphic objects.

For full details of the IBM ILOG scripting features, refer to the following documentation:

- This chapter explains how to program IBM ILOG Views graphic objects with IBM ILOG Script.

  > ***Note:*** *It includes material regarding programming of optional panels and gadgets. For complete information on these options, refer to the appropriate IBM ILOG Views documentation packages.*

- The IBM ILOG Script *Manual* shows you how to use IBM ILOG Script for IBM ILOG Views from `ivfstudio`, which includes the extension of IBM ILOG Views Studio that lets you write IBM ILOG Views applications in IBM ILOG Script.
- The *ILOG Views Foundation Reference Manual* provides all the information you need concerning the IBM ILOG Views objects supported by IBM ILOG Script for IBM ILOG Views.

Making IBM ILOG Views Applications Scriptable

To use IBM ILOG Script in an IBM ILOG Views application, you have to make this application *scriptable*. The interpreter of IBM ILOG Script for IBM ILOG Views is implemented as C++ libraries. Therefore, if you want to use an IBM ILOG Views application with IBM ILOG Script for IBM ILOG Views, you have to:

- Include the appropriate header file in the source files of your application, as described in *Including the Header File*.
- Link your application using the supplied IBM ILOG Script for IBM ILOG Views libraries, as described in *Linking with IBM ILOG Script for IBM ILOG Views Libraries*.

> ***Note:*** *You can also generate IBM ILOG Views scriptable applications from the extension of IBM ILOG Views Studio that lets you write IBM ILOG Views applications in IBM ILOG Script. For more information, refer to the IBM ILOG Views Studio User's Manual.*

Including the Header File

Include the following header file in the main source file of your application:

```
#include <ilviews/jvscript/script.h>
```

You need to include this file only once. You do not have to include it in each one of the source files of your application.

Linking with IBM ILOG Script for IBM ILOG Views Libraries

In addition to the IBM ILOG Views libraries, you must link your application with the following IBM ILOG Script for IBM ILOG Views libraries:

On Microsoft Windows

- `ilvjs.lib`
- `iljs.lib`
- `iljsgide.lib`

On UNIX

- `libilvjs`
- `libiljs`
- `libiljsgide`

Binding IBM ILOG Views Objects

To use IBM® ILOG® Views objects, such as gadgets and panels, in IBM ILOG Script, you must make these objects accessible using a binding procedure. To bind an object:

- First get an IBM ILOG Script context. This is described in *Getting the Global IBM ILOG Script Context*.
- Invoke the `bind` method, discussed in *Binding IBM ILOG Views Objects*.

A bound object becomes accessible from IBM ILOG Script.

Getting the Global IBM ILOG Script Context

An IBM ILOG Script context is a gateway between IBM® ILOG® Views and the scripting language that should be created before binding any IBM ILOG Views objects. If, as explained in the previous section, you have included the `script.h` header file in the files of your application and linked that application with the appropriate libraries, the global context will be created automatically.

To activate this context, invoke the following function:

```
IlvScriptLanguage* jvscript = IlvScriptLanguage::Get("JvScript");
IlvScriptContext* theContext = jvscript->getGlobalContext();
```

This function returns a pointer to the global IBM ILOG Script context.

Binding IBM ILOG Views Objects

To bind an IBM® ILOG® Views object, call the following function:

```
IlvScriptContext::bind(IlvValueInterface* object,
                       const char* name);
```

This function takes a pointer to the object to be bound as its first parameter and the character string to which the object is bound as its second parameter. IBM ILOG Script programmers can use this name to access the associated object. The pointer is of the type `IlvValueInterface`, which is a superclass for most of the IBM ILOG Views classes.

Thus, you can bind an `IlvApplication` object with the following code:

```
IlvScriptLanguage* jvscript = IlvScriptLanguage::Get("JvScript);
IlvScriptContext* theContext = jvscript->getGlobalContext();
theContext->bind(theApp, "Application");
// theApp is the pointer to an IlvApplication
```

The above code binds the `IlvApplication` object to IBM ILOG Script through the `Application` symbol. Consequently, you can access the properties attached to `Application` from IBM ILOG Script:

```
var name = Application.name;
```

Accessing IBM ILOG Views Objects in IBM ILOG Script

You might want to bind all the IBM ILOG Views objects in your application. To do so, the best solution is to bind only the root object. This is because you can access, either directly or indirectly, almost any other IBM ILOG Views object starting from that object.

In an IBM ILOG Views application generated from `ivfstudio`, for example, you can access the pointers to the panels through the pointer to the application by calling the `IlvApplication::getPanel` method. Similarly, you can then access the gadgets in the panels by invoking the `IlvContainer::getObject` method. This is the reason why, in such applications, the only object that should be bound is an `IlvApplication` object.

The Application Object

In applications generated with `ivfstudio`, the `IlvApplication` object is bound with the `Application` symbol. You can access any other IBM ILOG Views object starting from the `Application` object.

Let us suppose that your application contains one panel named `myPanel`. Here is how you can access it in IBM ILOG Script:

```
var panel = Application.getPanel("myPanel");
```

To change its title, you can type:

```
panel.title = "New title";
```

If the panel in your application contains one button called `myButton`, you can access it using the following code:

```
var button = panel.getObject("myButton");
```

To change the button label, type:

```
button.label = "A new label";
```

Accessing Panels and Gadgets

Below are easier ways to access an application's panels and gadgets.

To access the panel named `myPanel` in IBM ILOG Script, you can type:

```
var panel = Application.myPanel;
```

To change its title:

```
Application.myPanel.title = "A new title";
```

To access gadgets in the panel:

```
var button = panel.myButton;
```

Note that only panels and gadgets that have regular names can be accessed that way. If a panel or gadget name includes special characters, such as `&`, `+`, `-`, `=`, space, and so on, you will not be able to access them using the procedures described above. Be careful not to use these characters in the name of panels and gadgets.

Loading IBM ILOG Script Modules

Three different types of scripts can be loaded into a scriptable IBM® ILOG® Views application:

- *Inline Scripts*
- *Default IBM ILOG Script Files*
- *Independent IBM ILOG Script Files*

Static functions that may be defined in these scripts have a limited function, as discussed in *IBM ILOG Script Static Functions*.

Inline Scripts

Scripts that you create from `ivfstudio` while designing graphic panels are called inline scripts. These scripts are saved in `.ilv` files by `ivfstudio`. Inline scripts are loaded together with the `.ilv` files in which they are stored when these files are loaded into an IBM ILOG Views scriptable application. See *Making IBM ILOG Views Applications Scriptable*.

Default IBM ILOG Script Files

When an `.ilv` file, such as `panel.ilv`, that does not contain inline scripts is being loaded into an IBM ILOG Views scriptable application, IBM ILOG Views looks into the directory where the file is located for an IBM ILOG Script file with the same name that has the `.js` extension and automatically loads it. The `panel.js` file is called the default IBM ILOG Script file of `panel.ilv`.

Independent IBM ILOG Script Files

You can load independent IBM ILOG Script modules into an IBM ILOG Views scriptable application using the `IlvScriptContext::loadScript` method:

```
IlvScriptContext::loadScript("c:\\myscripts\\myscript.js");
```

Using this method, you can load IBM ILOG Script files that are shared by several applications.

IBM ILOG Script Static Functions

Inline and default scripts are associated with an `.ilv` file. The names of static functions defined in such scripts have a limited scope:

- The name scope of an IBM ILOG Script static function is limited to the module where it is defined. It is not visible from other IBM ILOG Script modules.
- It can only be used as a callback for the gadgets in the associated `.ilv` file and for the gadgets in the subpanels of the gadget container defined by the associated `.ilv` file, if any.

Here is an example of a static function:

```
static function OnClick(graphic)
{
    graphic.foreground = "red";
}
```

Using IBM ILOG Script Callbacks

IBM® ILOG® Views gadgets are able to recognize specific mouse or keyboard events that apply to them and invoke the associated predefined IBM ILOG Script callback functions.

To handle gadget events using callbacks, you must:

- Write the callback functions: see *Writing a Callback*
- Set the callback: see *Setting an IBM ILOG Script Callback*

Writing a Callback

In IBM ILOG Script for IBM ILOG Views, callbacks have the following signature:

```
function CallBack(gadget, value)
```

Here is an example of a callback:

```
function OnButtonClick(button, value)
{
    button.foreground = "red";
    writeln("The additional value is: " + value);
}
```

The second argument passed to the callback is an optional value that you can specify when you set a callback in `ivfstudio`. The contents of the callback function can therefore be the following:

```
function OnGadgetClick(gadget)
{
    gadget. foreground = "red";
}
```

Setting an IBM ILOG Script Callback

There are two ways to set a callback to a gadget:

- You can set callbacks in IBM ILOG Views Studio, while designing panels. This is the easiest way.
- You can set callbacks with the `IlvGraphic::setCallback` method. Using this method, you can set a callback to a gadget or modify it at run time. Here is an example:

```
myGadget.setCallback("Generic",  "myCallback", "JvScript");
```

The first argument is the callback type that identifies the event to handle. The second argument is the callback function you defined. The third argument is always `JvScript`.

Handling Panel Events

You can use IBM ILOG Script functions to be informed whenever the panels of your application are created, displayed, hidden, or deleted. To handle these events you use:

- *The OnLoad Function*
- *The onShow Property*
- *The onHide Property*
- *The onClose Property*

The OnLoad Function

When an `IlvContainer` object is created, it looks for an IBM ILOG Script function `OnLoad` and invokes it, passing the container as its argument. If there are several `OnLoad` functions, the container will look in the following modules in order and call the first `OnLoad` function encountered:

1. The inline script module
2. The default IBM ILOG Script module
3. Other IBM ILOG Script modules

The IBM ILOG Script function `OnLoad` should have the following signature:

```
function OnLoad(theContainer)
{
     // Initialization code
}
```

The `OnLoad` function is generally used to perform initialization once the panels in the application have been created.

The onShow Property

`IlvContainer` has an `onShow` property to which you can pass an IBM ILOG Script function. The specified function is called when the container is displayed on the screen. For example:

```
function OnShow(theContainer)
{
    writeln("Hi, " + theContainer.name + " is displayed.");
}

function OnLoad(theContainer)
{
   theContainer.onShow = OnShow;
}
```

In this example, the `OnShow` function is used to handle the `onShow` event applied to `theContainer`.

The onHide Property

The `onHide` property is similar to the `onShow` property except that the specified function is called when the container is hidden. For example:

```
function WhenPanelHides(theContainer)
{
    writeln("Hi, I am " + theContainer.name + ", see you later.");
}

function OnLoad(theContainer)
{
   theContainer.onHide = WhenPanelHides;
}
```

The onClose Property

The `onClose` property is similar to the `onShow` property except that the specified function is called when the container is being closed. For example:

```
function OnClose(theContainer)
{
    writeln("Hi, " + theContainer.name + " has terrible news ...");
}

function OnLoad(theContainer)
{
   theContainer.onClose = OnClose;
}
```

Creating IBM ILOG Views Objects at Run Time

In IBM ILOG Script for IBM® ILOG® Views, you can use the `new` operator to create IBM ILOG Views objects at run time just as you would create other IBM ILOG Script native objects, such as strings, numbers, and so on.

You can create the following types of objects at run time:

- `IlvPoint`
- `IlvRect`
- `IlvGadgetContainer`

To create a gadget, such as an `IlvButton`, we recommend that you use IBM ILOG Views Studio.

IlvPoint and IlvRect

Certain IBM ILOG Views object methods take an `IlvPoint` or an `IlvRect` as their arguments. These can be created at run time as shown below:

```
var myPoint = new IlvPoint(20, 20);
myPanel.move(myPoint);
```

IlvGadgetContainer

You can create new panels at run time. Here is an example:

```
var size = new IlvRect(20, 20, 400, 300);
var myNewPanel = new IlvGadgetContainer("Panel", "My panel", size);
myNewPanel.readFile("panel.ilv");
myNewPanel.readDraw();
Application.addPanel(myNewPanel);
```

We recommend that you add the new panel to `Application` after it has been created.

For more information, see "Accessors for class `IlvGadgetContainer`" in the *Gadgets Accessor Reference Manual*.

Common Properties of IBM ILOG Views Objects

The following properties useful in scripting are common to all IBM® ILOG® Views objects:

- *className*
- *name*
- *help*

className

`className` is a read-only string that indicates the type of the object. Object types are documented in the *ILOG Views Foundation Reference Manual*.

name

`name` is a string that identifies the object. A panel should have a unique name within the application. A gadget should have a unique name within its container.

help

`help` is a read-only string that gives the object description. This property is very useful when debugging a scriptable IBM ILOG Views application.

For example, in the IBM ILOG Views Studio script debugger, type `Application.help` to get a list of the supported properties and methods:

```
> Application.help
= ViewsObject :
Method getPanel;
Method addPanel;
Method removePanel;
Method setState;
Method quit;
Object rootState;
String name;
String className;
```

To get more information on the method `IlvApplication::getPanel`, just type `Application.getPanel.help`:

```
> Application.getPanel.help
= Object getPanel(String name)
```

You can also obtain information on the `IlvApplication::getPanel` method by typing `Application.getPanel`:

```
> Application.getPanel
= [Views method: Object getPanel(String name)]
```

Using Resources in IBM ILOG Script for IBM ILOG Views

Resources in IBM ILOG Script for IBM® ILOG® Views, such as colors, bitmaps, and fonts, are identified by a name or a string. The following sections show you how to use them:

- *Using Resource Names with IBM ILOG Script for IBM ILOG Views*
- *Using Bitmaps with IBM ILOG Script for IBM ILOG Views*
- *Using Fonts with IBM ILOG Script for IBM ILOG Views*

Using Resource Names with IBM ILOG Script for IBM ILOG Views

To modify the resource associated with a given IBM ILOG Views object (whether a color, a pattern, a line or fill style, an arc mode, and so on) use its name. Here are a few examples:

```
myButton.foreground = "red";
myButton.pattern = "solid";
myLabel.alignment = "right";
```

Resource names are listed in *Resource Names*.

Using Bitmaps with IBM ILOG Script for IBM ILOG Views

Bitmaps are identified by their names. To modify an IBM® ILOG® Views bitmap, use its name as shown in the example below:

```
myButton.bitmap = "ilog.ic";
myPanel.backgroundBitmap = "subdir/mybmp.gif";
```

The specified bitmaps must be stored in the directories defined in `ILVPATH`. If the bitmaps are located in another directory, indicate the complete access path:

```
myButton.bitmap = "/mybmps/myicon.gif";
```

Using Fonts with IBM ILOG Script for IBM ILOG Views

In IBM ILOG Script for IBM® ILOG® Views, fonts are usually identified by character strings with the following format:

```
%fontName-fontSize-fontFlags
```

`fontName` is the name of the font family, such as Courier, Helvetica or Times. `fontSize` is an integer that indicates the font size. `fontFlags` is a series of characters that indicates the font style: B for Bold, I for Italic and U for Underlined. Leave this field empty if you want the font to appear plain.

For example, to change the font of an `IlvLabel`, type the following:

```
myLabel.font = "%times-16-I";
```

Guidelines for Developing Scriptable Applications

To create a new scriptable IBM® ILOG® Views application or make an existing application scriptable, follow these guidelines:

1. Use an object of the class `IlvApplication`, or of a derived class, as your application's root object. Once your `IlvApplication` object is created, bind it using the name `Application`.
2. Add all the panels to the `IlvApplication` object so that they can be accessed from the `Application` object in IBM ILOG Script for IBM ILOG Views.
3. After creating `IlvDisplay`, initialize the IBM ILOG Script for IBM ILOG Views auxiliary library using the following code:

```
IlvJvScriptLanguage::InitAuxiliaryLib(appli->getDisplay());
```

Initialization is required when you want to use the `IlvCommonDialog` object or create `IlvPoint`, `IlvRect`, or `IlvGadgetContainer` objects in IBM ILOG Script for IBM ILOG Views.

Resource Names

In this section you will find tables listing the names of resources in IBM ILOG Script for IBM® ILOG® Views.

Table 14.1 *Color Names*

Color Name	RGB Definition
aliceblue	240, 248, 255
antiquewhite	250, 235, 215
aquamarine	127, 255, 212
azure	240, 255, 255
beige	245, 245, 220
bisque	255, 228, 196
black	0, 0, 0
blanchedalmond	255, 235, 205
blue	0, 0, 255
blueviolet	138, 43, 226
brown1	65, 42, 42
burlywood	222, 184, 135
cadetblue	95, 158, 160
chartreuse	127, 255, 0
chocolate	210, 105, 30
coral	255, 127, 80
cornflowerblue	100, 149, 237
cornsilk	255, 248, 220
cyan	0, 255, 255
darkgoldenrod	184, 134, 11
darkgreen	0, 100, 0
darkkhaki	189, 183, 107
darkolivegreen	85, 107, 47
darkorange	255, 140, 0
darkorchid	153, 50, 204
darksalmon	233, 150, 122
darkseagreen	143, 188, 143
darkslateblue	72, 61, 139
darkslategray	47, 79, 79
darkslategrey	47, 79, 79
darkturquoise	0, 206, 209
darkviolet	148, 0, 211
deeppink	255, 20, 147
deepskyblue	0, 191, 255
dimgray	105, 105, 105
dimgrey	105, 105, 105
dodgerblue	30, 144, 255
firebrick	178, 34, 34
floralwhite	255, 250, 240
forestgreen	34, 139, 34
gainsboro	220, 220, 220

```
ghostwhite        248, 248, 255
gold              255, 215, 0
goldenrod         218, 165,  32
gray              192, 192, 192
green             0, 255, 0
greenyellow       173, 255,  47
grey              192, 192, 192
honeydew          240, 255, 240
hotpink           255, 105, 180
indianred         205,  92,  92
ivory             255, 255, 240
khaki             240, 230, 140
lavender          230, 230, 250
lavenderblush     255, 240, 245
lawngreen         124, 252, 0
lemonchiffon      255, 250, 205
lightblue         173, 216, 230
lightcoral        240, 128, 128
lightcyan         224, 255, 255
lightgoldenrod    238, 221, 130
lightgoldenrod    250, 250, 210
lightgray         211, 211, 211
lightgrey         211, 211, 211
lightpink         255, 182, 193
lightsalmon       255, 160, 122
lightseagreen     32, 178, 170
lightskyblue      135, 206, 250
lightslateblue    132, 112, 255
lightslategray    119, 136, 153
lightslategrey    119, 136, 153
lightsteelblue    176, 196, 222
lightyellow       255, 255, 224
limegreen         50, 205,  50
linen             250, 240, 230
magenta           255,  0, 255
maroon            176,  48,  96
mediumaquamarine  102, 205, 170
mediumblue        0, 0, 205
mediumorchid      186, 85, 211
mediumpurple      147, 112, 219
mediumseagreen    60, 179, 113
mediumslateblue   123, 104, 238
mediumspringgreen 0, 250, 154
mediumturquoise   72, 209, 204
mediumvioletred   199,  21, 133
midnightblue      25, 25, 112
mintcream         245, 255, 250
mistyrose         255, 228, 225
moccasin          255, 228, 181
navajowhite       255, 222, 173
navy              0, 0, 128
navyblue          0, 0, 128
oldlace           253, 245, 230
olivedrab         107, 142,  35
orange            255, 165, 0
orangered         255,  69, 0
orchid            218, 112, 214
palegoldenrod     238, 232, 170
```

palegreen	152, 251, 152
paleturquoise	175, 238, 238
paleviolet	219, 112, 147
papayawhip	255, 239, 213
peachpuff	255, 218, 185
peru	205, 133, 63
pink	255, 192, 203
plum	221, 160, 221
powderblue	176, 224, 230
purple	160, 32, 240
red	255, 0, 0
rosybrown	188, 143, 143
royalblue	65, 105, 225
saddlebrown	139, 69, 19
salmon	250, 128, 114
sandybrown	244, 164, 96
seagreen	46, 139, 87
seashell	255, 245, 238
sienna	160, 82, 45
skyblue	135, 206, 235
slateblue	106, 90, 205
slategray	112, 128, 144
slategrey	112, 128, 144
snow	255, 250, 250
springgreen	0, 255, 127
steelblue	70, 130, 180
tan	210, 180, 140
thistle	216, 191, 216
tomato	255, 99, 71
turquoise	64, 224, 208
violet	238, 130, 238
violetred	208, 32, 144
wheat	245, 222, 179
white	255, 255, 255
whitesmoke	245, 245, 245
yellow	255, 255, 0
yellowgreen	154, 205, 50

Table 14.2 *Directions*

left
right
top
bottom
topLeft
bottomLeft
topRight
bottomRight
center
horizontal
vertical

Table 14.3 *Arc Modes*

ArcPie
ArcChord

Table 14.4 *Fill Rules*

```
EvenOddRule
WindingRule
```

Table 14.5 *Fill Styles*

```
FillPattern
FillColorPattern
FillMaskPattern
```

Table 14.6 *Patterns*

```
solid
clear
diaglr
diagrl
dark1
dark2
dark3
dark4
light1
light2
light3
light4
gray
horiz
vert
cross
```

Table 14.7 *Line Styles*

```
solid
dot
dash
dashdot
dashdoubledot
alternate
doubledot
longdash
```

C H A P T E R

15

Internationalization

IBM® ILOG® Views allows users to develop international versions of software. You will find information on:

- *What is i18n?* is a brief introduction to internationalization.
- *Checklist for Localized Environments* lists the requirements for success in creating and running your programs. Further topics detail requirements for locales, fonts, and localized message database files. A *Troubleshooting* checklist gives some problem-solving techniques if localized messages do not appear on your system during the development phase.
- *Using IBM ILOG Views with Far Eastern Languages* describes special considerations for multibyte character languages.
- *Data Input Requirements*
- *Limitations of Internationalization Features*
- There is a *Reference: Encoding Listings* of the encodings supported by IBM ILOG Views and extensive tables in *Reference: Supported Locales on Different Platforms*

> ***Note:*** *This chapter explains how to use IBM ILOG Views internationalization features. To learn how to write internationalized software, you should refer to general books on the subject.*

What is i18n?

Internationalization (or the common abbreviation “i18n”) is a software design methodology that lets users interact with a software application using their native language. Internationalized software handles data so that the rules of the users’ language are respected. Users expect their software to meet the following requirements:

- Allow input, processing, and display of characters in the language they use.
- Allow them to interact with the system using their own language. Prompts and error messages must be displayed in this language.
- Format and process data according to the user’s local rules and environment.

Locale

Support of i18n by IBM ILOG Views is based on the POSIX locale model. A *locale* is a collection of data and/or methods that allow internationalized C library and system-dependent library functions to comply with the users’ language, local customs, and data encoding. The locale determines the characters and fonts used to display the language. It also determines how programs display and sort dates, times, currency, and numbers.

Checklist for Localized Environments

Before you begin to use your program in the local language, there are certain things you must do to ensure that your program will run in the desired language.

- You must create the program to run in the localized environment. See *Creating a Program to Run in a Localized Environment*.
- Your system must support the locale (the language you want to use). See *Locale Requirements*.
- IBM® ILOG® Views must support the language you want to use. See *IBM ILOG Views Locale Support*.
- The fonts needed to display the language must be installed on your system. See *Required Fonts*.
- The files containing messages and other system text (the `.dbm` files) must be translated into the local language and available in the proper subdirectories. See *Localized Message Database Files in IBM ILOG Views*.

When all of these requirements are in place, you can then run your localized software.

Creating a Program to Run in a Localized Environment

When creating a program that you intend to use in an internationalized environment, you code as you normally would for any other program. You must make sure, however, that you call the `IlvSetLocale` global function at the beginning of your program. This call should appear before creating an instance of `IlvDisplay`. The`IlvSetLocale` call is necessary for IBM® ILOG® Views to set up the underlying information it needs to run correctly in the default locale environment.

> ***Note:*** *If you do not have a call to* `IlvSetLocale` *in your program, the localized messages will not appear on the screen and multibyte support will not be enabled. Your program will behave as if you are running in the C locale, thus displaying only English messages.*

The following example shows a simple program that is ready for an internationalized environment, meaning that the program can run in different languages. Notice the `IlvSetLocale()` call at the beginning of the program.

```
// --------------------------------------------------------------- -*- C++ -*-
//                                                  IlogViews userman source file
// File: doc/fondation/userman/src/internationalization/setLocale.cpp
// --------------------------------------------------------------------------
//  Copyright (C) 1990-2000 by ILOG.
//  All Rights Reserved.
// --------------------------------------------------------------------------

#include <ilviews/gadgets/gadcont.h>
#include <ilviews/gadgets/textfd.h>
#include <stdio.h>

static void
Quit(IlvView*, IlAny)
{
    IlvExit(0);
}

int main (int argc, char* argv[])
{
    if (!IlvSetLocale()) {
printf("Falling back to the C locale.\n");
    }

    IlvDisplay* display = new IlvDisplay("Test", 0, argc, argv);
    IlvRect rect(20,20,250,80);
    IlvGadgetContainer* cont = new IlvGadgetContainer(display, "Container",
"Container", rect);
    cont->setDestroyCallback(Quit, 0);
    IlvRect rect1(10,10,220,50);

    IlvTextField* tf = new IlvTextField(display, "This is a text field.",
rect1);
    cont->addObject(tf);
    IlvMainLoop();

    return 0;
}
```

Locale Requirements

The locale is the language you want your system to support. See the following topics for locale requirements:

- *Checking Your System's Locale Requirements*
- *Locale Name Format*
- *Current Default Locale*
- *Changing the Current Default Locale*

Checking Your System's Locale Requirements

To determine if your system meets the locale requirements:

- Ask your system administrator whether or not your operating system supports the locale you need. If your operating system does not support the locale, you cannot run your localized program.
- Depending on the system you are using, you can also do the following to find out if your system supports the locale:
  - *On UNIX Systems*
  - *X Library Support (UNIX only)*
  - *On Microsoft Windows Systems*

> ***Note:*** *Locale names are system-dependent. For each example of a system-dependent name, we will mention only the French and Japanese settings for the HP-UX (10.x or 11), Solaris (2.6 or 2.7), and Windows platforms.*

On UNIX Systems

Run the following utility program to get a list of the locales supported by your system:

```
$ locale -a
```

Here is an example of what you will get on an HP-UX system if only French (as spoken in France) and Japanese are supported:

```
fr_FR.iso88591
fr_FR.iso885915@euro
fr_FR.roman8
fr_FR.utf8
ja_JP.SJIS
ja_JP.eucJP
ja_JP.kana8
ja_JP.utf8
```

Here is an example of what you will get on a Solaris system, if only French and Japanese are supported:

```
fr
fr.ISO8859-15
fr.UTF-8
fr_FR
fr_FR.ISO8859-1
fr_FR.ISO8859-15
fr.ISO8859-15@euro
fr_FR.UTF-8
```

```
fr_FR.UTF-8@euro
ja
ja_JP.eucJP
ja_JP.PCK
ja_JP.UTF-8
japanese
```

On Microsoft Windows Systems

Look at the Regional Settings in the Control Panel:

1. On the Windows desktop click Start - Settings - Control Panel.
2. Double-click the Regional Settings icon to access the Regional Settings Properties dialog box.
3. In the Regional Settings notebook page, you will see a list of supported locales.

Locale Name Format

As you can see, locale names are system-dependent. Most systems, however, tend to follow the XPG (X/Open Portability Guide) naming convention, where a locale name has the following format:

`language_territory.encoding`
`language` is the language name; `territory` is the territory name (a language can be spoken in different areas or countries: French, for example, is spoken in France, Canada, Belgium, Switzerland, and other countries); and `encoding` is a code set or an encoding method by which characters are coded.

On UNIX Systems

The following examples show the format of a locale name as displayed on different UNIX systems. The locale is for French as spoken in France with the Latin1 encoding.

Solaris 8	`fr` or `fr_FR.iso8859-1`
HP-UX 11	`fr_FR.iso88591`
Red Hat Entreprise Linux 4.0	`fr_FR.iso88591`
Suze 10.0	`fr_FR`
AIX 5.1	`fr_FR` or `fr_FR.ISO8859-1`

On Microsoft Windows Systems

The following example shows the format of a locale on a Windows system. The locale is for French as spoken in France with Windows Code Page 1252.

Windows XP	`French_France.1252`

Current Default Locale

Your system will have a default locale. Normally, your default locale should be set to the language you want to use. To find out the current default locale on your system, you can run one of the following programs:

On UNIX Systems

```
// ------------------------------------------------------------- -*- C++ -*-
//                                                 IlogViews userman source file
// File: doc/fondation/userman/src/internationalization/checkUnixLocale.cpp
// ------------------------------------------------------------------------
//  Copyright (C) 1990-2008 by ILOG.
//  All Rights Reserved.
// ------------------------------------------------------------------------

#include <locale.h>
#include <stdio.h>
#include <langinfo.h>
#if defined(linux) && !defined(CODESET)
#define CODESET _NL_CTYPE_CODESET_NAME
#endif /* linux */

int main()
{
    char* loc = setlocale(LC_ALL, "");
    if (loc) {
        printf("default locale: %s\n",loc);
        printf("encoding %s\n", nl_langinfo(CODESET));
    } else
        printf("System does not support this locale\n");
    return 0;
}
```

If your system is set to the French language, here is an example of what you will get on HP-UX:

```
default locale: fr_FR.iso88591 fr_FR.iso88591 fr_FR.iso88591
fr_FR.iso88591 fr_FR.iso88591 fr_FR.iso88591
```

On Solaris, you will get:

```
default locale: fr
```

On Microsoft Windows Systems

```
// ------------------------------------------------------------- -*- C++ -*-
//                                                 IlogViews userman source file
// File: doc/foundation/userman/src/internationalization/checkWindowsLocale.cpp
// ------------------------------------------------------------------------
//  Copyright (C) 1990-2008 by ILOG.
//  All Rights Reserved.
// ------------------------------------------------------------------------

#include <locale.h>
#include <stdio.h>
#include <windows.h>

int main(int argc, char* argv[])
{
    printf("default locale: %s\n", setlocale(LC_ALL, ""));
    printf("encoding %d\n", GetACP());
    return 0;
}
```

On Windows XP, if your regional setting has been set to French (Standard), you will get:

```
default locale: French_France.1252
encoding 1252
```

Changing the Current Default Locale

For your localized messages to appear on the screen, you may need to change the current default locale.

On UNIX Systems

You can use one of the following environment variables: `LANG` or `LC_ALL`. See your system documentation to find out the appropriate locale names.

For example, if you want to use Japanese with the EUC encoding:

On HP-UX, type:

```
$ LANG=ja_JP.eucJP
```

On Solaris, type:

```
$ LANG=ja or LANG=japanese
```

On Microsoft Windows Systems

Change the language using the Regional Settings of the Control Panel.

X Library Support (UNIX only)

Your X Window system needs to support the desired language. You can run the following program to find out if the appropriate X libraries are available on your system:

```
// --------------------------------------------------------------- -*- C++ -*-
//                                                  IlogViews userman source file
// File: doc/fondation/userman/src/internationalization/checkXLocale.cpp
// ------------------------------------------------------------------------------
//  Copyright (C) 1990-2008 by ILOG.
//  All Rights Reserved.
// ------------------------------------------------------------------------------

#include <X11/Xlib.h>
#include <X11/Xlocale.h>
#include <stdlib.h>
#include <stdio.h>

int
main(int argc, char* argv[])
{
    char* loc = setlocale(LC_CTYPE, "");
    if (loc == NULL) {
        fprintf(stderr, "System does not support this locale.\n");
        exit(1);
    }
    if (!XSupportsLocale()) {
        fprintf(stderr, "X does not support locale %s.\n", loc);
        exit(1);
    }
    if (XSetLocaleModifiers("") == NULL) {
        fprintf(stderr, "Warning: cannot set locale modifiers for %s.\n", loc);
    } else
        fprintf(stderr, "Locale %s is supported by Xlib.\n", loc);
    exit(0);
}
```

For example, on an HP system where Arabic is not supported, setting LANG to ar_DZ.arabic8, gives the following output:

```
X does not support locale ar_DZ.arabic8.
```

IBM ILOG Views Locale Support

Although locale names are system-dependent, and each system has its own way of identifying the locale information, IBM® ILOG® Views supports a system-independent scheme for localization.

IBM ILOG Views Locale Names

For IBM ILOG Views to use locale-dependent information in a system-independent way, IBM ILOG Views defines the concept of an IBM ILOG Views *locale*, whose name is system-independent. This locale has the following format:

```
ll_TT.encoding
```

where:

`ll` is a two-letter, lowercase abbreviation of the language name.

`TT` is a two-letter, uppercase abbreviation of the territory name.

`encoding` is a string that identifies the code set or encoding method used.

For example, in the IBM ILOG Views locale name `fr_FR.ISO-8859-1`, the `fr` represents the language name, French; the `FR` represents the territory name, France; `ISO-8859-1` represents the encoding method used for the language, which is ISO 8859-1.

The following examples show several IBM ILOG Views locale names on UNIX platforms:

- `fr_FR.ISO-8859-1`
- `de_DE.ISO-8859-1`
- `ja_JP.EUC-JP`
- `ja_JP.Shift_JIS`

The following examples show several IBM ILOG Views locale names on Windows platforms:

- `fr_FR.windows-1252`
- `de_DE.windows-1252`
- `ja_JP.Shift_JIS`

Language Name Specification

In the IBM ILOG Views locale, the language names are specified using the abbreviations from the ISO 639 Code for the Representation of Names of Languages. Here are several examples:

- `en` (English)
- `fr` (French)
- `de` (German, from “Deutsch”)
- `ja` (Japanese)

The ISO 639 standard can be consulted on the following Web sites:

```
http://www.loc.gov/standards/iso639-2/ascii_8bits.html
```

or `ftp://std.dkuug.dk/i18n/iso-639-2.txt`

More generally, ISO codes can be consulted on the following Web site:

`http://userpage.chemie.fu-berlin.de/diverse/doc/ISO_639.html`

Territory Name Specification

In the IBM ILOG Views locale, the territory names are specified using the abbreviations from the ISO 3166 Codes for the Representation of Names of Countries.

The ISO 3166 standard can be consulted on the following Web site:

`http://www.iso.org/iso/country_codes/iso_3166_code_lists/english_country_names_and_code_elements.htm`

Here are several examples:

- `US` (United States)
- `NL` (the Netherlands)
- `FR` (France)
- `DE` (Germany, from “Deutschland”)
- `JP` (Japan)

Encoding Specification

In the IBM ILOG Views locale, the encoding identifies the code set or encoding method used for the language. Examples of encoding methods are:

- `ISO-8859-1` (ISO 8859/1)
- `Shift_JIS` (Shift Japanese Industrial Standard)

Any character encoding registered by IANA could be used. Currently only character sets listed in *Reference: Encoding Listings* are supported by IBM ILOG Views, which tends to use the preferred MIME notation.

For more information, you can consult the following Web site:

`http://www.iana.org/assignments/character-sets`

Determining IBM ILOG Views Support for the Locale

To determine if IBM ILOG Views supports the desired locale, you can run the following program:

> ***Note:*** *This example uses private code that you should not use in a real application.*

```
// ------------------------------------------------------------ -*- C++ -*-
//                                                 IlogViews userman source file
// File: doc/fondation/userman/src/internationalization/checkViewsLocale.cpp
// --------------------------------------------------------------------------
//  Copyright (C) 1990-2008 by ILOG.
//  All Rights Reserved.
// --------------------------------------------------------------------------

#include <ilviews/ilv.h>
#include <ilviews/base/locale.h>

int main(int argc, char* argv[])
{
    if (!IlvSetLocale()) {
exit(1);
    }

    char* stdLocale = IlLocale::GetStdLocaleName(setlocale(LC_CTYPE, NULL));
    if (stdLocale)
IlvPrint("Standard Views locale name: %s\n", stdLocale);
    else
IlvPrint("Views does not support this locale.\n");

    return 0;
}
```

For example, on an HP-UX system with your `LANG` set to fr_FR.iso88591 or on a Solaris system with your `LANG` set to fr, you get the following result:

```
Standard Views locale name: fr_FR.ISO-8859-1
```

On a Windows system set to Japanese, you get:

```
Standard Views locale name: ja_JP.Shift_JIS
```

Required Fonts

Your system must support the fonts required by your locale.

On UNIX Systems

Make sure that the X resources are set to the fonts used by IBM ILOG Views applications. To do this, edit your `.Xdefaults` file, which is located in your `home` directory. If this file does not exist, you can create it. You should add the following statements to the resource file:

```
IlogViews*font: a-valid-font-set-name-for-your-locale
IlogViews*normalfont: a-valid-font-set-name-for-your-locale
IlogViews*italicfont: a-valid-font-set-name-for-your-locale
IlogViews*boldfont: a-valid-font-set-name-for-your-locale
IlogViews*largefont: a-valid-font-set-name-for-your-locale
```

```
IlogViews*monospacefont: a-valid-font-set-name-for-your-locale
IlogViews*ButtonFont: a-valid-font-set-name-for-your-locale
IlogViews*MenuFont: a-valid-font-set-name-for-your-locale
```

The value of `a-valid-font-set-name-for-your-locale` depends on your language and environment. Fonts are not the same on every system.

If you are running under the Common Desktop Environment (CDE) and have started your desktop in your language, you can use the "-dt" aliases for the fonts, as shown in the following example:

```
IlogViews.font: -dt-interface user-medium-r-normal-m*-*-*-*-*-*-*-*-*
IlogViews.normalfont: -dt-interface user-medium-r-normal-s*-*-*-*-*-*-*-*-*
IlogViews.boldfont: -dt-interface user-bold-r-normal-m*-*-*-*-*-*-*-*-*
IlogViews.italicfont: -dt-interface user-medium-i-normal-m*-*-*-*-*-*-*-*-*
IlogViews.largefont: -dt-interface user-medium-r-normal-xl*-*-*-*-*-*-*-*-*
IlogViews.monospacefont: -dt-interface user-medium-r-normal-m*-*-*-*-*-*-*-*-*
IlogViews.MenuFont: -dt-interface user-bold-r-normal-m*-*-*-*-*-*-*-*-*
IlogViews.ButtonFont: -dt-interface user-bold-r-normal-m*-*-*-*-*-*-*-*-*
```

If you do not use the "-dt-" aliases for the fonts, you need to add your own font statements in the `.Xdefaults file`.

The following are examples of the font statements used on an HP-UX system Japanese:

```
IlogViews.ButtonFont: -hp-gothic-bold-r-normal--14-101-100-100-c-*-*-*,
                        -misc-fixed-bold-r-normal--14-130-75-75-c-70-iso8859-1
IlogViews.MenuFont: -hp-gothic-bold-r-normal--14-101-100-100-c-*-*-*,
                        -misc-fixed-bold-r-normal--14-130-75-75-c-70-iso8859-1
IlogViews.boldfont: -hp-gothic-bold-r-normal--14-101-100-100-c-*-*-*,
                        -misc-fixed-bold-r-normal--14-130-75-75-c-70-iso8859-1
IlogViews.font: -misc-fixed-medium-r-normal--14-*-75-75-c-*-*-*,
                       -misc-fixed-medium-r-normal--15-140-75-75-c-90-iso8859-1
IlogViews.italicfont: -misc-fixed-medium-r-normal--14-*-75-75-c-*-*-*,
                     -adobe-helvetica-bold-o-normal--14-140-75-75-p-82-iso8859-1
IlogViews.largefont: -hp-fixed-medium-r-normal--24-230-75-75-c-*-*-*,
                     -sony-fixed-medium-r-normal--24-170-100-100-c-120-iso8859-1
IlogViews.monospacefont: -misc-fixed-medium-r-normal--14-*-75-75-c-*-*-*,
                       -misc-fixed-medium-r-normal--15-140-75-75-c-90-iso8859-1
IlogViews.normalfont: -misc-fixed-medium-r-normal--14-*-75-75-c-*-*-*,
                        -misc-fixed-medium-r-normal--15-140-75-75-c-90-iso8859-1
```

The following are examples of font statements used on a Solaris system for Japanese:

```
IlogViews.ButtonFont: -sun-gothic-bold-r-normal--14-120-75-75-c-*-*-*,
                            -*-helvetica-bold-r-normal--14-*-*-*-*-*-iso8859-1
IlogViews.MenuFont: -sun-gothic-bold-r-normal--14-120-75-75-c-*-*-*,
                            -*-helvetica-bold-r-normal--14-*-*-*-*-*-iso8859-1
IlogViews.boldfont: -sun-gothic-bold-r-normal--14-120-75-75-c-*-*-*,
                            -*-helvetica-bold-r-normal--14-*-*-*-*-*-iso8859-1
IlogViews.font: -sun-gothic-medium-r-normal--14-*-75-75-c-*-*-*,
                          -*-helvetica-medium-r-normal--14-*-*-*-*-*-iso8859-1
IlogViews.italicfont: -sun-gothic-medium-r-normal--14-*-75-75-c-*-*-*,
                            -*-helvetica-medium-o-normal--14-*-*-*-*-*-iso8859-1
IlogViews.largefont: -sun-gothic-medium-r-normal--22-200-75-75-c-*-*-*,
                            -*-helvetica-medium-r-normal--24-*-*-*-*-*-iso8859-1
IlogViews.monospacefont: -sun-gothic-medium-r-normal--14-*-75-75-c-*-*-*,
```

```
                          -*-helvetica-medium-r-normal--14-*-*-*-*-*-iso8859-1
IlogViews.normalfont: -sun-gothic-medium-r-normal--14-*-75-75-c-*-*-*,
                          -*-helvetica-medium-r-normal--14-*-*-*-*-*-iso8859-1
```

On Microsoft Windows Systems

In many cases, the default font settings will be adequate. If you need to change the fonts used by your applications, edit the `views.ini` file (for all applications) to contain some or all of the following statements:

```
[IlogViews]
font=a-valid-font-for-your-language
normalfont=a-valid-font-for-your-language
italicfont=a-valid-font-for-your-language
boldfont=a-valid-font-for-your-language
largefont=a-valid-font-for-your-language
monospacefont: a-valid-font-for-your-locale
buttonFont=a-valid-font-name-for-your-locale
menuFont=a-valid-font-name-for-your-locale
```

The value `a-valid-font-for-your-language` depends on the language and environment you are using. You can use Microsoft Word or any text editor to find a suitable font name that will display text in your language. The entry in the `.ini` file should have the following form:

```
%<font name>-<font size>-<style>
```

For example, `%helvetica-12-B` for the font Helvetica Bold 12 points. If the font should appear plain, leave out the `style` parameter and type `%helvetica-12-` to display text in Helvetica 12.

The following example shows the fonts for Japanese on a Windows system:

```
font=%ＭＳ明朝-12-
normalfont=%ＭＳ明朝-12-
buttonfont=%ＭＳＰゴシック-12-
boldfont=%ＭＳゴシック-12-B
italicfont=%ＭＳゴシック-12-I
largefont=%ＭＳ明朝-16-
monospacefont=%ＭＳ明朝-12-
menuFont=%ＭＳＰゴシック-12-
toolBarFont=%ＭＳ明朝-12-
```

Localized Message Database Files in IBM ILOG Views

IBM® ILOG® Views uses message database files (`.dbm` files) for the message text, menu item text, and other text that appears in the user interface. These files are described in detail in the following topics:

- *The IlvMessageDatabase Class* describes the class mechanism for localization.

- For localization, the message database files must be translated into the local language as described in *Language of the Message Database Files*.
- The files must also be located in the proper directory so that IBM ILOG Views can find the files when it needs to load the message database. For details see *Location of the Message Database Files*.
- *Determining Parameters of the Message Database Files* provides a short program to find the locale, language and location of the message database files.
- *Loading the Message Database* discusses automatic loading of the default language and gives various methods of overriding the default language to load another language.
- The *.dbm File Format* describes the format of the message database files and how to handle old (pre-3.0) formatted files.
- You can change the language on the fly with `setCurrentLanguage` as described in *How to Dynamically Change Your Display Language*.

The IlvMessageDatabase Class

IBM ILOG Views provides a simple mechanism to help you manipulate multilingual applications. This mechanism is called the messages mechanism and is based on the `IlvMessageDatabase` class.

It uses a database that stores different translations of the same message. Depending on the current language, the appropriate message is accessed. Each instance of the `IlvDisplay` class creates its own message database. It reads the database description from the file name provided in the environment variable `ILVDB`, or `views .dbm` if this variable is not set. This file is searched for in the display path. You can access this database by calling the member function `IlvDisplay::getDatabase`.

Each string that you plan to show in different languages can be stored in the database with its different possible translations; that is, you associate a *message identifier* with different *message strings*, depending on the language you target. The language is specified in a symbol object (`IlSymbol` class). Following is some sample code:

```
IlvMessageDatabase database;
IlSymbol* en_US = IlGetSymbol("en_US");
IlSymbol* fr_FR = IlGetSymbol("fr_FR");
database.putMessage("&cancel", en_US, "Cancel");
database.putMessage("&cancel", fr_FR, "Annuler");
```

The IBM ILOG Views environment variable `ILVLANG` lets you override your current language (for example, English, French, or Japanese). For more information see *Localized Message Database Files in IBM ILOG Views*.

> ***Note:*** *Although IBM ILOG Views does not support multilingual applications, you can use multiple languages in the same application if they use compatible encodings.*

Language of the Message Database Files

IBM® ILOG® Views is released with the message database files in English and French. The files for each supported language are found in a separate directory to facilitate the use of multiple languages using different encoding methods. If you require a language other than English or French, you must translate the `.dbm` files into the desired language. Make sure your files are in the correct `.dbm` format. (See the section *.dbm File Format* for more information.)

Location of the Message Database Files

Localized message databases are located in a subdirectory under the `locale` directory. This subdirectory is named after the corresponding language and the encoding method used. The subdirectory name has the following format:

```
<ll_TT.encoding>
```

For example, on a UNIX system, the French message database files are found in the subdirectory `fr_Fr.ISO-8859-1` under the `locale` directory. See the section *IBM ILOG Views Locale Names* for more information on the IBM ILOG Views locale naming conventions from which the subdirectory name is derived.

On UNIX Systems

French message database files can be found in the following directories:

- `<$ILVHOME>/bin/data/locale/fr_FR.ISO-8859-1/editpnl.dbm`
- `<$ILVHOME>/bin/data/locale/fr_FR.ISO-8859-1/ilv2data.dbm`
- `<$ILVHOME>/bin/data/locale/fr_FR.ISO-8859-1/ilvedit.dbm`
- `<$ILVHOME>/data/ivprotos/locale/fr_FR.ISO-8859-1/protos.dbm`
- `<$ILVHOME>/data/iljscript/locale/fr_FR.ISO-8859-1/gide.dbm`
- `<$ILVHOME>/data/iljscript/locale/fr_FR.ISO-8859-1/messages.js`
- `<$ILVHOME>/data/ilviews/locale/fr_FR.ISO-8859-1/views.dbm`
- `<$ILVHOME>/studio/data/ivprotos/locale/fr_FR.ISO-8859-1 / prstudio.dbm`
- `<$ILVHOME>/studio/data/ivstudio/locale/fr_FR.ISO-8859-1/ jsstudio.dbm`
- `<$ILVHOME>/studio/data/ivstudio/locale/fr_FR.ISO-8859-1/ studio.dbm`
- `<$ILVHOME>/studio/data/ivstudio/locale/fr_FR.ISO-8859-1/ vrstudio.dbm`

On Microsoft Windows Systems

French message database files can be found in the following directories:

- <$ILVHOME>/bin/data/locale/fr_FR.windows-1252/editpnl.dbm
- <$ILVHOME>/bin/data/locale/fr_FR.windows-1252/ilv2data.dbm
- <$ILVHOME>/bin/data/locale/fr_FR.windows-1252/ilvedit.dbm
- <$ILVHOME>/data/ivprotos/locale/fr_FR.windows-1252/protos.dbm
- <$ILVHOME>/data/iljscript/locale/fr_FR.windows-1252/gide.dbm
- <$ILVHOME>/data/iljscript/locale/fr_FR.windows-1252/messages.js
- <$ILVHOME>/data/ilviews/locale/fr_FR.windows-1252/views.dbm
- <$ILVHOME>/studio/data/ivprotos/locale/fr_FR.windows-1252 / prstudio.dbm
- <$ILVHOME>/studio/data/ivstudio/locale/fr_FR.windows-1252/ jsstudio.dbm
- <$ILVHOME>/studio/data/ivstudio/locale/fr_FR.windows-1252/ studio.dbm
- <$ILVHOME>/studio/data/ivstudio/locale/fr_FR.windows-1252/ vrstudio.dbm

You can run the following program to help you find the location of your message database files.

```
// --------------------------------------------------------------- -*- C++ -*-
//                                                  IlogViews userman source file
// File: doc/fondation/userman/src/internationalization/checkLocalizedPath.cpp
// --------------------------------------------------------------------------
//  Copyright (C) 1990-2008 by ILOG.
//  All Rights Reserved.
// --------------------------------------------------------------------------


#include <ilviews/ilv.h>
#include <ilog/pathname.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    if (!IlvSetLocale()) {
exit(1);
    }

    IlPathName pname("");
    pname.localize();
    IlvPrint("\nLooking under directories: .../%s\n\n",
     pname.getString().getValue());

    return 0;
}
```

> ***Note:*** *The C locale (that is, the IBM ILOG Views locale* `en_US`*), which is considered the standard, is an exception to the above mentioned rule. IBM ILOG Views* `.dbm` *files are located in the directory of the library that uses them. For example,* `views.dbm` *can be found in the directory:*
> `<$ILVHOME>/data/ilviews/views.dbm`*.. You do not need to create the* `en_US.US-ASCII` *directory. IBM ILOG Views will automatically fall back to the regular data directory.*

For example, on an HP-UX system with your LANG set to fr_FR.iso88591 or on a Solaris system with your LANG set to fr, you get the following result:

```
Looking under directories: .../locale/fr_FR.ISO-8859-1/
```

On a Windows system set to Japanese, you get:

```
Looking under directories: .../locale\ja_JP.Shift_JIS\
```

Determining Parameters of the Message Database Files

You can run the following program to determine the current IBM® ILOG® Views display language and the path names where IBM ILOG Views looks for the message database files.

You need to set the `verboseFindInPath` environment variable to `true` for the path names to be printed out, and you can play with the `ILVPATH` environment variable to see the effect.

```
// ------------------------------------------------------------- -*- C++ -*-
//                                                 IlogViews userman source file
// File: doc/fondation/userman/src/internationalization/checkLocalizedDbm.cpp
// ------------------------------------------------------------------------
//  Copyright (C) 1990-2008 by ILOG.
//  All Rights Reserved.
// ------------------------------------------------------------------------

#include <ilviews/ilv.h>
#include <ilviews/base/message.h>
#include <ilog/pathlist.h>

int main(int argc, char* argv[])
{
    if (!IlvSetLocale()) {
        exit(1);
    }

    IlvDisplay* display = new IlvDisplay("CheckLocalizedDbm", 0, argc, argv);

    IlvPrint("Current Views display language: %s\n",
     display->getCurrentLanguage()->name());

    const char* path = display->getPath();

    IlPathList plist(path? path : "./");
    IlvPrint("Current path: %s\n", plist.getString().getValue());

    display->getDatabase()->read("my-file.dbm", display);

    return 0;
}
```

For example, if you run this program on an HP-UX system where `LANG=fr_FR.iso88591` or on a Solaris system where `LANG=fr`, you will get the following results:

```
IlvPathList::findInPath file ilviews/locale/fr_FR.ISO-8859-1/views.dbm not in ./.
IlvPathList::findInPath found: <$ILVHOME>/data/ilviews/locale/fr_FR.ISO-8859-1/views.dbm.

Current Views display language: fr_FR

 Current path: ./

IlvPathList::findInPath file locale/fr_FR.ISO-8859-1/my-file.dbm not in ./.
IlvPathList::findInPath file locale/fr_FR.ISO-8859-1/my-file.dbm not in <$ILVHOME>/data/.
IlvPathList::findInPath file locale/fr_FR.ISO-8859-1/my-file.dbm not in <$ILVHOME>/data/icon/.
IlvPathList::findInPath file locale/fr_FR.ISO-8859-1/my-file.dbm not in <$ILVHOME>/data/images/
.
 IlvDisplay::findInPath Couldn't find 'locale/fr_FR.ISO-8859-1/my-file.dbm'
IlvPathList::findInPath file my-file.dbm not in ./.
IlvPathList::findInPath file my-file.dbm not in <$ILVHOME>/data/.
IlvPathList::findInPath file my-file.dbm not in <$ILVHOME>/data/icon/.
IlvPathList::findInPath file my-file.dbm not in <$ILVHOME>/data/images/.
 IlvDisplay::findInPath Couldn't find 'my-file.dbm'
```

Loading the Message Database

IBM® ILOG® Views automatically loads the correct message database, which is located in the `locale/<ll_TT.encoding>` directory.

For example, if you work in an ISO 8859-1 French environment, the following call:

```
display->getDatabase()->read("/my-directory-path/my-file.dbm");
```

will automatically look for a file in the following directory on UNIX systems:

```
/my-directory-path/locale/fr_FR.ISO-8859-1/my-file.dbm
```

and in the following directory on Microsoft Windows systems:

```
/my-directory-path/locale/fr_FR.windows-1252/my-file.dbm
```

> ***Note:*** *In versions of IBM ILOG Views before 3.0, users had to set the* `ILVLANG` *environment variable to the language they wanted to use.*

Overriding the Default Behavior

If you want to override the default behavior and have IBM ILOG Views use another display language, you can use the `ILVLANG` environment variable. On a XPG4-compliant UNIX system, you can also use the `LC_MESSAGES` environment variable. IBM ILOG Views will look for the message database files in the following order:

On UNIX Systems

1. ILVLANG
2. LC_MESSAGES
3. LC_CTYPE category of your running locale

On Windows Systems

1. ILVLANG
2. LC_CTYPE category of your running locale

> ***Note:*** *If you want to change your IBM ILOG Views display language by overriding it with* LC_MESSAGES *or* ILVLANG *environment variables, you need to be sure your program runs in the same or a stronger encoding (that is, a superset of the encoding) than the one you plan to use for your messages. This is because the* .dbm *files are read based on the IBM ILOG Views locale encoding your program runs in. For example, if you run a program in Japanese or French, you can always read English messages, the opposite is not true.*

Overriding the Default Behavior Using the LANG Resource

On UNIX Systems

You can set the ILVLANG environment variable to use a language other than the default. ILVLANG applies to IBM ILOG Views applications only and is system-independent.

For example, if your current IBM ILOG Views locale is French and you want to see Spanish messages, you could use ILVLANG=es_ES on any UNIX system. Your settings would then be as follows:

```
Current Views locale running: fr_FR.ISO-8859-1
Current Views display language: es_ES
Looking under directories: .../locale/es_ES.ISO-8859-1/
```

In this case, only the messages in your IBM ILOG Views applications will be displayed in Spanish. The system messages are not affected.

On Microsoft Windows Systems

You can set the lang variable in your views.ini file to use a language other than the default. For example, if your current IBM ILOG Views locale is French and you want to see Spanish messages, you could use lang=es_ES. Your settings would then be as follows:

```
Current Views locale running: fr_FR.windows-1252
Current Views display language: es_ES
Looking under directories: .../locale/es_ES.windows-1252/
```

Overriding the Default Behavior using LC_MESSAGES (UNIX only)

You can set the LC_MESSAGES environment variable to use a language other than the default. You should note that if you use the LC_MESSAGES environment variable this will override all system messages as well.

For example, if your current IBM ILOG Views locale is French and you want to see Italian messages, you could use `LC_MESSAGES=it_IT.iso88591` on an HP-UX system or `LC_MESSAGES=it` on a Solaris system. Your settings would then be as follows:

```
Current Views locale running: fr_FR.ISO-8859-1
Current Views display language: it_IT
Looking under directories: .../locale/it_IT.ISO-8859-1/
```

In this case, not only will your IBM ILOG Views messages appear in Italian, but all your system messages will be in Italian as well.

.dbm File Format

In the `.dbm` format, each supported language is stored in a separate database in order to deal with multiple languages using different encoding methods.

A `.dbm` file has the following format:

```
// IlvMessageDatabase ...
// Language: <ll_TT>
// Encoding: <encoding>
"&message" "message translation..."
```

The first line is an information line containing information about `IlvMessageDatabase`, with the IBM® ILOG® Views version and the creation date.

The language is represented using the `ll_TT` naming convention of IBM ILOG Views where `ll` is the two-letter abbreviation for the language name and `TT` is the two-letter abbreviation for the territory name.

The encoding method must be one of the methods supported by IBM ILOG Views. See the section *Reference: Encoding Listings* for a list of supported encoding methods.

The following example shows part of a message database file for French:

```
// IlvMessageDatabase
// Language: fr_FR
// Encoding: ISO-8859-1
"&AlignmentLabelPicture" "Alignement texte / image"
"&Appearance" "Apparence"
"&April" "avril"
```

If you are translating your .dbm files to your local language, make sure your files are in this .dbm format.

***Note:** It is recommended that you do the following for localized database message files for American English. **Do not** create an* en_US.US-ASCII *subdirectory in the* locale *directory for your files. Put your files directly in your data directory; for example,* views.dbm *located in* <$ILVHOME>/data/ilviews/*. Set the contents of the files as shown in the following example, even if the encoding that you are running for American English is not US-ASCII. You can do this because US-ASCII is the weakest encoding and can be read by any other encoding that IBM ILOG Views supports.*

```
// IlvMessageDatabase
// Language: en_US
// Encoding: US-ASCII
"&AlignmentLabelPicture" "Alignment text / picture"
"&Appearance" "Appearance"
"&April" "April"
```

.dbm File Format in Versions Before 3.0

In IBM ILOG Views version 3.0, the .dbm file format was enhanced to support different languages that use different and incompatible encoding methods. In the .dbm file format of versions before 3.0, a message database contains the translation of each message to each of the supported languages. In other words, all supported language translations are found in the same database. Although files in the old .dbm format can still be read using IBM ILOG Views 3.0 and later, new files are generated in the new format.

If you have database files in the old .dbm format, it is a good idea to split your databases into several files, one for each supported language. Not only will this make maintenance easier, but it will also avoid encoding incompatibilities.

To split your database files with the old format, use the following program:

```
$ILVHOME/bin/src/splitdbm.cpp
```

This program finds the various languages in your .dbm file and suggests a new language name for each one of them (we recommend that you use the IBM ILOG Views naming

convention `ll_TT`), an encoding method (the selected encoding method must be compatible with the current one), and a name for the file with the new format.

> ***Note:*** *You should run the* `splitdbm` *program using the strongest encoding. The strongest encoding is the one that encompasses the others. For example, if you want to split a file that contains English (US-ASCII) and Japanese (Shift_JIS), you should run* `splitdbm` *using a Japanese locale. Shift_JIS contains US-ASCII, but the opposite is not true. Therefore, Shift_JIS is the strongest encoding and should be used for running the program.*

> ***Note:*** *It is recommended that you do the following when you are splitting a file containing American English messages. When the* `splitdbm` *program prompts you for information, choose the US-ASCII encoding and store the localized file in your data directory, not under a* `locale` *subdirectory.*

Example

This example shows how to split a database message file.

The following file called `your_data_dir/testall.dbm` contains the message text for three languages, American English, French, and Italian:

```
// IlvMessageDatabase 3 Web Jun 3 11:50:35 1998
"&Hello" 3
"en_US" "Hello"
"fr_FR" "Bonjour"
"it_IT" "Buongiórno"
"&Goodbye" 3
"en_US" "Goodbye"
"fr_FR" "Au revoir"
"it_IT" "Ciao"
```

To split this file into three files, one file for each language, you should run the `splitdbm` program in the French or Italian locale. When you run the program, you will be prompted for the information the program needs to complete its run. When the run finishes, you should have three files, each file containing the message text for a single language. On a UNIX system, the resulting files would be:

```
your_data_dir/test.dbm
locale/fr_FR.ISO-8859-1/test.dbm
locale/it_IT.ISO-8859-1/test.dbm
```

The contents of each file would be as follows:

```
// IlvMessageDatabase
// Language: en_US
// Encoding: US-ASCII
"&Goodbye" "Goodbye"
"&Hello" "Hello"

// IlvMessageDatabase
```

```
// Language: fr_FR
// Encoding: ISO-8859-1
"&Goodbye" "Au revoir"
"&Hello" "Bonjour"

// IlvMessageDatabase
// Language: it_IT
// Encoding: ISO-8859-1
"&Goodbye" "Ciao"
"&Hello" "Buongiórno"
```

Encoding Compatibility of .dbm Files

Note that an IBM ILOG Views application will only load `.dbm` files that were written using an encoding that is compatible with the system environment. Encodings are compatible if they share the same character set. If you load `.dbm` files that still use the old format and that do not contain encoding information, these files are supposed to be in the encoding of the current locale. Otherwise, information might be improperly or incompletely loaded.

How to Dynamically Change Your Display Language

You can dynamically change your display language. You simply need to call `IlvDisplay::setCurrentLanguage` on the display. IBM ILOG Views will reload automatically all the data files that you currently have loaded and display the new language. In order to do this, you have to provide the localized versions of these files and they must be available on your system.

> ***Note:*** *If you plan to switch languages, you need to start your application in the strongest encoding. For example, if you plan to switch between French and English, start your application in French. If however, you want your application to show English messages when started, override your start-up display language by using the* `ILVLANG` *environment variable or the* `lang` *resource on Windows.*

Let's say you have created an application where all the messages are defined in a file named `my_messages.dbm`.

If `display` is your display, just load this file at the beginning of your application through a call to:

```
display->getDatabase()->read("my_messages.dbm", display);
```

If you have started your program in a French locale, IBM ILOG Views will load the file located in `locale/fr_FR.ISO-8859-1/my_messages.dbm`.

Now, to change your display language, just call `IlvDisplay::setCurrentLanguage` with the new language you want. For example, if you want your display to be in Italian, call:

```
display->setCurrentLanguage(IlGetSymbol("it_IT"));
```

IBM ILOG Views will automatically load the file located in:

```
locale/it_IT.ISO-8859-1/my_messages.dbm
```

It will also load all the other data files that are already open.

To go back to French, call `IlvDisplay::setCurrentLanguage` again as follows:

```
display->setCurrentLanguage(IlGetSymbol("fr_FR"));
```

Note: *This can only be done if the encodings are compatible.*

The sample that you can find in `samples/foundation/i18n/changelang` is an illustration of this feature.

Using IBM ILOG Views with Far Eastern Languages

You should read this section if you want your system to support Far Eastern languages, such as Japanese, Korean, or Chinese. Far Eastern languages are multibyte character languages that present certain distinctive characteristics that should be taken into account when using IBM® ILOG® Views.

Although the API remains unchanged, you should keep in mind that a `char*` value can contain multibyte characters in a Far Eastern locale.

For example, you can pass a multibyte string to:

```
void IlvListLabel::setText(const char* text);
```

and get a multibyte string as the return value of the following:

```
const char* IlvListLabel::getText() const;
```

This is true for all the gadget classes (that is, `IlvText`, `IlvTextField` and its subclasses, `IlvMessageLabel`, `IlvStringList`, and so on) and the manager view interactors, such as `IlvManagerMakeStringInteractor` or `IlvManagerMakeTextInteractor`.

To help programmers control input in text areas, the `mbCheck` method has been added to the `IlvTextField` and `IlvText` gadgets. The APIs are defined as follows:

For `IlvText`:

```
virtual IlBoolean  mbCheck(const char* text);
```

For `IlvTextField`:

```
virtual const char* mbCheck(const char* text);
```

Note: *The* `mbCheck` *method calls the* `check` *method when running in a monobyte locale.*

`IlvPasswordTextField` supports multibyte strings, and the mask is applied by drawable characters. This means that you can create or perform a `setLabel` on an `IlvPasswordTextField` using multibyte strings.

Internally, IBM ILOG Views manipulates `wide-char*` values, but there are no documented public APIs. If you want to use `wide-char*` values, you must convert back and forth to `char*` values before calling any public API. Note that you can use the `IlvWChar` type, which is defined as `wchar_t` in the `macros.h` file, for your own internationalized API.

Data Input Requirements

- IBM® ILOG® Views provides Input Method support, described in *Input Method (IM)* on page 227.
- Controlling input contexts is discussed in *Far Eastern Input Method Servers Tested with IBM ILOG Views* on page 228.
- You can disallow localized input as shown in the example in *How to Control the Language Used for Data Input* on page 228.

Input Method (IM)

Some languages, such as Far Eastern languages, use a lot of characters. The concept of Input Method (IM) has been developed to allow entering these characters using the keyboard. An Input Method is a procedure, a macro, or sometimes a separate process that converts keystrokes to characters that are encoded in the code set of the current locale.

On UNIX systems, European Input Methods are directly supported in the X library. However, Far Eastern languages require a separate process to be run.

For these languages, you must have an Input Method (also called a Front-End processor) running on your system, and set your environment accordingly. On UNIX systems, for example, this could mean setting the `XMODIFIERS` environment variable. Please check your local system documentation to see what needs to be done.

Input through an Input Method server is supported in the following classes:

- `IlvText`, `IlvTextField` and its subclasses (except for `IlvDateField`, `IlvNumberField`, and `IlvPasswordTextField`).
- Any class that uses an `IlvTextField` to enter text is also able to use an Input Method server. This is true, for example, for an `IlvMatrix` or the `IlvManagerMakeStringInteractor`.

Far Eastern Input Method Servers Tested with IBM ILOG Views

On a UNIX system, you can control the way you use your input contexts. The default is to share one input context from the top level window. This means that all the input text areas of this top level window share the input context.

If you want to use different input contexts for each of your input text areas, you can set the `ILVICSHARED` environment variable to "`no`".

On HP-UX, the following IM servers have been tested successfully with IBM ILOG Views:

- For Japanese: `xjim`, `atok8`
- For Chinese: `xtim`, `xsim`
- For Korean: `xkim`

On Solaris, the following IM servers have been tested successfully with IBM ILOG Views:

- For Japanese: `htt`
- For Chinese: `htt`
- For Korean: `htt`

On Windows IBM ILOG Views directly connects to the default IME server.

How to Control the Language Used for Data Input

Any input field object, a subclass of `IlvText` or `IlvTextField`, will automatically connect to an Input Method (as long as `SetLocale` has been called) so that input can be done in the current locale.

If an application wants to disallow this behavior (that is, it does not want localized input, but only ASCII input), it must call the `setNeedsInputContext` method on the `SimpleGraphic` object with the parameter value `IlFalse`.

```
virtual void setNeedsInputContext(IlBoolean val)
```

Example

The code sample in `samples/foundation/i18n/controlinput` creates two text field gadgets. The first text field connects to an Input Method so that input can be done in the current locale. The second text field does not connect to an Input Method, meaning that you may only be able to type English characters in it.

Limitations of Internationalization Features

The following limitations apply to the current internationalization support features:

- Saving or reading a string in an `.ilv` file is carried out in the encoding of the current locale. Trying to read a file which is not encoded in the current locale will fail.
- No input through an Input Method server is supported for the password text field, since the user doesn't have to see what he is typing.
- No input through an Input Method server is currently supported for the date field and the number field. On Windows platforms, the user must disconnect from the `FEP` in order to be able to enter text in these gadgets.
- `IlvAnnoText` has no internationalization support.
- Depending on the fonts or the language you use, the IBM ILOG Views Studio Main Window may be too small. But as with any IBM ILOG Views Studio panel, the Main Window size can be customized by setting the width in the `studio.pnl` file:

```
panel "MainPanel" {
 // ....
width 900;
}
```

- Multibyte variables are not supported in this version. For information on the variables module of IBM ILOG Views, see the Manager documentation.
- You can define any single-byte character with IBM ILOG Views for mnemonics. We recommend, however, you install mnemonics on single-byte characters that have a corresponding keyboard key. For example, we do not recommend using accents in European languages and Hankaku characters in Japanese.
- On UNIX systems, only Input Methods implemented using a Back-End architecture are supported. For example, if you are using the `htt` import server on Solaris, you should set the `XIMP_TYPE` environment variable to `XIMP_SYNC_BE_TYPE2` before running your application.

Troubleshooting

If your localized messages do not appear on your screen, follow these steps:

1. Check that you called `IlvSetLocale` at the beginning of your program.
2. Check that your system supports the locale and the fonts to display it. On most UNIX systems, you can run the `locale -a` command. See the section *Locale Requirements*.
3. Do not set the `ILVLANG` environment variable.
4. On UNIX platforms, set `LANG` to a locale supported by your system and by the X Window system. For example, to set the LANG variable to French, type:

   `LANG=fr` on Solaris

`LANG=fr_FR.iso88591` on HP_UX

5. Check that the localized `.dbm` file in a subdirectory named `.../locale/<ll_TT.encoding>/your_file.dbm`. See the section *Location of the Message Database Files*.

6. Check that the contents of your `.dbm` file has the following new format. See the section *.dbm File Format*.

```
// IlvMessageDatabase ...
// Language: <ll_TT>
// Encoding: <encoding>
"&message" "message translation..."
```

7. If you read a `.dbm` file with the old format from the IBM ILOG Views Studio editor (`.dbm` files with the old format are files created with versions of IBM ILOG Views before 3.0), and your file appears truncated, this means the encodings are not compatible. In this case, split your `.dbm` file. See the section *.dbm File Format in Versions Before 3.0*.

Reference: Encoding Listings

The following encodings are supported by IBM ILOG Views:

- US-ASCII
- ISO-8859-1 (Latin1)
- ISO-8859-2 (Latin2)
- ISO-8859-3 (Latin3)
- ISO-8859-4 (Latin4)
- ISO-8859-5 (LatinCyrillic)
- ISO-8859-6 (LatinArabic)
- ISO-8859-7 (LatinGreek)
- ISO-8859-8 (LatinHebrew)
- ISO-8859-9 (Latin5)
- ISO-8859-10 (Latin6)
- ISO-8859-11 (LatinThai)
- ISO-8859-13 (Latin7)
- ISO-8859-14 (Latin8)
- SO-8859-15 (Latin9)

- EUC-JP
- Shift_JIS
- EUC-KR
- GB2312
- Big5
- Big5-HKSCS
- EUC-TW
- hp-roman8
- IBM850
- windows-1250
- windows-1251
- windows-1252
- windows-1253
- windows-1254
- windows-1255
- windows-1256
- windows-1257
- windows-1258
- windows-874
- windows-949
- UTF-8

ISO-8859-1

Latin1 covers most West European languages, such as:

- Afrikaans (af)
- Albanian (sq)
- Basque (eu)
- Catalan (ca)
- Danish (da)
- Dutch (nl)

- English (en)
- Faroese (fo)
- Finnish (fi)
- French (fr)
- Galician (gl)
- German (de)
- Icelandic (is)
- Irish (ga)
- Italian (it)
- Norwegian (no)
- Portuguese (pt)
- Scottish (gd)
- Spanish (es)
- Swedish (sv)

ISO-8859-2

Latin2 covers the languages of Central and Eastern Europe:

- Croatian (hr),
- Czech (cs),
- Hungarian (hu),
- Polish (pl),
- Romanian (ro),
- Slovak (sk),
- Slovenian (sl)

ISO-8859-3

Latin3 is popular with authors of Esperanto (eo), Maltese (mt), and it covered Turkish before the introduction of Latin5.

ISO-8859-4

Latin4 introduced letters for Estonian, Baltic languages, Latvian and Lithuanian, Greenlandic and Lappish. It is an incomplete precursor of Latin6.

ISO-8859-5

With these Cyrillic letters you can type Bulgarian (bg), Byelorussian (be), Macedonian (mk), Russian (ru), Serbian (sr) and Ukrainian (uk).

ISO-8859-6

This is the Arabic (ar) alphabet.

Note: *This version of IBM ILOG Views does not support bidirectional text.*

ISO-8859-7

This is modern Greek (el).

ISO-8859-8

This is Hebrew (iw).

Note: *This version of IBM ILOG Views does not support bidirectional text.*

ISO-8859-9

Latin5 replaces the rarely needed Icelandic letters in Latin1 with the Turkish (tr) ones.

ISO-8859-10

Latin6 rearranged Latin4, added the last missing Inuit (Greenlandic Eskimo) and non-Skolt Sami (Lappish) letters, and reintroduced the rarely Icelandic letters to cover the entire Nordic area:

- Estonian (et)
- Lapp
- Latvian (lv)
- Lithuanian (lt)

Skolt Sami still needs a few more accents.

ISO-8859-11

To cover the Thai language. On UNIX systems, this is similar to the tis620 encoding.

ISO-8859-13

To cover the Baltic Rim. Latin7 is going to cover the Baltic Rim and re-establish the Latvian (lv) support lost in Latin6 and may introduce the local quotation marks. It resembles WinBaltic, that is, windows-1257.

ISO-8859-14

To cover Celtic. Latin8 adds the last Gaelic and Welsh (cy) letters to Latin1 to cover all Celtic languages.

ISO-8859-15

Similar to Latin1 with euro and oe ligature. The new Latin9 nicknamed Latin0 aims to update Latin1 by replacing the less needed symbols ¨´¸ with forgotten French and Finnish letters and placing the U+20AC Euro sign in the cell =A4 of the former international currency sign ¤.

EUC-JP

Extended UNIX Code for Japanese.

Standardized by OSF, UNIX International, and UNIX Systems Laboratories Pacific. Uses ISO 2022 rules to select:

- code set 0: JIS Roman (a single 7-bit byte set)
- code set 1: JIS X0208-1990 (a double 8-bit byte set) restricted to A0-FF in both bytes
- code set 2: Half Width Katakana (a single 7-bit byte set) requiring SS2 as the character prefix
- code set 3: JIS X0212-1990 (a double 7-bit byte set) restricted to A0-FF in both bytes requiring SS3 as the character prefix

Shift_JIS

A Microsoft code that extends csHalfWidthKatakana to include kanji by adding a second byte when the value of the first byte is in the ranges 81-9F or E0-EF.

EUC-KR (KS C 5861-1992)

Extended UNIX Code for Korean.

GB2312

Multibyte encoding standardized by the People's Republic of China.

Big5

Multibyte encoding standardized by Taiwan

Big5-HKSCS

Hong-Kong Supplementary Character Set

EUC-TW (cns11643)

Extended UNIX Code for Traditional Chinese

hp-roman8

HP specific

IBM850

IBM specific

windows-1250

Windows 3.1 Eastern European languages.

windows-1251

Windows 3.1 Cyrillic

windows-1252

Windows 3.1 US (ANSI)

windows-1253

Windows 3.1 Greek

windows-1254

Windows 3.1 Turkish

windows-1255

Hebrew

Note: *This version of IBM ILOG Views does not support bidirectional text.*

windows-1256

Arabic

Note: *This version of IBM ILOG Views does not support bidirectional text.*

windows-1257

Baltic

windows-1258

Vietnamese

windows-874

Thai

windows-949

Korean (Wansung)

UTF-8

Unicode UTF-8

Reference: Supported Locales on Different Platforms

The following tables list, by platform, locales that have been successfully tested as Views locales.

The first table lists the locales currently supported on Microsoft Windows platforms. If support of a locale is limited to particular platforms, this is listed in the last colum. If this column is blank then all platforms (2000 to Vista) are supported.

.

Table 15.1 *Microsoft Windows Locale Support*

Windows Locale Name	Code Page	Views Locale Name	Limited to Windows
Afrikaans_South Africa	1252	af_ZA.windows-1252	
Albanian_Albania	1250	sq_AL.windows-1250	
Arabic_Algeria	1256	ar_DZ.windows-1256	
Arabic_Bahrain	1256	ar_BH.windows-1256	
Arabic_Egypt	1256	ar_EG.windows-1256	
Arabic_Iraq	1256	ar_IQ.windows-1256	
Arabic_Jordan	1256	ar_JO.windows-1256	
Arabic_Kuwait	1256	ar_KW.windows-1256	
Arabic_Lebanon	1256	ar_LB.windows-1256	
Arabic_Libya	1256	ar_LY.windows-1256	
Arabic_Morocco	1256	ar_MA.windows-1256	
Arabic_Oman	1256	ar_OM.windows-1256	
Arabic_Qatar	1256	ar_QA.windows-1256	
Arabic_Saudi Arabia	1256	ar_SA.windows-1256	
Arabic_Syria	1256	ar_SY.windows-1256	
Arabic_Tunisia	1256	ar_TN.windows-1256	

Table 15.1 *Microsoft Windows Locale Support (Continued)*

Windows Locale Name	Code Page	Views Locale Name	Limited to Windows
Arabic_U.A.E.	1256	ar_AE.windows-1256	
Arabic_Yemen	1256	ar_YE.windows-1256	
Azeri (Cyrillic)_Azerbaijan	1251	az_AZ.windows-1251	
Azeri (Latin)_Azerbaijan	1254	az_AZ.windows-1254	
Basque_Spain	1252	eu_ES.windows-1252	
Belarusian_Belarus	1251	be_BY.windows-1251	
Bulgarian_Bulgaria	1251	bg_BG.windows-1251	
Catalan_Spain	1252	ca_ES.windows-1252	
Chinese_Hong Kong	950	zh_HK.Big5	2000
Chinese_Hong Kong S.A.R	950	zh_HK.Big5-HKSCS	XP (see *Support of HKSCS* below)
Chinese_Macau	950	zh_MO.Big5	2000
Chinese_People's Republic of China	936	zh_CN.GB2312	
Chinese_Singapore	936	zh_SG.GB2312	
Chinese_Taiwan	950	zh_TW.Big5	
Croatian_Croatia	1250	hr_HR.windows-1250	
Czech_Czech Republic	1250	cs_CZ.windows-1250	
Danish_Denmark	1252	da_DK.windows-1252	
Dutch_Belgium	1252	nl_BE.windows-1252	
Dutch_Netherlands	1252	nl_NL.windows-1252	
English_Australia	1252	en_AU.windows-1252	
English_Belize	1252	en_BZ.windows-1252	
English_Ireland	1252	en_IE.windows-1252	
English_Jamaica	1252	en_JM.windows-1252	
English_New Zealand	1252	en_NZ.windows-1252	

Table 15.1 *Microsoft Windows Locale Support (Continued)*

Windows Locale Name	Code Page	Views Locale Name	Limited to Windows
English_Republic of the Philippines	1252	en_PH.windows-1252	
English_South Africa	1252	en_ZA.windows-1252	
English_Trinidad y Tobago	1252	en_TT.windows-1252	2000
English_Zimbabwe	1252	en_ZW.windows-1252	
English_United States	1252	en_US.windows-1252	
English_United Kingdom	1252	en_GB.windows-1252	
Estonian_Estonia	1257	et_EE.windows-1257	
Faeroese_Faeroe Islands	1252	fo_FO.windows-1252	2000
Farsi_Iran	1256	fa_IR.windows-1256	
Finnish_Finland	1252	fi_FI.windows-1252	
French_Belgium	1252	fr_BE.windows-1252	
French_Canada	1252	fr_CA.windows-1252	
French_France	1252	fr_FR.windows-1252	
French_Luxembourg	1252	fr_LU.windows-1252	
French_Principality of Monaco	1252	fr_MC.windows-1252	
French_Switzerland	1252	fr_CH.windows-1252	
German_Austria	1252	de_AT.windows-1252	
German_Germany	1252	de_DE.windows-1252	
German_Liechtenstein	1252	de_LI.windows-1252	
German_Luxembourg	1252	de_LU.windows-1252	
German_Switzerland	1252	de_CH.windows-1252	
Greek_Greece	1253	el_GR.windows-1253	
Hebrew_Israel	1255	iw_IL.windows-1255	
Hungarian_Hungary	1250	hu_HU.windows-1250	

Table 15.1 *Microsoft Windows Locale Support (Continued)*

Windows Locale Name	Code Page	Views Locale Name	Limited to Windows
Icelandic_Iceland	1252	is_IS.windows-1252	
Indonesian_Indonesia	1252	in_ID.windows-1252	
Italian_Italy	1252	it_IT.windows-1252	
Italian_Switzerland	1252	it_CH.windows-1252	
Kazakh_Kazakstan	1251	kk_KZ.windows-1251	
Japanese_Japan	932	ja_JP.Shift_JIS	
Korean_Korea	949	ko_KR.windows-949	
Latvian_Latvia	1257	lv_LV.windows-1257	
Lithuanian_Lithuania	1257	bo_LT.windows-1257	
Macedonian_Former Yugoslav Republic of Macedonia	1251	mk_MK.windows-1251	
Malay_Brunei Darussalam	1252	ms_BN.windows-1252	
Malay_Malaysia	1252	ms_MY.windows-1252	
Norwegian (Bokmål)_Norway	1252	no_NO.windows-1252	
Norwegian (Nynorsk)_Norway	1252	no_NO.windows-1252	
Norwegian_Norway	1252	no_NO.windows-1252	2000
Polish_Poland	1250	pl_PL.windows-1250	
Portuguese_Brazil	1252	pt_BR.windows-1252	
Portuguese_Portugal	1252	pt_PT.windows-1252	
Romanian_Romania	1250	ro_RO.windows-1250	
Russian_Russia	1251	ru_RU.windows-1251	
Serbian (Latin)_Serbia	1250	sh_YU.windows-1250	2000
Serbian (Cyrillic)_Serbia	1251	sr_YU.windows-1251	2000

Table 15.1 *Microsoft Windows Locale Support (Continued)*

Windows Locale Name	Code Page	Views Locale Name	Limited to Windows
Slovak_Slovakia	1250	sk_SK.windows-1250	
Slovenian_Slovenia	1250	sl_SI.windows-1250	
Spanish_Argentina	1252	es_AR.windows-1252	
Spanish_Bolivia	1252	es_BO.windows-1252	
Spanish_Chile	1252	es_CL.windows-1252	
Spanish_Colombia	1252	es_CO.windows-1252	
Spanish_Costa Rica	1252	es_CR.windows-1252	
Spanish_Dominican Republic	1252	es_DO.windows-1252	
Spanish_Ecuador	1252	es_EC.windows-1252	
Spanish_El Salvador	1252	es_SV.windows-1252	
Spanish_Guatemala	1252	es_GT.windows-1252	
Spanish_Mexico	1252	es_MX.windows-1252	
Spanish_Honduras	1252	es_HN.windows-1252	
Spanish_Nicaragua	1252	es_NI.windows-1252	
Spanish_Panama	1252	es_PA.windows-1252	
Spanish_Paraguay	1252	es_PY.windows-1252	
Spanish_Peru	1252	es_PE.windows-1252	
Spanish - Modern Sort_Spain	1252	es_ES.windows-1252	2000
Spanish_Puerto Rico	1252	es_PR.windows-1252	
Spanish - Traditional Sort_Spain	1252	es_ES.windows-1252	2000
Spanish_Spain	1252	es_ES.windows-1252	
Spanish_Uruguay	1252	es_UY.windows-1252	
Spanish_Venezuela	1252	es_VE.windows-1252	

Table 15.1 *Microsoft Windows Locale Support (Continued)*

Windows Locale Name	Code Page	Views Locale Name	Limited to Windows
Swahili_Kenya	1252	sw_KE.windows-1252	
Swedish_Finland	1252	sv_FI.windows-1252	
Swedish_Sweden	1252	sv_SE.windows-1252	
Tatar_Tatarstan	1251	tt_TS.windows-1251	2000
Thai_Thailand	874	th_TH.windows-874	
Turkish_Turkey	1254	tr_TR.windows-1254	
Ukrainian_Ukraine	1251	uk_UA.windows-1251	
Urdu_Islamic Republic of Pakistan	1256	ur_PK.windows-1256	
Uzbek_Republic of Uzbekistan	1251	uz_UZ.windows-1251	2000

Support of HKSCS

You will need to install a specific package in order to support the Hong Kong Supplementary Chararcter Set on Windows 2000 and Windows XP (see `http://www.microsoft.com/hk/hkscs/`).

Table 15.2 *HP-UX 11 Locale Support*

HP-UX Locale Name	Encoding	Views Locale Name
C	roman8	en_US.US-ASCII
POSIX	roman8	en_US.hp-roman8
C.iso88591	iso88591	en_US.ISO-8859-1
C.utf8	utf8	en_US.UTF-8
univ.utf8	utf8	en_US.UTF-8
ar_SA.iso88596	iso88596	ar_SA.ISO-8859-6
bg_BG.iso88595	iso88595	bg_BG.ISO-8859-5
cs_CZ.iso88592	iso88592	cs_CZ.ISO-8859-2

Table 15.2 *HP-UX 11 Locale Support (Continued)*

HP-UX Locale Name	Encoding	Views Locale Name
da_DK.iso88591	iso88591	da_DK.ISO-8859-1
da_DK.roman8	roman8	da_DK.hp-roman8
de_DE.iso88591	iso88591	de_DE.ISO-8859-1
de_DE.roman8	roman8	de_DE.hp-roman8
el_GR.iso88597	iso88597	el_GR.ISO-8859-7
en_GB.iso88591	iso88591	en_GB.ISO-8859-1
en_GB.roman8	roman8	en_GB.hp-roman8
en_US.iso88591	iso88591	en_US.ISO-8859-1
en_US.roman8	roman8	en_US.hp-roman8
es_ES.iso88591	iso88591	es_ES.ISO-8859-1
es_ES.roman8	roman8	es_ES.hp-roman8
fi_FI.iso88591	iso88591	fi_FI.ISO-8859-1
fi_FI.roman8	roman8	fi_FI.hp-roman8
fr_CA.iso88591	iso88591	fr_CA.ISO-8859-1
fr_CA.roman8	roman8	fr_CA.hp-roman8
fr_FR.iso88591	iso88591	fr_FR.ISO-8859-1
fr_FR.roman8	roman8	fr_FR.hp-roman8
hr_HR.iso88592	iso88592	hr_HR.ISO-8859-2
hu_HU.iso88592	iso88592	hu_HU.ISO-8859-2
is_IS.iso88591	iso88591	is_IS.ISO-8859-1
is_IS.roman8	roman8	is_IS.hp-roman8
it_IT.iso88591	iso88591	it_IT.ISO-8859-1
it_IT.roman8	roman8	it_IT.hp-roman8
iw_IL.iso88598	iso88598	iw_IL.ISO-8859-8
ja_JP.SJIS	SJIS	ja_JP.Shift_JIS

Table 15.2 *HP-UX 11 Locale Support (Continued)*

HP-UX Locale Name	Encoding	Views Locale Name
ja_JP.eucJP	eucJP	ja_JP.EUC-JP
ko_KR.eucKR	eucKR	ko_KR.EUC-KR
nl_NL.iso88591	iso88591	nl_NL.ISO-8859-1
nl_NL.roman8	roman8	nl_NL.hp-roman8
no_NO.iso88591	iso88591	no_NO.ISO-8859-1
no_NO.roman8	roman8	no_NO.hp-roman8
pl_PL.iso88592	iso88592	pl_PL.ISO-8859-2
pt_PT.iso88591	iso88591	pt_PT.ISO-8859-1
pt_PT.roman8	roman8	pt_PT.hp-roman8
ro_RO.iso88592	iso88592	ro_RO.ISO-8859-2
ru_RU.iso88595	iso88595	ru_RU.ISO-8859-5
sk_SK.iso88592	iso88592	sk_SK.ISO-8859-2
sl_SI.iso88592	iso88592	sl_SI.ISO-8859-2
sv_SE.iso88591	iso88591	sv_SE.ISO-8859-1
sv_SE.roman8	roman8	sv_SE.hp-roman8
tr_TR.iso88599	iso88599	tr_TR.ISO-8859-9
zh_CN.hp15CN	hp15CN	zh_CN.GB2312
zh_TW.big5	big5	zh_TW.Big5
zh_TW.eucTW	eucTW	zh_TW.EUC-TW

Table 15.3 *Solaris Locale Support*

Solaris Locale Name	Encoding	Views Locale Name
POSIX	646	en_US.US-ASCII
C	646	en_US.US-ASCII
iso_8859_1	ISO8859	en_US.US-ASCII

Table 15.3 *Solaris Locale Support (Continued)*

Solaris Locale Name	Encoding	Views Locale Name
ar	ISO8859-6	ar_AA.ISO-8859-6
bg_BG	ISO8859-5	bg_BG.ISO-8859-5
cz	ISO8859-2	cs_CZ.ISO-8859-2
da	ISO8859-1	da_DK.ISO-8859-1
da.ISO8859-15	ISO8859-15	da_DK.ISO-8859-15
da.ISO8859-15@euro	ISO8859-15	da_DK.ISO-8859-15
de	ISO8859-1	de_DE.ISO-8859-1
de.ISO8859-15	ISO8859-15	de_DE.ISO-8859-15
de.ISO8859-15@euro	ISO8859-15	de_DE.ISO-8859-15
de.UTF-8	UTF-8	de_DE.UTF-8
de.UTF-8@euro	UTF-8	de_DE.UTF-8
de_AT	ISO8859-1	de_AT.ISO-8859-1
de_AT.ISO8859-15	ISO8859-15	de_AT.ISO-8859-15
de_AT.ISO8859-15@euro	ISO8859-15	de_AT.ISO-8859-15
de_CH	ISO8859-1	de_CH.ISO-8859-1
el	ISO8859-7	el_GR.ISO-8859-7
el.sun_eu_greek	sun_eu_greek	
en_AU	ISO-8859-1	en_AU.ISO-8859-1
en_CA	ISO8859-1	en_CA.ISO-8859-1
en_GB	ISO8859-1	en_GB.ISO-8859-1
en_GB.ISO8859-15	ISO8859-15	en_GB.ISO-8859-15
en_GB.ISO8859-15@euro	ISO8859-15	en_GB.ISO-8859-15
en_IE	ISO8859-1	en_IE.ISO-8859-1
en_IE.ISO8859-15	ISO8859-15	en_IE.ISO-8859-15
en_IE.ISO8859-15@euro	ISO8859-15	en_IE.ISO-8859-15

Table 15.3 *Solaris Locale Support (Continued)*

Solaris Locale Name	Encoding	Views Locale Name
en_NZ	ISO8859-1	en_NZ.ISO-8859-1
en_US	ISO-8859-1	en_US.ISO-8859-1
en_US.UTF-8	UTF-8	en_US.UTF-8
es	ISO-8859-1	es_ES.ISO-8859-1
es.ISO8859-15	ISO8859-15	es_ES.ISO-8859-15
es.ISO8859-15@euro	ISO8859-15	es_ES.ISO-8859-15
es.UTF-8	UTF-8	es_ES.UTF-8
es.UTF-8@euro	UTF-8	es_ES.UTF-8
es_AR	ISO8859-1	es_AR.ISO-8859-1
es_BO	ISO8859-1	es_BO.ISO-8859-1
es_CL	ISO8859-1	es_CL.ISO-8859-1
es_CO	ISO8859-1	es_CO.ISO-8859-1
es_CR	ISO8859-1	es_CR.ISO-8859-1
es_EC	ISO8859-1	es_EC.ISO-8859-1
es_GT	ISO8859-1	es_GT.ISO-8859-1
es_MX	ISO8859-1	es_MX.ISO-8859-1
es_NI	ISO8859-1	es_NI.ISO-8859-1
es_PA	ISO8859-1	es_PA.ISO-8859-1
es_PE	ISO8859-1	es_PE.ISO-8859-1
es_PY	ISO8859-1	es_PY.ISO-8859-1
es_SV	ISO8859-1	es_SV.ISO-8859-1
es_UY	ISO8859-1	es_UY.ISO-8859-1
es_VE	ISO8859-1	es_VE.ISO-8859-1
et	ISO8859-1	et_EE.ISO-8859-1
fi	ISO8859-1	fi_FI.ISO-8859-1

Table 15.3 *Solaris Locale Support (Continued)*

Solaris Locale Name	Encoding	Views Locale Name
fi.ISO8859-15	ISO8859-15	fi_FI.ISO-8859-15
fi.ISO8859-15@euro	ISO8859-15	fi_FI.ISO-8859-15
fr	ISO8859-1	fr_FR.ISO-8859-1
fr.ISO8859-15	ISO8859-15	fr_FR.ISO-8859-15
fr.ISO8859-15@euro	ISO8859-15	fr_FR.ISO-8859-15
fr.UTF-8	UTF-8	fr_FR.UTF-8
fr.UTF-8@euro	UTF-8	fr_FR.UTF-8
fr_BE	ISO8859-1	fr_BE.ISO-8859-1
fr_BE.ISO8859-15	ISO8859-15	fr_BE.ISO-8859-15
fr_BE.ISO8859-15@euro	ISO8859-15	fr_BE.ISO-8859-15
fr_CA	ISO8859-1	fr_CA.ISO-8859-1
fr_CH	ISO8859-1	fr_CH.ISO-8859-1
hr_HR	ISO8859-2	hr_HR.ISO-8859-2
he	ISO8859-8	iw_IL.ISO-8859-8
hu	ISO8859-2	hu_HU.ISO-8859-2
it	ISO8859-1	it_IT.ISO-8859-1
it.ISO8859-15	ISO8859-15	it_IT.ISO-8859-15
it.ISO8859-15@euro	ISO8859-15	it_IT.ISO-8859-15
it.UTF-8	UTF-8	it_IT.UTF-8
it.UTF-8@euro	UTF-8	it_IT.UTF-8
lv	ISO8859-13	lv_LV.ISO-8859-13
lt	ISO8859-13	lt_LT.ISO-8859-13
mk_MK	ISO8859-5	mk_MK.ISO-8859-5
nl	ISO8859-1	nl_NL.ISO-8859-1
nl.ISO8859-15	ISO8859-15	nl_NL.ISO-8859-15

Table 15.3 *Solaris Locale Support (Continued)*

Solaris Locale Name	Encoding	Views Locale Name
nl.ISO8859-15@euro	ISO8859-15	nl_NL.ISO-8859-15
nl_BE	ISO8859-1	nl_BE.ISO-8859-1
nl_BE.ISO8859-15	ISO8859-15	nl_BE.ISO-8859-15
nl_BE.ISO8859-15@euro	ISO8859-15	nl_BE.ISO-8859-15
no	ISO8859-1	no_NO.ISO-8859-1
no_NY	ISO8859-1	no_NY.ISO-8859-1
nr	ISO8859-2	nr_NA.ISO-8859-2
pl	ISO8859-2	pl_PL.ISO-8859-2
pt	ISO8859-1	pt_PT.ISO-8859-1
pt.ISO8859-15	ISO8859-15	pt_PT.ISO-8859-15
pt.ISO8859-15@euro	ISO8859-15	pt_PT.ISO-8859-15
pt_BR	ISO8859-1	pt_BR.ISO-8859-1
ro_RO	ISO8859-2	ro_RO.ISO-8859-2
ru	ISO8859-5	ru_RU.ISO-8859-5
sk_SK	ISO8859-2	sk_SK.ISO-8859-2
sl_SI	ISO8859-2	sl_SI.ISO-8859-2
sq_AL	ISO8859-2	sq_AL.ISO-8859-2
sr_SP	ISO8859-5	sr_SP.ISO-8859-5
sv	ISO-8859-1	sv_SE.ISO-8859-1
sv.ISO8859-15	ISO8859-15	sv_SE.ISO-8859-15
sv.ISO8859-15@euro	ISO8859-15	sv_SE.ISO-8859-15
sv.UTF-8	UTF-8	sv_SE.UTF-8
sv.UTF-8@euro	UTF-8	sv_SE.UTF-8
th_TH	TIS620.2533	th_TH.ISO-8859-11

Table 15.3 *Solaris Locale Support (Continued)*

Solaris Locale Name	Encoding	Views Locale Name
th	TIS620.2533	th_TH.ISO-8859-11
tr	ISO8859-9	tr_TR.ISO-8859-9

Table 15.4 *AIX Locale Support*

Aix Locale Name	Encoding	Views Locale Name
C	ISO8859-1	en_US.US-ASCII
POSIX	ISO8859-1	en_US.ISO-8859-1
ar_AA	ISO8859-6	ar_AA.ISO-8859-6
ar_AA.ISO8859-6	ISO8859-6	ar_AA.ISO-8859-6
Ar_AA		
Ar_AA.IBM-1046		
bg_BG	ISO8859-5	bg_BG.ISO-8859-5
bg_BG.ISO8859-5	ISO8859-5	bg_BG.ISO-8859-5
ca_ES	ISO8859-1	ca_ES.ISO-8859-1
ca_ES.ISO8859-1	ISO8859-1	ca_ES.ISO-8859-1
Ca_ES	IBM-850	ca_ES.IBM850
Ca_ES.IBM-850	IBM-850	ca_ES.IBM850
cs_CZ	ISO8859-2	cs_CZ.ISO-8859-2
cs_CZ.ISO8859-2	ISO8859-2	cs_CZ.ISO-8859-2
da_DK	ISO8859-1	da_DK.ISO-8859-1
da_DK.ISO8859-1	ISO8859-1	da_DK.ISO-8859-1
Da_DK	IBM-850	da_DK.IBM850
Da_DK.IBM-850	IBM-850	da_DK.IBM850
de_CH	ISO8859-1	de_CH.ISO-8859-1
de_CH.ISO8859-1	ISO8859-1	de_CH.ISO-8859-1

Table 15.4 *AIX Locale Support (Continued)*

Aix Locale Name	Encoding	Views Locale Name
De_CH	IBM-850	de_CH.IBM850
De_CH.IBM-850	IBM-850	de_CH.IBM850
de_DE	ISO8859-1	de_DE.ISO-8859-1
de_DE.ISO8859-1	ISO8859-1	de_DE.ISO-8859-1
De_DE	IBM-850	de_DE.IBM850
De_DE.IBM-850	IBM-850	de_DE.IBM850
el_GR	ISO8859-7	el_GR.ISO-8859-7
el_GR.ISO8859-7	ISO8859-7	el_GR.ISO-8859-7
en_GB	ISO8859-1	en_GB.ISO-8859-1
en_GB.ISO8859-1	ISO8859-1	en_GB.ISO-8859-1
En_GB	IBM-850	en_GB.IBM850
En_GB.IBM-850	IBM-850	en_GB.IBM850
en_US	ISO8859-1	en_US.ISO-8859-1
en_US.ISO8859-1	ISO8859-1	en_US.ISO-8859-1
En_US	IBM-850	en_US.IBM850
En_US.IBM-850	IBM-850	en_US.IBM850
es_ES	ISO8859-1	es_ES.ISO-8859-1
es_ES.ISO8859-1	ISO8859-1	es_ES.ISO-8859-1
Es_ES	IBM-850	es_ES.IBM850
Es_ES.IBM-850	IBM-850	es_ES.IBM850
Et_EE		
Et_EE.IBM-922		
ET_EE	UTF-8	et_EE.UTF-8
ET_EE.UTF-8	UTF-8	et_EE.UTF-8
fi_FI	ISO8859-1	fi_FI.ISO-8859-1

Table 15.4 *AIX Locale Support (Continued)*

Aix Locale Name	Encoding	Views Locale Name
fi_FI.ISO8859-1	ISO8859-1	fi_FI.ISO-8859-1
Fi_FI	IBM-850	fi_FI.IBM850
Fi_FI.IBM-850	IBM-850	fi_FI.IBM850
fr_BE	ISO8859-1	fr_BE.ISO-8859-1
fr_BE.ISO8859-1	ISO8859-1	fr_BE.ISO-8859-1
Fr_BE	IBM-850	fr_BE.IBM850
Fr_BE.IBM-850	IBM-850	fr_BE.IBM850
fr_CA	ISO8859-1	fr_CA.ISO-8859-1
fr_CA.ISO8859-1	ISO8859-1	fr_CA.ISO-8859-1
Fr_CA	IBM-850	fr_CA.IBM850
Fr_CA.IBM-850	IBM-850	fr_CA.IBM850
fr_CH	SO8859-1	fr_CH.ISO-8859-1
fr_CH.ISO8859-1	ISO8859-1	fr_CH.ISO-8859-1
Fr_CH	IBM-850	fr_CH.IBM850
Fr_CH.IBM-850	IBM-850	fr_CH.IBM850
fr_FR	ISO8859-1	fr_FR.ISO-8859-1
fr_FR.ISO8859-1	ISO8859-1	fr_FR.ISO-8859-1
Fr_FR	IBM-850	fr_FR.IBM850
Fr_FR.IBM-850	IBM-850	fr_FR.IBM850
hr_HR	ISO8859-2	hr_HR.ISO-8859-2
hr_HR.ISO8859-2	ISO8859-2	hr_HR.ISO-8859-2
hu_HU	ISO8859-2	hu_HU.ISO-8859-2
hu_HU.ISO8859-2	ISO8859-2	hu_HU.ISO-8859-2
is_IS	ISO8859-1	is_IS.ISO-8859-1
is_IS.ISO8859-1	ISO8859-1	is_IS.ISO-8859-1

Table 15.4 *AIX Locale Support (Continued)*

Aix Locale Name	Encoding	Views Locale Name
Is_IS	IBM-850	is_IS.IBM850
Is_IS.IBM-850	IBM-850	is_IS.IBM850
it_IT	ISO8859-1	it_IT.ISO-8859-1
it_IT.ISO8859-1	ISO8859-1	it_IT.ISO-8859-1
It_IT	IBM-850	it_IT.IBM850
It_IT.IBM-850	IBM-850	it_IT.IBM850
iw_IL	ISO8859-8	iw_IL.ISO-8859-8
iw_IL.ISO8859-8	ISO8859-8	iw_IL.ISO-8859-8
Iw_IL		
Iw_IL.IBM-856		
ja_JP	IBM-eucJP	ja_JP.EUC-JP
ja_JP.IBM-eucJP	IBM-eucJP	ja_JP.EUC-JP
Ja_JP	IBM-932	ja_JP.Shift_JIS
Ja_JP.IBM-932	IBM-932	ja_JP.Shift_JIS
Jp_JP.pc932		
Jp_JP		
ko_KR	IBM-eucKR	ko_KR.EUC-KR
ko_KR.IBM-eucKR	IBM-eucKR	ko_KR.EUC-KR
Lt_LT		
Lt_LT.IBM-921		
LT_LT	UTF-8	lt_LT.UTF-8
LT_LT.UTF-8	UTF-8	lt_LT.UTF-8
Lv_LV		
Lv_LV.IBM-921		
LV_LV	UTF-8	lv_LV.UTF-8

Table 15.4 *AIX Locale Support (Continued)*

Aix Locale Name	Encoding	Views Locale Name
LV_LV.UTF-8	UTF-8	lv_LV.UTF-8
mk_MK	ISO8859-5	mk_MK.ISO-8859-5
mk_MK.ISO8859-5	ISO8859-5	mk_MK.ISO-8859-5
nl_BE	ISO8859-1	nl_BE.ISO-8859-1
nl_BE.ISO8859-1	ISO8859-1	nl_BE.ISO-8859-1
Nl_BE	IBM-850	nl_BE.IBM850
Nl_BE.IBM-850	IBM-850	nl_BE.IBM850
nl_NL	ISO8859-1	nl_NL.ISO-8859-1
nl_NL.ISO8859-1	ISO8859-1	nl_NL.ISO-8859-1
Nl_NL	IBM-850	nl_NL.IBM850
Nl_NL.IBM-850	IBM-850	nl_NL.IBM850
no_NO	ISO8859-1	no_NO.ISO-8859-1
no_NO.ISO8859-1	ISO8859-1	no_NO.ISO-8859-1
No_NO	IBM-850	no_NO.IBM850
No_NO.IBM-850	IBM-850	no_NO.IBM850
pl_PL	ISO8859-2	pl_PL.ISO-8859-2
pl_PL.ISO8859-2	ISO8859-2	pl_PL.ISO-8859-2
pt_BR	ISO8859-1	pt_BR.ISO-8859-1
pt_BR.ISO8859-1	ISO8859-1	pt_BR.ISO-8859-1
pt_PT	ISO8859-1	pt_PT.ISO-8859-1
pt_PT.ISO8859-1	ISO8859-1	pt_PT.ISO-8859-1
Pt_PT	IBM-850	pt_PT.IBM850
Pt_PT.IBM-850	IBM-850	pt_PT.IBM850
ro_RO	ISO8859-2	ro_RO.ISO-8859-2
ro_RO.ISO8859-2	ISO8859-2	ro_RO.ISO-8859-2

Table 15.4 *AIX Locale Support (Continued)*

Aix Locale Name	Encoding	Views Locale Name
ru_RU	ISO8859-5	ru_RU.ISO-8859-5
ru_RU.ISO8859-5	ISO8859-5	ru_RU.ISO-8859-5
sh_SP	ISO8859-2	sh_SP.ISO-8859-2
sh_SP.ISO8859-2	ISO8859-2	sh_SP.ISO-8859-2
sk_SK	ISO8859-2	sk_SK.ISO-8859-2
sk_SK.ISO8859-2	ISO8859-2	sk_SK.ISO-8859-2
sl_SI	ISO8859-2	sl_SI.ISO-8859-2
sl_SI.ISO8859-2	ISO8859-2	sl_SI.ISO-8859-2
sq_AL	ISO8859-1	sq_AL.ISO-8859-1
sq_AL.ISO8859-1	ISO8859-1	sq_AL.ISO-8859-1
sr_SP	ISO8859-5	sr_SP.ISO-8859-5
sr_SP.ISO8859-5	ISO8859-5	sr_SP.ISO-8859-5
sv_SE	ISO8859-1	sv_SE.ISO-8859-1
sv_SE.ISO8859-1	ISO8859-1	sv_SE.ISO-8859-1
Sv_SE	IBM-850	sv_SE.IBM850
Sv_SE.IBM-850	IBM-850	sv_SE.IBM850
tr_TR	ISO8859-9	tr_TR.ISO-8859-9
tr_TR.ISO8859-9	ISO8859-9	tr_TR.ISO-8859-9
zh_CN	IBM-eucCN	zh_CN.GB2312
zh_CN.IBM-eucCN	IBM-eucCN	zh_CN.GB2312
ZH_CN	UTF-8	zh_CN.UTF-8
ZH_CN.UTF-8	UTF-8	zh_CN.UTF-8
zh_TW	IBM-eucTW	zh_TW.EUC-TW
zh_TW.IBM-eucTW	IBM-eucTW	zh_TW.EUC-TW

Table 15.4 *AIX Locale Support (Continued)*

Aix Locale Name	Encoding	Views Locale Name
Zh_TW	big5	zh_TW.Big5
Zh_TW.big5	big5	zh_TW.Big5

Table 15.5 *OSF Locale Support*

Osf Locale Name	Encoding	Views Locale Name
C	ISO8859-1	en_US.US-ASCII
POSIX	ISO8859-1	en_US.ISO-8859-1
da_DK.ISO8859-1	ISO8859-1	da_DK.ISO-8859-1
de_CH.ISO8859-1	ISO8859-1	de_CH.ISO-8859-1
de_DE.ISO8859-1	ISO8859-1	de_DE.ISO-8859-1
el_GR.ISO8859-7	ISO8859-7	el_GR.ISO-8859-7
en_GB.ISO8859-1	ISO8859-1	en_GB.ISO-8859-1
en_US.ISO8859-1	ISO8859-1	en_US.ISO-8859-1
en_US.cp850	cp850	en_US.IBM850
es_ES.ISO8859-1	ISO8859-1	es_ES.ISO-8859-1
fi_FI.ISO8859-1	ISO8859-1	fi_FI.ISO-8859-1
fr_BE.ISO8859-1	ISO8859-1	fr_BE.ISO-8859-1
fr_CA.ISO8859-1	ISO8859-1	fr_CA.ISO-8859-1
fr_CH.ISO8859-1	ISO8859-1	fr_CH.ISO-8859-1
fr_FR.ISO8859-1	ISO8859-1	fr_FR.ISO-8859-1
is_IS.ISO8859-1	ISO8859-1	is_IS.ISO-8859-1
it_IT.ISO8859-1	ISO8859-1	it_IT.ISO-8859-1
nl_BE.ISO8859-1	ISO8859-1	nl_BE.ISO-8859-1
nl_NL.ISO8859-1	ISO8859-1	nl_NL.ISO-8859-1

Table 15.5 *OSF Locale Support (Continued)*

Osf Locale Name	Encoding	Views Locale Name
no_NO.ISO8859-1	ISO8859-1	no_NO.ISO-8859-1
pt_PT.ISO8859-1	ISO8859-1	pt_PT.ISO-8859-1
sv_SE.ISO8859-1	ISO8859-1	sv_SE.ISO-8859-1
tr_TR.ISO8859-9	ISO8859-9	tr_TR.ISO-8859-9

A

Packaging IBM ILOG Views Applications

This section explains how to use `ilv2data`, a tool provided with IBM® ILOG® Views to safely package application data files together with your IBM ILOG Views application in the same executable.

What is ilv2data?

The `ilv2data` executable file allows you to put all the application resources, such as `.ilv` and .`dbm` files and bitmaps (`.gif`, `.bmp`, `.pbm`, and so on), in a file generated by `ilv2data` that you will then add to an application project (on PCs) or compile and link with your IBM ILOG Views application (on UNIX).

This file is a *data resource file* on Microsoft Windows (`.rc`) that can be compiled with the Microsoft Resource Compler (`RC.EXE`). On UNIX platforms, this file is a regular C++ source file that contains only the definition of static data. This file can be compiled with your regular C++ compiler. We call this file a *resource file* in the rest of this section.

A resource file stores a set of data blocks that can be retrieved at run time using the name with which they were associated when building the resource file.

In This Section

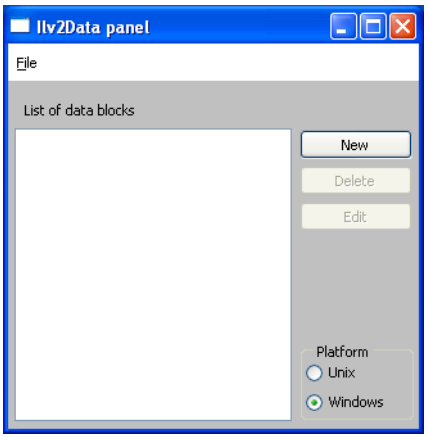
- *Adding a Resource File to a UNIX Library*
- *Adding a Resource File to a Windows DLL*

Launching ilv2data

To launch `ilv2data`:

1. Go to the directory `<ILVHOME>/bin/<system>`.
2. Compile the executable, if it is not already there (note that `ilv2data` uses the Gadgets package).
3. Launch the executable by typing `ilv2data`.

   The following panel appears:



The ilv2data Panel

The `ilv2data` panel is composed of the following elements:

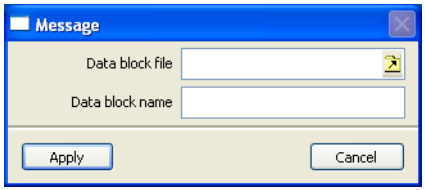
- A File menu that is used to handle resource files. A resource file is the file where you will put all the resources you want to package together with your application. Once completed and validated, this file will be saved as a `.rc` file or as a `.cpp` file depending on the platform you have selected (Microsoft Windows or UNIX). The **File** menu has the following menu items:
  - New—Creates a new resource file.

- Open—Opens a resource file.
- Save—Saves the data in the resource file and generates a `.rc` or a `.cpp` file (depending on the selected platform).

◆ Three buttons:

- New—Adds a data block to the list.
- Delete—Removes a data block from the list.
- Edit—Modifies the values associated with the selected item in the list.

When you click New or Edit, the following dialog box is displayed:



The Data block file entry field is where you type the physical name of the resource file you want to add to the list. It you want to use a file browser to locate the file, click the icon to the right of the entry field to display a file chooser.

The Data block name field is set by default to the logical name that the program uses to read the data block. Initially, this name is the same as the one you entered in the Data block file entry field.

The Apply button validates the data, and the Cancel button cancels the procedure.

> ***Note:*** *If you add a Views (or extension Views like Data Access) data file like a* `.dbm` *file, you must not forget the path of the file from* `$ILVHOME/data`*. For example, il you want to add the* `dataccess.dbm` *file, the Data black name must be* `dataccess/dataccess.dbm` *because the full filename of* `dataccess.dbm` *is* `$ILVHOME/data/dataccess/dataccess.dbm`.

Launching ilv2data with a Batch Command

You can launch `ilv2data` via a command line in which you can specify a number of options to perform various basic operations.

The available options, along with their description, are given below:

```
ilv2data [-a key[=val]] [-c] [-d key] [-h] [-i dir] [-l]
```

```
[-m key[=val]] [-u|w] [-v 0|1] file
```

- `-a key[=val]` : Add option

  Adds the data block name `key` to the list of resources. `val` specifies the file to be inserted. The default value is `key`.

- `-c` : Check option

  Checks the consistency of `file`.

- `-d key` : Delete option

  Deletes the data block key from the list of resources.

- `-h` : Help option

  Displays the usage of the command.

- `-i dir` : Include option

  Adds the directory `dir` to the list of paths where data block files are searched for.

- `-l` : List option

  Lists all data blocks available in `file`.

- `-m key[=val]` : Modify option

  Refreshes the data block `key` with the file `val`.

- `-u|w` : Regenerate option

  Regenerates the file `file` in UNIX mode (`-u`) or in Windows mode (`-w`). To find a file in the display path, use the name of the data block. Use this file if it exists, otherwise use the old definition of the data block contained in `file`.

- `-v 0|1` : Verbose option

  Prints comments during execution if the option is set to 1. Errors and warnings are displayed even when the option is set to 0.

This command returns `0` if execution succeeds and `1` if it fails.

Adding a Resource File to a UNIX Library

To add a resource file to a UNIX library, add the following two lines to a module in the library that you know will be called by your final application (such as, the library initialization module):

```
extern IlUInt IL_MODINIT(<name>Resources)();
static IlUInt forceRes = IL_MODINIT(<name>Resources)();
```

`ilv2data` generates a file name with the following format: `<name>.cpp`.

Adding a Resource File to a Windows DLL

You must add the following lines to any dll module:

```
#include <windows.h>
#include <ilviews/macros.h>

extern "C" {
    void _declspec(dllimport) IlvAddHandleToResPath(long, int);
    void _declspec(dllimport) IlvRemoveHandleFromResPath(long);
}

BOOL WINAPI
DllEntryPoint(HINSTANCE instance, DWORD reason, LPVOID)
{
    switch (reason) {
    case DLL_PROCESS_ATTACH:
        IlvAddHandleToResPath((long)instance, -1);
        return 1;
    case DLL_PROCESS_DETACH:
        IlvRemoveHandleFromResPath((long)instance);
        return 0;
    }
    return 0;
}
BOOL WINAPI
DllMain(HINSTANCE hinstance, DWORD reason, LPVOID reserved)
{
    return DllEntryPoint(hinstance, reason, reserved);
}
```

B

Using IBM ILOG Views on Microsoft Windows

This section is aimed at programmers who develop their applications on Microsoft Windows or want to merge IBM® ILOG® Views and Windows code. It gives information on:

- *Creating a New IBM ILOG Views Application on Microsoft Windows*
- *Incorporating Windows Code into an IBM ILOG Views Application*
- *Integrating IBM ILOG Views Code into a Windows Application*
- *Exiting an Application Running on Microsoft Windows*
- *Windows-specific Devices*
- *Using GDI+ Features with IBM ILOG Views*
- *Using Multiple Display Monitors with IBM ILOG Views*

Creating a New IBM ILOG Views Application on Microsoft Windows

To create a new IBM® ILOG® Views application that does not contain any Windows code, all you have to do is create the `main` function and instantiate the `IlvDisplay` class by providing the application name to its constructor:

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

Note that 'main' is not the regular entry point of an application running on Microsoft Windows (it should be 'WinMain'). Because of the source code portability that IBM ILOG Views provides, and for easier command line parameter parsing, we choose to use the regular C++ 'main' entry point. The impact of this choice is discussed further in this topic.

The application name is used for resource scanning (see *Display System Resources*). The second argument is not used on Microsoft Windows, and therefore it is replaced by an empty string. (It is used on X Window where it corresponds to an X display.) The last two parameters are also not used on Microsoft Windows.

Then you can build your view structure and objects and call the global function `IlvMainLoop`.

```
int
main(int argc, char* argv[])
{
    IlvDisplay* display = new IlvDisplay("IlogViews", "", argc, argv);
    ...
    ...
    IlvMainLoop();
    return 0;
}
```

Here, since a `main` function is provided instead of the `WinMain` entry point that Microsoft Windows expects to start an application, you have to link your object files with the `ILVMAIN.OBJ` file. This file, supplied with IBM ILOG Views, defines a default `WinMain` function that does all the necessary initialization operations and calls the `main` function.

> ***Note:*** *To avoid conflicts with other definitions of the* `main` *function, which might be provided by some compilers, a preprocessor macro redefines the* `main` *function as* `IlvMain`*. This macro is declared in the header file* `<ilviews/ilv.h>`*.*

For examples, look at the `make` or `project` files in the `BIN` directory.

Incorporating Windows Code into an IBM ILOG Views Application

You can easily incorporate into your IBM® ILOG® Views application Windows menus and panels that were created with one of the numerous interface generators that Microsoft Windows supports. Examples can be found in the subdirectory `foundation\windows`,

which is located under <ILVHOME>\samples. Refer also to the Views Foundation Tutorials in the online documentation.

The following example displays a panel that was created by any interface builder, and linked with your application with the resource compiler.

```
#define VIEW_ID 1010 // The ID of a sub-window in the panel
int PASCAL ILVEXPORTED
DialogProc(HWND dlg, UINT msg, WPARAM wParam, LPARAM lParam)
{
  switch (msg) {
  case WM_INITDIALOG:
     // Create some IlogViews object in the dialog.
     InitIlogViews((IlvDisplay*)lParam, GetDlgItem(dlg, VIEW_ID));
     return 1;
  case WM_COMMAND:
     if (wParam == QUIT_ID) {
         EndDialog(dlg, 1);    // Close the dialog
         ReleaseIlogViews();   // Delete IlogViews objects
         PostQuitMessage(0);   // Exit the event loop
         return 1;
     }
  }
  return 0;
}


int
main(int argc, char* argv[])
{
     // Connect to the windowing system.
     IlvDisplay* display = new IlvDisplay("IlogViews", "", argc, argv);
     if (display->isBad()) {
         IlvFatalError("Couldn't connect to display system");
         delete display;
         return 1;
     }
     // Create the dialog box.
     if (DialogBoxParam(display->getInstance(), "MY_PANEL", 0,
                        (FARPROC)DialogProc, (long)display) == -1)
        IlvFatalError("Couldn't create dialog");
     delete display;
     return 1;
}

void
InitIlogViews(IlvDisplay* display, HWND wnd)
{
    // For example: a container that uses the 'wnd' window.
    container = new IlvContainer(display, wnd);
    ...
}void
ReleaseIlogViews()
{
    delete container;
}
```

In the `InitIlogViews` member function, a new `IlvContainer` object holding an existing Windows panel, `wnd`, is created. In your user interface generator, you must specify that `IlogViewsWndClass` is the WindowsClass to be used for that window.

In this example, since a `main` function is provided instead of the `WinMain` entry point that Microsoft Windows expects to start an application, you have to link your object files with the `ILVMAIN.OBJ` file. This file, supplied with IBM ILOG Views, defines a default `WinMain` function that does all the necessary initialization operations and calls the `main` function.

Integrating IBM ILOG Views Code into a Windows Application

To integrate IBM® ILOG® Views code into an existing application running on Microsoft Windows, all you have to do is use the second constructor of the `IlvDisplay` class, which takes an instance of your application as its argument:

```
int PASCAL
WinMain(HANDLE appInstance, HANDLE, LPSTR, int)
{
    ...
    IlvDisplay* display = new IlvDisplay((IlAny)appInstance,
                                         "ApplicationName");
    ...
}
```

Note, however, that deleting the `IlvDisplay` object does not post a `QUIT` message. This is done in order not to exit the event loop, as you might want to do some more work after the IBM ILOG Views session is closed.

Here, because you provide a `WinMain` entry point to your application, you do not have to link your executable file using the `ILVMAIN.OBJ` file.

Exiting an Application Running on Microsoft Windows

Releasing the memory and the system resources before exiting an application is a good practice on all operating systems. On early versions of Microsoft Windows (3.1, 95), it was critical; the system had a very limited number of GDI resources (colors, fonts, and so on) and they were not automatically released. Later versions (NT 4, 2000, XP) have improved this behaviour. However, it is still highly recommanded to provide a clean way to quit an application, freeing the memory and releasing system resources before exiting. A convenient way to do so is to write a function that frees the application data, deletes the `IlvDisplay` and `call IlvExit(0)`. This function may then be used as an accelerator, a button callback, a top window destroy callback, or the like.

> ***Note:*** *All display instances must be deleted, as well as all managers. Remember that containers and managers delete the objects they store when they are destroyed. For information on managers, see the IBM ILOG Views Managers documentation.*

> ***Note:*** *IBM ILOG Views uses internal memory that is allocated dynamically. This memory is freed only when the application exits.*

Windows-specific Devices

In order to manage Windows devices (such as the printer or the metafile generation), IBM® ILOG® Views provides two classes: `IlvWindowsVirtualDevice` and `IlvWindowsDevice`.

Printing

You can use the `IlvWindowsDevice` dump device to print your IBM ILOG Views output to any printer controlled by Microsoft Windows.

Selecting a Printer

You can select a printer by calling the following global function:

```
const char*
IlvGetWindowsPrinter (Ilboolean dialog = IlTrue);
```

This function returns a string that describes which printer is about to be used. That string is internally managed and must not be modified nor deleted.

When called with an `IlTrue` value for the dialog parameter, a dialog box is displayed that lets the user specify which printer to use and what size and orientation parameters should be applied. If this function is called with an `IlFalse` parameter, a string that describes the current default printer is returned. If there is an error, or if the user clicks the Cancel button, NULL is returned.

Using GDI+ Features with IBM ILOG Views

What is GDI+

GDI+ is a way of drawing on Microsoft Windows platforms. It comes with interesting features such as transparency and anti-aliasing. For more information about GDI+, take a look at the Microsoft internet site.

Using Dynamic Link Libraries

When using the dynamic IBM ILOG Views libraries (`dll_mda`), using GDI+ is very simple: Microsoft provides a DLL (`gdiplus.dll`) that must be accessible by the IBM ILOG Views application. This DLL is shipped in the same directory as the dynamic IBM ILOG Views libraries (`dll_mda`). To download the latest `gdiplus.dll` redistributable, go to `http://www.microsoft.com/msdownload/platformsdk/sdkupdate/psdkredist.htm`.

Using Static Libraries

When using the static IBM ILOG Views libraries (`stat_mda`, `stat_mta`), you need to install the Microsoft Platform SDK, because you must link your application with the `gdiplus.lib` library. To get this SDK, go to `http://www.microsoft.com/msdownload/platformsdk/sdkupdate`.

You must also include the <`ilviews/windows/ilvgdiplus.h`> file when compiling, and link your application with the `ilvgdiplus.lib` library. This library can be found in the directory `ILVHOME/lib/[platform]/[subplatform]`, where `ILVHOME` is the root directory in which IBM ILOG Views was installed, and where *`subplatform`* is `stat_mda or stat_mta`, and where platform is one of the following:

- `x86_.net2003_7.1`
- `x86_.net2005_8.0`
- `x86_.net2008_9.0`

GDI+ and IBM ILOG Views

When GDI+ is installed, IBM ILOG Views provides its benefits by proposing a dedicated API to the `IlvPalette` and `IlvPort` classes. The following methods have been added to handle transparency and anti-aliasing:

- `IlvPalette::setAlpha`
- `IlvPalette::getAlpha`
- `IlvPort::setAlpha`
- `IlvPort::getAlpha`
- `IlvPalette::setAntialiasingMode`

- `IlvPalette::getAntialiasingMode`
- `IlvPort::setAntialiasingMode`
- `IlvPort::getAntialiasingMode`

See also the sections *Alpha Value* on page 83 and *Anti-Aliasing Mode* on page 84.

Controlling GDI+ Features at Run Time

It is possible to specify through resources whether you want GDI+ to be used or not. The following table summarizes the different resources, their possible values, and the effect of each value:

Table B.1 *GDI+ Resources*

Resource (.ini file)	Environment Variable	Value
`UseGdiPlus`	`ILVUSEGDIPLUS`	**needed**: GDI+ is used only when it is needed, for example, when transparency or anti-aliasing is required. This is the default. **true**: GDI+ is used each time it is possible. **false**: GDI+ is never used.
`Antialiasing`	`ILVANTIALIASING`	**false**: The anti-aliasing mode of the display is set to `IlvNoAntialiasingMode`. This is the default. **true**: The anti-aliasing mode of the display is set to `IlvUseAntialiasingMode`.

For example, the following `views.ini` file enables anti-aliasing for the entire application. As the `UseGdiPlus` resource is not specified, the default is used, that is, GDI+ will be used only when needed.

```
[IlogViews]

Antialiasing=true
```

Limitations

The following table summarizes the limitations and unsupported features of the use of GDI+ through IBM ILOG Views:

Table B.2 *GDI+ Limitations with IBM ILOG Views*

Fonts	GDI+ supports only True Type Fonts. When using a font that is not a True Type Font, GDI+ will not be used. Furthermore, when drawing strings with a transformer, to draw a vertical string for example, the result may not exactly match the rendering done with GDI. This is because the rendering engines used in GDI+ and GDI are not exactly the same.
Brushes	The Windows HATCHED patterns are not exactly mapped to the GDI+ HatchStyle. Thus, switching to GDI+ may change the drawing of the pattern slightly.
Printing	GDI+ is not used by IBM ILOG Views when drawing on a printer.
Arcs	Flat arcs are badly drawn by GDI+. Thus, GDI+ will not be used to draw flat arcs.
Draw Mode	GDI+ does not support other modes than `IlvModeSet`. For example, when using `IlvModeXor`, GDI+ will not be used.
Windows XP Look and Feel	Transparency and anti-aliasing are not available with gadgets drawn using Windows XP.

Using Multiple Display Monitors with IBM ILOG Views

Multiple display monitors provides a set of features that allow an application to make use of multiple display devices at the same time. Several monitors can be seen as one big monitor, making it possible to move windows from one screen to the other one.

IBM® ILOG® Views has taken into account this feature, and the following API has been added to retrieve the coordinates of a monitor: `IlvDisplay::screenBBox`. This method allows an application to retrieve the monitor coordinates in which a specific rectangle is located. For example, it can be used to center a window inside a single monitor. See the Reference Manual for more details.

The impact of this feature on existing applications is restricted to the management of top windows: Each time a top window is displayed, its position must be computed carefully. To avoid problems, top windows should be relative to other top windows and not to the screen. For example, most applications have a main panel and several dialogs, all being top windows. It is better to specify the dialog location relative to the main panel position (using the `IlvView::moveToView` method) than to center the dialogs into the whole screen (using the `IlvView::ensureInScreen` method).

> ***Note:*** *The method* `IlvView::ensureInScreen` *places a view inside a monitor. It considers the monitor in which the view is located as the working monitor. For example, if a view is located in monitor 2, calling* `IlvView::ensureInScreen` *on the view will leave the view in monitor 2.*

C

Using IBM ILOG Views on X Window Systems

This appendix provides information about using IBM® ILOG® Views on X Window systems in the UNIX environment.

- *Libraries* describes the two versions of IBM ILOG Views, based on Xlib or Motif.
- The capability of *Adding New Sources of Input*
- Performing *ONC-RPC Integration*
- *Integrating IBM ILOG Views with a Motif Application Using libmviews*
- *Integrating IBM ILOG Views with an X Application Using libxviews*

Libraries

IBM® ILOG® Views libraries are delivered in two versions:

- `libxviews`, which is based on Xlib.
- `libmviews`, which is based on Motif.

When developing an IBM ILOG Views application, you can create at link time either a pure Xlib application or an application that will be easier to integrate with Motif. Depending on

what kind of application you want to obtain, you will link your files either with `libxviews` for a pure Xlib application or with `libmviews` for a Motif-based application. Your source code is independent of the library you choose to link with.

For details on using these libraries, see:

- *Using the Xlib Version, libxviews*
- *Using the Motif Version, libmviews*

Using the Xlib Version, libxviews

Creating the `IlvDisplay` object establishes a regular connection with the display system. From the X Window point of view, the `IlvSystemView` type provided to `IlvDisplay` is equivalent to the `Window` type. The event loop management is based on a call to `select` on the file descriptor corresponding to the connection to the display system. You link with the `libxviews` and `libX11` libraries.

Restrictions

In early IBM ILOG Views releases, when gadgets were not yet available, some basic portable GUI components, mainly standard dialogs, were implemented using Motif on UNIX and Microsoft SDK on Windows. These features, though replaced by more recent equivalent components in IBM ILOG Views, have been kept for backward compatibility. They are implemented in `libmviews` but not in `libxviews`, which is not based on Motif. These are:

- Standard system dialogs: `IlvPromptDialog`, `IlvInformationDialog`, `IlvQuestionDialog`, `IlvFileSelector` and `IlvPromptStringsDialog`.

  These classes are declared in the header file `ilviews/dialogs.h`. The Gadgets library `libilvgadgt` provides portable versions (in pure IBM ILOG Views code) of similar dialogs. See the `ilviews/stdialog.h` header file.

- The `IlvScrollView`, based on Motif `XmScrolledWindow`.

  The classes `IlvScrolledView` and `IlvScrolledGadget` offer similar services.

Using the Motif Version, libmviews

Creating the `IlvDisplay` object initializes the Xt library and creates a top shell widget. The values returned by the member functions `IlvAbstractView::getSystemView` or `IlvAbstractView::getShellSystemView` are actual Motif widgets. The event loop management is strictly equivalent to a call to `XtAppMainLoop`. You must have Motif installed on your platform and you must link with the `libmviews` library and with the `libXm`, `libXt` and `libX11` libraries.

These differences are discussed in detail in the rest of this appendix, and examples on how to use one or the other mode are provided in the distribution.

Important restriction:

The use of `libmviews` is deprecated in shared library format. Since version 4.0, all shared libraries provided by IBM ILOG Views are built using `libxviews` and are incompatible with `libmviews`.

`libmviews` can only be used with the static version of other IBM ILOG Views libraries

Adding New Sources of Input

IBM® ILOG® Views allows the application to add file descriptors as new sources of input. See the member functions `IlvEventLoop::addInput` and `IlvEventLoop::addOutput` for details.

ONC-RPC Integration

You can use the BSD sockets or the ONC-RPC with IBM® ILOG® Views once you have access to the `XtAppAddInput` functions.

For more information about ONC-RPC, you can take a look at the *Sun Network* documentation or the equivalent on your system.

Integrating IBM ILOG Views with a Motif Application Using libmviews

IBM® ILOG® Views was designed to be easily integrated with existing Motif applications. The library `libmviews` provides a way to connect an `IlvView` with an existing Motif widget and the mechanism required to respond to user action.

In the following sections, you will find information on:

- *Initializing Your Application*
- *Retrieving Connection Information*
- *Using an Existing Widget*
- *Running the Main Loop*
- *Sample Program Using Motif and IBM ILOG Views*

Initializing Your Application

When integrating IBM ILOG Views code with a Motif-based application, you can create an IBM ILOG Views session in two different ways: you can use either the standard

IBM ILOG Views initialization procedure or use the initialization block of the Motif application and call the second constructor of the `IlvDisplay` class, as shown below:

Standard IBM ILOG Views Initialization Procedure

```
IlvDisplay* display = new IlvDisplay("Program", "", argc, argv);
```

Here, IBM ILOG Views establishes the connection with the display system.

Motif Application Initialization Procedure

```
Widget top = XtInitialize("", "Program", NULL, NULL, (Cardinal*)&argc, argv);
if (!top) {
    IlvFatalError("Couldn't open display");
    exit(1);
}
IlvDisplay* display = new IlvDisplay(XtDisplay(top), "X");
```

Here, standard Xt function calls initialize the connection. You have to specify the application name in the constructor of `IlvDisplay` to be able to find display resources from this string.

Retrieving Connection Information

You can access the topmost shell created by IBM ILOG Views by calling the member function `topShell` of the `IlvDisplay` class. The returned value must be converted to a `Widget`.

The Xt application context is returned by the function `IlvApplicationContext`. The returned value must be converted to an `XtAppContext`.

To get all the information about a connection to an X Window application, you must use the following functions:

```
XtAppContext appContext = (XtAppContext)IlvApplicationContext();
Widget topLevel = (Widget)display->topShell();
```

Before using the `IlvApplicationContext` function, add the following to the application code:

```
extern XtAppContext IlvApplicationContext();
```

To use the `XtAppContext` object, refer to the Xt documentation.

Using an Existing Widget

Most of the classes that inherit from the `IlvView` class define a constructor that specifies an existing widget to be used, instead of your having to create one. Here is how to create an `IlvView` object from a widget:

```
IlvDisplay* display = ... // display initialization
// Create a DrawingArea widget
Widget drawingArea =
    XtVaCreateManagedWidget("ilvview",
                            xmDrawingAreaWidgetClass, parent,
                            XmNwidth, 100,
                            XmNheight, 100,
                            0);
// Realize this widget
XtRealizeWidget(drawingArea);
// Create a IlvView object from this widget.
IlvView* aView = new IlvView(display, drawingArea);
```

The only restriction is that the widget you use must already be *realized* (in the Xt terminology, that is, the widget must have a window) before you call the constructor of `IlvView`.

Running the Main Loop

In `libmviews`, the function `IlvMainLoop` does exactly the same job as `XtAppMainLoop`. You can use the one you want, but `IlvMainLoop` is provided to make the code portable between different platforms.

We recommend that you clearly separate IBM ILOG Views code from Motif code if you plan to port your applications to other platforms.

Sample Program Using Motif and IBM ILOG Views

The following sample program is a full example of how to integrate IBM ILOG Views code into a Motif application (`samples/foundation/xlib/src/ilvmotif.cpp`):

```
// ------------------------------------------------------------- -*- C++ -*-
// IBM ILOG Views samples source file
// File: samples/foundation/xlib/src/ilvmotif.cpp
// --------------------------------------------------------------------------
// Using the grapher in a Motif widget
// --------------------------------------------------------------------------
#include <ilviews/contain/contain.h>
#include <ilviews/graphics/all.h>
#include <stdio.h>
#include <stdlib.h>

// --------------------------------------------------------------------------
// Integration Part with Motif
// --------------------------------------------------------------------------
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Xlib.h>
#include <X11/Shell.h>
#include <Xm/Xm.h>
#include <Xm/DrawingA.h>
#include <Xm/PushB.h>
```

```
// --------------------------------------------------------------------------
static void
Quit(Widget, XtPointer display, XtPointer)
{
    delete (IlvDisplay*)display;
    exit(0);
}

#define INPUT_MASK (unsigned long)(ButtonPressMask | ButtonReleaseMask |\
   KeyPressMask | KeyReleaseMask |\
   ButtonMotionMask | EnterWindowMask |\
   LeaveWindowMask | PointerMotionMask | \
   ExposureMask | StructureNotifyMask)

extern "C" void IlvDispatchEvent(XEvent* xevent);

static void
ManageInput(Widget, XtPointer, XEvent* xevent, Boolean*)
{
    IlvDispatchEvent(xevent);
}

// --------------------------------------------------------------------------
IlvDisplay*
IlvGetDisplay(Display* xdisplay)
{
    static IlvDisplay* ilv_display = 0;
    if (!ilv_display)
       ilv_display = new IlvDisplay(xdisplay, "IlvMotif");
    return ilv_display;
}

// --------------------------------------------------------------------------
IlvContainer*
CreateContainer(Widget widget)
{
    IlvContainer* c = new IlvContainer(IlvGetDisplay(XtDisplay(widget)),
       (IlvSystemView)XtWindow(widget));
    XtAddEventHandler(widget, INPUT_MASK, False,
       ManageInput, (XtPointer)c);
    return c;
}

// --------------------------------------------------------------------------
int
main(int argc, char* argv[])
{
    Widget toplevel = XtInitialize("", "IlvMotif", NULL, 0,
                                   &argc, argv);
    if (!toplevel)
        exit(1);
    Widget drawArea = XtVaCreateManagedWidget("ilvview",
      xmDrawingAreaWidgetClass,
      (Widget)toplevel,
      XtNwidth,  400,
      XtNheight, 400,
      (IlAny) 0);
    Widget pushb = XtVaCreateManagedWidget("Quit",
```

```
                    xmPushButtonWidgetClass,
                    drawArea,
                    (IlAny) 0);
    XtRealizeWidget(toplevel);
    IlvContainer* container = CreateContainer(drawArea);
     XtAddCallback(pushb, XmNactivateCallback, Quit, container->getDisplay());
    container->readFile("demo2d.ilv");
    XtMainLoop();
    return 0;
}
```

Integrating IBM ILOG Views with an X Application Using libxviews

The Xlib version has the capability of integrating any Xlib-based application as soon as it is provided a pointer to a `Display` object, a `Window` to draw to, and a way to receive events from it.

In the following sections, you will find information on:

- *Integration Steps*
- *Complete Template*
- *Complete Example with Motif*

Integration Steps

To use IBM ILOG Views with any Xlib-based toolkit, you have to:

1. Create an `IlvDisplay` instance using an existing X Display.

   Use the `IlvDisplay` constructor:

   ```
   IlvDisplay::IlvDisplay(IlAny exitingXDisplay, const char* name);
   ```

   For example:

```
Display* xdisplay;
// ... initialize this Display*: xdisplay = XOpenDisplay(...);
IlvDisplay* ilvdisplay = new IlvDisplay((IlAny)xdisplay, "Views");
```

2. Create some `IlvView` or `IlvContainer` instances using an existing X Window:

   Use the `IlvView` constructor:

   ```
   IlvView::IlvView(IlvDisplay* display,
                    IlvSystemView existingXWindow)
   ```

   For example:

   ```
   IlvDisplay* display;
   // initialize this 'display'
   Window xWindow;
   // initialize this X window
   ```

```
IlvView* view = new IlvView(display, (IlvSystemView)xWindow);
```

or

```
IlvContainer* container = new IlvContainer(display,
                                           (IlvSystemView)xWindow);
```

3. Manage the events in these `IlvView` views:

   Once you receive an X event, you must call

   ```
   IlvEventLoop::getEventLoop()->dispatchEvent(&xevent);
   ```

Complete Template

The main procedure looks like this:

```
main()
{
    // Initialize your toolkit
    Display* xdisplay;
    xdisplay = // XOpenDisplay...;
    // Initialize an IlvDisplay
    IlvDisplay* ilvdisplay = new IlvDisplay((IlAny)xdisplay, "Views");
    // Create an X window:
    Window xwindow;
    xwindow = // ...;
    // Create an IlvContainer
    IlvContainer* container = new IlvContainer(display,(IlvSystemView)xwindow);
    container->addObject(new IlvLabel(...));
    // Now call the toolkit main event loop
}
```

Complete Example with Motif

Motif is chosen only as an example of an X-based toolkit. A better way to integrate IBM ILOG Views with Motif is to use the standard IBM ILOG Views library `libmviews` that already does the integration for you. The following example is just meant to illustrate what would need to be done if `libmviews` was not available (`samples/xlib/ilvmotif.cc`):

```
// --------------------------------------------------------------
// Integration of IlogViews, pure XLib version into a Motif
// application
// --------------------------------------------------------------
#include <ilviews/contain.h>
#include <ilviews/label.h>

#include <X11/Intrinsic.h>
#include <Xm/Xm.h>
#include <Xm/DrawingA.h>
#include <X11/StringDefs.h>

// Define the default input mask for the window
#define INPUT_MASK (unsigned long)(ButtonPressMask    | \
```

```
                                    ButtonReleaseMask  | \
                                    KeyPressMask       | \
                                    KeyReleaseMask     | \
                                    ButtonMotionMask   | \
                                    EnterWindowMask    | \
                                    LeaveWindowMask    | \
                                    PointerMotionMask  | \
                                    ExposureMask       | \
                                    StructureNotifyMask)

// -------------------------------------------------------------
// This will be called by Xt when events of any of the
// types specified in INPUT_MASK occur.
// To do this, we call upon the XtAddEventHandler function call
// (see main()).
// -------------------------------------------------------------

static void
ManageInput(Widget, XtPointer view, XEvent* xevent, Boolean*)
{
    IlvEventLoop::getEventLoop()->dispatchEvent(xevent);
}
// -------------------------------------------------------------
void
main(int argc,char** argv)
{
    // Initialize X Window:
    Widget toplevel = XtInitialize("", "IlvXlib", NULL, NULL,
    // XtInitialize has a new specific signature in X11r5
#if defined(XlibSpecificationRelease) && (XlibSpecificationRelease >= 5)
                                    &argc,
#else
                                    (Cardinal*)&argc,
#endif
                                    argv);
    // If the top shell couldn't be created, exit
    if (!toplevel)
        exit(1);

    // Create a Motif widget to draw to
    Widget drawArea = XtVaCreateManagedWidget("ilvview",
                                              xmDrawingAreaWidgetClass,
                                              (Widget)toplevel,
                                              XtNwidth, 400,
                                              XtNheight, 400,
                                              0);
    XtRealizeWidget(toplevel);

    // Create an IlvDisplay instance from the existing Display
    IlvDisplay* display = new IlvDisplay(XtDisplay(drawArea), "Views");

    // Create a container associated with the drawing area:
    IlvContainer* container =
          new IlvContainer(display, (IlvSystemView)XtWindow(drawArea));

    // Create a graphic object in the container
    container->addObject(new IlvLabel(display,
                                      IlvPoint(30, 30),
```

```
                                        "an IlvLabel instance"));
    // Let IlogViews know about the events
    XtAddEventHandler(drawArea, INPUT_MASK, IlFalse, ManageInput, NULL);

    // Wait for events to occur
    XtMainLoop();
}
```

The directory `samples/xlib/` contains more examples applying to various toolkits: `ilvmotif.cpp` is another integration with Motif, similar to this one, `ilvolit.cpp` illustrates an integration with OpenWindow and `ilvxview.cpp` an integration with XView.

D

Portability Limitations

This section provides you with a list of IBM® ILOG® Views features whose portability might be limited because they are system-dependent. In the following sections, these features are grouped into:

- *Non-Supported or Limited Features*: Those that are partially or not supported on certain systems.
- *The Main Event Loop*: Features whose result varies depending on the system on which they are used, in particular the main event loop.

Non-Supported or Limited Features

The table below gives you a list of IBM® ILOG® Views features that are either partially or not supported on certain systems.

Table D.1 *Non-supported or Limited Features*

BitPlanes	Not Supported on Microsoft Windows.
Modal mode	Not supported on Windows NT.
Pattern size	Microsoft Windows patterns are limited in size. You can create larger patterns, but only the upper-left corner will define the final pattern.

Table D.1 *Non-supported or Limited Features (Continued)*

Transparent patterns	On Microsoft Windows, transparent patterns are available only for Microsoft Windows predefined HATCHED brushes. This means that user-defined patterns and some IBM ILOG Views predefined patterns cannot be transparent. The list of the IBM ILOG Views patterns built on a predefined Microsoft Windows HATCHED pattern is: `dialoglr`, `dialogrl`, `horiz`, `vert`, `cross`. This limitation is not applicable when using GDI+, which supports all kind of transparent brushes.
Line style	The following pattern styles are not valid when drawing lines on Microsoft Win9x: `dashdot`, `doubledot`, and `longdash`; these all result in the `dash` style. Setting the line width to a value greater than 1 causes the line pattern to disappear.
Cursor size	On Microsoft Windows, the size of the cursor is fixed and depends on the driver. When bitmaps with bad sizes are given to the `IlvCursor` constructor, an error message is sent. IBM ILOG Views provides the method `IlvCursor::isBad` for testing the success of the creation of a cursor.
Mouse buttons	Certain types of mouse have only two buttons. In this case, the events linked to the right button are set as `IlvMiddleButton`. This results from the fact that, historically, the first interactors used the `IlvMiddleButton` and almost never the `IlvRightButton`. You can modify this behavior using the `UseRightButton` application resource.
Windows icon	On Windows 95 and Windows NT4, the icon associated with each of the views is the same for all the views of an application.
Transparency Anti-aliasing	Available on Windows with GDI+ only. See Appendix B.
Polygons	On Windows 95, the maximal number of points of a polygon is 16381. However, in some cases, a polygon can be composed of more points (convex polygons for example).
Mutable colors	Mutable colors may only be used with the *pseudo color* model. The pseudo color model is an arbitrary mapping of pixel to color that depends on the screen depth and is stored in a color map (UNIX Systems) or a palette (PCs). Mutable colors do not work on *direct color* or on *true color* models.
Window opacity	Not supported on UNIX platforms.

Table D.1 *Non-supported or Limited Features (Continued)*

Zoomable labels	On UNIX, `IlvZoomableLabel` objects are bitmaps that can be zoomed, rotated, and so on. On Microsoft Windows, bitmaps cannot be used because Microsoft Windows is not able to rotate bitmaps, and therefore `IlvZoomableLabel` objects are implemented using True Type fonts. The limitation is due to the fact that True Type fonts are not true vectorial fonts since they work in a step by step way. Moreover, the Microsoft Windows system is not able to give the real size of a font (see Microsoft Win32 Programmer's Reference, Volume 1, page 688: "In Windows, the size of a font is an imprecise value"). **Note***: The same limitation appears for the Vectorial fonts contribution given in the* `<ILVHOME>/tools/vectfont` *directory. Vectorial fonts are implemented using Hershey fonts on UNIX platforms, and True Type or Hershey fonts on Microsoft Windows platforms.*
Strings in XOR mode	This works on X Window. Since Microsoft Windows cannot draw strings in the Xor mode, IBM ILOG Views draws an Xor dotted rectangle that has the same size as the text. To display a real string, display an Xor label using the methods `IlvPort::drawString` or `IlvPort::drawIString`.

The Main Event Loop

The main IBM® ILOG® Views event loop, defined by the global function `IlvMainLoop`, does not work in the same way on X Window and on Microsoft Windows systems. While X Window servers operate in asynchronous mode, Microsoft Windows works in synchronous mode. Also, timer management varies depending on the system used.

- **Synchronous vs. asynchronous mode:** On X Window, a request sent to the server is not immediately processed even if the function returns. It is processed only after it returns to the main loop. For example, a request for displaying a view is performed only when the X Window server sends back a map notify event and this event is processed by the IBM ILOG Views API.
- **Timers management:** On Microsoft Windows, a timer notification is a Windows event that can be processed in the event loop. On X Window, a timer notification is not an event and therefore the main loop is not aware of it, whether it is active or not.

E

Error Messages

This section discusses IBM® ILOG® Views error message generation, based on *The IlvError Class*. It contains a list of the messages that IBM ILOG Views may generate when running your applications. You will find the message text, the explanation of why the error message may have been produced, and a possible workaround.

The lists are alphabetical reference lists and describe:

- *Fatal Errors*
- *Warnings*

If you have not overloaded the default error handler, the fatal error messages are prefixed by two number-sign characters (**#**), and warnings are prefaced by two dashes (**-**).

> ***Note:*** *These are consolidated lists, so some of the error messages are for IBM ILOG Views packages that you may not have.*

The IlvError Class

IBM® ILOG® Views provides an error message mechanism based on the `IlvError` class. There is a default `IlvError` instance that is automatically installed for every IBM ILOG Views application.

This class implements warnings and fatal errors by simply printing out the message parameter. You can create subtypes of this class to perform more complex actions, and make IBM ILOG Views use them.

Two global functions get and set the current error handler:

```
extern "C" IlvError* IlvGetErrorHandler();
extern "C" void      IlvSetErrorHandler(IlvError* errorHandler);
```

To make IBM ILOG Views call the error handler, send each error message through one of these global functions:

```
extern "C" void IlvWarning(const char* format, ...);
extern "C" void IlvFatalError(const char* format, ...);
```

The parameter `format` has the same format as the regular C function `printf`. The above two global functions expect their parameters in the same way as `printf` does.

Fatal Errors

xxx was called with no arguments

In an arithmetic expression execution, the indicated predefined function should be called with at least one parameter.

Bad image description header

Unrecognizable bitmap header in an XPM file.

Bad image colors description

Unrecognizable color descriptor in an XPM file.

Cannot open xxx for writing

This indicated filename could not be opened for writing. The UNIX version gives more information.

couldn't open dump file

Could not open dump file for writing.

couldn't open xxx

The indicated filename cannot be opened for reading or writing.

File xxx has a bad format.

The indicated filename is not an IBM ILOG Views data file.

File IlogViews versions do not match

You are trying to read an IBM ILOG Views data file that was produced with a later version than the library you are running.

Format not implemented.

This BMP format is not implemented.

IlvBitmap::read: couldn't open file xxx

IlvDisplay::readBitmap: couldn't open file xxx

The indicated filename could not be opened for reading.

IlvBitmap::read: bad format xxx

Could not read file as a predefined format (XBM, XPM or PBM P0 or P4).

IlvBitmap::read: unknown color index xxx

This indicates a bad color allocation.

IlvBitmap::save: couldn't open file xxx

Could not open file for writing.

IlvBitmap::saveAscii: Too many colors for ascii format

...(continued)

Too many colors were allocated to read this bitmap with all the required colors. IBM ILOG Views will try to find the closest existing colors to represent the image as accurately as possible.

IlvContainer::readFile: couldn't open file xxx (check ILVPATH)

Specified file could not be loaded. Check your `ILVPATH` environment variable.

IlvContainer::read: wrong format

The file contents could not be loaded.

IlvDisplay::readAsciiBitmap: wrong type xxx

In `IlvDisplay::readAsciiBitmap` this is not a recognized file type.

IlvDisplay::readBitmap: unknown format xxx

The indicated file does not contain a known bitmap format.

IlvEventPlayer::load: couldn't open xxx

Could not open event file for reading.

IlvEventPlayer::save: couldn't open xxx

Could not open event file for writing.

IlvGetViewInteractor: xxx not registered

The indicated view interactor class name is not registered. You may need to add a call to the macro `IlvLoadViewInteractor`.

IlvGifFile() - xxx

A GIF error message. Self explanatory.

IlvInputFile::readNext: unknown class: xxx

The indicated class is unknown by your binary. Try to include the header file where this class appears in one of your modules source file.

IlvInputFile::readObject: bad format for: xxx

Not a valid IBM ILOG Views header.

IlvInputFile::readObjectBlock: no object

An object block could not be successfully read.

IlvManagerViewInteractor: no such view

You tried to set a view interactor to a view that is not connected to the manager.

IlvReadAttribute: unknown attribute class xxx

The indicated attribute class name does not match a known class. You may need to register this attribute class.

IlvReadPBMBitmap: bad format

The header of the PBM file is wrong.

IlvReadPBMBitmap: unknown bitmap format

Wrong PBM format. Known formats are P1 to P6.

IlvPSDevice::drawTransparentBitmap: cannot use image mask

Trying to dump a transparent bitmap that actually is a colored image.

IlvPSDevice::setCurrentPalette: file not opened

The dump file is not opened, but the dump process has started!

IlvVariable::setFormula: error in xxx

There has been an error when trying to read the formula.

IlvVariableContainer::connect: unknown attribute class xxx

IlvVariableManager::connect: unknown attribute class xxx

Need to register this attribute's type. This error message is deprecated.

Not an IlogViews data file

The indicated file is not an IBM ILOG Views data file.

Not a valid IlogViews message database file

`IlvMessageDatabase::read` could not convert this file contents into a database format.

Not a XPM format

Not a valid XPM format. IBM ILOG Views can read XPM 2 and C-coded formats.

PolyPoints with zero points

Trying to create an empty polypoints object.

ReadAsciiColorBitmap: couldn't open xxx

Could not open file for reading.

ReadMonochromeX11Bitmap: couldn't read bitmap. Data=xxx

Only occurs on Microsoft Windows versions. Could not read XBM bitmap file.

Unknown bitmap format: xxx

In `IlvBitmap::read`: unknown bitmap format.

Unknown event type: xxx

Reading an event file: could not find a match with a know event type.

Unknown requested type xxx in isSubtypeOf

The parameter to `IlvGraphic::isSubtypeOf` is not a known class.

Unknown proposed type xxx in isSubtypeOf

The class name of the object that called `isSubtypeOf` is invalid.

Warnings

(<<IlvPattern*): Pattern has no name. Using 'noname'

(<<IlvColorPattern*): Pattern has no name. Using 'noname'

When saving a pattern or a colored pattern, the pattern's bitmap has no name. You may need to set the bitmap name before saving it, otherwise it will not be correctly loaded.

CreateBitmapCell: bitmap xxx not found using default

When creating a bitmap cell, the bitmap name was not found internally. You may need to pre-read the indicated bitmap file.

Found object xxx without IlvPalette

When saving an object, the palette has been replaced. This indicates that you modified your object when saving it.

Icon bitmap has no name. Using 'noname'

When saving a transparent icon, the bitmap has no internal name and will not be loaded properly.

IlvButton::read: could not find bitmap xxx. Using default

When creating a bitmap button, the bitmap name was not found internally. You may need to pre-read the indicated bitmap file.

IlvDisplay::copyStretchedBitmap: can't stretch from pixmap to bitmap

Trying to stretch a color bitmap into a monochrome destination device.

IlvGadgetContainer::read: couldn't allocate background color

The container background color that was saved in the data file cannot be allocated. Release some colors to the system.

IlvGrapher::duplicate: object selection not removed

An error has occured when removing the selection on an object.

IlvGrapher::duplicate: object not found

Trying to duplicate objects that are not stored in the grapher.

IlvIcon::read: could not find bitmap xxx. Using default

When creating an icon, the bitmap name was not found internally. You may need to pre-read the indicated bitmap file.

IlvIcon::write: no name. Using 'noname'...

When saving the icon's bitmap, the bitmap has no name. You may need to set the bitmap name before saving it, otherwise it will not be correctly loaded.

IlvManager::align: invalid value for align : xxx

Invalid direction parameter for `IlvManager::align`.

IlvManager::cleanObj: no properties

Trying to clean an object that is not stored in a manager. You may have removed this object twice, or deleted the object before the manager has removed it.

IlvManager::duplicate: object not found

Trying to duplicate an object that is not in this manager.

IlvManager::reshapeObject: no properties

Trying to reshape an object that is not in this manager.

IlvManager::translateObject: no properties

Trying to translate an object that is not in this manager.

IlvManager::zoomView: invalid transformer

The requested zoom operation would result in a non-reversible transformer.

IlvReadPBMBitmap: bad values

The image description is wrong.

IlvSetLanguage: locale not supported by Xlib

The X11 library to which you linked your application does not support your current locale. You may need to relink with a shared version of libX11 that supports your locale.

IlvTransformer::inverse(IlvPoint&): bad transformer xxx

IlvTransformer::inverse(IlvFloatPoint&): bad transformer xxx

IlvTransformer::inverse(IlvRect&): bad transformer xxx

The transformer cannot perform the inverse call, because it is not reversible. The indicated value is the address of the transformer, provided for debugging purposes. Check the transformer values.

IlvTransparentIcon::read: couldn't find bitmap xxx. Using default

When reading a transparent icon, the name does not match an internally known bitmap. You may need to pre-load the corresponding bitmap.

IlvZoomableIcon::read: couldn't find bitmap xxx. Using default

The bitmap name of the zoomable icon does not match a know bitmap.

Object not removed xxx

In `IlvIndexedSet::removeObject`, the object is not stored in this indexed set.

Quadtree::add: xxx [bbox] Already in quadtree

An object is stored twice in a manager. The object type and its bounding box are provided.

Quadtree::remove: object xxx [bbox] not in quadtree

An object is removed from the manager but it was not stored in it.

ReadBitmap: Bitmap xxx not found! Using default

ReadColorPattern: Pattern xxx not found!

ReadPattern: Pattern xxx not found! Using 'solid'

When reading a bitmap, may be used in a pattern. The bitmap name was not found internally. You may need to pre-read the indicated bitmap file.

ReadLineStyle: LineStyle xxx not found! Using 'solid'

When reading a line style, could not find the indicated line style identifier.

Too many colors. We'll keep xxx

Color allocation request failed. IBM ILOG Views tries to find the closest existing color to complete the bitmap.

WriteBitmap: Bitmap has no name using 'noname'

When saving a bitmap, the bitmap has no name. You may need to set the bitmap name before saving it, otherwise it will not be correctly loaded.

F

IBM ILOG Script 2.0 Language Reference

This reference covers the syntax of IBM ILOG Script. IBM ILOG Script is an ILOG implementation of the JavaScript™ scripting language from Netscape Communications Corporation.

Language Structure

- *Syntax*
- *Expressions*
- *Statements*

Built-In Values and Functions

- *Numbers*
- *Strings*
- *Booleans*
- *Arrays*
- *Objects*
- *Dates*
- *The null Value*
- *The undefined Value*

- *Functions*
- *Miscellaneous*

Syntax

The topics are:

- *IBM ILOG Script Program Syntax*
- *Compound Statements*
- *Comments*
- *Identifier Syntax*

IBM ILOG Script Program Syntax

An IBM ILOG Script program is made of a sequence of *statements*. Statements may include conditional statements, loops, function definitions, local variable declarations, and so forth. An *expression* can also be used any time a statement is expected, in which case its value is ignored and only its side effect is taken into account. Expressions may include assignments, function calls, property access, etc.

Multiple statements or expressions may occur on a single line if they are separated by a semicolon (;). For example, the two following programs are equivalent:

Program1:

```
writeln("Hello, world")
x = x+1
if (x > 10) writeln("Too big")
```

Program2:

```
writeln("Hello, World"); X = X+1; If (X > 10) Writeln("Too Big")
```

Compound Statements

A *compound statement* is a sequence of statements and expressions enclosed in curly brackets ({}). It can be used to perform multiple tasks any time a single statement is expected. For example, in the following conditional statement, the three statements and expressions in curly brackets are executed when the condition a > b is true:

```
if (a > b) {
var c = a
a = b
b = c
}
```

The last statement or expression before a closing curly bracket does not need to be followed by a semicolon, even if it is on the same line. For example, the following program is syntactically correct and is equivalent to the previous one:

```
if (a > b) { var c = a; a = b; b = c }
```

Comments

IBM ILOG Script supports two different styles of comments:

- Single line comments. A single line comment starts with `//` and stops at the end of the line. Example:

  ```
  x = x+1 // Increment x,
  y = y-1 // then decrement y.
  ```

- Multiple line comments. A multiple line comment starts with a `/*` and stops with a `*/`; it can span multiple lines. Nested multiple line comments are not allowed. Example:

  ```
  /* The following statement
  increments x. */
  x = x+1
  /* The following statement
  decrements y. */
  y = y /* A comment can be inserted here */ -1
  ```

Identifier Syntax

Identifiers are used in IBM ILOG Script to name variables and functions. An identifier starts with either a letter or an underscore, and is followed by a sequence of letters, digits, and underscores.

Here are some examples of identifiers:

```
car
x12
main_window
_foo
```

IBM ILOG Script is case sensitive, thus the uppercase letters A-Z are distinct from the lowercase letters a-z. For example, the identifiers car and Car are distinct.

The names in the table below are reserved and cannot be used as identifiers. Some of these names are keywords used in IBM ILOG Script; others are reserved for future use.

The names in the table below are reserved and cannot be used as identifiers. Some of these names are keywords used in IBM ILOG Script; others are reserved for future use.

abstract	else	int	switch
boolean	extends	interface	synchronized
break	false	long	this
byte	final	native	throw
case	finally	new	throws
catch	float	null	transient
char	for	package	true
class	function	private	try
const	goto	protected	typeof
continue	if	public	var
default	implements	return	void
delete	import	short	while
do	in	static	with
double	instanceof	super	

Expressions

The topics are:

- *IBM ILOG Script Expressions*
- *Literals*
- *Variable Reference*
- *Property Access*
- *Assignment Operators*
- *Function Call*
- *Special Keywords*
- *Special Operators*
- *Other Operators*

IBM ILOG Script Expressions

Expressions are a combination of literals, variables, special keywords, and operators.

> ***Note:*** For C/C++ programmers: *The syntax of IBM ILOG Script expressions is very close to the C/C++ syntax.*

The precedence of operators determines the order in which they are applied when evaluating an expression. Operator precedence can be overridden by using parentheses.

The following table lists the IBM ILOG Script operators and gives their precedence, from lowest to highest:

Table F.1 *IBM ILOG Script Operator Precedence*

Category	Operators
Sequence	`,`
Assignment	`= += -= *= /= %= <<= >>= >>>= &= ^= |=`
Conditional	`?:`
Logical-or	`||`
Logical-and	`&&`
Bitwise-or	`|`
Bitwise-xor	`^`
Bitwise-and	`&`
Equality	`== !=`
Relational	`< <= > >=`
Bitwise shift	`<< >> >>>`
Addition, substraction	`+ -`
Multiply, divide	`* / %`
Negation, increment, typeof	`! ~ - ++ -- typeof`
Call	`( )`
New	`new`
Property	`. [ ]`

Literals

Literals can represent:

- *Numbers*, for example: `12 14.5 1.7e-100`
- *Strings*, for example: `"Ford" "Hello world\n"`
- *Booleans*, either `true` or `false`.
- *The null Value*: `null`.

See *Number Literal Syntax* on page 316 and *String Literal Syntax* on page 322 for further details about number and string literal syntax.

Variable Reference

Variable reference syntax is shown in the following table.

Table F.2 *IBM ILOG Script Variable Syntax*

Syntax	Effect
variable	Returns the value of *variable*. See *Identifier Syntax* on page 297 for the syntax of variables. If *variable* doesn't exist, an error is signalled. This is not the same as referencing an existing variable whose value is the undefined value—which is legal and returns the undefined value. When used in the body of a `with` statement, a variable reference is first looked up as a property of the current default value.

Property Access

There are two syntaxes for accessing a value property:

Table F.3 *IBM ILOG Script Property Access Syntax*

Syntax	Effect

value.*name*	Returns the value of the *name* property of *value*, or the undefined value if this property is not defined. See *Identifier Syntax* on page 297 for the syntax of *name*.
	Examples:
	`str.length`
	
	`getCar().name`
	Because *name* must be a valid identifier, this form cannot be used to access properties which don't have a valid identifier syntax. For example, the numeric properties of an array cannot be accessed this way:
	`myArray.10 // Illegal syntax`
	For these properties, use the second syntax.

value[*name*]	Same as the previous syntax, except that this time *name* is an evaluated expression which yields the property name.
	Examples:
	`str["length"] // Same as str.length`
	`getCar()[getPropertyName()]`
	`myArray[10]`
	`myArray[i+1]`

Assignment Operators

The = operator can be used to assign a new value to a variable or a property:

Table F.4 *IBM ILOG Script Assignment Operator Syntax*

Syntax	Effect
variable = *expression*	Assigns the value of *expression* to *variable*. If *variable* does not exist, it is created as a global variable. Examples: `x = y+1` The whole expression returns the value of *expression*.

value.*name* = *expression*
value[*name*] = *expression*
	Assigns the value of *expression* to the given property.
	If *value* doesn't have such a property, then if it is either an array or an object, the property is created; otherwise, an error is signalled.
	Examples:
	`car.name = "Ford"`
	`myArray[i] = myArray[i]+1`
	The whole expression returns the value of *expression*.

In addition, the following shorthand operators are also defined:

Table F.5 *Shorthand Operators*

Syntax	Shorthand For
++*X*	*X* = *X*+1
X++	Same as ++*X*, but returns the initial value of *X* instead of its new value.
--*X*	*X* = *X*–1
X--	Same as --*X*, but returns the initial value of *X* instead of its new value.
X += *Y*	*X* = *X* + *Y*
X -= *Y*	*X* = *X* - *Y*
X *= *Y*	*X* = *X* * *Y*
X /= *Y*	*X* = *X* / *Y*
X %= *Y*	*X* = *X* % *Y*
X <<= *Y*	*X* = *X* << *Y*
X >>= *Y*	*X* = *X* >> *Y*

Table F.5 *Shorthand Operators (Continued)*

Syntax	Shorthand For
X >>>= *Y*	*X* = *X* >>> *Y*
X &= *Y*	*X* = *X* & *Y*
X ^= *Y*	*X* = *X* ^ *Y*
X |= *Y*	*X* = *X* | *Y*

Function Call

The syntax for calling a function is:

Table F.6 *IBM ILOG Script Function Call Syntax*

Syntax	Effect

function(*arg1, ..., argn*)	Calls *function* with the given arguments, and returns the result of the call.
	Examples:
	`parseInt(field)`
	`writeln("Hello ", name)`
	`doAction()`
	`str.substring(start, start+length)`
	Function is typically either a variable reference or a property access, but it can be any arbitrary expression; the expression must yield a function value, or an error is signalled.
	Examples:
	`// Calls the function in callbacks[i]`
	`callbacks[i](arg)`
	`// Error:  a string is not a function`
	`"foo"()`

Special Keywords

Special keywords that can be used are:

Table F.7 *IBM ILOG Script Special Keywords*

Syntax	Effect
`this`	When referenced in a method, returns the current calling object; when referenced in a constructor, returns the object currently being initialized. Otherwise, returns the global object. See *Objects* on page 336 for examples.

`arguments`	Returns an array containing the arguments of the current function. When used outside of a function, an error is signalled.
	For example, the following function returns the sum of all its arguments:
	`function sum() {`
	`  var res = 0`
	`  for (var i=0; i<arguments.length; i++)`
	`  res = res+arguments[i]`
	`  return res`
	`}`
	The call `sum(1, 3, 5)` returns 9.

Special Operators

The special operators are:

Table F.8 *IBM ILOG Script Special Operator Syntax*

Syntax	Effect

`new` *constructor*`(`*arg1*, ..., *argn*`)`	Calls the *constructor* with the given *arguments*, and returns the created value.
	Examples:
	`new Array()`
	`new MyCar("Ford", 1975)`
	Constructor is typically a variable reference, but it can be any arbitrary expression.
	Example:
	`new ctors[i](arg) // Invokes constructor ctors[i]`

Table F.8 *IBM ILOG Script Special Operator Syntax (Continued)*

<table>
<tr><th>Syntax</th><th colspan="2">Effect</th></tr>
<tr><td rowspan="10"><code>typeof</code> value</td><td colspan="2">Returns a string representing the type of value, as follows:</td></tr>
<tr><td>Type of value</td><td>Result of <code>typeof</code> value</td></tr>
<tr><td>array</td><td>"object"</td></tr>
<tr><td>boolean</td><td>"boolean"</td></tr>
<tr><td>date</td><td>"date"</td></tr>
<tr><td>function</td><td>"function"</td></tr>
<tr><td>null</td><td>"object"</td></tr>
<tr><td>number</td><td>"number"</td></tr>
<tr><td>object</td><td>"object"</td></tr>
<tr><td>string</td><td>"string"</td></tr>
<tr><td>undefined</td><td>"undefined"</td></tr>
<tr><td><code>delete</code> variable</td><td colspan="2">Delete the global variable variable. This doesn't mean that the value in variable is deleted, but that variable is removed from the global environment.
Example:
<pre> myVar = "Hello, world" // Create the
global variable myVar
  delete myVar
  writeln(myVar) // Signals an error
because myVar is undefined</pre>If variable is a local variable, an error is signalled; if variable is not a known variable, nothing happens.
The whole expression returns the true value.
Note for C/C++ programmers: This operator has a radically different meaning than in C++, where it is used to delete objects, not variables and properties.</td></tr>
</table>

Table F.8 *IBM ILOG Script Special Operator Syntax (Continued)*

Syntax	Effect
`delete` *value*`.`*name* `delete` *value*`[`*name*`]`	Remove the property *name* from the object *value*. If *value* doesn't contain the *name* property, this expression does nothing. If the property does exist but cannot be deleted, an error is signalled. If *value* is not an object, an error is signalled. The whole expression returns the true value.

expression1 , *expression2*	Evaluates sequentially *expression1* and *expression2*, and returns the value of *expression2*. The value of *expression1* is ignored.
	The most common use for this operator is inside for loops, where it can be used to evaluate several expressions where a single expression is expected:
	`for (var i=0, j=0; i<10; i++, j+=2) {`
	`  writeln(j, " is twice as big as ", i);`
	`}`

Other Operators

Other operators are described in the section dedicated to the datatype they operate on. They are:

Table F.9 *Other IBM ILOG Script Operators*

Syntax	Effect
`-` *X* *X* `+` *Y* *X* `-` *Y* *X* `*` *Y* *X* `/` *Y* *X* `%` *Y*	Arithmetic operators. These operators perform the usual arithmetic operations. In addition, the `+` operator can be used to concatenate strings. See *Numeric Operators* on page 320 and *String Operators* on page 328.
X `==` *Y* *X* `!=` *Y*	Equality operators. These operators can be used to compare numbers and strings; see *Numeric Operators* on page 320 and *String Operators* on page 328. For other types of values, such as dates, arrays, objects, and so forth, the `==` operator is true if, and only if, *X* and *Y* are the exact same value. For example: `new Array(10) == new Array(10)` –> *false* `var a = new Array(10); a == a` –> *true*

Table F.9 *Other IBM ILOG Script Operators (Continued)*

Syntax	Effect
X > *Y* *X* >= *Y* *X* < *Y* *X* <= *Y*	Relational operators. These operators can be used to compare numbers and strings. See *Numeric Operators* on page 320 and *String Operators* on page 328.
~ *X* *X* & *Y* *X* | *Y* *X* ^ *Y* *X* << *Y* *X* >> *Y* *X* >>> *Y*	Bitwise operators. See *Numeric Operators* on page 320.
! *X* *X* || *Y* *X* && *Y* *condition* ? *X* : *Y*	Logical operators. See *Logical Operators* on page 331.

Statements

The topics are:

- *Conditional Statement* (if)
- *Loops* (while, for, for..in, break, continue)
- *Variable Declaration* (var)
- *Function Definition* (function, return)
- *Default Value* (with)

Conditional Statement

The conditional (if) statement has the following syntax:

Table F.10 *IBM ILOG Script Conditional Statement Syntax*

Syntax	Effect

`if (`*expression*`)`
statement1
[`else` *statement2*]
	Evaluate *expression*; if it is true, execute *statement1*; otherwise, if *statement2* is provided, execute *statement2*. If *expression* yields a non-boolean value, this value is converted to a boolean.
	Examples:
	`if (a == b) writeln("They are equal")`
	`  else writeln("They are not equal")`
	
	`if (s.indexOf("a") < 0) {`
	`  write("The string ", s)`
	`  writeln(" doesn't contains the letter a")`
	`}`

Loops

The loop statements have the following syntax:

Table F.11 *IBM ILOG Script Loop Statement Syntax*

Syntax	Effect

`while` (*expression*)
statement
	Execute *statement* repeatedly as long as *expression* is true. The test takes place before each execution of *statement*.
	If *expression* yields a non-boolean value, this value is converted to a boolean.
	Examples:
	`while (a*a < b) a = a+1`
	
	`while (s.length) {`
	`r = s.charAt(0)+r`
	`s = s.substring(1)`
	`}`

`for (` [*initialize*]`;`
[*condition*] `;`
[*update*] `)`
statement
where *condition* and *update* are expressions, and *initialize* is either an expression or has the form:
`var` *variable* = *expression*
	Evaluate *initialize* once, if present. Its value is ignored. If it has the form:
	`var` *variable* = *expression*
	then *variable* is declared as a local variable and initialized as in the var statement.
	Then, execute *statement* repeatedly as long as *condition* is true. If *condition* is omitted, it is taken to be true, which results in an infinite loop. If *condition* yields a non-boolean value, this value is converted to a boolean.
	If present, *update* is evaluated at each pass through the loop, after *statement* and before *condition*. Its value is ignored.
	Example:
	`for (var i=0; i < a.length; i++) {`
	`sum = sum+a[i]`
	`prod = prod*a[i]`
	`}`

Table F.11 *IBM ILOG Script Loop Statement Syntax (Continued)*

Syntax	Effect

`for (`[`var`] *variable* `in` *expression*`)` *statement*	Iterate over the properties of the value of *expression*: For each property, *variable* is set to a string representing this property, and *statement* is executed once.
	If the var keyword is present, *variable* is declared as a local variable, as with the `var` statement.
	For example, the following function takes an arbitrary value and displays all its properties and their values:
	`function printProperties(v) {`
	`  for (var p in v)`
	`  writeln(p, " -> ", v[p])`
	`}`
	Properties listed by the *for..in* statement include method properties, which are merely regular properties whose value is a function value. For example, the call `printProperties("foo")` would display:
	`length -> 3`
	`toString -> [primitive method toString]`
	`substring -> [primitive method substring]`
	`charAt -> [primitive method charAt]`
	etc.
	The only properties which are not listed by *for..in* loops are the numeric properties of arrays.

`break`	Exit the current *while*, *for* or *for..in* loop, and continue the execution at the statement immediately following the loop. This statement cannot be used outside of a loop.
	Example:
	`while (i < a.length) {`
	`  if (a[i] == "foo") {`
	`  foundFoo = true`
	`  break`
	`  }`
	`  i = i+1`
	`}`
	`// Execution continues here`

`continue`	Stop the current iteration of the current *while*, *for* or *for..in* loop, and continue the execution of the loop with the next iteration. This statement cannot be used outside of a loop.
	Example:
	`for (var i=0; i < a.length; i++) {`
	`  if (a[i] < 0) continue`
	`  writeln("A positive number: ", a[i])`
	`}`

Variable Declaration

The variable declaration has the following syntax:

Table F.12 *IBM ILOG Script Variable Declaration Syntax*

Syntax	Effect

`var` *decl1, ..., decln*
where each *decli* has the form
variable [= *expression*]
	Declare each *variable* as a local variable. If an *expression* is provided, it is evaluated and its value is assigned to the variable as its initial value. Otherwise, the variable is set to the undefined value.
	Examples:
	`var x`
	`var name = "Joe"`
	`var average = (a+b)/2, sum, message="Hello"`

Table F.12 *IBM ILOG Script Variable Declaration Syntax (Continued)*

Syntax	Effect

var inside a function definition	When `var` is used inside of a function definition, the declared variables are local to the function, and they hide any global variables with the same name; actually, they have the same status as function arguments.
	For example, in the following program, the variables `sum` and `res` are local to the `average` function, as well as the arguments `a` and `b`; when `average` is called, the global variables with the same names, if any, are temporarily hidden until the function is exited:
	`function average(a, b) {`
	`  var sum = a+b`
	`  var res = sum/2`
	`  return res`
	`}`
	Variables declared with var at *any* place in a function body have a scope which is the *entire* function body. This is different from local variable scope in C or C++.
	For example, in the following function, the variable *res* declared in the first branch of the `if` statement is used in the other branch and in the `return` statement:
	`function max(x, y) {`
	`  if (x > y) {`
	`  var res = x`
	`  } else {`
	`  res = y`
	`  }`
	`  return res`
	`}`

Table F.12 *IBM ILOG Script Variable Declaration Syntax (Continued)*

Syntax	Effect

var outside a function definition	When `var` is used outside of a function definition, that is, at the same level as function definitions, the declared variables are local to the current *program unit.* A program unit is a group of statements which are considered a whole; the exact definition of a program unit depends on the application in which IBM ILOG Script is embedded. Typically, a script file loaded by the application is treated as a program unit. In this case, variables declared with var at the file top level are local to this file, and they hide any global variables with the same names.
	For example, suppose that a file contains the following program:
	`var count = 0`
	
	`function NextNumber() {`
	`  count = count+1`
	`  return count`
	`}`
	When this file is loaded, the function `NextNumber` becomes visible to the whole application, while `count` remains local to the loaded program unit and is visible only inside it.
	It is an error to declare the same local variable twice in the same scope. For example, the following program is incorrect because `res` is declared twice:
	`function max(x, y) {`
	`  if (x > y) {`
	`  var res = x`
	`  } else {`
	`  var res = y // Error`
	`  }`
	`  return res`
	`}`

Function Definition

A function definition has the following syntax:

Table F.13 *IBM ILOG Script Function Definition*

Syntax	Effect

[`static`]
`function` *name*`(`*v1*`, ... , `*vn*`)`
`{` *statements* `}`
	Defines a function *name* with the given parameters and body. A function definition can only take place at the top level; function definitions cannot be nested.
	When the function is called, the variables *v1, ... , vn* are set to the corresponding argument values. Then, the *statements* are executed. If a `return` statement is reached, the function returns the specified value; otherwise, after the *statements* are executed, the function returns the undefined value.
	The number of actual arguments does not need to match the number of parameters: If there are less arguments than parameters, the remaining parameters are set to the undefined value; if there are more arguments than parameters, the exceeding arguments are ignored.
	Independently of the parameter mechanism, the function arguments can be retrieved using the arguments keyword.
	Defining a function *name* is operationally the same as assigning a specific function value to the variable *name*; thus a function definition is equivalent to:
	`var` *name* = *some function value*
	The function value can be retrieved from the variable and manipulated like any other type of value. For example, the following program defines a function `add` and assigns its value to the variable `sum`, which makes `add` and `sum` synonyms for the same function:
	`function add(a, b) {`
	`  return a+b`
	`}`
	
	`sum = add`
	Without the `static` keyword, the defined function is global and can be accessed from the whole application. With the `static` keyword, the function is local to the current program unit, exactly like if *name* was declared with the `var` keyword:
	`var` *name* = *some function value*

`return` [*expression*]	Returns the value of *expression* from the current function. If *expression* is omitted, returns the undefined value. The `return` statement can only be used in the body of a function.

Default Value

A default value is used with the following syntax:

Table F.14 *IBM ILOG Script Default Value*

Syntax	Effect

`with (`*expression*`)`
statement
	Evaluate *expression*, then execute *statement* with the value of *expression* temporarily installed as the default value.
	When evaluating a reference to an identifier *name* in *statement*, this identifier is first looked up as a property of the default value; if the default value does not have such a property, *name* is treated as a regular variable.
	For example, the following program displays "The length is 3", because the identifier `length` is taken as the `length` property of the string `"abc"`.
	`with ("abc") {`
	`  writeln("The length is ", length)`
	`}`
	`With` statements can be nested; in this case, reference to identifiers are looked up in the successive default values, from the innermost to the outermost `with` statement.

Numbers

The topics are:

- *Number Literal Syntax*
- *Special Numbers*
- *Automatic Conversion to a Number*
- *Number Methods*
- *Numeric Functions*
- *Numeric Constants*
- *Numeric Operators*

Number Literal Syntax

Numbers can be expressed in decimal (base 10), hexadecimal (base 16), or octal (base 8.)

> ***Note:*** For C/C++ programmers: *Numbers have the same syntax as C and C++ integers and doubles. They are internally represented as 64-bit double precision floating-point numbers.*

A decimal number consists of a sequence of digits, followed by an optional fraction, followed by an optional exponent. The fraction consists of a decimal point (`.`) followed by a sequence of digits; the exponent consists of an `e` or `E` followed by an optional `+` or `-` sign and a sequence of digits. A decimal number must have at least one digit.

Here are some examples of decimal number literals:

```
15
3.14
4e100
.25
5.25e-10
```

A hexadecimal number consists of `0x` or `0X` prefix, followed by a sequence of hexadecimal digits, which include digits (`0-9`) and the letters `a-f` or `A-F`. For example:

```
0x3ff
0x0
```

An octal number consists of a `0` followed by a sequence of octal digits, which include the digits `0-7`. For example:

```
0123
0777
```

Special Numbers

There are three special numbers: *NaN* (Not-A-Number), *Infinity* (positive infinity), and *-Infinity* (negative infinity.)

The special number NaN is used to indicate errors in number manipulations. For example, the square root function `Math.sqrt` applied to a negative number returns NaN. There is no representation of NaN as a number literal, but the global variable `NaN` contains its value.

The NaN value is contagious, and a numeric operation involving NaN always returns NaN. A comparison operation involving NaN always returns false—even the NaN == NaN comparison.

Examples:

`Math.sqrt(-1)` –> NaN

`Math.sqrt(NaN)` –> NaN

`NaN + 3` –> NaN

`NaN == NaN` –> *false*

`NaN <= 3` –> *false*

`NaN >= 3` –> *false*

The special numbers Infinity and -Infinity are used to indicate infinite values and overflows in arithmetic operations. The global variable `Infinity` contains the positive infinity. The negative infinity can be computed using the negation operator (*-Infinity*).

Examples:

`1/0` –> Infinity

`-1/0` –> -Infinity

`1/Infinity` –> 0

`Infinity == Infinity` –> true

Automatic Conversion to a Number

When a function or a method which expects a number as one of its arguments is passed a non-numeric value, it tries to convert this value to a number using the following rules:

- A string is parsed as a number literal. If the string does not represent a valid number literal, the conversion yields NaN.
- The boolean *true* yields the number 1;
- The boolean *false* yields the number 0;
- The null value yields the number 0;
- A date yields the corresponding number of milliseconds since 00:00:00 UTC, January 1, 1970.

For example, if the `Math.sqrt` function is passed a string, this string is converted to the number it represents:

`Math.sqrt("25")` –> 5

Similarly, operators which take numeric operands attempt to convert any non-numeric operands to a number:

`"3" * "4"` –> 12

For operators which can take both strings and numbers, such as +, the conversion to a string takes precedence over the conversion to a number (see *Automatic Conversion to a String* on

page 323.) In other words, if at least one of the operands is a string, the other operand is converted to a string; if none of the operands is a string, the operands are both converted to numbers. For example:

```
"3" + true –> "3true"
3 + true  –> 4
```

For comparison operators, such as == and >=, the conversion to a number takes precedence over the conversion to a string. In other words, if at least one of the operands is a number, the other operand is converted to a number. If both operands are strings, the comparison is made on strings. For example:

```
"10" > "2" –> false
"10" > 2 –> true
```

Number Methods

The only number method is:

Table F.15 *IBM ILOG Script Number Method*

Syntax	Effect
number.`toString`( )	Returns a string representing the number as a literal. Example: `(14.3e2).toString() –> "1430"`

Numeric Functions

The following numeric functions are defined:

> ***Note:*** For C/C++ programmers*: Most of these functions are wrap-ups for standard math library functions.*

Table F.16 *IBM ILOG Script Numeric Functions*

Syntax	Effect
`Math.abs`(*x*)	Returns the absolute value of *x*.
`Math.max`(*x*, y) `Math.min`(*x*, y)	`Math.max`(*x*, *y*) returns the larger of *x* and *y*, and `Math.min`(*x*, *y*) returns the lowest of the two.
`Math.random( )`	Returns a pseudo-random number between 0, inclusive, and 1, exclusive.

Table F.16 *IBM ILOG Script Numeric Functions (Continued)*

Syntax	Effect

`Math.ceil(`*x*`)`
`Math.floor(`*x*`)`
`Math.round(`*x*`)`
	`Math.ceil(`*x*`)` returns the least integral value greater or equal to *x*. `Math.floor(`*x*`)` returns the greatest integral value less or equal to *x*. `Math.round(`*x*`)` returns the nearest integral value of *x*.

`Math.sqrt(`*x*`)`	Returns the square root of *x*.

`Math.sin(`*x*`)`
`Math.cos(`*x*`)`
`Math.tan(`*x*`)`
`Math.asin(`*x*`)`
`Math.acos(`*x*`)`
`Math.atan(`*x*`)`
`Math.atan2(y, `*x*`)`
	`Math.sin(`*x*`)`, `Math.cos(`*x*`)` and `Math.tan(`*x*`)` return trigonometric functions of radian arguments.
	`Math.asin(`*x*`)` returns the arc sine of *x* in the range -PI/2 to PI/2.
	`Math.acos(`*x*`)` returns the arc cosine of *x* in the range 0 to PI.
	`Math.atan(`*x*`)` returns the arc tangent of *x* in the range -PI/2 to PI/2.
	`Math.atan2(`*y*`, `*x*`)` converts rectangular coordinates `(`*x*, *y*`)` to polar `(`*r*, *a*`)` by computing *a* as an arc tangent of *y*/*x* in the range -PI to PI.

`Math.exp(`*x*`)`
`Math.log(`*x*`)`
`Math.pow(`*x*`, y)`
	`Math.exp(`*x*`)` computes the exponential function `ex`.
	`Math.log(`*x*`)` computes the natural logarithm of *x*.
	`Math.pow(`*x*`, `*y*`)` computes *x* raised to the power *y*.

Numeric Constants

The following numeric constants are defined:

Table F.17 *IBM ILOG Script Numeric Constants*

Syntax	Value
`NaN`	Contains the NaN value.
`Infinity`	Contains the Infinity value.
`Number.NaN`	Same as NaN.
`Number.MAX_VALUE`	The maximum representable number, approximately 1.79E+308.
`Number.MIN_VALUE`	The smallest representable positive number, approximately 2.22E-308.
`Math.E`	Euler's constant and the base of natural logarithms, approximately 2.718.

Table F.17 *IBM ILOG Script Numeric Constants (Continued)*

Syntax	Value
`Math.LN10`	The natural logarithm of 10, approximately 2.302.
`Math.LN2`	The natural logarithm of two, approximately 0.693.
`Math.LOG2E`	The base 2 logarithm of `e`, approximately 1.442.
`Math.LOG10E`	The base 10 logarithm of `e`, approximately 0.434.
`Math.PI`	The ratio of the circumference of a circle to its diameter, approximately 3.142.
`Math.SQRT1_2`	The square root of one-half, approximately 0.707.
`Math.SQRT2`	The square root of two, approximately 1.414.

Numeric Operators

The following numeric operators are available:

Note: For C/C++ programmers*: These operators are the same as in C and C++*.

Table F.18 *IBM ILOG Script Numeric Operators*

Syntax	Effect
x `+` *y* *x* `-` *y* *x* `*` *y* *x* `/` *y*	The usual arithmetic operations. Examples: `3 + 4.2` –> 7.2 `100 - 120` –> -20 `4 * 7.1` –> 28.4 `6 / 5` –> 1.2
`-` *x*	Negation. Examples: `- 142` –> -142
x `%` *y*	Returns the floating-point remainder of dividing *x* by *y*. Examples: `12 % 5` –> 2 `12.5 % 5` –> 2.5

Table F.18 *IBM ILOG Script Numeric Operators (Continued)*

Syntax	Effect
x == *y* *x* != *y*	The operator == returns *true* if *x* and *y* are equal, and *false* otherwise. The operator != is the converse of ==. Examples: `12 == 12` –> *true* `12 == 12.1` –> *false* `12 != 12.1` –> *true*
x < *y* *x* <= *y* *x* > *y* *x* >= *y*	The operator < returns *true* if *x* is smaller than *y*, and *false* otherwise. The operator <= returns *true* if *x* is smaller or equal to *y*, and *false* otherwise; and so on. Examples: `-1 < 0` –> *true* `1 < 1` –> *false* `1 <= 1` –> *true*
x & *y* *x* | *y* *x* ^ *y*	The bitwise operations AND, OR, and XOR. *X* and *y* must be integers in the range of -2**32+1 to 2**32-1 (-2147483647 to 2147483647.) Examples: `14 & 9` –> 8 (1110 & 1001 –> 1000) `14 | 9` –> 15 (1110 | 1001 –> 1111) `14 ^ 9` –> 7 (1110 ^ 1001 –> 111)
~ *x*	Bitwise NOT. *X* must be an integer in the range of -2**32+1 to 2**32-1 (-2147483647 to 2147483647.) Examples: `~ 14` –> 1 (~ 1110 –> 0001)
x << *y* *x* >> *y* *x* >>> *y*	Binary shift operations. *X* and *y* must be integers in the range of -2**32+1 to 2**32-1 (-2147483647 to 2147483647.) The operator << shifts to the left, >> shifts to the right (maintaining the sign bit), and >>> shifts to the right, shifting in zeros from the left. Examples: `9 << 2` –> 36 (1001 << 2 –> 100100) `9 >> 2` –> 2 (1001 >> 2 –> 10) `-9 >> 2` –> -2 (1..11001 >> 2 –> 1..11110) `-9 >>> 2` –> 1073741821 (1..11001 >>> 2 –> 01..11110)

Strings

- *String Literal Syntax*
- *Automatic Conversion to a String*
- *String Properties*
- *String Methods*
- *String Functions*
- *String Operators*

String Literal Syntax

A string literal is zero or more characters enclosed in double (`"`) or single (`'`) quotes.

> ***Note:*** For C/C++ programmers: *Except for the use of single quotes, string literals have the same syntax as in C and C++.*

Here are examples of string literals:

```
"My name is Hal"
'My name is Hal'
'"Hi there", he said'
"3.14"
"Hello, world\n"
```

In these examples, the first and the second strings are identical.

The backslash character (`\`) can be used to introduce an escape sequence, which stands for a character which cannot be directly expressed in a string literal. Escape sequences allowed in strings are:

Table F.19 *IBM ILOG Script Escape Sequences*

Escape Sequence	Stands for
`\n`	Newline
`\t`	Tab
`\\`	Backslash character (\)
`\"`	Double quote (")

Table F.19 *IBM ILOG Script Escape Sequences (Continued)*

Escape Sequence	Stands for
`\'`	Single quote (')
`\b`	Backspace
`\f`	Form feed
`\r`	Carriage return
`\x`*hh*	The character whose ASCII code is *hh*, where *hh* is a sequence of two hexadecimal digits.
`\`*ooo*	The character whose ASCII code is *ooo*, where *ooo* is a sequence of one, two, or three octal digits.

Here are examples of string literals using escape sequences:

Table F.20 *IBM ILOG Script Escape Sequence Examples*

String Literal	Stands for
`"Read \"The Black Bean\""`	Read "The Black Bean"
`'\'Hello\', he said'`	'Hello', he said
`"c:\\temp"`	c:\temp
`"First line\nSecond line\nThird line"`	First line Second line Third line
`"\xA9 1995-1997"`	© 1995-1997

When a string is converted to a number, an attempt is made to parse it as a number literal. If the string does not represent a valid number literal, the conversion yields NaN.

Automatic Conversion to a String

When a function or a method which expects a string as one of its arguments is passed a non-string value, this value is automatically converted to a string. For example, if the string method indexOf is passed a number as its first argument, this number is treated like its string representation:

`"The 10 commandments".indexOf(10)` –> 4

Similarly, operators which take string operands automatically convert non-string operands to strings:

`"The " + 10 + " commandments"` –> "The 10 commandments"

The conversion to a string uses the `toString` method of the given value. All built-in values have a `toString` method.

String Properties

Strings have the following properties:

Table F.21 *IBM ILOG Script String Properties*

Syntax	Value
string`.length`	Number of characters in *string*. This is a read-only property. Examples `"abc".length` –> 3 `"".length` –> 0

String Methods

Characters in a string are indexed from left to right. The index of the first character in a string *string* is 0, and the index of the last character is *string*.length-1.

Strings have the following methods:

Table F.22 *IBM ILOG Script String Methods*

Syntax	Effect
string`.substring (`*start* `[ ,` *end* `] )`	Returns the substring of *string* starting at the index *start* and ending at the index *end*-1. If *end* is omitted, the tail of *string* is returned. Examples: `"0123456".substring(0, 3)` –> "012" `"0123456".substring(2, 4)` –> "23" `"0123456".substring(2)` –> "23456"
string`.charAt(`*index*`)`	Returns a one-character string containing the character at the specified index of *string*. If *index* is out of range, an empty string is returned. Examples: `"abcdef".charAt(0)` –> "a" `"abcdef".charAt(3)` –> "d" `"abcdef".charAt(100)` –> ""

Table F.22 *IBM ILOG Script String Methods (Continued)*

Syntax	Effect
string`.charCodeAt(`*index*`)`	Returns the ASCII code of the character at the specified index of *string*. If *index* is out of range, return NaN. Examples: `"abcdef".charCodeAt(0)` –> 97 `"abcdef".charCodeAt(3)` –> 100 `"abcdef".charCodeAt(100)` –> NaN
string`.indexOf(`*substring* `[ , `*index*` ] )`	Returns the index in *string* of the first occurrence of *substring*. *String* is searched starting at *index*. If *index* is omitted, *string* is searched from the beginning. This method returns -1 if *substring* is not found. Examples: `"abcdabcd".indexOf("bc")` –> 1 `"abcdabcd".indexOf("bc", 1)` –> 1 `"abcdabcd".indexOf("bc", 2)` –> 5 `"abcdabcd".indexOf("bc", 10)` –> -1 `"abcdabcd".indexOf("foo")` –> -1 `"abcdabcd".indexOf("BC")` –> -1
string`.lastIndexOf` `(`*substring* `[ , `*index*` ] )`	Returns the index in *string* of the last occurrence of *substring*. *String* is searched backwards, starting at *index*. If *index* is omitted, *string* is searched from the end. This method returns -1 if *substring* is not found. Examples: `"abcdabcd".lastIndexOf("bc")` –> 5 `"abcdabcd".lastIndexOf("bc", 5)` –> 5 `"abcdabcd".lastIndexOf("bc", 4)` –> 1 `"abcdabcd".lastIndexOf("bc", 0)` –> -1 `"abcdabcd".lastIndexOf("foo")` –> -1 `"abcdabcd".lastIndexOf("BC")` –> -1
string`.toLowerCase( )`	Returns *string* converted to lowercase. Examples: `"Hello, World".toLowerCase()` –> "hello, world"
string`.toUpperCase( )`	Returns *string* converted to uppercase. Examples: `"Hello, World".toUpperCase()` –> "HELLO, WORLD"

Table F.22 *IBM ILOG Script String Methods (Continued)*

Syntax	Effect
string`.split(`*separator*`)`	Returns an array of strings containing the substrings of *string* which are separated by *separator*. See also the array method join. Examples: `"first name,last name,age".split(",")` –> an array *a* such that a.length is 3, a[0] is "first name", a[1] is "last name", and a[2] is "age". If *string* does not contain *separator*, an array with one element containing the whole *string* is returned. Examples: `"hello".split(",")` –> an array *a* such that a.length is 1 and a[0] is "hello",
string`.toString( )`	Returns the string itself.

String Functions

The following functions operate on strings:

Table F.23 *IBM ILOG Script String Functions*

Syntax	Effect
`String.fromCharCode(`*code*`)`	Returns a single character string containing the character with the given ASCII code. Examples: `String.fromCharCode(65)` –> "A" `String.fromCharCode(0xA9)` –> "©"
`parseInt(`*string* `[ , `*base*` ] ) IBM`	Parses *string* as an integer written in the given base, and returns its value. If the string does not represent a valid integer, NaN is returned. Leading white space characters are ignored. If `parseInt` encounters a character that is not a digit in the specified base, it ignores it and all succeeding characters and returns the integer value parsed up to that point. If *base* is omitted, it is taken to be 10, unless *string* starts with `0x` or `0X`, in which case it is parsed in base 16, or with `0`, in which case it is parsed in base 8. Examples: `parseInt("123")` –> -123 `parseInt("-123")` –> -123 `parseInt("123.45")` –> 123 `parseInt("1001010010110", 2)` –> 4758 `parseInt("a9", 16)` –> 169 `parseInt("0xa9")` –> 169 `parseInt("010")` –> 8 `parseInt("123 poodles")` –> 123 `parseInt("a lot of poodles")` –> NaN
`parseFloat(`*string*`)`	Parses *string* as a floating-point number and return its value. If the string does not represent a valid number, NaN is returned. Leading white space characters are ignored. The string is parsed up to the first unrecognized character. If no number is recognized, the function returns NaN. Examples: `parseFloat("-3.14e-15")` –> -3.14e-15 `parseFloat("-3.14e-15 poodles")` –> -3.14e-15 `parseFloat("a fraction of a poodle")` –> NaN

String Operators

The following operators can be used to manipulate strings:

Table F.24 *IBM ILOG Script String Operators*

Syntax	Effect
string1 + *string2*	Returns a string containing the concatenation of *string1* and *string2*. Examples: `"Hello," + " world"` –> "Hello, world" When the operator + is used to add a string to a non-string value, the non-string value is first converted to a string. Examples: `"Your age is " + 23` –> "Your age is 23" `23 + " is your age"` –> "23 is your age"

Table F.24 *IBM ILOG Script String Operators (Continued)*

Syntax	Effect
string1 `==` *string2* *string1* `!=` *string2*	The operator `==` returns the boolean *true* if *string1* and *string2* are identical, and *false* otherwise. Two strings are identical if they have the same length and contain the same sequence of characters. The operator `!=` is the converse of `==`. Examples: `"a string" == "a string"` –> *true* `"a string" == "another string"` –> *false* `"a string" == "A STRING"` –> *false* `"a string" != "a string"` –> *false* `"a string" != "another string"` –> *true* When the operators `==` and `!=` are used to compare a string with a number, the string is first converted to a number and the two numbers are compared numerically. Examples: `"12" == "+12"` –> *false* `12 == "+12"` –> *true*

Table F.24 *IBM ILOG Script String Operators (Continued)*

Syntax	Effect
string1 `<` *string2* *string1* `<=` *string2* *string1* `>` *string2* *string1* `>=` *string2*	The operator `<` returns *true* if *string1* strictly precedes *string2* lexicographically, and *false* otherwise. The operator `<=` returns *true* if *string1* strictly precedes *string2* lexicographically or is equal to it, and *false* otherwise; and so on. Examples: `"abc" < "xyz"` –> *true* `"a" < "abc"` –> *true* `"xyz" < "abc"` –> *false* `"abc" < "abc"` –> *false* `"abc" > "xyz"` –> *false* `"a" > "abc"` –> *false* `"xyz" > "abc"` –> *true* Etc. When one of these operators is used to compare a string with a non-string value, the non-string value is first converted to a string. Examples: `"2" >= 123` –> *true* `123 < "2"` –> *false* When one of these operators is used to compare a string with a number, the string is first converted to a number and the two numbers are compared numerically. Examples: `"10" > "2"` –> *false* `10 > "2"` –> *true*

Booleans

- *Boolean Literal Syntax*
- *Automatic Conversion to a Boolean*
- *Boolean methods*
- *Logical Operators*

Boolean Literal Syntax

There are two boolean literals: `true`, which represents the boolean value *true*, and `false`, which represents the boolean value *false*.

When converted to a number, *true* yields 1 and *false* yields 0.

Automatic Conversion to a Boolean

When a function, method or statement which expects a boolean as one of its arguments is passed a non-boolean value, this value is automatically converted to a boolean as follows:

- The number 0 yields *false*;
- The empty string "" yields *false*;
- The null value yields *false*;
- The undefined value yields *false*;
- Any other non-boolean values yield *true*.

For example:

```
if ("") writeln("True"); else writeln("False");
if (123) writeln("True"); else writeln("False");
```

displays "False", then "True".

Boolean methods

The only boolean method is:

Table F.25 *IBM ILOG Script Boolean Method*

Syntax	Effect
boolean`.toString()`	Returns a string representing the boolean, either "true" or "false". Example: `true.toString` –> "true" `false.toString` –> "false"

Logical Operators

The following boolean operators are available:

Note: For C/C++ programmers*: These operators are the same as in C and C++*.

Table F.26 *IBM ILOG Script Logical Operators*

Syntax	Effect
`!` *boolean*	Logical negation. Examples: `! true` –> *false* `! false` –> *true*
exp1 `&&` *exp2*	Returns *true* if both boolean expressions *exp1* and *exp2* are true. Otherwise, returns *false*. If *exp1* is false, this expression immediately returns *false* without evaluating *exp2*, so any side effects of *exp2* are not taken into account. Examples: `true && true` –> *true* `true && false` –> *false* `false &&` *whatever* –> *false*; *whatever* is not evaluated.
exp1 `||` *exp2*	Returns *true* if either boolean expression *exp1* or *exp2* is true. Otherwise, returns *false*. If *exp1* is true, this expression immediately returns *true* without evaluating *exp2*, so any side effects of *exp2* are not taken into account. Examples: `false || true` –> *true* `false || false` –> *false* `true ||` *whatever* –> *true*; *whatever* is not evaluated.
condition `?` *exp1* `:` *exp2*	If *condition* is true, this expression returns *exp1*; otherwise, it returns *exp2*. When *condition* is true, the expression *exp2* is not evaluated, so any side effects it may contain are not taken into account. Similarly, when *condition* is false, *exp1* is not evaluated. Examples: `true ? 3.14 :` *whatever* –> 3.14 `false ?` *whatever* `: "Hello"` –> "Hello"

Arrays

The topics are:

- *IBM ILOG Script Arrays*
- *Array Constructor*
- *Array Properties*

- *Array Methods*

IBM ILOG Script Arrays

Arrays provide a way of manipulating ordered sets of values referenced through an index starting from 0. Unlike arrays in other languages, IBM ILOG Script arrays do not have a fixed size and are automatically expanded as new elements are added. For example, in the following program, an array is created empty, and is then added new elements:

```
a = new Array() // Create an empty array
a[0] = "first" // Set the element 0
a[1] = "second" // Set the element 1
a[2] = "third" // Set the element 2
```

Arrays are internally represented as sparse objects, which means that an array where only the element 0 and the element 10000 have been set occupies just enough memory to store these two elements, not the 9999 which are between 0 and 10000.

Array Constructor

The array constructor has two distinct syntaxes:

Table F.27 *IBM ILOG Script Array Constructor Syntax*

Syntax	Effect
`new Array`(*length*)	Returns a new array of length *length* with its elements from 0 to *length*-1 set to null. If *length* is not a number, and its conversion to a number yields NaN, the second syntax is used. Examples: `new Array(12)` –> an array *a* with length 12 and a[0] to a[11] containing *null*. `new Array("5")` –> an array *a* with length 5 and a[0] to a[4] containing *null*. `new Array("foo")` –> see second syntax.
`new Array`(*element1*, ..., *elementn*)	Returns a new array *a* of length *n* with a[0] containing *element1*, a[1] containing *element2*, and so on. If no argument is given, that is *n*=0, an empty array is created. If *n*=1 and *element1* is a number or can be converted to a number, the first syntax is used. Examples: `new Array(327, "hello world")` –> an array *a* of length 2 with a[0] == 327 and a[1] == "hello world". `new Array()` –> an array with length 0. `new Array("327")` –> see first syntax.

Array Properties

The array properties are:

Table F.28 *IBM ILOG Script Array Properties*

Syntax	Effect

array[*index*]	If *index* can be converted to a number between 0 and 2e32-2 (see *Automatic Conversion to a Number* on page 317), *array*[*index*] is the value of the *index*th element of the array. Otherwise, it is considered as a standard property access.If this element has never been set, *null* is returned.
	Example: Suppose that the array *a* has been created with:
	`a = new Array("foo", 12, true)`
	Then:
	`a[0]` –> "foo"
	`a[1]` –> 12
	`a[2]` –> *true*
	`a[3]` –> *null*
	`a[1000]` –> *null*
	When an element of an array is set beyond the current length of the array, the array is automatically expanded:
	`a[1000] = "bar"`
	`// the array is automatically expanded.`
	Unlike other properties, the numeric properties of an array are *not* listed by the for..in statement.

array.`length`	The length of *array*, which is the highest index of an element set in *array*, plus one. It is always included in 0 and 2e31-1. When a new element is set in the array, and its index is greater or equal to the current array length, the `length` property is automatically increased.
	Example: Suppose that the array *a* has been created with:
	`a = new Array("a", "b", "c")`
	Then:
	`a.length` –> 3
	`a[100] = "bar"`;`a.length` –> 101
	You can also change the length of an array by setting its length property.
	`a = new Array(); a[4] = "foo"; a[9] = "bar";`
	`a.length` –> 10
	`a.length` = 5
	`a.length` –> 5
	`a.length` –> 5
	`a[4]` –> "foo"
	`a[9]` –> null

Array Methods

Arrays have the following methods:

Table F.29 *IBM ILOG Script Array Methods*

Syntax	Effect
array.`join`([*separator*])	Returns a string which contains the elements of the array converted to strings, concatenated together and separated with *separator*. If *separator* is omitted, it is taken to be ",". Elements which are not initialized are converted to the empty string. See also the string method split. Example: Suppose that the array *a* has been created with `a = new Array("foo", 12, true)` Then: `a.join("//")` –> "foo//12//true" `a.join()` –> "foo,12,true"

array.`sort`([*function*])	Sorts the array. The elements are sorted in place; no new array is created.
	If *function* is not provided, *array* is sorted lexicographically: Elements are compared by converting them to strings and using the <= operator. With this order, the number 20 would come before the number 5, since "20" < "5" is true.
	If *function* is supplied, the array is sorted according to the return value of this function. This function must take two arguments *x* and *y* and return:
	◆ -1 if *x* is smaller than *y*;
	◆ 0 if *x* is equal to *y*;
	◆ 1 if *x* is greater than *y*.
	Example: Suppose that the function `compareLength` is defined as
	`function compareLength(x, y) {`
	`  if (x.length < y.length) return -1;`
	`  else if (x.length == y.length) return 0;`
	`  else return 1;`
	`}`
	and that the array *a* has been created with:
	`a = new Array("giraffe", "rat", "brontosaurus")`
	Then `a.sort()` will reorder its elements as follows:
	`"brontosaurus" "rat" "giraffe"`
	while `a.sort(compareLength)` will reorder them as follows:
	`"rat" "giraffe" "brontosaurus"`

Table F.29 *IBM ILOG Script Array Methods (Continued)*

Syntax	Effect
array.`reverse`( )	Transposes the elements of the array: The first element becomes the last, the second becomes the second to last, etc. The elements are reversed in place; no new array is created. Example: Suppose that the array *a* has been created with `a = new Array("foo", 12, "hello", true, false)` Then a.reverse() changes *a* so that: `a[0]` –> *false* `a[1]` –> *true* `a[2]` –> "hello" `a[3]` –> 12 `a[4]` –> "foo"
array.`toString`( )	Returns the string "[object Object]".

Objects

The topics are:

- *IBM ILOG Script Objects*
- *Defining Methods*
- *The this Keyword*
- *Object Constructor*
- *User-defined Constructors*
- *Built-in Methods*

IBM ILOG Script Objects

Objects are values which do not contain any predefined properties or methods (except the `toString` method), but where new ones can be added. A new, empty object can be created using the Object constructor. For example, the following program creates a new object, stores it in the variable `myCar`, and adds the properties "name" and "year" to it:

```
myCar = new Object() // o contains no properties
myCar.name = "Ford"
myCar.year = 1985
```

Now:

```
myCar.name –> "Ford"
```

```
myCar.year -> 1985
```

Defining Methods

Since a method is really a property which contains a function value, defining a method simply consists in defining a regular function, then assigning it to a property.

For example, the following program adds a method "start" to the `myCar` object defined in *IBM ILOG Script Objects*:

```
function start_engine() {
  writeln("vroom vroom\n")
}

myCar.start = start_engine
```

Now, the expression `myCar.start()` will call the function defined as `start_engine`. Note that the only reason for using a different name for the function and for the method is to avoid confusion; we could have written:

```
function start() {
  writeln("vroom vroom\n")
}

myCar.start = start
```

The this Keyword

Inside methods, the `this` keyword can be used to reference the calling object. For example, the following program defines a method `getName`, which returns the value of the `name` property of the calling object, and adds this method to *myCar*:

```
function get_name() {
  return this.name
}

myCar.getName = get_name
```

Inside constructors, `this` references the object created by the constructor. When used in a non-method context, `this` returns a reference on the global object. The global object contains variables declared at toplevel, and built-in functions and constructors.

Object Constructor

Objects are created using the following constructor:

Table F.30 *IBM ILOG Script Object Constructor*

Syntax	Effect
`new Object()`	Returns a new object with no properties.

User-defined Constructors

In addition to the `Object` constructor, any user-defined function can be used as an object constructor, using the following syntax:

Table F.31 *IBM ILOG Script User-defined Constructor*

Syntax	Effect
`new` *function*(*arg1*, ..., *argn*)	Creates a new object, then calls *function*(*arg1*, ..., *argn*) to initialize it.

Inside the constructor, the keyword `this` can be used to make reference to the object being initialized.

For example, the following program defines a constructor for cars:

```
function Car(name, year) {
  this.name = name
  this.year = year
  this.start = start_engine
}
```

Now, calling

```
new Car("Ford", "1985")
```

creates a new object with the properties `name` and `year`, and a `start` method.

Built-in Methods

The only object built-in method is:

Table F.32 *IBM ILOG Script Built-in Method*

Syntax	Effect
object`.toString( )`	Returns the string "[object Object]". This method can be overridden by assigning the `toString` property of an object.

Dates

The topics are:

- *IBM ILOG Script Date Values*
- *Date Constructor*

- *Date Methods*
- *Date Functions*
- *Date Operators*

IBM ILOG Script Date Values

Date values provide a way of manipulating dates and times. Dates can be best understood as internally represented by a number of milliseconds since 00:00:00 UTC, January 1, 1970. This number can be negative, thus expressing a date before 1970.

> ***Note:*** For C/C++ programmers: *Unlike dates manipulated by the the standard C library, date values are not limited to the range of 1970 to 2038, but span over approximately 285,616 years before and after 1970.*

When converted to a number, a date yields the number of milliseconds since 00:00:00 UTC, January 1, 1970.

Date Constructor

The date constructor has four distinct syntaxes:

Table F.33 *IBM ILOG Script Date Constructor*

Syntax	Effect
`new Date()`	Returns the date representing the current time.
`new Date(`*milliseconds*`)`	Returns the date representing 00:00:00 UTC, January 1, 1970, plus *milliseconds* milliseconds. The argument can be negative, thus expressing a date before 1970. If the argument cannot be converted to a number, the third constructor syntax is used. Examples: `new Date(0)` –> a date representing 00:00:00 UTC, January 1, 1970. `new Date(1000*60*60*24*20)` –> a date representing twenty days after 00:00:00 UTC, January 1, 1970. `new Date(-1000*60*60*24*20)` –> a date representing twenty days before 00:00:00 UTC, January 1, 1970.

Table F.33 *IBM ILOG Script Date Constructor (Continued)*

Syntax	Effect
`new Date`(*string*)	Returns the date described by *string*, which must have the form: *month*/*day*/*year hour*:*minute*:*second msecond* The date expressed in *string* is taken in local time. Example: `new Date("12/25/1932 14:35:12 820")` –> a date representing December the 25th, 1932, at 2:35 PM plus 12 seconds and 820 milliseconds, local time.
`new Date`(*year*, *month*, [, *date* [, *hours* [, *minutes* [, *seconds* [, *mseconds*]]]]]]])	Returns a new date representing the given year, month, date, etc., taken in local time. The arguments are: ◆ *year*: Any integer. ◆ *month*: range 0-11 (0=January, 1=February, etc) ◆ *day*: range 1-31, defaults to 1. ◆ *hours*: range 0-23, defaults to 0. ◆ *minutes*: range 0-59, defaults to 0. ◆ *seconds*: range 0-59, defaults to 0. ◆ *mseconds*: range 0-999, defaults to 0. Example: `new Date(1932, 11, 25, 14, 35, 12, 820)` –> a date representing December the 25th, 1932, at 2:35 PM plus 12 seconds and 820 milliseconds, local time. `new Date(1932, 11, 25)` –> a date representing December the 25th, 1932, at 00:00, local time.

Date Methods

Dates have the following methods:

Table F.34 *IBM ILOG Script Date Methods*

Syntax	Effect
date`.getTime()` *date*`.setTime(`*milliseconds*`)`	Returns (or sets) the number of milliseconds since 00:00:00 UTC, January 1, 1970. Example: Suppose that the date *d* has been created with: `d = new Date(3427)` Then: `d.getTime()` –> 3427
date`.toLocaleString()` *date*`.toUTCString()`	Returns a string representing the date in local time (respectively in UTC.) Example: Suppose that the date *d* has been created with: `d = new Date("3/12/1997 12:45:00 0")` Then: `d.toLocaleString()` –> "03/12/1997 12:45:00 000" `d.toUTCString()` –> "03/12/1997 10:45:00 000", assuming a local time zone offset of +2 hours with respect to the Greenwich meridian.
date`.getYear()` *date*`.setYear(`*year*`)`	Returns (or sets) the year of *date*.
date`.getMonth()` *date*`.setMonth(`*month*`)`	Returns (or sets) the month of *date*.
date`.getDate()` *date*`.setDate(`*day*`)`	Returns (or sets) the day of *date*.
date`.getHours()` *date*`.setHours(`*day*`)`	Returns (or sets) the hour of *date*.
date`.getMinutes()` *date*`.setMinutes(`*day*`)`	Returns (or sets) the minute of *date*.
date`.getSeconds()` *date*`.setSeconds(`*day*`)`	Returns (or sets) the second of *date*.
date`.getMilliseconds()` *date*`.setMilliseconds(`*day*`)`	Returns (or sets) the millisecond of *date*.
date`.toString()`	Returns the same value as *date*`.toLocaleString()`

Date Functions

The following functions manipulate dates:

Table F.35 *IBM ILOG Script Date Functions*

Syntax	Effect
`Date.UTC`(*string*)	Same as `new Date`(*string*), but *string* is taken in UTC and the result is returned as a number rather than as a date object.
`Date.parse`(*string*)	Same as `new Date`(*string*), but the result is returned as a number rather than as a date object.

Date Operators

There are no specific operators for dealing with dates, but, since numeric operators automatically convert their arguments to numbers, these operators can be used to compute the time elapsed between two dates, to compare dates, or to add a given amount of time to a date. For example:

date1 `-` *date2* –> the number of milliseconds elapsed between *date1* and *date2*.

date1 `<` *date2* –> true if *date1* is before *date2*, false otherwise.

`new Date(`*date*`+10000)` –> a date representing 10000 milliseconds after *date*.

The following program displays the number of milliseconds spent for executing the statement <*do something*>:

```
before = new Date()
<do something>
after = new Date()
writeln("Time for doing something: ", after-before, " milliseconds.")
```

The null Value

The topics are:

- *The IBM ILOG Script null Value*
- *Methods of null*

The IBM ILOG Script null Value

The *null* value is a special value used in some places to specify an absence of information. For example, an array element which hasn't been set yet has a default *null* value. The *null*

value is not to be confused with the undefined value, which also specifies an absence of information in some contexts.

The *null* value can be referenced in programs with the keyword `null`:

`null` –> the *null* value

When converted to a number, *null* yields 0.

Methods of null

The only method of *null* is:

Table F.36 *IBM ILOG Script Null Method*

Syntax	Effect
null`.toString( )`	Returns the string "null".

The undefined Value

The topics are:

- *The IBM ILOG Script undefined Value*
- *Methods of undefined*

The IBM ILOG Script undefined Value

The *undefined* value is a special value used in some places to specify an absence of information. For example, accessing a property of a value which is not defined, or a local variable which has been declared but not initialized, yields the *undefined* value.

There is no way of referencing the *undefined* value in programs. Checking if a value is the undefined value can be done using the `typeof` operator:

`typeof(`*value*`) == "undefined"` –> *true* if *value* is undefined, *false* otherwise.

Methods of undefined

The only method of *undefined* is:

Table F.37 *IBM ILOG Script Undefined Method*

Syntax	Effect
undefined`.toString( )`	Returns the string "undefined".

Functions

The topics are:

- *IBM ILOG Script Functions*
- *Function Methods*

IBM ILOG Script Functions

In IBM ILOG Script, functions are regular values (also known as "first class" values) which can be manipulated like any other type of value: They can be passed to functions, returned by functions, stored into variables or into objects properties, etc.

For example, the function `parseInt` is actually a function value which is stored in the `parseInt` variable:

```
parseInt –> a function value
```

This function value can be, for example, assigned to another variable:

```
myFunction = parseInt
```

and then called through this variable:

```
myFunction("-25") –> -25
```

Function Methods

The only method of functions is:

Table F.38 *IBM ILOG Script Function Method*

Syntax	Effect
function`.toString()`	Returns a string which contains some information about the function. Examples: `"foo".substring.toString()` –> "[primitive method substring]" `eval.toString()` –> "[primitive function eval]"

Miscellaneous

Miscellaneous functions are described in the following table.

Table F.39 *Miscellaneous Functions*

Syntax	Effect
`stop`( )	Stops the execution of the program at the current statement and, if the debugger is enabled, enters in debug mode.
`write`(*arg1*, ..., *argn*) `writeln`(*arg1*, ..., *argn*)	Converts the arguments to strings and prints them to the current debug output. The implementation of this depends on the application in which IBM ILOG Script is embedded. The function `writeln` prints a newline at the end of the output, while `write` does not.
`loadFile`(*string*)	Loads the script file whose path is *string*. The path can be either absolute or relative. If this path does not designate an existing file, the file is looked up using a method which depends on the application in which IBM ILOG Script is embedded; typically, a file with the name *string* is searched in a list of directories specified in the application setup.
`eval`(*string*)	Executes *string* as a program, and returns the value of the last evaluated expression. The program in *string* can use all the features of the language, except that it cannot define functions; in other words, the function statement is not allowed in *string*. Examples: `eval("2*3")` –> 6 `eval("var i=0; for (var j=0; j<100; j++) i=i+j; i")` –> 4950 `n=25; eval("Math.sqrt(n)")` –> 5 `eval("function foo(x) { return x+1 }")` –> *error*

Index

D

E

F

G

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z