IBM ILOG JViews Diagrammer V8.6

Using the Symbol Compiler

Copyright

Copyright notice

Trademarks

Notices

For further copyright information see ***<installdir>*** **`/license/notices.txt`**.

Table of contents

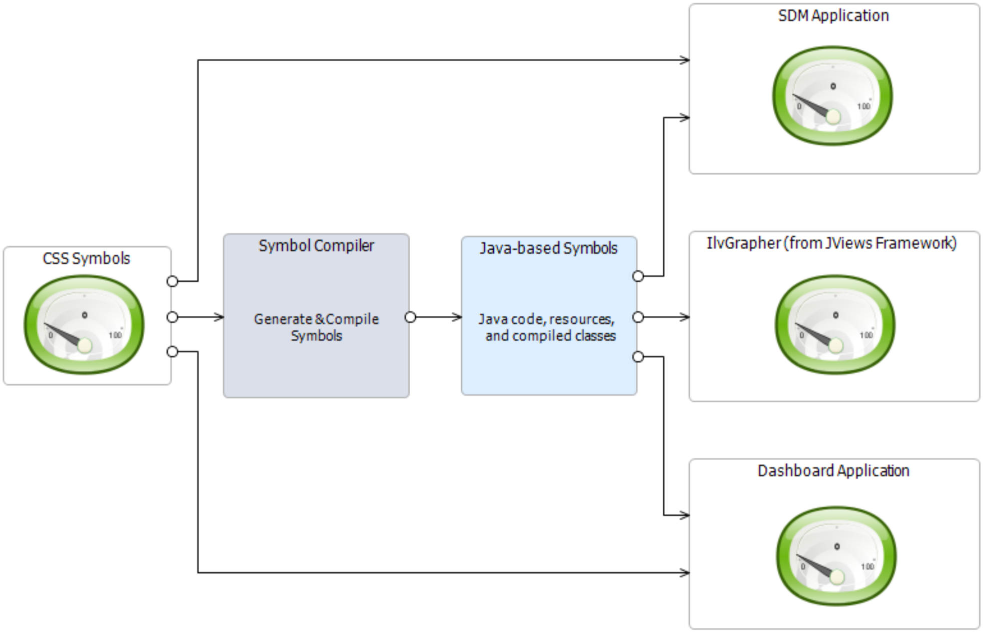
Introducing the Symbol Compiler

Symbols designed with the Symbol Editor can be used directly in JViews Diagrammer applications to create live diagrams and dashboards. Internally, these symbols are fully described using CSS directives and are executable at runtime without any compilation. In some cases, it can be interesting to use a slightly different approach and have Java™ code available instead of interpreted versions of symbols.

Here are a few cases of interest:

- Improving the performance. With the compiled version of a symbol, the design and internal logic of the symbol are directly written in Java code, the execution is faster and the footprint smaller.
- Using symbols for object-oriented programming. You can derive and specialize the symbols using your own Java code.
- Using symbols within the pure JViews Graphic Framework, when the added abstraction level of JViews Diagrammer is not required. This is not possible with interpreted CSS-based symbols.

The following picture illustrates how CSS symbols as well as generated and compiled symbols can be used in applications:


SDM Application
CSS Symbols
Symbol Compiler
Generate & Compile Symbols
Java-based Symbols
Java code, resources, and compiled classes
IlvGrapher (from JViews Framework)
Dashboard Application

The Symbol Compiler is a point-and-click interface to manage the lifecycle of code-based symbols. Taking existing palettes of symbols, it manages code generation, code review, compilation and testing. It also keeps track of your previously generated and compiled symbols to allow you to compile new versions of your symbols.

Getting started with the Symbol Compiler

Explains how to use the IBM® ILOG® JViews Symbol Compiler user interface to translate CSS palette symbols into Java™ classes.

In this section

Running the Symbol Compiler

Explains how to launch Symbol Compiler under Windows® and UNIX® .

Compiling symbols

Explains how to translate a CSS symbol into Java™ code.

Testing compiled symbols

Explains how to execute a compiled symbol and test its behavior.

Running the Symbol Compiler

The Symbol Compiler is a tool that allows you to translate the CSS symbols created with the Symbol Editor into Java™ classes. The symbols that you create in the Symbol Editor use a CSS file to store the information necessary to their graphical representation. A compiled symbol provides improvements in terms of performance, as it is created in Java code instead of CSS, and flexibility, as its features can be extended through the JViews Framework API.

You can run the Symbol Compiler under the Windows® or UNIX® operating systems. The startup conditions are the same in both cases.

To run the Symbol Compiler under Windows:

- In the **Start** menu, select **All Programs>IBM ILOG>IBM ILOG JViews Diagrammer 8.6>Symbol Compiler**.

  IBM® ILOG® is the default program group and may be different if you have installed JViews Diagrammer into a different program group.

—or—

1. Go to the directory `<installdir>/jviews-diagrammer86/bin/symbolcompiler`.

   You need to have a JDK version 1.5 or higher installed on your system and to set its base directory to the JAVA_HOME environment variable.

2. Double-click `run.bat`.

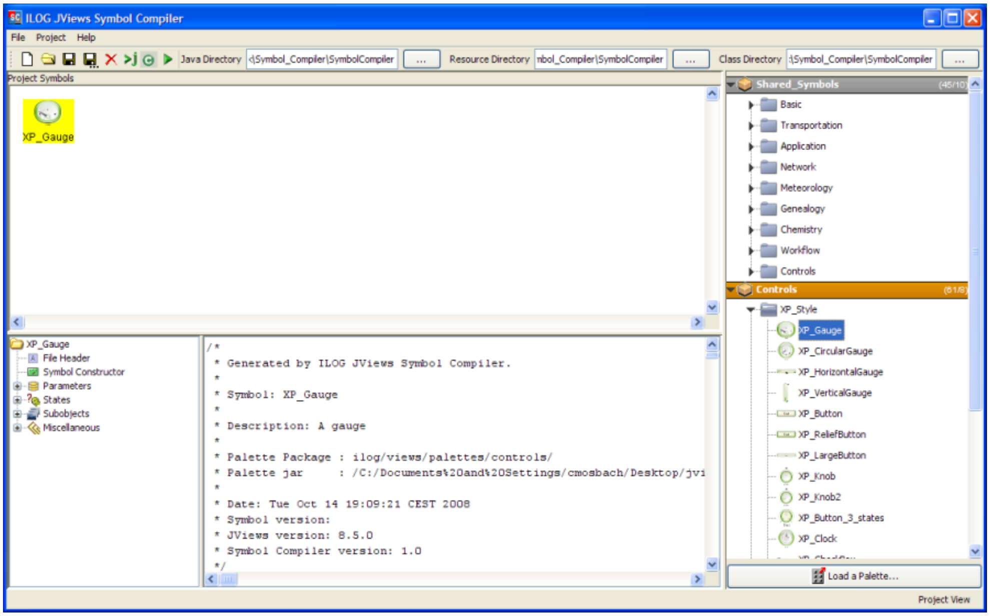
To run the Symbol Compiler under UNIX:

1. Go to the directory `<installdir>/jviews-diagrammer86/bin/symbolcompiler`.

   You need to have a JDK version 1.5 or higher installed on your system and to set its base directory to the JAVA_HOME environment variable.

2. Enter `run.sh`.

When you start the Symbol Compiler, the interface is displayed as shown in the following figure.



Java Directory ‹\Symbol_Compiler\SymbolCompiler ...

Resource Directory mbol_Compiler\SymbolCompiler ...

Class Directory :\Symbol_Compiler\SymbolCompiler ...

```
/*
 * Generated by ILOG JViews Symbol Compiler.
 *
 * Symbol: XP_Gauge
 *
 * Description: A gauge
 *
 * Palette Package : ilog/views/palettes/controls/
 * Palette jar     : /C:/Documents%20and%20Settings/cmosbach/Desktop/jvi
 *
 * Date: Tue Oct 14 19:09:21 CEST 2008
 * Symbol version:
 * JViews version: 8.5.0
 * Symbol Compiler version: 1.0
 */
```

Compiling symbols

When you launch the Symbol Compiler, it opens with the two default palettes: Shared Symbols and Controls.

To compile a symbol:

1. Drag and drop (or double-click) a symbol from an open palette into the Project Symbols pane.

   A tree view in the bottom left pane displays the elements of the symbol class. The Java™ code of the symbol displays in the bottom center pane.

2. Choose **Project > Generate and Compile** or click the C button in the toolbar.

   Symbol Compiler compiles the Java source file into a Java class and places it in the Class Directory specified in the toolbar.

   Any image files are placed in the Resource Directory specified in the toolbar.

To generate the Java source file:

- Choose **Project > Generate Source Files** or click the >j button in the toolbar.

  Symbol Compiler generates the Java code of the symbol in the Java Directory specified in the toolbar

Testing compiled symbols

Once you have compiled a symbol, you can test it in a separate window.

To test a compiled symbol:

1. Click the ▶ button in the toolbar.

   A new window opens and displays the symbol. There are two modes: Test Symbol Interaction and Selection Mode.

2. In **Test Symbol Interaction** mode, play with the symbol.

   For example, if you have a gauge, you can move the needle or the slider and see the values change.

3. In **Selection Mode**, select the symbol to display its property sheet.

   a. Edit some property values.

   b. Switch back to **Test Symbol Interaction** mode to see the result of your changes.

   You can also resize the symbol in selection mode.

Getting to know the Symbol Compiler

Describes the main features of the Symbol Compiler and explains how to use them.

In this section

What you can do with the Symbol Compiler
Lists the main actions you can perform with the Symbol Compiler.

Loading symbol palettes
Explains how to load other symbol palettes.

Displaying the Java source code
Explains how to navigate in the source code of a compiled symbol.

Working with project symbols
Explains what you can do with symbols in the Project Symbols pane.

Output directories
Describes the content of the three output directories that are created when you compile a symbol.

Generating source files
Explains how to create a Java™ source file from a CSS symbol.

Compiling Java classes
Explains how to create Java™ classes from the generated Java source files.

Testing the compiled symbol classes
Explains how to test the behavior of your compiled symbols.

Compound symbols and subsymbols
Explains what compound symbols are and the conditions to use them.

Symbol compiler project file
Describes the content of a project file created in the Symbol Compiler.

Basic settings and predefined CSS functions
Describes the settings that are saved by default when you exit a Symbol Compiler session and how to register CSS functions.

What you can do with the Symbol Compiler

The Symbol Compiler application provides you with a user-friendly environment that allows you to compile your palette symbols.

The Symbol Compiler allows you to:

- Load the symbol palettes you want to use
- Select the symbols that you want to translate into Java™ classes
- Display the Java code of a symbol and use navigation facilities
- Specify the directories for the generated files
- Generate the Java source code and extract used resources
- Compile the generated Java source files
- Test the compiled symbol classes

Loading symbol palettes

The Symbol Compiler works with symbol palettes created with the Symbol Editor.

By default, the Symbol Compiler loads some default palettes provided by IBM® ILOG® JViews. They are displayed in the palette viewer in the right pane of the Symbol Compiler window. The symbols in each palette are grouped into folders by category.

To load other symbol palettes:

1. Click the **Load a Palette** button at the bottom of the palette viewer.
2. Select a palette in the file chooser.
3. Click **Open**.

   The new palette appears in the palette viewer.

Displaying the Java source code

When you select a symbol in the **Project Symbols** pane, its Java™ source code is displayed in the source code viewer in the bottom-center pane of the Symbol Compiler window.

The class tree view

To the left of the source code viewer, a tree shows the different parts of the symbol class.

When you click a node in the tree, the source code viewer shows you the corresponding source.

The source code viewer provides you with incremental search facilities.

To search for a text:

1. Click the source code viewer to put the keyboard focus on it.
2. Type the search string.

A Searching field displays the character sequence you are searching and the first occurrence found is highlighted in the code viewer.

To remove the last entered characters from the search text:

- Type `Backspace`.

To exit search mode and empty the Searching field:

- Type `Esc`.

> **Note**: The search is case sensitive.

The source code viewer allows you to quickly reach a specific line of code.

To access a specific line of code:

1. Click the source code viewer to put the keyboard focus on it.
2. Type `Ctrl+L`.

   A text field appears at the bottom of the source code viewer.
3. Enter the line number.

Working with project symbols

The **Project Symbols** pane displays the symbols you have added in your project.

To select a symbol:

- Click the symbol in the **Project Symbols** pane.

To extend the selection:

- Hold down the `SHIFT` key and click symbols.

To add a symbol to your project:

1. Locate the symbol in the palette viewer.
2. Double-click it or drag and drop it into the **Project Symbols** pane.

To remove a symbol from your project:

1. Select the symbol in the `Project Symbols` pane.
2. Click the Delete Selection button ✕ in the toolbar.

Output directories

The Symbol Compiler generates the Java™ source files, the resource files used by the symbols, and the `.class` files.

The output directories can be specified by using the corresponding fields in the toolbar.

Java directory
: Use the Java Directory field to specify the directory where the Java source code of your symbol and its BeanInfo are generated.

Resource directory
: Use the **Resource Directory** field to specify the directory where the resource files used by your symbols (for example, image files) are generated. These resources are extracted from the palette jars.

Class directory
: Use the **Class Directory** field to specify the directory where the Java compiler (javac) generates the Java `.class` files. The specified directory is added to the class loader class path, so that the compiled symbols can be used in the Symbol Compiler by compound symbols (symbols that use subsymbols) and by the Test window.

Generating source files

For each symbol of your project, the Symbol Compiler generates a Java™ source file and its BeanInfo. If the symbol uses resource files (for example, image files), these files are also extracted form the symbol palette.

To generate the source files of your project:

♦ Choose **Project>Generate Source Files**.

--or—

Click >j in the toolbar.

Compiling Java classes

To use this functionality, you need to have a JDK (1.5 or higher version) installed on your system and set its base directory to the JAVA_HOME environment variable.

To generate the symbol source files and compile their .class:

1. Select the symbol(s) in the **Project Symbols** pane.
2. Choose **Project>Generate and Compile**.

You may not need this functionality and simply integrate the generated Java™ source code in your favorite development environment. However, you need to compile the generated code if you want to test your symbol in the Symbol Compiler, or if your symbol is to be used as a subsymbol in a compound symbol.

Testing the compiled symbol classes

Once you have compiled your symbol classes (see *Compiling Java classes*), you can test your project symbols by using the **Test Compiled Symbols** button in the toolbar.

> **Important**: The compiled symbol class is read only once by the class loader. If a version of the same class is found in the class path and loaded before you compile your symbol, the new version is not loaded.

You can test all the symbols in your project or only the selected symbols. If there is no selected symbol, all the symbols of the project are tested.

To open the Test window:

- Click the **Test Compiled Symbols** icon in the toolbar.

  The Test window displays the instances of the compiled symbols inside an `IlvGrapher` object.

  The **Test Symbol Interaction** mode ▶ of the toolbar allows you to test the interactors of the symbols.

  The **Selection Mode** ↖ allows you to select a symbol, resize it, or inspect its parameters.

Compound symbols and subsymbols

A compound symbol is a symbol that uses other symbols (called subsymbols) as graphical elements.

Before you can add a compound symbol in your project, you need to have the compiled classes of its subsymbols available in the class path of the Symbol Compiler environment. If you plan to have a compound symbol and its subsymbols in the same project, you need to add the subsymbols first and execute the **Generate and Compile** action before adding your compound symbol.

Symbol compiler project file

The Symbol Compiler saves the following information in a project file:

- output directories
- symbol palettes used
- project symbols

When you load a Symbol Compiler project file, it also reads the used symbol palettes that are not yet loaded.

Basic settings and predefined CSS functions

When you exit the Symbol Compiler application, it automatically saves some basic settings, such as the window bounds and the path to the current project. At the next session, the current project is automatically opened. If you want to launch the Symbol Compiler with the default context without opening your previous project, you can run it with the `-clean` command line argument.

Predefined CSS functions

The Symbol Compiler knows how to translate some predefined CSS functions, see Using custom functions. If you want to register your own predefined CSS functions in the Symbol Compiler, extract the file `ilog/views/util/css/cssfunctions.properties` from `jviews-framework-all.jar`. This file contains the settings for predefined CSS functions. Follow the instructions in this file to add your own CSS functions.

Using compiled symbols in applications

Describes the integration of compiled symbols in three different types of application.

In this section

What is a compiled symbol

A compiled symbol is a subclass of `IlvCompositeGraphic`. It can directly be used in an `IlvManager` or an `IlvGrapher`, independently of SDM or `IlvDiagrammer` objects.

You can use a compiled symbol class at different levels:

- in an `IlvDiagrammer` application
- in a runtime Dashboard application
- at the Graphic Framework level

All the CSS rules of the symbol are translated into Java™ code in applicable conditions, with direct function calls to customize the symbol elements. The constant values specified in the Symbol Editor are statically cast or converted to expected types in the generated code.

The benefits in terms of performance can be seen at runtime:

- no need for rule matching operations
- no method invocations through introspection
- less value conversions

In addition to the performance improvements, the compiled symbol, as a Java class, can be extended to offer additional flexibility. You may, for example, use the Symbol Editor to build your symbol with different states based on simple rules that use Boolean parameters. Then you can override the parameter functions in your subclass to do whatever you want in Java using the IBM® ILOG® JViews Graphic Framework SDK.

Generated class of a compiled symbol

For each palette symbol, the Symbol Compiler generates a subclass of `IlvCompositeGraphic` and a corresponding Java™ BeanInfo class to expose the symbol parameters as JavaBean™ properties.

Class name and package

The package of the generated Java class is defined by the symbol palette package name. The class name is the ID of the symbol with the first character converted to uppercase if it is not a capital letter.

A palette symbol can only be compiled if its ID is a valid Java class name and its palette package name is a valid Java package.

Constructor

The generated symbol constructor of the symbol class creates the elements of the composite graphic as they are specified in the Symbol Editor, with the attachments and initial properties, according to the default values of the parameters.

Parameters

Each symbol parameter is translated into a JavaBean property with the same name. All the properties corresponding to the parameters are exported to a BeanInfo class. The getter and setter functions are generated with the additional Parameter prefix. For this reason, the parameter IDs must be valid Java identifiers.

For example, for a string parameter named label, you will have:

- a bean property named label, of type String
- the getter function: `String getLabelParameter()`
- the setter function: `void setLabelParameter(String value)`

The generated code of a parameter setter function updates the properties of the symbol elements that reference the parameter in their value definitions with direct function calls. If the parameter is referenced in rule conditions, the applicable symbol element properties are also updated in the conditional bloc of the generated Java code.

Selectable symbols

If your palette symbol has elements that respond to selection, the generated class implements the `IlvSelectableSymbol` interface and defines the `selectionStateChanged` method that you can call to update the symbol when it is selected or unselected.

Using compiled symbols in an IlvDiagrammer

When you use a symbol in an `IlvDiagrammer` component, the SDM engine of `IlvDiagrammer` customizes by default an `IlvSDMCompositeNode` (a subclass of `IlvCompositeGraphic`) using the CSS of the palette symbol.

Instead of using a CSS-customized SDM composite node, you can use the compiled version of your symbol by enabling an option of your `IlvDiagrammer` SDM engine, as follows:

```
diagrammer.getEngine.setCompiledSymbolAutoLoad(true);
```

This requires that your compiled symbol classes are available in the class path of your application. When creating the graphic object of a node, if the SDM engine finds, in the class path, a Java™ class with the full name corresponding to the palette package name and the palette symbol ID, it creates an instance of the compiled symbol. Otherwise, it creates a regular SDM composite node customized with the symbol CSS. This automatic use of compiled symbols is also applicable to an `IlvDashboardDiagram` component.

Using compiled symbols in a runtime dashboard

A dashboard diagram edited with the Dashboard Editor can be loaded by an `IlvDashboardDiagram` into your application. However, if you do not need any SDM feature, you can have your runtime dashboard instantiated as an `IlvGrapher` containing compiled symbols. IBM® ILOG® JViews Diagrammer provides you with a Compiled Symbol Dashboard Reader that allows you to read a dashboard binary file and create instances of compiled symbol classes in a given grapher, without using SDM and symbol CSS. This is faster than creating symbols and customizing them with their CSS in an `IlvDashboardDiagram`. Furthermore, when you use compiled symbols in a lightweight grapher, the animation can be more than 3 times faster than with an `IlvDashboardDiagram` containing symbols customized by CSS, especially for dashboards that make use of complex symbols.

Reading a dashboard binary file in a grapher

Typical steps to build a dashboard diagram using compiled symbols in a grapher:

1. You create your symbols using the Symbol Editor.
2. You create your dashboard diagram using the Dashboard Editor.
3. You compile your palette symbols using the Symbol Compiler.
4. You integrate the compiled symbol classes in your application class path.
5. You use the Compiled Symbol Dashboard Reader to read your dashboard binary file in an `IlvGrapher` object.

The following function loads a dashboard binary file in a grapher using the compiled symbols:

```
static void readDashboard(URL dashboardURL, IlvGrapher targetGrapher)
 {
   IlvCompiledSymbolDashboardReader reader =
     new IlvCompiledSymbolDashboardReader(targetGrapher);
   try {
     reader.read(dashboardURL);
   } catch (IlvDashboardReaderException e) {
     // Do something
   } catch (IOException e) {
     // Do something
   }
 }
```

Parameter accessors and mappings

In addition to the direct parameter getter and setter functions, the generated symbol class allows you to access your parameters through their IDs and mappings.

For a String parameter named label, you can either call:

```
mySymbol.setLabelParameter(“My label”);
```

or

```
mySymbol.setParameterValue(“label”, “My label”);
```

If you have defined mapping names for your symbol parameters in the Dashboard Editor, you will have to call the `setParameterValue` function with the mapping name instead of the parameter ID.

For example, if the label parameter of your symbol is mapped to name in the Dashboard Editor, you will write the following:

```
mySymbol.setParameterValue(“name”, “My label”);
```

Using compiled symbols at the Graphic Framework level

The Symbol Compiler allows you to use the palette symbols designed with the Symbol Editor in a simple IlvManager or IlvGrapher without SDM model, SDM engine, IlvDiagrammer, or dashboard diagram. You can create instances of the compiled symbol classes and add them to an IlvGraphicBag (for example, IlvManager or IlvGrapher) with the full access to the Graphic Framework API.

You can also extend the generated class and add more elements or more behavior to your symbols.

When programmatically setting your symbol parameter values, keep in mind that the bounding box of your object may change and you may need to do it in an IlvApplyObject. See the documentation of IlvGraphicBag.applyToObject or more information. If the layout of graphical elements in your symbol is changed when you set a parameter value, you have to invalidate it using the IlvCompositeGraphic.invalidate method .