Developing Java Applications: A Tutorial

Sections

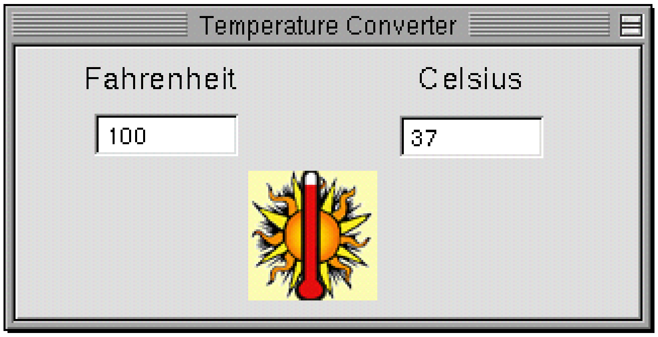
You can now write Mac OS X and Yellow Box for Windows programs in Java™ as well as in Objective-C, C++, C, and PostScript. You can build a Yellow Box application that is written exclusively in Java or that is a mix of Java and another supported language.

This tutorial walks through the basic steps for developing a Java Yellow Box application with the features available in the current release. The feature set will be extended and refined in future releases, and the procedure will thus be even easier.

Fast Track to Java Development summarizes the different steps in Java development for programmers with experience in developing Objective-C Yellow Box applications.

What You'll Learn in This Tutorial

In this tutorial you will build a simple application that converts temperature values between Celsius and Fahrenheit. The application will display a different image depending on the temperature range. Here's what the finished application looks like:



The tutorial has three parts, which you should complete in the following order:

1. **Building a simple application**. Explains how to create a project and a graphical user interface, define a custom controller class, and connect an instance of that class to other objects in the application. It also shows how you must change the source code files generated by Inteface Builder to be valid Java files.
2. **Creating a custom view**. Shows how to create a custom view object using Interface Builder and Project Builder. (The procedure varies from that for controller classes.)
3. **Debugging Java applications**. Illustrates the use of Project Builder and its JavaDebug facility to debug Yellow Box Java applications. It also shows how you can debug projects that contain both Java code and Objective-C code.

Related Concepts

Fast Track to Java Development

The Yellow Box's Java Feature

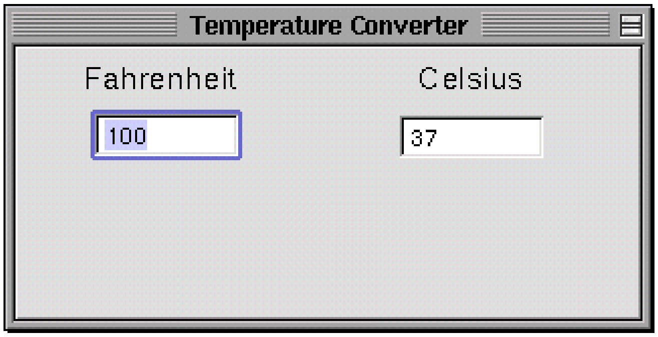
The Java Bridge

Developing 100% Pure Java Applications

Building a Simple Application

Major Tasks:

This part of the tutorial guides you through the creation of a very simple application and in the process teaches you the steps essential to building a Yellow Box application written in Java. By the end of this tutorial you will have created an application called Temperature Converter that looks like this:



This application does exactly what its name suggests, converting Celsius values to Fahrenheit, and vice versa. The user just types a value in one of the text fields and presses the Return key. The converted value is shown in the other field.

Although this application is simple, the experience of putting it together will clarify a few central techniques and paradigms in Yellow Box Java development, among them:

- Creating graphical user interfaces with Interface Builder
- Defining custom Java controller classes with Interface Builder
- Connecting an instance of the controller class with other objects in the application
- Generating source files from Interface Builder definitions and modifying those files to be suitable for Java compilation
- Building the project, after writing the necessary Java code

Important
In future releases you won't have to prepare the generated source files for Java, as you now must do in the current release. Interface Builder will automatically create proper Java files from the definitions of subclasses. Thus some of the details in this tutorial are applicable only to the development environment in the current release.

Creating a Project

Launch Project Builder

Choose the New Command

Name the Project

The procedure for creating an Objective-C project is the same as the procedure for creating a Java project.

Launch Project Builder

Project Builder is the application typically used for creating and managing development projects in the Yellow Box. To launch Project Builder:

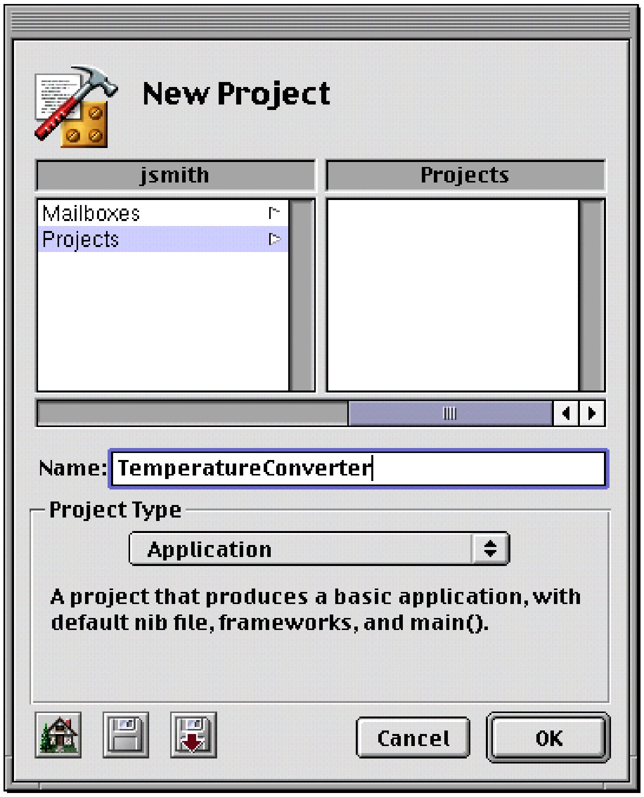
1. Find **ProjectBuilder.app** in **/System/Developer/Applications** and select it.
2. Double-click the icon in the File Viewer.

Choose the New Command

When Project Builder is launched, only its menus appear. To create a project, choose New from the Project menu. This action causes the New Project panel to appear.

Name the Project

All projects must have a name, a location in the file system, and a type designation. The New Project panel allows you to set all these things.



Name: TemperatureConverter

Project Type: Application

A project that produces a basic application, with default nib file, frameworks, and main().

1. Using the file-system browser, navigate to the directory where you want your project to be.

   One convention, as shown in the example, is to have a subdirectory in your home directory named Projects.

2. Type the name of the project in the Name field. For the current project, type the name "TemperatureConverter."

   The name of the project becomes, by default, the name of the project directory and the resulting program.

3. Make sure the project type, as displayed in the pop-up list, is Application.

4. Click OK.

When you click OK, Project Builder creates and displays a project window. After it opens the window, it indexes the project.

Related Concepts

A Project Window

Project Indexing

You might want to look in the project directory to see what kind of files it now contains. Among the project files are:

Makefiles
: Three files contain build information related to the project. The **Makefile** file is maintained by Project Builder itself using the choices you make in inspector panels and elsewhere. Do not modify this file. You can however, customize the **Makefile.preamble** and **Makefile.postamble** files.

Templates
: Templates for both Objective-C and Java souce code files.

English.lproj/
: A directory containing resources localized to your preferred language. In this directory are nib files automatically created for the project.

TemperatureConverter_main.m
: A file, generated for each project, that contains the entry-point code for the application in main().

You'll also see a file named **PB.project**. This file contains information that defines the project (don't modify this file either). You can open the project for subsequent sessions by double-clicking this file.

Related Concepts

What's a Nib File?

Creating the Interface

Open the Main Nib File

Resize the Window

Rename the Window

Put Text Fields in the Window

Set Attributes of the Text Fields

Add Labels for the Text Fields

The procedure for creating a graphical user interface in an Objective-C project is the same as the procedure for creating a graphical user interface in a Java project.

Open the Main Nib File

Each application project, when first created, includes a blank nib file called the "main nib file." This nib file contains the application menu and perhaps one or more windows. Applications automatically load the main nib file when they are launched.

1. Locate the file **TemperatureConverter.nib** in the Interfaces category in the project browser.

   The main nib file has the same name as the project. A default main nib file is also provided for other platforms (in this case, Windows NT).

2. Double-click the icon for nib files (four screws in a plate) in the upper-right corner of the project window.

Related Concepts

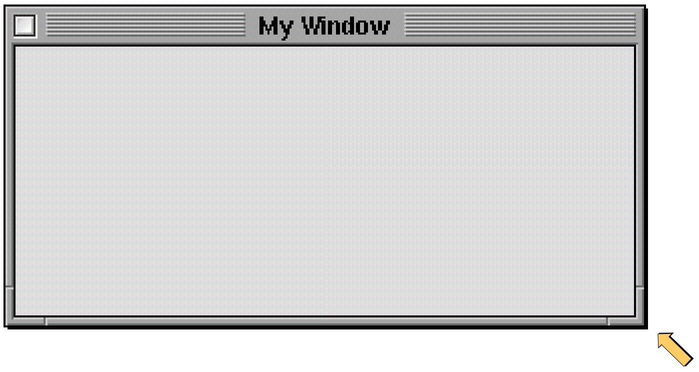
A Project Window

The Windows of Interface Builder

What's a Nib File?

Resize the Window

The window provided for you in the main nib file is too large. To resize a window, drag either lower corner of the window in any direction (up, down, diagonal).



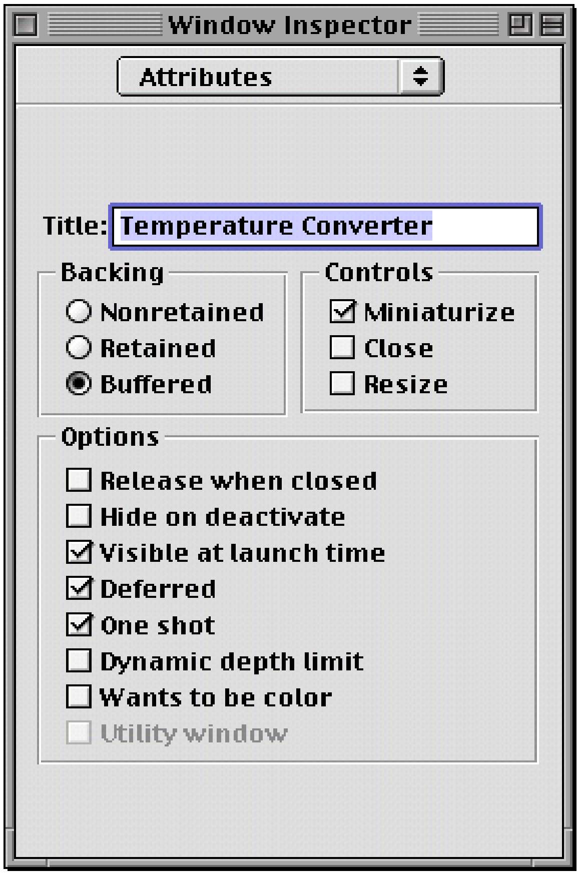
You can resize a window to exact dimensions by entering pixel values in the Size display of Interface Builder's Inspector (see Place and Resize the CustomView Object in the second part of the tutorial for an example of how this is done).

Rename the Window

Windows usually carry distinctive titles in, of course, their title bars. In the Yellow Box, each window in a screen is based on an instance of NSWindow. The title of a window is an attribute of this object. Interface Builder allows you to set the attributes of NSWindows and many other objects in its Inspector.

To set the title of the TemperatureConverter window:

1. Select the window by clicking it.
2. Choose Inspector from Interface Builder's Tools menu.
3. Choose Attributes from the pop-up menu (if it is not already chosen).
4. In the Attributes display of the Inspector, enter "Temperature Converter" in the Title field, replacing "My Window."



Title: Temperature Converter

Backing
- ( ) Nonretained
- ( ) Retained
- (●) Buffered

Controls
- [✓] Miniaturize
- [ ] Close
- [ ] Resize

Options
- [ ] Release when closed
- [ ] Hide on deactivate
- [✓] Visible at launch time
- [✓] Deferred
- [✓] One shot
- [ ] Dynamic depth limit
- [ ] Wants to be color
- [ ] Utility window

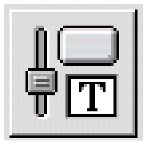
5. Uncheck the Close and Resize checkboxes. These window attributes don't make sense with an application as simple as this.

Put Text Fields in the Window

Interface Builder's Palette window has several palettes full of Application Kit objects. The Views palette holds many of the smaller objects, among which is the text-field object. You now will add two text fields to the TemperatureConverter interface:

1. Select the Views palette.

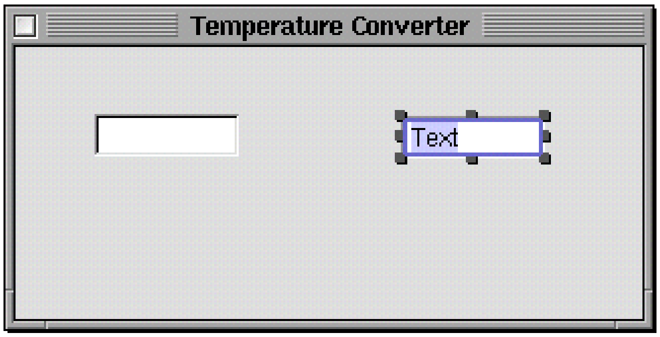
This is what the button for the Views palette looks like this when it's selected:



2. Drag a text field object from the palette toward the Temperature Converter window.
3. Drop the object (by releasing the mouse button) in the window at one of the locations shown below.

   Once you drop the object, you can reposition it in the window by dragging it.
4. Delete the string "Text" from the object.

   To delete the string, double-click to highlight it, then press the Delete key.



5. Repeat steps 2 through 4 for the other text field.

If you need to delete an object in the interface, just select it and press the Delete key.

Set Attributes of the Text Fields

Earlier you set an attribute of the TemperatureConverter window (its title). Now you need to set an attribute of each of the text fields. All objects on Interface Builder's palettes have attributes that you can set through the Inspector.

1. Select a text field.
2. Choose Inspector from the Tools menu.

3. Choose Attributes from the Inspector's pop-up list, if it is not already selected.
4. Click on the radio button labeled "Only on enter" in the Send Action group.

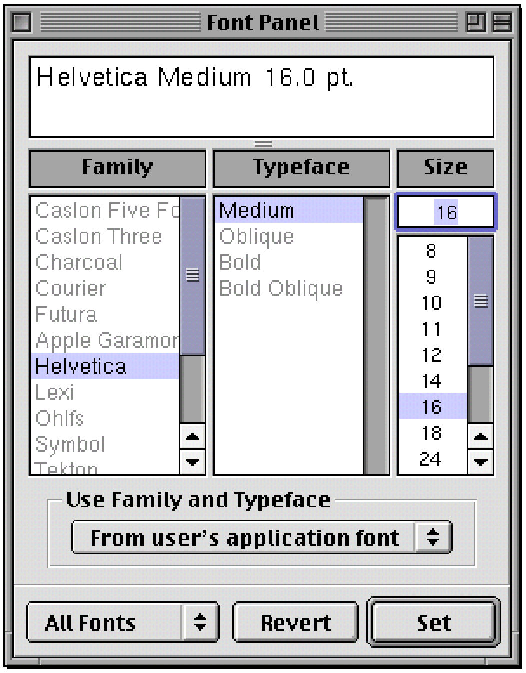
Repeat this sequence for the other text field.

Important
You do not have to set the "Send Action—Only on enter" attribute. If what you want is the the default behavior for text fields—the field sends its action message to its target whenever the insertion point leaves it—then leave the "Only on enter" radio button turned off. When this button is turned on, the text field sends its action message only when the user presses the Enter or Return key while the insertion point is in the field.

Add Labels for the Text Fields

Text fields without labels would be confusing to users, so solve that problem by labeling each field.

1. Drag the "Message Text" object from the Views palette and drop it so it's just above the left text field.
2. Triple-click the text to select all of it.
3. Type "Fahrenheit".
4. Change the font size of the label (since it's too large):
   a. Double-click the label to select it.
   b. Choose Format>Font>Fonts.
   c. Select "From user's application font" from the Use Family and Typeface pop-up list.
   d. In the Font panel, select 16 under Size and click Apply (Set on Yellow Box for Windows).



Repeat the steps for the other label ("Celsius").

Important
Occasionally you should save the nib file containing your work. Now is a good time: choose Save from the Document menu.

Defining the Controller Class

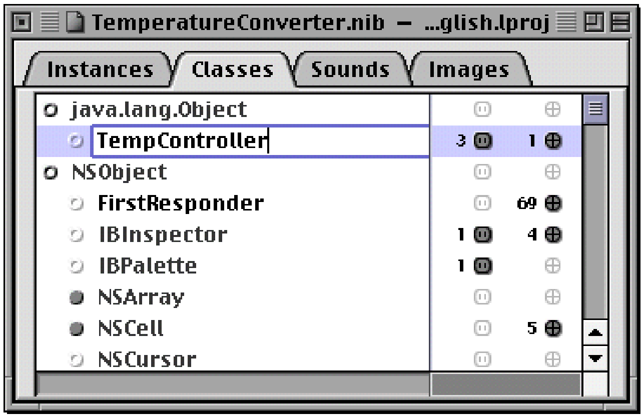
Interface Builder not only lets you construct the user interface of an application from real objects stored on palettes, but lets you partially define a class in terms of its name, its superclass, its outlets, and its actions.

Identify the Class and Its Superclass

You define a custom class using the Classes menu and the Classes display of the nib file window.

1. Click the Classes tab of the **TemperatureConverter nib** file window.
2. Highlight java.lang.Object in the list of classes.
3. Choose Subclass from the Classes menu.

   "MyObject" appears in an editable field below java.lang.Object.
4. Type "TempController" in place of "MyObject" and press Return.



java.lang.Object		
TempController	3	1
NSObject		
FirstResponder		69
IBInspector	1	4
IBPalette	1	
NSArray		
NSCell		5
NSCursor		

Specify the Outlets of the Class

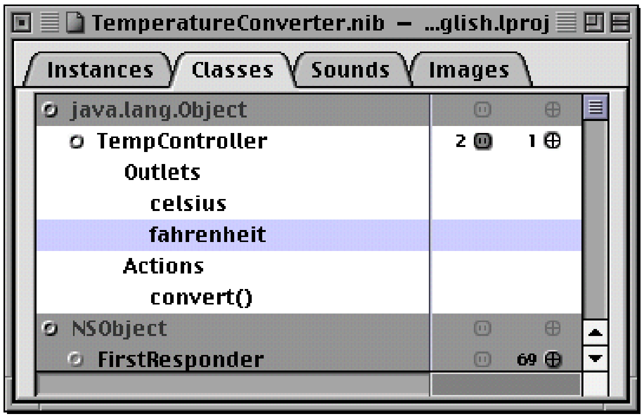
An outlet is a reference one object holds to another object so that it can easily send that object messages; it is an instance variable of type id or IBOutlet. The TempController class has two outlets, one to each of the text fields in the user interface.

1. Click the small electric-outlet icon to the right of TempController in the Classes display.

   Tthe area under TempController expands to include “Outlets” and “Actions.”
2. Select “Outlets.”
3. Choose Add Outlet from the Classes menu.

   You can press the Return key instead of choosing the menu command.
4. Type “celsius” in place of “myOutlet.”

Repeat steps 2 through 4, this time naming the outlet “fahrenheit”.



To collapse the TempController item, click any other class. in the Classes display.

Specify the Action of the Class

An action refers to a method invoked in a target object when a user event occurs, such a the click of a button or the movement of a slider. We want a method in TempController to be invoked whenever the user presses the Return key in a text field.

1. Click the small target icon to the right of TempController in the Classes display.

   The area under TempController expands to include “Outlets” and “Actions.”
2. Select “Actions.”

3. Choose Add Action from the Classes menu.

   You can press the Return key instead of choosing the menu command.

4. Type “convert” in place of “myAction” (parentheses are automatically appended to the method name).

See the illustration above for an example of what things look like when you complete this task.

Related Concepts

The Target/Action Paradigm

Connecting Objects

Interface Builder enables you to connect a custom object to its outlets and to the objects in the user interface that invoke action methods of the custom object. This connection information is stored in the nib file along with the user interface objects, class definitions, and nib resources.

Create an Instance of the Controller Class

Before you can connect a custom object to objects in the user interface, you must create an instance of the object. (This is not a real instance, but a “proxy” instance representing the connections to the object. The real instance is created when the nib file is loaded.)

1. Select the TempController class in the Classes display of the nib file window.
2. Choose Instantiate from the Classes menu.

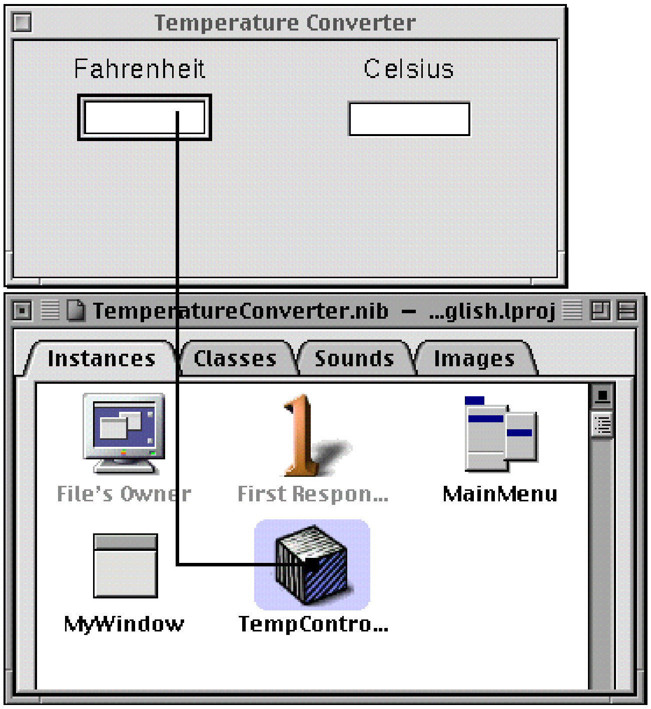
The nib file window automatically changes to the Instances display, and an instance of the TempController class (depicted as a cube) appears in the display.

Connect the Controller to Its Outlets

Follow this Interface Builder procedure to connect the TempController custom object to its outlets:

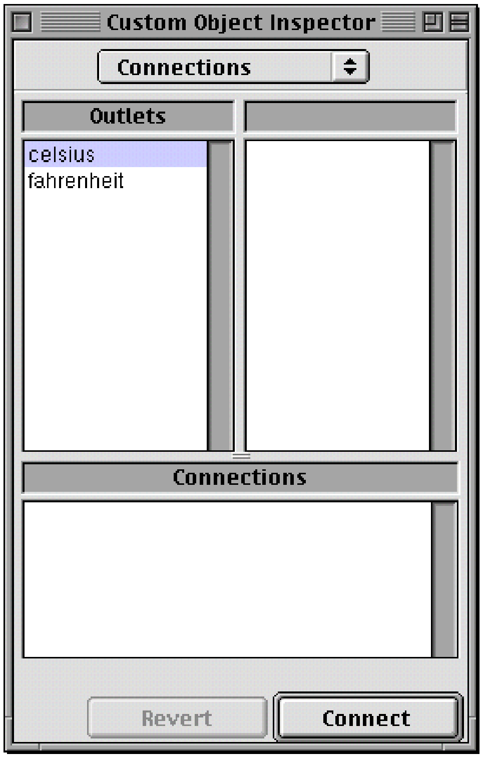
1. Control-drag from the cube representing the custom object to the Fahrenheit text field (the editable field, not the label). A thick black line follows the cursor while you drag.

   "Control-drag" means to hold down the Control key while dragging the mouse (moving it with the mouse button pressed).



2. When a box encloses the Fahrenheit field, release the mouse button.

Interface Builder shows the Connections display of its Inspector. The left column of this display lists the outlets defined by TempController.



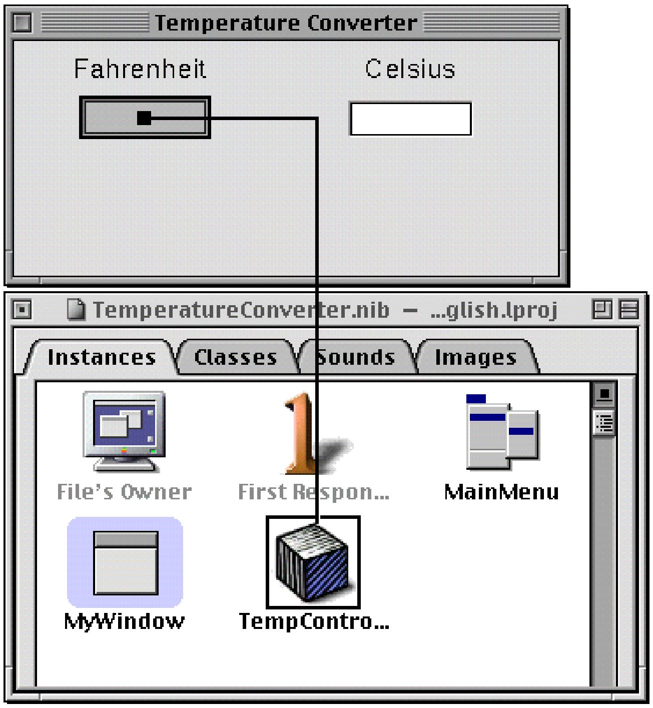
3. Select the fahrenheit outlet.
4. Click the Connect button.

Repeat steps 1 through 4 for the celsius outlet.

Connect the Action of the Controller

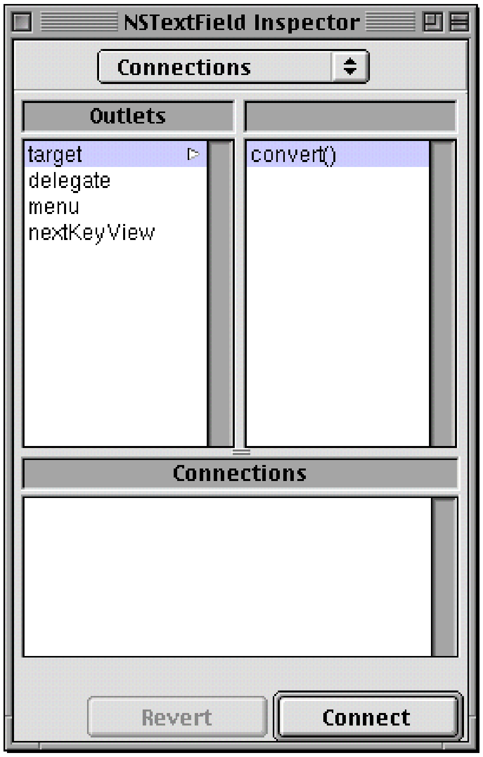
Follow this Interface Builder procedure to connect the action method defined by TempController to the objects that might invoke that method:

1. Control–drag from the the Fahrenheit field (the editable field, not the label) to the cube representing the custom object. A thick black line follows the cursor while you drag.



2. When a box encloses the cube, release the mouse button.

   Interface Builder shows the Connections display of its Inspector. The right column of this display lists the action defined by TempController.



3. Select the convert() action.

   You might first have to click the target item under Outlets to get to the action.

4. Click the Connect button.

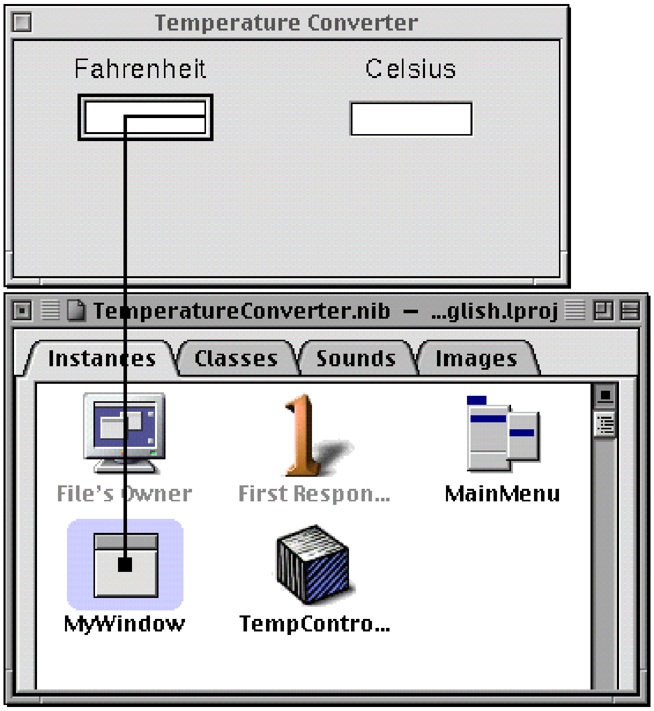
Repeat steps 1 through 4 for the Celsius field.

Connect the Responders

As a convenience to users, you want the insertion point to be in a certain field after the application is launched. For the same reason (convenience), you want users to be able to switch between the fields without having to use the mouse—they should be able to tab between the fields. You can specify this behavior entirely in Interface Builder:

1. Click the Instances tab of the nib file window.

2. Control–drag a connection line from the window icon to the Fahrenheit field.



3. In the Connections display of the inspector, select initialFirstResponder.
4. Click the Connect button of the inspector.
5. Control–drag a connection line from the Fahrenheit field to the Celsius field.
6. In the Connections display, select nextKeyView and click Connect.
7. Control–drag a connection line from the Celsius field to the Fahrenheit field.
8. In the Connections display, select nextKeyView and click Connect.

What have you just done? You've specified the sequence of responder objects in the user interface that are to receive the focus of keyboard events when users press the Tab key.

Related Concepts

The View Hierarchy and the First Responder

Test the User Interface

You can now test the user interface you've constructed with Interface Builder. Save the nib file and choose Test Interface from the Document menu. Interface Builder goes into test mode and the window and text fields you've just created behave as they would in the final application—except, of course, there is yet no custom behavior.

Notice that the insertion point is initially in the Celsius field. Press the Tab key; note how the insertion point jumps between the fields. Type something into one of the fields, then select it and choose Cut from the Edit menu. Click in the other text field and choose Paste from the Edit menu. These are but a couple of examples of features you get in any application with little or no work on your part.

Implementing the Controller Class

You're now ready to generate source-code template files from the nib file you've created with Interface Builder. After that, you'll work solely with the other major development application, Project Builder.

Generate the Source Code Files

To generate the source-code templates files for TempController, in Interface Builder:

1. Click the Classes tab in the nib file window.
2. Select the TempController class.
3. Choose Create Files from the Classes menu.
4. Respond to the query "Create TempController.java?" by clicking Yes.

5. Respond to the query "Insert file in project?" by clicking Yes.

Interface Builder creates a **TempController.java file** and puts it in the Classes category of Project Builder. You can now quit Interface Builder (or, better still, hide it) and click in Project Builder's project window to bring it to the front.

Modify the Source Code Files

In Project Builder, perform the following steps to modify the generated files:

1. Click the Classes category in the left column of the project browser.
2. Click **TempController.java** in the second column.

   The following code is displayed in the code editor:

```
import com.apple.yellow.application.*;
import com.apple.yellow.foundation.*;
public class TempController {
        Object celsius;
        Object fahrenheit;
        public void convert(Object sender){
}
}
```

3. Modify the above code so that it looks like this:

```
import com.apple.yellow.application.*;
public class TempController {
        NSTextField celsius;
        NSTextField fahrenheit;
        public void convert(NSTextField sender) {
    }
}
```

Why is this modification necessary? Java is a strongly typed language and has no equivalent for the Objective-C dynamic object type `id`. When Interface Builder generates source-code files for Objective-C classes, it gives `id` as the type of outlets and as the type of the object sending action messages. This `id` is essential to the method signature for outlets and actions. However, when it generates Java source-code files, it substitutes the static Java type Object for `id`.

Related Concepts

[A Project Window]

Implement the convert Method

Finally implement the convert method in Java, as shown here:

```
import com.apple.yellow.application.*;
public class TempController {
        NSTextField celsius;
        NSTextField fahrenheit;
        public void convert(NSTextField sender) {
                if (sender == celsius) {
                        int f = (int)((9.0/5.0 * celsius.intValue()) + 32);
                        fahrenheit.setIntValue(f);
                } else if (sender == fahrenheit) {
                        int c = (int)((fahrenheit.intValue()-32) * 5.0/9.0);
                        celsius.setIntValue(c);
                }
    }
}
```

You can freely intermix Yellow Box and native Java objects in the code. And you can use any Java language element, such as the try/catch exception handler.

Write a Patch for Windows Applications

If you're writing the application to run on Yellow Box for Windows, you must write some code that works around a problem with the Java virtual machine (VM) on Windows. Because the way the VM implements security-manager features, it will otherwise not allow any native method to be loaded through a class loader.

The best place to put the code shown below is in:

- A static initializer in the principal class
- Your application delegate's applicationDidFinishLaunching method

The code to add is the following:

```
try {
        com.apple.security.NullSecurityManager.installSystemSecurityManager();
} catch (Exception e) {
        // Can't install it
}
```

If the exception is raised, the “null” security manager cannot be installed and thus native code might not be invoked properly.

Building and Running the Application

You’ve completed the work required from you for the Temperature Converter project. Now it’s Project Builder’s turn to work.

Build the Project

To build the project:

1. Click the Build icon in the project window to display the Build panel.

2. Click the same icon in the Build panel.

You can also press Command–Shift–B to start building directly, bypassing step 1.

Project Builder begins compiling and linking the project code. It reports progress in the Build panel. If there are errors, Project Builder lists them in the upper part of the display area. Click a line reporting an error to have Project Builder scroll to the site of the error in the code editor.

Related Concepts

The Build Panel

The build target for Application projects is, by default, “app” (for application). By clicking the checkmark button on the Build panel, you can bring up the Build

Options panel, where you can set the target to “debug” (which creates an executable with extra symbols for debugging) or set other per-build parameters.

Launch and Test the Project

Of course, once the application has been built, you’ll want to launch the application to see if it works as planned. You have at least two ways of doing this:

- Locate the file **TemperatureConverter.app** in the project directory. Double-click this file to launch the application.
- Click the Launch button on the project window to display the Launch panel.

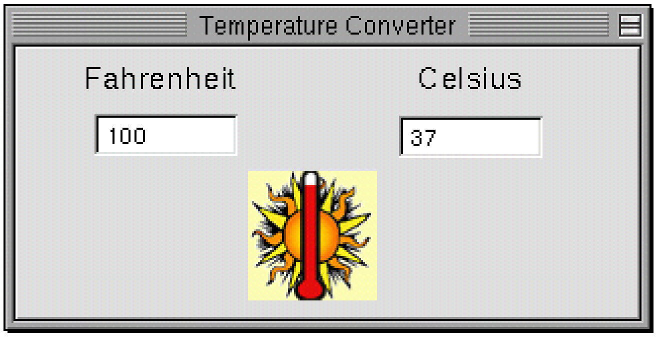
  This button looks like a monitor:

Then click the Launch button (same icon) on the Launch panel to launch the application.

Creating a Custom View Class

Major Tasks:

This section of the tutorial describes the basic steps for creating a custom view and, more specifically, shows how to create a sublcass of an Application Kit class that itself inherits from NSView. In this section, you will add a custom "image view" to the user interface you created in the first part of this tutorial. This custom object will respond to messages from the controller object, TempController, and change its image depending on the temperature entered. Here's what the final TemperatureConverter application will look like:



The behavior that your custom view object adds to its superclass, NSImageView, is trivial. You could just as well accomplish the same behavior by sending messages to an "off-the-shelf" instance of NSImageView. But the subclass illustrates the basic procedure for making subclasses of Yellow Box classes that don't inherit from java.lang.Object.

Creating a Subclass of NSView summarizes the procedure and provides example code for creating a subclass of NSView whose instances can draw themselves. This example subclass can replace the one you will create in this section, because it draws graphical shapes instead of displaying images when the temperature changes to another range.

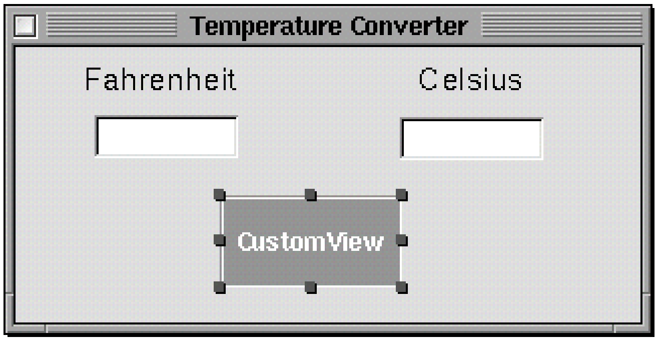
Defining the Custom View Subclass

Place and Resize the CustomView Object

The CustomView object on Interface Builder's Views palette represents an instance of any custom subclass of NSView or of any Application Kit class that inherits from NSView. The CustomView object lets you specify the basic attributes of all view objects: their location in a window and their size.

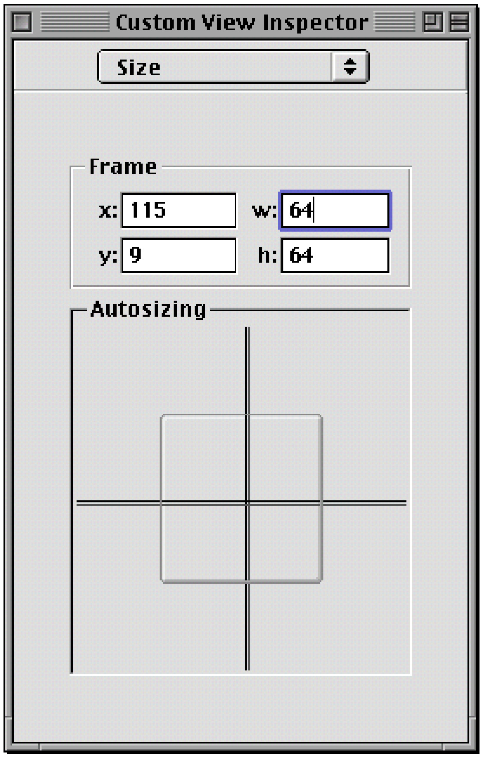
1. Drag the CustomView object from the Views palette and drop it in the window.

   Center it in the window beneath the text fields.



2. Resize the CustomView using the Size inspector.

   Choose Inspector from the Tools menu, select the Size display, and enter 64 in both the width (w) and height (h) fields.



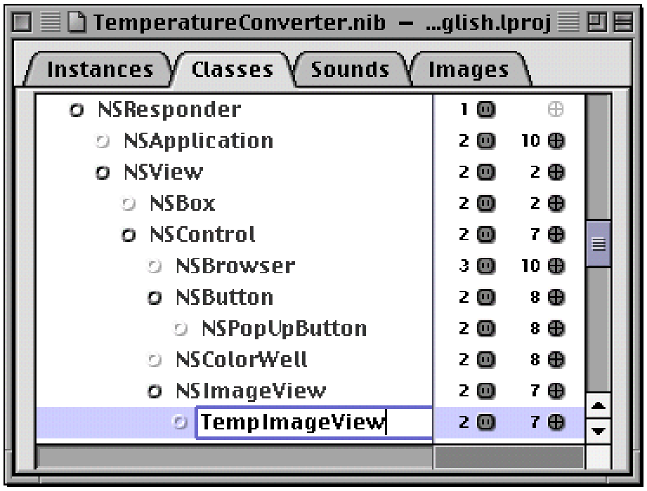
x: 115
w: 64
y: 9
h: 64

Specify the Subclass

As you did earlier with the controller object TempController, you must provide the name and superclass of your custom class. But now, instead of inheriting from java.lang.Object, your class inherits from an Application Kit class.

1. In the Classes display of the nib file window, select the NSImageView class.

   If a class in the display has a filled-in circle next to it, you can click the circle to reveal the subclasses of that class. The path you want to follow is this: NSObject, NSResponder, NSView, NSControl, NSImageView.



NSResponder	1	
NSApplication	2	10
NSView	2	2
NSBox	2	2
NSControl	2	7
NSBrowser	3	10
NSButton	2	8
NSPopUpButton	2	8
NSColorWell	2	8
NSImageView	2	7
TempImageView	2	7

2. Choose Subclass from the Classes menu.
3. Name the class "TempImageView".

There is no need to specify any outlets or actions for this class.

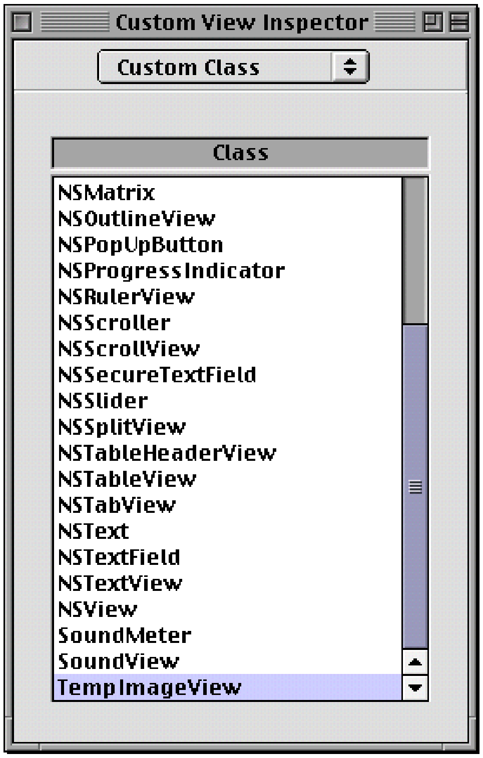
Important
Currently, Interface Builder lists the Objective-C set of Yellow Box classes in its Classes display. This set does not map exactly to the Java set. The release notes for the Yellow Box Java APIs (**JavaAPI.html** in **/System/Documentation/Developer/YellowBox/ReleaseNotes**) describes which Objective-C Foundation and Application Kit classes and protocols were exposed as

Java classes and interfaces, and which Java classes are new. For those Objective-C classes that were not exposed, it indicates the JDK counterparts that you can use instead.

Assign the Class to the CustomView

Unlike custom controller classes, where you use the Classes>Instantiate command to make an instance, you make an instance of a custom view in Interface Builder by assigning the class to the CustomView object.

1. Select the CustomView object.
2. Select the Custom Class display of the inspector.
3. Select the TempImageView class in the list provided by that display.



NSMatrix
NSOutlineView
NSPopUpButton
NSProgressIndicator
NSRulerView
NSScroller
NSScrollView
NSSecureTextField
NSSlider
NSSplitView
NSTableHeaderView
NSTableView
NSTabView
NSText
NSTextField
NSTextView
NSView
SoundMeter
SoundView
TempImageView

Notice how the title of the custom view object changes to “TempImageView.”

Connecting the View Object

The TempImageView itself has no outlets or actions, but the controller object TempController needs to communicate with it to tell it when the temperature value changes. One additional outlet in TempController is needed for this purpose.

Specify a Controller Outlet

You can always add an action or outlet to an existing custom class. Just make sure the header and implementation files of the class (if created) reflect the new outlet or action.

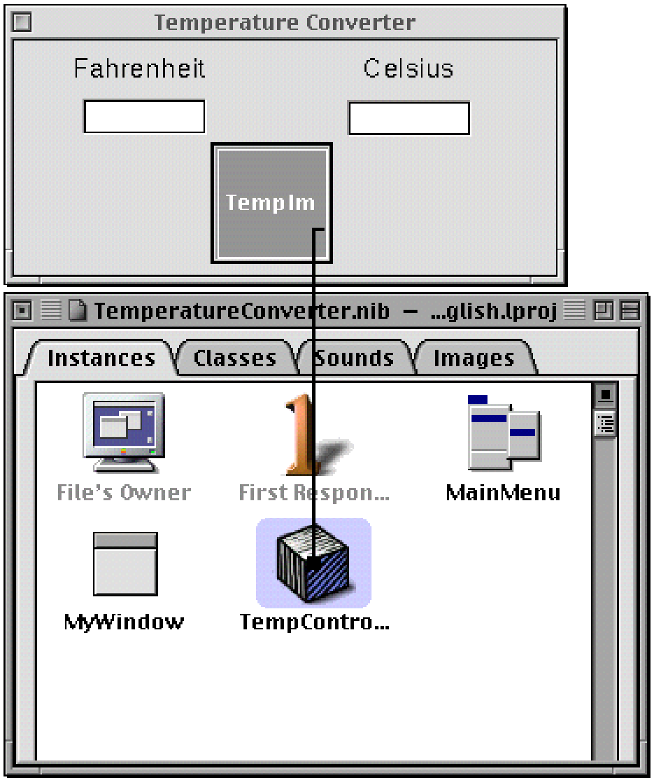
1. Select the TempController class in the Classes display of the nib file window.
2. Click the electrical-outlet icon next to the class.
3. Choose Add Outlet from the Classes menu (or just press Return).
4. Type the name of the outlet: “tempImage”.

Before you move on the next step, be sure to collapse the listing of outlets and actions by clicking another class.

Connect the Instances

You’ve already created an instance of TempController; all you need to do now is connect it to the TempImageView instance through the tempImage outlet.

1. Click the nib file window and the application window to bring them both to the front of the screen.
2. Drag a connection from the TempController instance in the Instances display to the custom view object (TempImageView) in the application window.



3. Select the tempImage outlet in the Connections view of the inspector and click Connect.
4. Save the nib file.

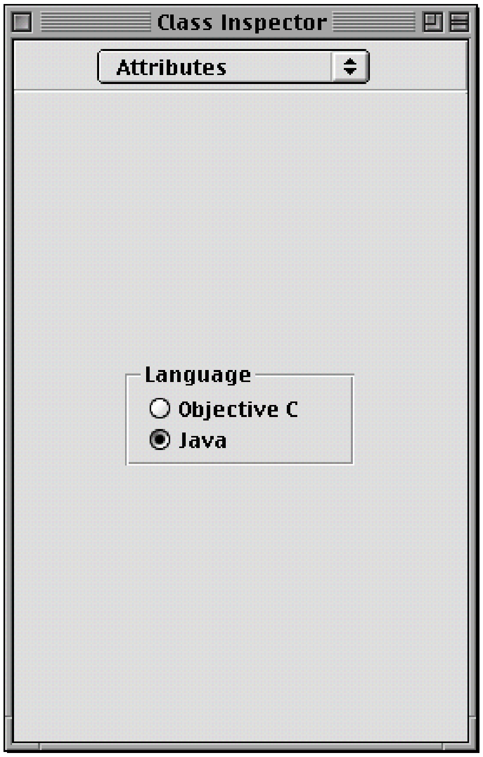
Implementing the Custom View

Call the Image-Setting Method

Generate the .java File

The classes under NSObject in the Classes display of the nib file window represent, in most cases, both Java and Objective-C versions of the same class. When you create source code files from your nib-file definitions, you must specify which language you want.

1. Select TempImageView in the Classes display of the nib file window.
2. Select the Attributes display of the inspector.
3. Click the Java radio button.



Language
( ) Objective C
[✓ Java]

4. Choose Create Files from the Classes menu.

5. Respond Yes to both confirmation prompts.

Implement the Constructor

The constructor for TempImageView loads image files from the application's resources, converts them to NSImage objects, and assigns these to instance variables. (You'll add these images to the project later in this tutorial.) It also sets certain inherited attributes of the image view. The following procedure approaches the implementation of this constructor in three steps.

1. Open **TempImageView.java** in Project Builder's project browser by clicking the filename listed in the Classes category.
2. Add the instance variables for the three NSImages as shown here:

```
/* TempImageView */

import com.apple.yellow.application.*;
import com.apple.yellow.foundation.*;

public class TempImageView extends NSImageView {
   protected NSImage coldImage; // add this
   protected NSImage moderateImage; // add this
   protected NSImage hotImage; // add this
```

3. Load the images, create NSImages, and assign them to instance variables.

```
public TempImageView(NSRect frame) {
                // load images
      super(frame);
      coldImage = new NSImage("Cold.tiff", true);
      if (coldImage == null) {
         System.err.println("Image Cold.tiff not found.");
      }
      moderateImage = new NSImage("Moderate.tiff", true);
      if (moderateImage == null) {
         System.err.println("Image Moderate.tiff not found.");
      }
      String h = NSBundle.mainBundle().pathForResource("Hot", "tiff");
      if (h != null) {
         hotImage = new NSImage(h, true);
      } else {
        System.err.println("Image Hot.tiff not found.");
    }
```

There are a few things to observe about the above excerpt of code:

- TempImageView's constructor is based on NSImageView's NSImageView(NSRect) method, so the first thing done is a call to super's constructor.
- It uses NSImage's constructor NSImage(String, boolean) to locate the specified image resource in the application bundle and create an NSImage with it.
- For the third image (just to show how it can be done differently) NSBundle's pathForResource(String, String) method is called to locate the image and return a path to it within the application bundle. This path is used in the NSImage(String, boolean) constructor.
- The error handling in this constructor is rudimentary. In a real application, you would probably want to implement something more useful.

4. Set attributes of the image view:

```
        setEditable(false);
        setImage(moderateImage);
        setImageAlignment(NSImageCell.ImageAlignCenter);
        setImageFrameStyle(NSImageCell.ImageFrameNone);
        setImageScaling(NSImageCell.ScaleProportionally);
    }
```

The foregoing procedure might lead you to wonder how you can learn more about the methods of a Yellow Box class, especially their arguments and return types. Mac OS X and Yellow Box for Windows provide a tool to help you, JavaBrowser. They also include a version of the Java Yellow Box API reference documentation which documents many classes; for those that are not documented, a "skeletal" class specification displays the prototype for each method and includes an HTML link to the method's Objective-C counterpart in the reference documentation.

Implement the Image-Setting Method

TempImageView has one public method that TempController calls whenever the user enters a new temperature value: the tempDidChange method. Implement this method as shown here:

```
public void tempDidChange(int degree) {
    if (degree < 45) {
        setImage(coldImage);
    } else if (degree > 75) {
        setImage(hotImage);
    } else setImage(moderateImage);
}
```

These ranges are completely arbitrary, but could be influenced by a California climate. If you have lower or higher thresholds for hot and cold temperatures, you can specify your own ranges.

Call the Image-Setting Method

In the convert method of TempController, call TempImageView's tempDidChange method after converting the entered value. Copy the following example:

```
public void convert(NSTextField sender) {
        if (sender == celsius) {
                int f = (int)((9.0/5.0 * celsius.intValue()) + 32);
                fahrenheit.setIntValue(f);
        } else if (sender == fahrenheit) {
                int c = (int)((fahrenheit.intValue()-32) * 5.0/9.0);
                celsius.setIntValue(c);
        }
        tempImage.tempDidChange(fahrenheit.intValue()); // add this
}
```

You must also add the tempImage outlet you defined earlier in Interface Builder as an instance variable in **TempController.java**.

Completing the Application

Add Images to the Project

The TempImageView object must, of course, have images to show. Ready-made "climate" images come provided for this tutorial. You must add these image files to the application.

1. In Project Builder, double-click the Images category in the project browser.
2. In the Add Images panel, navigate to the following directory:

   /System/Documentation/Developer/YellowBox/TasksAndConcepts/JavaTutorial/ApplicationImages

   If you are on a Yellow Box for Windows system, the above path starts with the value of NEXT_ROOT rather than **/System**.
3. Add the following images: **Hot.tiff**, **Cold.tiff**, **Moderate.tiff**.

   Shift-click each file to select all images, then click OK to add them to the project.
4. Choose Save from the Project menu.

Build the Project

Now you're ready to build the project. Click the Build button in the project window, then the same button in the Build panel. You can circumvent these buttons by pressing Command–Shift–B. If there are errors in the code, they will appear in the two lower displays of the Build panel. Click a line describing an error in the upper display to go to the line in the code containing the error; fix the error and rebuild.

Related Concepts

The Build Panel

Test Drive the Application

Launch the TemperatureConverter application and see what it can do; this includes not only what you specifically programmed it to do, but the behavior the application gets "for free."

- Enter a low temperature value in the Fahrenheit field and press Return.

  The Celsius field displays the converted temperature and the image changes to the "cold" picture.

- Enter a high temperature value in the Celsius field and press Return.

  The Fahrenheit field displays the converted temperature and the image changes to the “hot” picture.

Now check out the “free” behavior.

- Click the window of another application or anywhere in the workspace.

  The TemperatureConverter window loses key status (and as a result its title bar loses its detail) and it might become tiered beneath other windows on your screen. TemperatureConverter is no longer the active application.

- Click the TemperatureConverter window.

  It is brought to the front tier of the window system and is made key.

- Choose Hide TemperatureConverter from the Application menu (the menu at the far right of the menu bar).

  The TemperatureConverter window disappears from the screen.

- In the same Application menu, choose TemperatureConverter from the list of applications currently running on your system.

  The TemperatureConverter window reappears.

- Select a number in one of the text fields, choose Copy from the Edit menu, select the number in the other text field, and choose Paste from the Edit menu.

  The number is copied from one field to the other.

- Select a number again and choose a suitable command from the Services menu, such as Make Sticky.

  The Services menu lists those applications that can accept selected data from your application and process it in specific ways. When you choose a Services command, the application associated with the command starts up (if it is not already running) and processes the selected number. (In the case of Make Sticky, the number appears in a Stickies window.)

- Chose Quit from the File menu.

Creating a Subclass of NSView

If you have completed the tutorial so far, you're done. You need not go any further—unless you would like to try a more interesting and challenging variation of the TempImageView class you created earlier. In this "extra credit" part of the tutorial, you will create a class for objects that know how to draw themselves and know how to respond to user events. These classes are custom subclasses of NSView.

Custom subclasses of NSView are ususally constructed differently than subclasses of other Application Kit classes because the custom NSView subclass is responsible for drawing itself and, optionally, for responding to user actions. Of course, you can do custom drawing in a subclass that doesn't inherit directly from NSView, but usually instances of these classes draw themselves adequately. This section describes in general terms what you must do to create a custom NSView subclass.

Define a Custom Subclass of NSView

The differences are slight between the Interface Builder procedures for defining a direct subclass of NSView and for defining a subclass of any Application Kit class that inherits, directly or indirectly, from NSView. The following is a summary of the required procedure in Interface Builder:

1. Drag a CustomView object from the Views palette and drop it in the window.
2. Resize the CustomView object to the dimensions you would like it to have.
3. Select NSView in the Classes display of the nib file window, choose Subclass from the Class menu, and name your subclass.
4. Add any necessary outlets or actions.
5. Assign the class you defined to the CustomView object.

   Do this by selecting the object and selecting the class in the Classes display of the inspector.

6. Make any necessary connections.
7. Generate the "skeletal" **.java** file; before you choose the Classes>Create Files command, be sure to select first the class and then Java in the Attributes display of the inspector.

If you are unsure how to complete any of these steps, refer to the Defining the Custom View Subclass and Connecting the View Object sections of this tutorial.

Implement the Code for a Custom NSView

You can implement your custom NSView to do one or two general things: to draw itself and to respond to user actions. The basic procedures for these and related tasks are given below.

Important
The information provided in this section barely scratches the surface of the concepts related to NSView, including drawing, the imaging model, event handling, the view hierarchy, and so on. This section intends only to give you an idea of what is involved in creating a custom view. For a much more complete picture, see the description of the NSView class in the Objective-C API reference.

Drawing

All objects that inherit from NSView must override the drawRect method to render themselves on the screen. The invocation of NSView's display method, or one of the display variants, leads to the invocation of drawRect. Before drawRect is invoked, NSView "locks focus," setting the Window Server up with information about the view, including the window device it draws in, the coordinate system and clipping path it uses, and other PostScript graphics state information.

In the drawRect method, you must write the code that transmits drawing instructions to the Window Server. The drawRect method has one argument: the NSRect object defining the area in which the drawing is to occur (usually the bounds of the NSView itself or a subrectangle of it). The range of options the Java Yellow Box APIs provide is currently more limited than on the Objective-C side, which has the whole suite of PostScript client-side functions and operators. For drawing in Java, you can use the following classes:

- NSBezierPath offers methods for constructing straight or curved lines, rectangles, ovals, arcs, and polygons with bezier paths.
- NSAffineTransforms has methods for translating, rotating, and resizing graphical objects, such as those created with NSBezierPath.
- The static methods of the NSGraphics class draw rectangles, including buttons of various styles. They also perform bitmap operations and provide various information about the Window Server and graphics context.
- Foundation's geometry classes—NSRect, NSSize, and NSPoint (and their mutable variants)—help you to compute the location and size of graphical objects.
- NSColor and NSFont, among other classes, have methods that directly set a parameter of the current graphics context.

Invalidating the View

With each cycle of the event loop, the Window Server ensures that each NSView in a window that requires redrawing is given an opportunity to redisplay itself. Besides implementing drawRect to draw your custom NSView, your application must indicate that an NSView requires redrawing when data affecting the view changes.

This indication is called "invalidation." Invalidation marks an entire view or a portion of a view as "invalid," and thus requiring a redisplay. NSView defines two methods for marking a view's image as invalid: setNeedsDisplay, which invalidates the view's entire bounds rectangle, and setNeedsDisplayInRect, which invalidates a portion of the view.

You can also force an immediate redisplay of a view with the display and displayRect methods, which are the counterparts to the methods mentioned above. However, you should use these and related display... methods sparingly, and only when necessary. Constant forced displays can markedly affect application performance.

You should never invoke drawRect directly.

Event Handling

If an NSView expresses a willingness to respond to user events, it is made the potential recipient of any event detected by the window system. The view then just must implement the appropriate NSResponder method (or methods) that correspond to the event the view is interested in. (NSView inherits from NSResponder.)

What this means in practical terms is that an NSView must at a bare minimum do two things:

- Override NSResponder's acceptsFirstResponder method to return true.
- Implement an NSResponder method such as mouseDown, mouseDragged, or keyUp. The argument of each of these methods is an NSEvent, which provides information related to the event.

See the NSResponder and NSEvent class descriptions in the API reference for further information.

An Example

The TemperatureView class is similar to the TempImageView class implemented in the second part of the tutorial. Instead of displaying a different image when the temperature changes to a certain range, it draws a circle of a different color. To illustrate basic event handling, the TemperatureView class changes the thickness of the view's border each time the user clicks the view.

```
/* TemperatureView */

import com.apple.yellow.application.*;
import com.apple.yellow.foundation.*;

public class TemperatureView extends NSView {
   protected NSBezierPath sun;
   protected int temperature;
   protected int thickness;

   static public final int SpringSun=0;
   static public final int SummerSun=1;
   static public final int WinterSun=2;

   public TemperatureView(NSRect frame) {
      super(frame);

      float shortest = frame.width() >= frame.height()?frame.height():frame.width();
      NSRect rect;
```

```
        NSColor color;

        shortest *= 0.75;
        rect = new NSRect(((frame.width() - shortest) /2),
                                  ((frame.height() - shortest) /2), shortest, shortest);
        sun = NSBezierPath.bezierPathWithOvalInRect(rect);
        thickness = 1;
    }

    public void drawRect(NSRect frame) {
        NSColor color;
        if (temperature == WinterSun) {
            color = NSColor.lightGrayColor();
        } else if (temperature == SummerSun) {
            color = NSColor.orangeColor();
        } else {
            color = NSColor.yellowColor();
        }
        color.set();
        sun.fill();
        NSGraphics.frameRectWithWidth(frame, (float)thickness);
    }

    public void tempDidChange(int degree) {
        if (degree < 45) {
            temperature = WinterSun;
        } else if (degree > 75) {
            temperature = SummerSun;
        } else temperature = SpringSun;
        setNeedsDisplay(true);
    }

    public void mouseDown(NSEvent e) {
        if (thickness == 3) {
            thickness = 1;
        } else {
            thickness++;
        }
        setNeedsDisplay(true);
    }
    public boolean acceptsFirstResponder() {
        return true;
    }

}
```

Debugging Java Applications

Major Tasks:

This part of the tutorial show you the basic steps for debugging Java Yellow Box applications using the facilities provided by Project Builder. It tells you the steps you must take to prepare for debugging, illustrates several debugging commands, and describes how to debug applications built from Java and Objective-C code.

Important
Project Builder *might* not support the debugging of 100% Pure Java applications in the current release; check the release notes to verify the status of this feature. You can use **/usr/bin/jdb** for debugging 100% Pure Java applications.

Project Builder uses the Java Debugger for debugging Java Yellow Box executables. The Java Debugger is a tool that uses a customized subset of **jdb** commands. Because it is implemented as a set of threads in the Java VM, it is always active when the VM is running. Thus you can interact with the Java Debugger even when the target is running (unlike **gdb**, which requires that the target be stopped before it can process commands).

Project Builder integrates the Java Debugger and **gdb** so that, with projects that consist of Java code and other code (Objective-C, C, or C++), you can use both debuggers in the same session, switching between them as necessary. Currently, there is no mixed-stack backtrace; in other words, the stack backtrace when the Java Debugger is used shows only Java frames, and the **gdb** stack backtrace shows only Objective-C, C, and C++ frames.

Preparing to Debug an Application

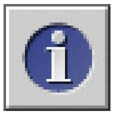
Before you can debug a Yellow Box Java application, you must build the application with the proper debugging symbols and then run the Java Debugger, after which you can set breakpoints and begin debugging.

Install Debug Libraries on Windows NT

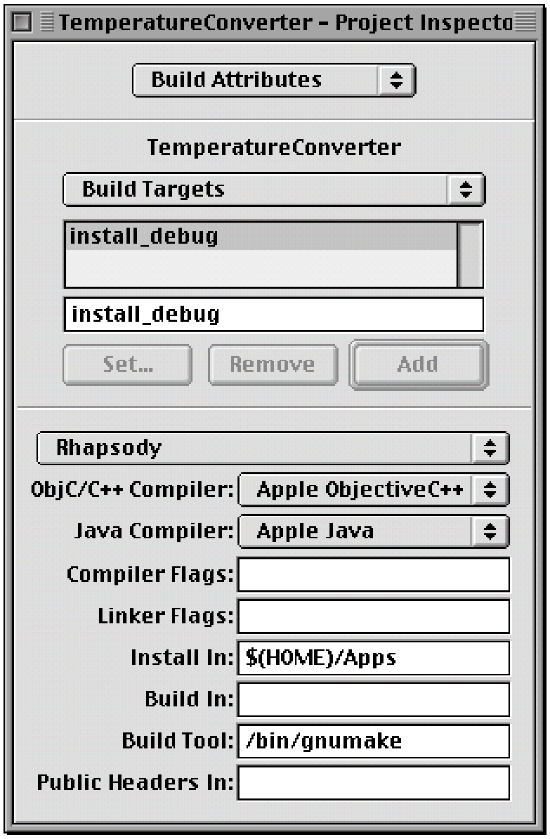
To debug "wrapped" frameworks of yours on Yellow Box for Windows systems, you must first install the debug versions of the Yellow Box libraries (DLLs).

> **Important**
> If you have a Mac OS X system, you do not need to complete this task. Skip ahead to the next task, Build for Debugging.

1. Click the Inspector button on the main window:



2. In the Build Attributes display of the Project Inspector, select Build Targets from the middle pop-up menu.
3. Enter "install_debug" in the text field and click Add.


TemperatureConverter - Project Inspecto

Build Attributes

TemperatureConverter

Build Targets

install_debug

install_debug

Set... Remove Add

Rhapsody

ObjC/C++ Compiler: Apple ObjectiveC++

Java Compiler: Apple Java

Compiler Flags:

Linker Flags:

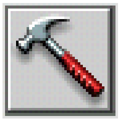
Install In: $(HOME)/Apps

Build In:

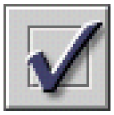
Build Tool: /bin/gnumake

Public Headers In:

4. To get the Build panel, click the Build button on Project Builder's main window:



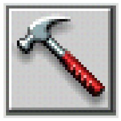
5. Click the Options button on the Build panel:



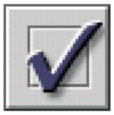
6. Choose “install_debug” from the Target pop-up menu.
7. Click the Build button on the Build panel to install the debug libraries.

Build for Debugging

1. Display the Build Options panel.
   a. Click the Build button on Project Builder’s main window:

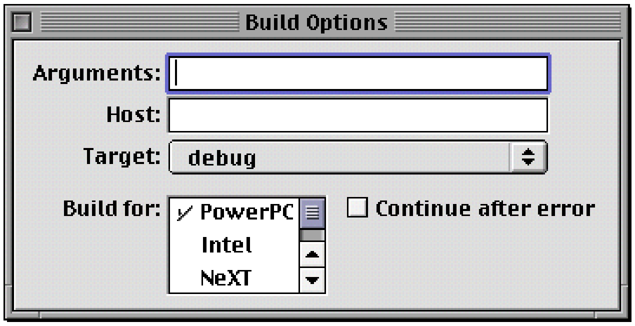


   b. Click the Options button on the Build panel:



2. Set the debug target.

   Select the debug item from the Target pop-up menu, as shown in the illustration below:



Target: debug

3. Remove object files and other files generated by previous builds by clicking the “make clean” button. This step is necessary only if you had previously built an application executable without debugging symbols.)

The “make clean” button on the Build panel looks like a broom:

4. Build the project.

Access the Java Debugger

When you have built the project and have an executable containing symbols understood by **jdb**, run the Java Debugger from Project Builder. Project Builder knows which debugger to run, based on the type of executable it is debugging.

For the following exercise, assume that you are debugging a program containing only (or mostly) Java code, such as Temperature Converter.

1. Click the Launch button on Project Builder’s main window:

   This brings up the Launch panel.

2. Make sure that **gdb** is turned off as a debugging feature.

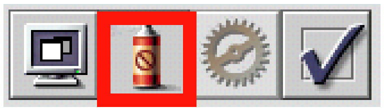
   a. Click the Options button on the Launch panel:

   b. Click the Debugger tab to display the debugger options; make sure that the Java Debugger checkbox is marked and the **gdb** checkbox is *not* marked.

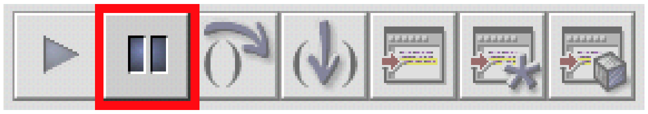
   Important
   The Java Debugger is automatically selected when your project contains Java code. The **gdb** debugger is also selected if the “Use GDB when debugging Java

programs" preference (in the Debugging display of the Preferences panel) is selected. If you do not want **gdb** active when you debug Java programs, turn off this preference.

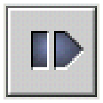
3. Click the Debug button on the Launch panel.



The Java Debugger starts up and displays its "JavaDebug>>" prompt. Because the Java VM is an interpreter, you do not need to suspend the Java Debugger to perform tasks such as setting breakpoints. You can, however, suspend and resume the Java Debugger if you wish. The Suspend/Continue button looks like this in Suspend mode (in the setting of its neighboring controls):



And in Continue mode the button looks like this:



Using the Java Debugger

Because many JavaDebug commands semantically correspond to **gdb** commands, Project Builder reuses the same controls and displays for controlling a Java debugging session. Some JavaDebug commands, however, cannot be invoked from the user interface. You must execute these commands from the command line. If you need to find out which Java Debugger commands are available, you can display a list of valid commands by entering anything that is not a valid command (such as "help") after the "JavaDebug>>" prompt:

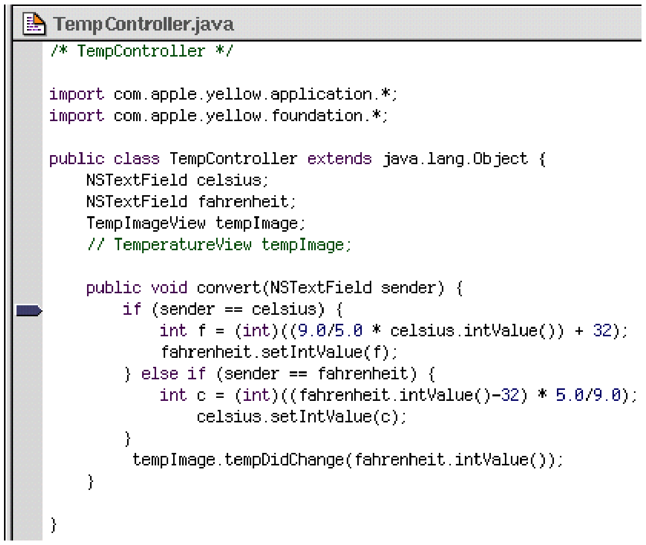
```
JavaDebug> help
```

Set and Manipulate Breakpoints

When JavaDebug starts up, a light gray band appears on the left edge of source-code files in the code editor. As in Objective-C debugging, you can set a breakpoint in Java code by double-clicking in this gray band next to the line of code where you want a breakpoint.

1. Locate the convert method in **TempController.m**.
2. Double-click in the gray band on the first line after the initial brace.

   A small pointer appears in the gray band.



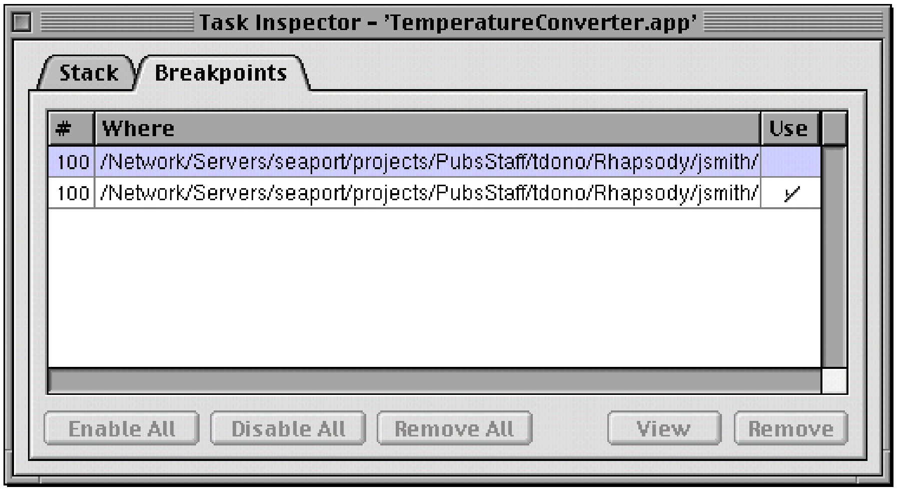
```
/* TempController */

import com.apple.yellow.application.*;
import com.apple.yellow.foundation.*;

public class TempController extends java.lang.Object {
    NSTextField celsius;
    NSTextField fahrenheit;
    TempImageView tempImage;
    // TemperatureView tempImage;

    public void convert(NSTextField sender) {
        if (sender == celsius) {
            int f = (int)((9.0/5.0 * celsius.intValue()) + 32);
            fahrenheit.setIntValue(f);
        } else if (sender == fahrenheit) {
            int c = (int)((fahrenheit.intValue()-32) * 5.0/9.0);
                celsius.setIntValue(c);
        }
         tempImage.tempDidChange(fahrenheit.intValue());
    }

}
```

Once you set a breakpoint you can move it within a file by dragging a pointer up and down the gray band. You can remove it by dragging the pointer off the gray band into the code editor. And you can temporarily disable a breakpoint, by Control–double-clicking the pointer, after which it turns gray (re-enable it by

Control–double-clicking the pointer again). You can also use the Breakpoint display of the Task Inspector to enable and disable breakpoints:

1. Click the Task Inspector button on the Launch panel:

2. Click the Breakpoints tab to display the current breakpoints.
3. Double-click the breakpoint line under the Use column. (The breakpoint is disabled when the checkmark does not appear.)



#	Where	Use
100	/Network/Servers/seaport/projects/PubsStaff/tdono/Rhapsody/jsmith/	
100	/Network/Servers/seaport/projects/PubsStaff/tdono/Rhapsody/jsmith/	✓

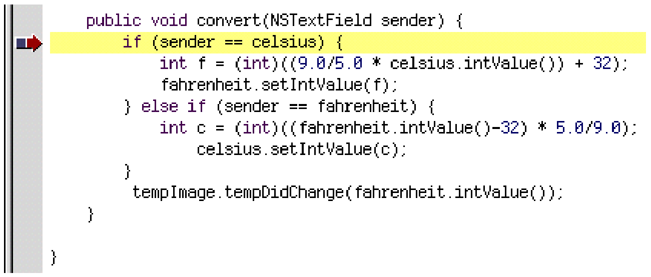
Several Java Debugger commands let you set, disable, and otherwise manipulate breakpoints from the command line. For example, to set the same breakpoint as above, you can just specify the class and method, separated by a colon:

```
JavaDebug>> b TempController:convert

 Set breakpoint 2000 in method: convert at line 13 in file "TempController.java"
```

Step Into and Over Code

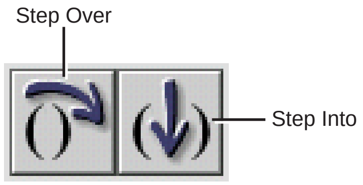
As in any kind of debugging, before you can perform any useful debugging task, such as stepping through code, you must run the program until it hits the next breakpoint.

1. Enter a number in a TemperatureConverter field and press Return. Execution stops at the breakpoint you set in the previous exercise; the program counter is indicated by a red pointer:



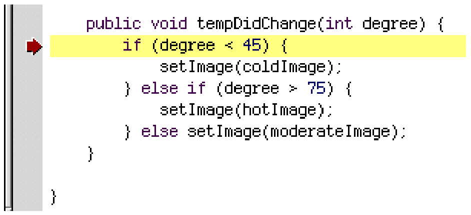
```
    public void convert(NSTextField sender) {
        if (sender == celsius) {
            int f = (int)((9.0/5.0 * celsius.intValue()) + 32);
            fahrenheit.setIntValue(f);
        } else if (sender == fahrenheit) {
            int c = (int)((fahrenheit.intValue()-32) * 5.0/9.0);
                celsius.setIntValue(c);
        }
         tempImage.tempDidChange(fahrenheit.intValue());
    }

}
```

2. Step over the first few lines of code by clicking the appropriate control on the Launch panel:



Notice how as you step over lines the program counter changes to the next line to be executed.

3. When you arrive at the last statement of the method (which calls tempDidChange), click the Step Into button. The code editor displays **TempImageView.java** (or **TemperatureView.java** if you are using the view object implemented by that code) and the program counter points to the first line of the tempDidChange method.



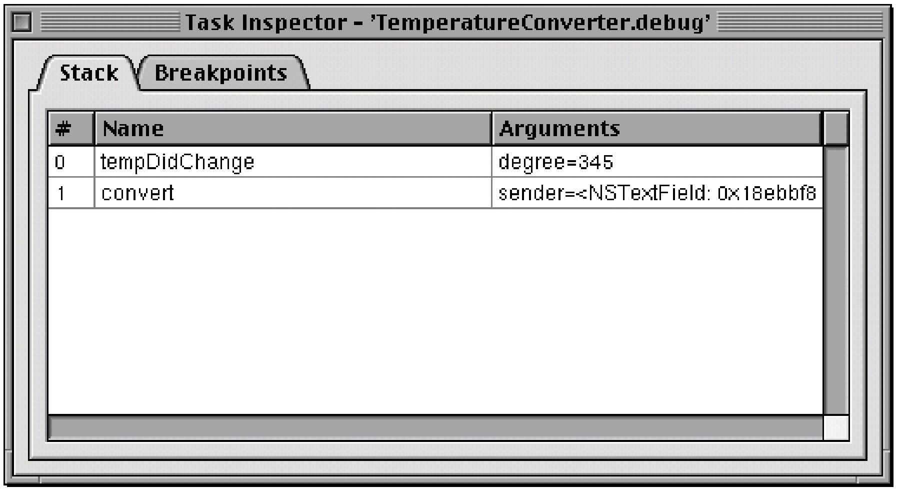
```
    public void tempDidChange(int degree) {
        if (degree < 45) {
            setImage(coldImage);
        } else if (degree > 75) {
            setImage(hotImage);
        } else setImage(moderateImage);
    }

}
```

You can also use Java Debugger commands to step into (stepi, step, and si) and step over (next) lines of code. The finish command lets you "step out," or complete the execution of the current stack frame. Note that pressing Return repeats the last step command entered.

Get a Backtrace and Examine a Frame

At any point in debugging you can view the current stack frame.

1. Click the Task Inspector button.
2. Click the Stack tab in the Task Inspector.



#	Name	Arguments
0	tempDidChange	degree=345
1	convert	sender=<NSTextField: 0x18ebbf8

The display lists the methods in the stack frame in the order of invocation; each line shows the arguments of a method. (Of course, the arguments of methods for which your project has no source code are not shown.) Double-click a line to display the method in the code editor.

The Java Debugger has several commands that you can use to get a backtrace and examine a frame. To show the stack frame, enter the backtrace or bt command:

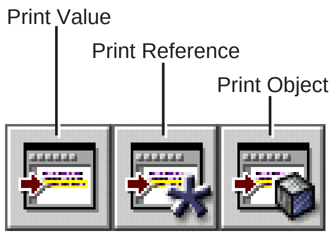
```
JavaDebug>> bt

 [0]   TempImageView.tempDidChange (TempImageView:38) pc = 0
 [1]   TempController.convert (TempController:20) pc = 73
```

To examine an individual frame, enter frame followed by the frame number:

```
JavaDebug>> frame 1
[1]     convert(sender=<NSTextField: 0x1813508>)
```

Examine Objects and Variables

Project Builder lets you examine data of three general types: value, reference, and object. Buttons on the Launch panel correspond to these types:



- The “Print Value” button (and command) prints the value of a variable.
- The “Print Reference” button (and command) prints the value referenced by a variable; this command is useful for printing the instance variables of a class or object.
- The “Print Object” button (and command) displays an object’s self-description by calling the object’s toString method.

Because the Java Debugger treats primitive types as objects, the “Print Value” and “Print Object” commands are equivalent.

To use the print buttons:

1. Highlight the variable or expression in the code editor.

   Unless a variable (such as an instance variable) has global scope or is an argument, you must highlight it after the statement that assigns it a value has been executed.

2. Click the appropriate print button on the Launch panel.

Of course, you can give the same commands from the Java Debugger command line.

Printing a value:

```
JavaDebug>> print degree
degree = 120
```

Printing a reference (object):

```
JavaDebug>> p* sender
sender = (com.apple.yellow.application.NSTextField)0x368470 {
   private int instance = 53568
}
```

Printing a reference (class):

```
JavaDebug>> p* NSTextField
NSTextField = 0x3646c0:class(com.apple.yellow.application.NSTextField) {
   superclass = 0x366408:class(com.apple.yellow.application.NSControl)
   loader = null

   private static boolean _alreadySqwaked = false
   public static final String ViewFrameDidChangeNotification = NSViewFrameDidChangeNotification
   public static final String ViewFocusDidChangeNotification = NSViewFocusDidChangeNotification
  public static final String ViewBoundsDidChangeNotification = NSViewBoundsDidChangeNotification
   public static final int NoBorder = 0
   public static final int LineBorder = 1
   public static final int BezelBorder = 2
   public static final int GrooveBorder = 3
   public static final int ViewNotSizable = 0
   public static final int ViewMinXMargin = 1
   public static final int ViewWidthSizable = 2
   public static final int ViewMaxXMargin = 4
   public static final int ViewMinYMargin = 8
   public static final int ViewHeightSizable = 16
   public static final int ViewMaxYMargin = 32
   public static final String ControlTextDidBeginEditingNotification =
NSControlTextDidBeginEditingNotification
   public static final String ControlTextDidEndEditingNotification =
NSControlTextDidEndEditingNotification
   public static final String ControlTextDidChangeNotification = NSControlTextDidChangeNotification
}
```

Printing an object:

```
JavaDebug>> po sender
sender = <NSTextField: 0xd140>
```

Printing the receiver (this):

```
JavaBug>> p* this
this = (TempImageView)0x39f5c8 {
   private int instance = 25616120
   protected NSImage coldImage = (com.apple.yellow.application.NSImage)0x39f670
   protected NSImage moderateImage = (com.apple.yellow.application.NSImage)0x39f688
   protected NSImage hotImage = (com.apple.yellow.application.NSImage)0x39f3f0
}
```

Debug Multiple Threads

With the Java Debugger, you can debug multiple threads in an application, switching among them to set breakpoints, examine stack frames and data, and perform other debugging tasks. The Java Debugger displays threads in groups; each thread has its own unique number within a group. To switch to another thread, you enter the `thread` command with the thread's group number as an argument.

To find out a thread's number, enter the `group` command (Project Builder has no controls equivalent to the `group` and `thread` commands):

```
JavaDebug>> group
```

```
Group: system
 1     Finalizer thread   cond. waiting
 2     JavaDebug   cond. waiting
 3     Debugger agent   running
 4     Breakpoint handler   cond. waiting
 5     Step handler   cond. waiting
 6     Agent input   cond. waiting
 7     main    suspended
 8     myThread   cond. waiting

Group: main
 1     main     suspended

••CURRENT GROUP is "system"
••CURRENT THREAD within the group is "main"
```

In addition to the names and numbers of threads, the output shows the current states. To switch to a thread, enter the thread command with a thread number.

```
JavaDebug>> thread 8

  Current thread now myThread, state=cond. waiting
```

Debugging Java and Objective-C Simultaneously

This section of the tutorial uses the Java TextEdit example application for illustration. This application project contains both Java classes and an Objective-C class, and so it provides a good test case for exploring how to debug in this dual world. Before you begin the following tasks, copy the TextEdit example project, located at **/System/Developer/Examples/Java/AppKit/TextEdit**, to your temporary directory (**/tmp**).

Set Up For Debugging

To set up for the tasks illustrated in this section—debugging both Objective-C and Java code in the same executable—do the following:

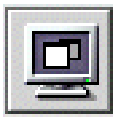
1. Build the TextEdit application with debugging symbols (that is, with the target set to "debug"; see Preparing to Debug an Application for details).
2. Open the Launch Options panel, click the Debugger tab, and make sure both the "gdb" and "Java Debugger" checkboxes are checked.

Important
If you are debugging a project of type JavaPackage, you must select the class with the entry point (main()) before you begin debugging. To do this, select the class containing main() in the project browser; then open the Project Inspector and, in the File Attributes display, click the "Has Java main" checkbox.

Run both gdb and the Java Debugger

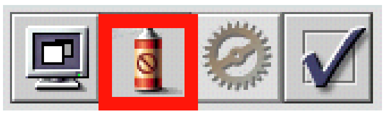
When you have built the TextEdit application and have an executable containing symbols understood by the Java Debugger, run the debuggers from Project Builder. Project Builder starts up both **gdb** and the Java Debugger. In Yellow Box applications, **gdb** is started before the Java Debugger because the entry point, main(), contains Objective-C code.

1. Click the Launch button on Project Builder's main window:



This brings up the Launch panel.

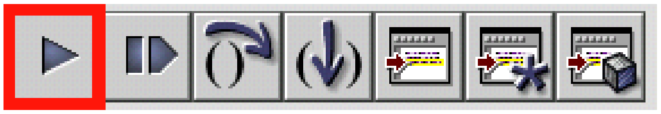
2. Click the Debug button on the Launch panel.



The **gdb** debugger starts up, and prints output to the console view similar to this:

```
> Debugging 'TextEdit'...
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 970507 (powerpc-apple-rhapsody), Copyright 1995 Free Software Foundation, Inc.
Reading symbols from /tmp/TextEdit/TextEdit.debug/TextEdit...done.

(gdb)
```

3. Run the **TemperatureConverter.debug** executable in **gdb**. To do this, click the arrow button:



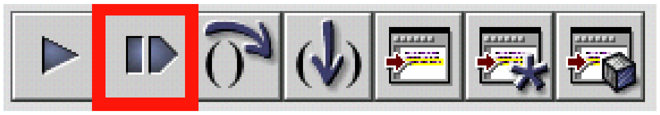
This action runs **gdb** until it stops at a breakpoint automatically set at the entry point; it writes output to the console view that is similar to the following:

```
gdb) Starting program: /tmp/TextEdit/TextEdit.debug/TextEdit -NSPBDebug
donote_1345_84422926_1994577324_1
/usr/local/standards/commonalias: No such file or directory.

Breakpoint 1, 0x3b04 in start ()
Dynamic Linkeditor at 0x41100000 offset 0x0
Executable at 0x2000 offset 0x0
/System/Library/Frameworks/AppKit.framework/Versions/C/AppKit at 0x43300000 offset 0x0
/System/Library/Frameworks/Foundation.framework/Versions/C/Foundation at 0x42500000 offset 0x0
/System/Library/Frameworks/System.framework/Versions/B/System at 0x41300000 offset 0x0
(gdb)
```

4. Set breakpoints in Objective-C code. A good place to set one is the `find:` method in the TextFinder class (**TextFinder.m**).

   The procedure is the same as for Java code; see Set and Manipulate Breakpoints for details.

5. Click the Continue button to resume the debugging operation:



The gdb debugger writes more output to the console view, after which it links in the necessary dynamic libraries and runs the Java Debugger (indicated by the prompt "JavaDebug>>"):

```
(gdb) Continuing.
/System/Library/Frameworks/JavaVM.framework/Versions/A/JavaVM at 0x5cd00000 offset 0x0
/usr/lib/java/libObjCJava.A.dylib at 0x53d00000 offset 0x0
/System/Library/Frameworks/JavaVM.framework/Libraries/libjava.A.dylib at 0x5c100000 offset 0x0
/System/Library/Frameworks/JavaVM.framework/Libraries/libzip.A.dylib at 0x5cb00000 offset 0x0
/System/Library/Frameworks/JavaVM.framework/Libraries/libdebugit.A.dylib at 0x5bf00000 offset 0x0
/System/Library/Frameworks/JavaVM.framework/Libraries/libagent.A.dylib at 0x5bd00000 offset 0x0

JavaDebug>> /usr/lib/java/libFoundationJava.B.dylib at 0x50300000 offset 0x0
/System/Library/Frameworks/JavaVM.framework/Libraries/libmath.A.dylib at 0x5c500000 offset 0x0
/usr/lib/java/libAppKitJava.B.dylib at 0x50c00000 offset 0x0
```

You are now ready to set Java breakpoints and begin debugging the application. A suggested breakpoint location is at one of the constructors of the Document class.

Now do something with the TextEdit application that causes execution to stop at one of your breakpoints.

- Create a new document or open an existing one. If you set a breakpoint at a Document constructor, the code editor displays **Document.java** and the program counter aligns with the breakpoint. The "JavaDebug>>" prompt is shown in the Launch panel's console.
- Type some text in the new document and then choose Edit>Find>Find. Enter a matching string in the Find Panel's Find field and click Next. This displays the **TextFinder.m** file at the find: breakpoint. Notice how the "JavaDebug>>" prompt changes to "(gdb)"; you are now using the **gdb** debugger.

Important
The Java Debugger runs in the Java VM and so it is running whether the target is running or not. However, **gdb** requires that the entire target process be stopped, and to ensure this it stops the Java VM. This

means that when you switch from the Java Debugger to **gdb**, not only the target is stopped but the Java VM is stopped. You must click the Continue button (or enter the continue command) to restart the Java VM and the Java Debugger.

Switching Between JavaDebug and gdb

With both **gdb** and the Java Debugger running on a debugging target, you can use Project Builder's controls to perform most common debugging tasks in both Java and non-Java code. Using the "breakpoint band" next to source-code files, you can set and manipulate breakpoints in any file. When a breakpoint is hit, you can click the appropriate button on the Launch panel to step into or over code and to inspect frames, values, and objects. The correct debugger is used for the task at hand.

However, there might be occasions when you want more explicit control, and for this you need to switch between debuggers on the command line. When debugging Objective-C (and C and C++) code and Java code simultaneously, you must keep in mind that the Java VM is still running when the Java Debugger is debugging code in the stopped target; however, **gdb** can only debug code when all processes involving the target—including the Java VM—are stopped.

Here is how you switch between debuggers in a session. The following scenario assumes you have both debuggers active and that the target is currently executing Java code:

1. Choose Tools>Debugger>Suspend Java VM to use the Java Debugger.

   The Java Debugger prompt ("JavaDebug>>") appears in the Launch console. You can now perform Java debugging tasks that require the target to be stopped.
2. Click the Continue button on the Launch panel to have the target continue execution.
3. Choose Tools>Debugger>Suspend Process to switch to **gdb**.

   The Java Debugger relinquishes control to **gdb**: the "(gdb)" prompt appears and all process are stopped, including the Java VM. You can now perform debugging tasks in **gdb**.
4. Click the Continue button to have the target (plus the Java VM) resume running.

Java Debugger Command Reference

This section describes the complete set of Java Debugger commands. The Java Debugger recognizes command truncations as long as they are unique. For example, it recognizes du as dump but it cannot interpret "d". If a truncation is not unique, the debugger informs you. Several commands have shortened aliases (for example, si is equivalent to stepi and bt is equivalent to backtrace). The case of a command does not matter; Break is the same break which is the same as BrEaK. However, the case of arguments often does matter; for example, class and file names must match case ("Exception" is not the same as "exception").

Getting Help

JavaDebug>> non-command

To get a list of Java Debugger commands, enter any string that is not a command after the prompt.

Thread Commands

The Java Debugger recognizes a current thread group and a current thread. When an exception occurs, the current thread is reset. The current thread group is always the thread group containing the current thread.

Group [*groupName*]

Lists thread groups and threads if used by itself. If *groupName* is given, the current thread group is set to the group identified by that name. To avoid confusion, thread groups are referred to by name, while threads within a group are referred to by number.

Thread *threadNum*

Sets the thread within the current thread group to *threadNum*. To find out what the current group and thread are, enter the group command.

Example:

```
JavaDebug>> group

Group: system
1 Finalizer thread cond. waiting
2 JavaDebug cond. waiting
3 Debugger agent running
4 Breakpoint handler cond. waiting
5 Step handler cond. waiting
6 Agent input cond. waiting
7 main suspended

Group: main
1 main suspended

CURRENT GROUP is "system"
CURRENT THREAD within the group is "main"

JavaDebug>> thread 6

Current thread now Agent input, state=cond. waiting
```

Stack and Data Inspection

The stack and data inspection commands frame, up, down, print, and dump require that the thread being inspected be suspended. Threads are suspended by program execution hitting a breakpoint or by users giving the suspend command.

Backtrace | bt

Displays the contents of the current thread's Java stack. Currently, only Java stack frames are displayed. No "native" (non-Java) frames appear.

Example:

```
JavaDebug>> bt
0] java.io.PipedInputStream.read (PipedInputStream:201) pc = 80
[1] java.io.PipedInputStream.read (PipedInputStream:242) pc = 43
[2] java.io.BufferedInputStream.fill (BufferedInputStream:135) pc = 164
[3] java.io.BufferedInputStream.read (BufferedInputStream:162) pc = 12
[4] java.io.FilterInputStream.read (FilterInputStream:81) pc = 4
[5] sun.tools.debug.AgentIn.run (AgentIn:46) pc = 20
[6] java.lang.Thread.run (Thread:474) pc = 11
```

Frame [*frameNum*]

Displays the arguments and local variables for the current frame or the frame number *frameNum*. Frame numbers, which are the numbers displayed in the backtrace, are absolute. If you do not compile Java code with **javac**'s **-g** flag, local and argument information is not available to the debugger. The frame command resets the current frame.

Example:

```
JavaDebug>> backtrace

[0] com.apple.alpha.core.MutableDictionary.<init> (MutableDictionary:213) pc = 0
[1] Document.setRichText (Document:534) pc = 10<
[2] Document.toggleRich (Document:596) pc = 69

JavaDebug>> frame 1

[1] setRichText(flag=true)
<local> view = <NSTextView: 0x194b8c0>
Frame = {{0.00, 0.00}, {490.00, 420.00}}, Bounds = {{0.00, 0.00}, {490.00, 420.00}}
Horizontally resizable: NO, Vertically resizable: YES
MinSize = {490.00, 420.00}, MaxSize = {340282346638528859811704183484516925440.00,
340282346638528859811704183484516925440.00}
```

Up [*numFrames*]

Resets the current frame upwards toward older frames (that is, frames with higher numbers) relative to the current frame. If *numFrames* is not given, 1 is assumed.

Down [*numFrames*]

Resets the current frame downwards toward newer frames (that is, frames with lower numbers) relative to the current frame. If *numFrames* is not given, 1 is assumed.

Print | po *anObject*

Prints the object *anObject* by calling its toString() method.

Example:

```
JavaDebug>> po view

view = <NSTextView: 0x194b8c0>
Frame = {{0.00, 0.00}, {490.00, 420.00}}, Bounds = {{0.00, 0.00}, {490.00, 420.00}}
Horizontally resizable: NO, Vertically resizable: YES
```

```
MinSize = {490.00, 420.00}, MaxSize = {340282346638528859811704183484516925440.00,
340282346638528859811704183484516925440.00}
```

Dump *anObject*

Prints the object *anObject* structurally. An alias for dump is p* (from **gdb** usage for Print *=reference)

Example:

```
JavaDebug>> p* view

view = (com.apple.alpha.app.TextView)0x3bc0c0 {
private int instance = 26523840
}

JavaDebug>> p* 0x3bc0c0

0x3bc0c0 = (com.apple.alpha.app.TextView)0x3bc0c0 {
private int instance = 26523840
}
```

In the above case, the object is obviously a native object with a peer class.

List

Displays source text for the current frame—when it can find it. (See the dir command in Convenience Functions, below).

```
JavaDebug>> break Document:setRichText

Set breakpoint 2000 in method: setRichText at line 533 in file "Document.java"

JavaDebug>> cont

JavaDebug>> // breakpoint hit because of user action
Broken at 533 in "Document.java"

JavaDebug>> list

File: "Document.java"
529 return layoutManager().hyphenationFactor();
530 }
531
```

```
532 public void setRichText(boolean flag) {
533 => TextView view = firstTextView();
534 MutableDictionary textAttributes = new MutableDictionary(2);
535
536 isRichText = flag;
537
```

Control Functions

Suspend

Suspends all Java threads that are not involved in debugging.

Resume

Resumes all Java threads that are not involved in debugging.

Break [*fileName*:*lineNumber*]

Sets a breakpoint in file *fileName* at line *lineNumber*. An alias for break is b. With no arguments, it prints the current breakpoints.

Break [*className*:*methodName*]

Sets a breakpoint at the start of method *methodName* in class *className* . An alias for break is b. With no arguments, it prints the current breakpoints.

Example:

```
JavaDebug>> break Document:setRichText

Set breakpoint 2000 in method: setRichText at line 533 in file "Document.java"

JavaDebug>> break

break <fileName>:<lineNum> | <class>:<method> -- set a breakpoint
2000 Document.java:533

JavaDebug>> disable 2000

JavaDebug>> break
break <fileName>:<lineNum> | <class>:<method> -- set a breakpoint

2000 Document.java:533 [disabled]
```

Clear [*breakNum*]

Clears (forgets) the breakpoint numbered *breakNum*. With no argument, it prints the current breakpoints.

Disable [*breakNum*]

Disables, but does not clear, the breakpoint numbered *breakNum*. If no argument is given, prints the current breakpoints.

Enable [breakNum]

Re-enables the disabled breakpoint numbered *breakNum*. If no argument is given, prints the current breakpoints.

Continue

Continues (resumes) execution of a suspended thread (must be the current thread).

Step

“Step Into.” Steps forward a line of code. If the line is a statement that has a call to a Java method, it steps into the call. If the call is to "native" (non-Java) code, it steps over the code (that is, it acts like next). If the step returns to native code, it acts like continue.

Stepi

Steps forward a single bytecode instruction. It otherwise acts like step. (Currently, there is no command to display the bytecodes. Use javap -c *classFile* to see the bytecodes).

Next

“Step Over.” Steps forward a line of code. If the code is a method call, it steps over the call. If the step returns to native code, it acts like continue.

Finish

Steps to the end of the code in the current stack frame (also know as "step out").

Catch *exceptionClass*

Causes exceptions for class *exceptionClass* and all its subclasses to act like a breakpoint. At this time you cannot step through exception code; you can only inspect the stack and continue.

Drop *exceptionClass*

Reverses the effect of catch. The system no longer breaks when an exception of class *exceptionClass* is thrown.

Convenience Functions

Dir [*newClassPath*]

If *newClassPath* is given, sets the CLASSPATH environment variable used to search for debug and source information (for example, for the list and frame commands). If the command has no argument, it prints the CLASSPATH variable.

GetProp [*propName*]

Prints the value for the specified property *propName* or, if no property is specified, prints all properties of the Java VM. Java VM properties cannot be changed.

```
JavaDebug>> getprop user.home

/Local/Users/kend

JavaDebug>> getprop

Property Names:
user.language = "en"
java.home = "/System/Library/Frameworks/JavaVM.framework/Home"
awt.toolkit = "com.apple.rhapsody.awt.RToolkit"
file.encoding.pkg = "sun.io"
java.version = "internal_build:kend:03/05/98-09:08"
file.separator = "/"
line.separator = "

file.encoding = "MacRoman"
java.compiler = "jitc_ppc"
java.protocol.handler.pkgs = "com.apple.net.protocol"
java.vendor = "Apple Computer, Inc."
user.timezone = "PST"
user.name = "kend"
os.arch = "ppc"
os.name = "Rhapsody"
java.vendor.url = "http://www.apple.com/"
user.dir = "/Local/Users/kend/Projects/TextEdit"
java.class.path = "/Local/Users/kend/Projects/TextEdit/TextEdit.app/Resources/Java/.:.:/Local/Users/kend/
jdk20build/build/classes:/System/Library/Java:/System/Library/Frameworks/JavaVM.framework/Classes/
classes.jar:/System/Library/Frameworks/JavaVM.framework/Classes/awt.jar"
```

```
java.class.version = "45.3"
os.version = "Premier Release"
path.separator = ":"
user.home = "/Local/Users/kend"
```

Developing Java Applications—Concepts

This document presents the concepts related to the tutorial Developing Java Applications: A Tutorial.

Fast Track to Java Development

If you are an Objective-C programmer who is familiar with the Yellow Box development environment and the Yellow Box APIs, you're probably interested only in the differences in the development procedure between Objective-C and Java:

- Controller classes must inherit from java.lang.Object. You can specify this relationship in Interface Builder's Classes display when you define the class.
- In Interface Builder, when you create a source-code file for a Java controller class (by choosing Classes>Create Files), Java code is generated. However, since Java has no notion of dynamically typed objects (id), it substitutes java.lang.Object as the type of outlets and senders of action messages. You must specify the correct class types in place of Object. In other words,

  ```
  Object myTextField;
  public void doThis(Object sender)
  ```

  must be translated to this:

  ```
  NSTextField myTextField;
  public void doThis(NSButton sender)
  ```

  If you don't specify the correct class type, you must cast to that type in the code.
- The classes that Interface Builder presents in its Classes display represent, in most cases, both Objective-C and Java Yellow Box classes. And the process for

defining a class and connecting its outlets is the same for both Yellow Box language versions. However, before you create the "skeletal" source files to be added to Project Builder, select the class and then select the language in the inspector's Attributes display, Interface Builder then generates the source file in the requested language.

The Yellow Box's Java Feature

With the Yellow Box development environment, you can create applications written in Java™ but built both from objects in the Yellow Box frameworks and from pure Java objects. These applications run on any Mac OS X or Yellow Box for Windows system.

There are four major parts to the Yellow Box's Java feature for this release:

- **Java virtual machine** (VM). This is the Java "runtime," an interpreter that loads Java class files and interprets the bytecode. The VM is packaged in a framework (**JavaVM.framework**) which also includes the latest version of JavaSoft's JDK and a copy of JavaSoft's reference documentation.
- **Tools integration**. Project Builder integrates the Java compiler (**javac**), debugger (**jdb**), and packaging technology (creation of **.jar** or **.zip** files). During a build, Project Builder handles source-code files in a project in a manner appropriate to their extensions. Project Builder also features an integrated debugger interface that allows you to debug Java and Objective-C (as well as C and C++) code simultaneously. Interface Builder incorporates a few Java features such as definition of java.lang.Object subclasses and generating of "skeletal" **.java** files from class definitions.
- **Java bridge**. This technology links the Java programmatic interfaces of Yellow Box classes and interfaces with their Objective-C implementations. It makes it possible for developers to "wrap" their Objective-C classes in Java APIs. Currently, most Yellow Box classes are "wrapped."

  The Yellow Box development environment includes project types and tools for wrapping Objective-C code in Java interfaces. For more on Apple's bridging technology, see The Java Bridge.
- **New Java classes**. Apple has developed several new Java classes—both Yellow Box and native Java—primarily to resolve "unbridgable" differences between

the languages. Some of these classes perform class loading, while others provide object wrappers for Objective-C structures.

The Yellow Box also helps you build and lets you run 100% Pure Java™ applets and applications. See Developing 100% Pure Java Applications for details.

The Java Bridge

The Java bridge is an Apple technology that lets Objective-C objects and Java objects communicate freely. With it developers can transparently instantiate Objective-C objects in Java code and treat them as if they were Java objects; for example, it allows Objective-C protocols to appear in the guise of Java interfaces. Conversely, it can expose any Java class or interface as an Objective-C class or protocol.

The core Yellow Box frameworks—the Application Kit and Foundation—have been “bridged” to Java. This means that developers can write Yellow Box applications using nothing but Java code.

The Java bridge offers the following features:

- It exposes Objective-C classes as Java classes that can be directly subclassed.
- Java objects are passed across the bridge to the Objective-C world where they are manipulated by the code as if they were Objective-C objects. (This happens whether the object is an instance of a 100% Pure Java object or not.)
- Some Java classes, such as String and Exception, are mapped to Objective-C classes, such as NSString and NSException; objects of these classes are transparently “morphed” into each other as they cross the bridge between the Java and Objective-C worlds.
- Developers need not worry whether a class comes from the Java or the Objective-C world. The bridge transparently loads any needed Objective-C framework whenever a bridged class is used.

The Yellow Box development environment provides a set of tools and specifications that enable you to bridge your own Objective-C classes and protocols, exposing them as Java classes and interfaces. (And, if you wish, you can expose Java classes and interfaces as Objective-C classes and protocols.) The essential tool, **bridget**, reads and processes a “jobs” file —a file with an extension of **.jobs** (the letters of which stand for Java to Objective-C Bridging Specification). The jobs file is a text file that contains specifications mapping Objective-C classes, interfaces, and method selectors to Java classes, interfaces, and methods.

Other tools in the development environment facilitate the process of bridging frameworks. The general procedure for bridging is as follows:

1. Create a project in Project Builder that is of type JavaWrapper.
2. Create the jobs file. A demo application, WrapIt, assists developers with this task.
3. Build the project. Project Builder and bridget use the jobs file to generate Java classes and a dynamic library providing the implementation of the native methods these classes declare.

Developing 100% Pure Java Applications

You can use the Yellow Box development environment to develop 100% Pure Java applications: that is, applications developed exclusively with JavaSoft's Java Development Kit™ (JDK). The Yellow Box includes the latest version of the JDK in **/System/Library/Frameworks/JavaVM. framework**.

To create an 100% Pure Java application with the Yellow Box development environment:

1. Launch Project Builder.
2. Choose New from the Project menu.
3. Select the Java Package project type from the pop-up menu.
4. Specify a directory location for your application.
5. For each **.java** file in your project:
    a. Choose New in Project from the File menu.
    b. In the New panel select the Classes suitcase, name the file, and give it an extension of **.java**. Make sure that the Include Header checkbox is not selected.

c. Click OK to add the file to the Classes category of the project.

6. Write the Java code needed to implement your project (you can remove the lines importing the Yellow Box packages).

   Project Builder supports syntax coloring and indentation for Java code. You can use all other Project Builder features that do not depend on project indexing.

7. Build the project. As with Objective-C projects, this merely requires you to choose Tools>Build>Build Project.

   The build process automatically invokes **javac** with the correct arguments and does whatever else is requred to build the project, such as creating the archive (a **.zip** file, by default). If there are Java coding errors, Project Builder reports them in its Build panel; you can navigate to the code containing an error by clicking the reporting line in the panel.

When you're ready to create and install the **.zip** package containing your Java classes, do the following:

1. Chose the Build Attributes display of the Project Inspector and examine the path in the Install In field. If the default installation location is not what you want, change it.
2. Chose Tools>Project Build>Build to open the Build panel.
3. Click the Options button on the Build panel (the checkmark icon).
4. Change the selected item in the Target pop-up menu to "install".
5. Click the Build button (the hammer icon).

The **.zip** package is created and installed.

You can use the Java interpreter (**java**) and the applet previewer (**appletviewer**) to run Java applications and applets, respectively. These tools are in **/usr/bin**.

Important
To compile and run Java applicatons, the CLASSPATH environment variable must be correctly set. This variable is usually set for you by the installation script and by Project Builder. But if CLASSPATH becomes faulty, you can reset it with the **setenv** and **javaconfig** commands on the command line:

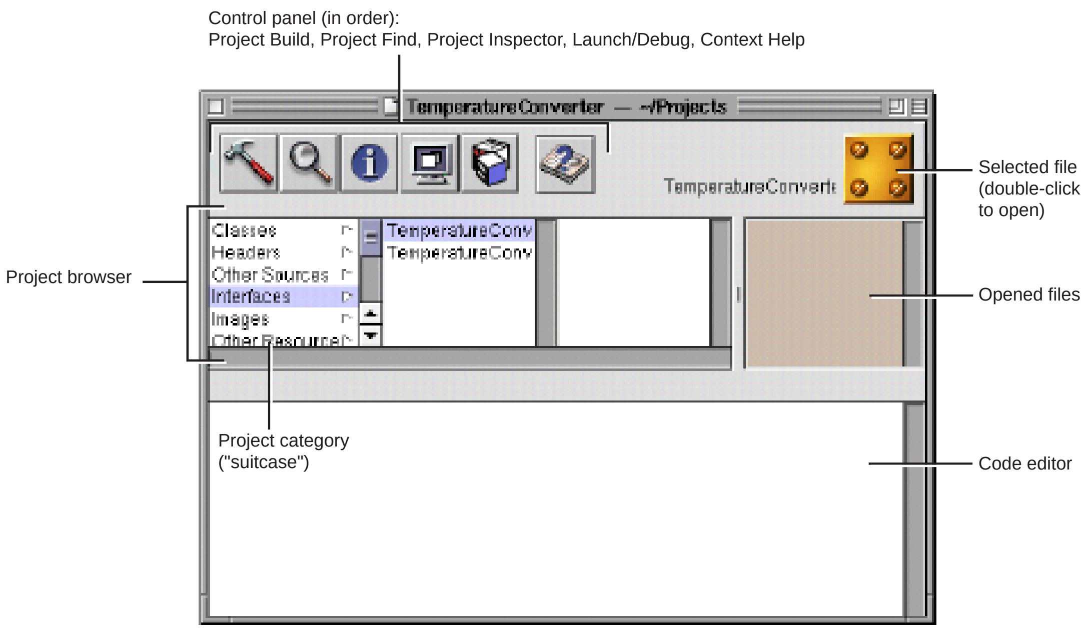
```
#setenv CLASSPATH .:‘javaconfig DefaultClasspath‘
```

CHAPTER

A Project Window

Project Builder presents the elements of a project in a project window.

Figure 0-1 Project Builder's Main Window


Control panel (in order):
Project Build, Project Find, Project Inspector, Launch/Debug, Context Help
TemperatureConverter — ~/Projects
TemperatureConverte
Selected file (double-click to open)
Project browser
Classes
Headers
Other Sources
Interfaces
Images
Other Resources
TemperatureConv
TemperatureConv
Opened files
Project category ("suitcase")
Code editor

Project Indexing

When you create or open a project, after some seconds you may notice triangular "branch" buttons appearing after source code files in the browser. Project Builder has indexed these files.

During indexing, Project Builder stores all symbols of the project (classes, methods, globals, and such) in virtual memory. This allows Project Builder to access project-wide information quickly. Indexing is indispensable to such features as name completion and Project Find. Usually indexing happens automatically when you create or open a project. You can turn off this option if you wish. Choose Preferences from the Edit menu and then choose the Indexing display. Turn off the "Index when project is opened" checkbox.

You can also index a project at any time by choosing Tools>Indexer>Index Project. If you want to do without indexing (maybe you have memory constraints), choose Tools>Indexer>Purge Indices.

What's a Nib File?

Every application has at least one nib file. The main nib file contains the application menu and often a window and other objects. An application can have other nib files as well. Each nib file contains the following information:

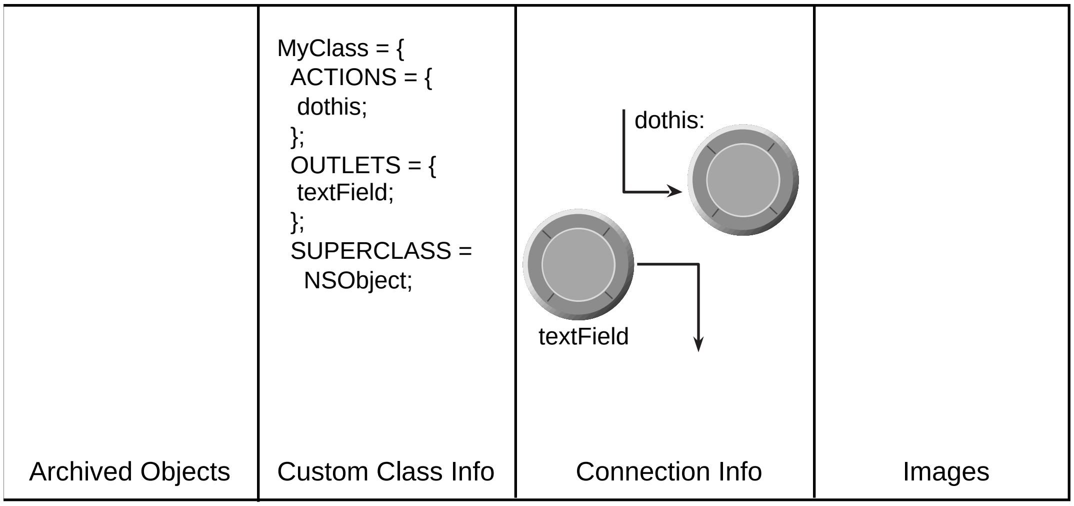
Archived objects. Encoded information on Yellow Box objects, including their size, location, and position in the object hierarchy (for view objects, determined by superview/subview relationship). At the top of the hierarchy of archived objects is the File's Owner object, a proxy object that points to the actual object that owns the nib file.

Custom class information. Interface Builder can store the details of Yellow Box objects and objects that you palettize (static palettes), but it does not know how to archive instances of your custom classes since it doesn't have access to the code. For these classes, Interface Builder stores a proxy object to which it attaches class information.

Connection information. Information about how objects within the object hierarchy are interconnected. Connector objects special to Interface Builder store this information. When you save the document, connector objects are archived in the nib file along with the objects they connect.

Images and sounds. Image files and sound files that you drag and drop over the nib file window or over an object that can accept them (such as a button or image view).

Figure 0-2 Contents of a Nib File



When You Load a Nib File

In your code, you can load a nib file by sending the NSBundle class loadNibNamed:owner: or loadNibFile:externalNameTable:withZone: messages. When you do this, the runtime system does the following: It unarchives the objects from the object hierarchy, sending each object an initWithCoder: message after allocating memory for it.

1. It unarchives each proxy object and queries it to determine the identity of the class that the proxy represents. Then it creates an instance of this custom class and frees the proxy.
2. It unarchives the connector objects and allows them to establish connections, including connections to File's Owner.
3. It sends awakeFromNib to all objects that were derived from information in the nib file, signalling that the loading process is complete.

Connections and Accessor Methods

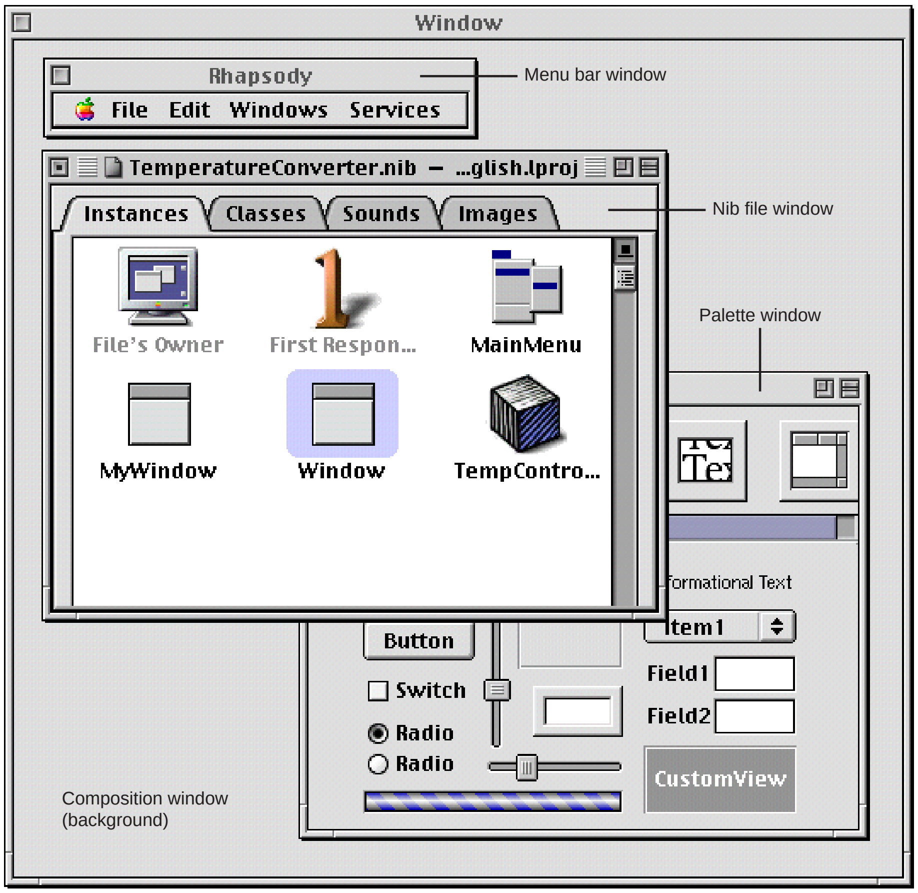
When the Yellow Box establishes connections during the course of loading a nib file, it sets the values of the source object's outlets. It first tries to set an outlet through the “set” accessor method if the source object implements it. For example, if the source object has an outlet named “contraption,” the system first sees if that object responds to “setContraption” and, if it does, it invokes the accessor method. If the source object doesn’t implement the accessor method, the system sets the outlet directly.

Problems naturally ensue if a “set” accessor method does something other than directly set the outlet. One common example is an accessor method that sets the string value of an outlet referring to a text field (setStringValue). After loading, the value of the outlet is null because the “set” accessor method did not directly assign the value.

The Windows of Interface Builder

When you open a nib file, Interface Builder opens several windows: the nib file window, a menu bar panel, a palette window, and an empty window in which you can put elements of a graphical user interface. The nib file window gives access to the objects, class definitions, and resources of a nib file. The menu panel allows you to constuct your application menus. And the palette window holds various objects of the Application Kit and any custom objects that you or third parties palettize. To show a palette, click on the of the icons that runs across the top of the window.

Figure 0-3 The Standard Windows of Interface Builder



Not shown in Figure 0-3 is Interface Builder's Inspector (Tools>Inspector), which lets you set attributes and the size of objects, specify objects to be connected, identify help files, and set many other variables.

The View Hierarchy and the First Responder

Just inside each window's content area—the area enclosed by the title bar and the other three sides of the frame—is the "content view." The content view is the root (or top) NSView in a window's view hierarchy. Conceptually like a tree, one or more NSViews may branch from the content view, one or more other NSViews may branch from these subordinate NSViews, and so on. Except for the content view, each NSView has one (and only one) NSView above it in the hierarchy. An NSView's subordinate views are called its subviews; its superior view is known as the superview.

On the screen, enclosure determines the relationship between superview and subview: a superview encloses its subviews. This relationship has several implications for drawing:

- Subviews are positioned in the coordinates of their superview, so when you move an NSView or transform its coordinate system, all subviews are moved and transformed in concert.
- It permits construction of a superview simply by arrangement of subviews. (An NSBrowser object is an instance of a compound NSView.)
- Because an NSView has its own coordinate system for drawing, its drawing instructions remain constant regardless of any change in position in itself or of its superview.

The view hierarchy also affects how events are handled, particularly through the first-responder mechanism.

Figure 0-4 A View Hierarchy

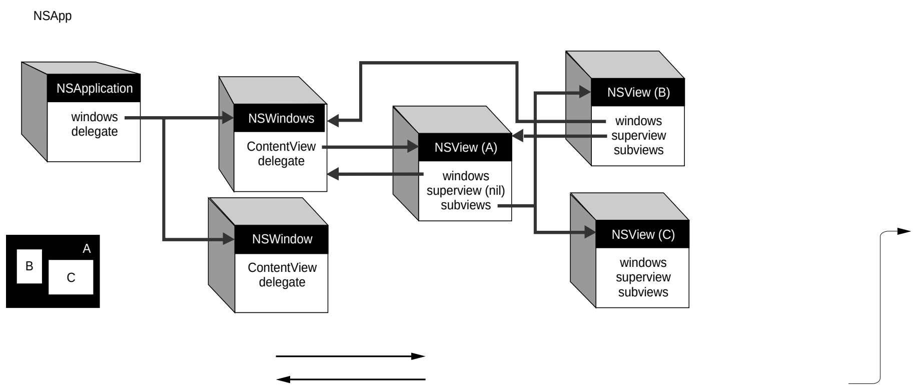


Figure 0-4shows how NSApplication, NSWindow, and NSView objects are connected through their instance variables.

First Responder and the Responder Chain

Each NSWindow in an application keeps track of the object in its view hierarchy that has "first responder" status. The first responder is the NSView that currently is the focus of keyboard events in the window. By default, an NSWindow is its own first responder, but any NSView within the window can become first responder when the user clicks it with the mouse.

You can also set the first responder programmatically with the NSWindow's makeFirstResponder method. Moreover, the first-responder object can be a target of an action message sent by an NSControl, such as a button or a matrix. Programmatically, you do this by sending setTarget to the NSControl (or its cell) with an argument of **nil**. You can do the same thing in Interface Builder by making a target/action connection between the NSControl and the First Responder icon in the Instances display of the nib file window.

All NSViews of an application, as well as all NSWindows and the application object itself, inherit from NSResponder, which defines the default message-handling behavior: events are passed up the responder chain. Many Application Kit objects, of course, override this behavior, so events are passed up the chain until they reach an object that does respond.

The series of next responders in the responder chain is determined by the interrelationships between the application's NSView, NSWindow, and NSApplication objects). For an NSView, the next responder is usually its superview; the content view's next responder is the NSWindow. From there, the event is passed to the NSApplication object.

For action messages sent to the first responder, the trail back through possible respondents is even more detailed. The messages are first passed up the responder chain to the NSWindow and then to the NSWindow's delegate. Then, if the previous sequence occurred in the key window, the same path is followed for the main window. Then the NSApplication object tries to respond, and failing that, it goes to NSApp's delegate.

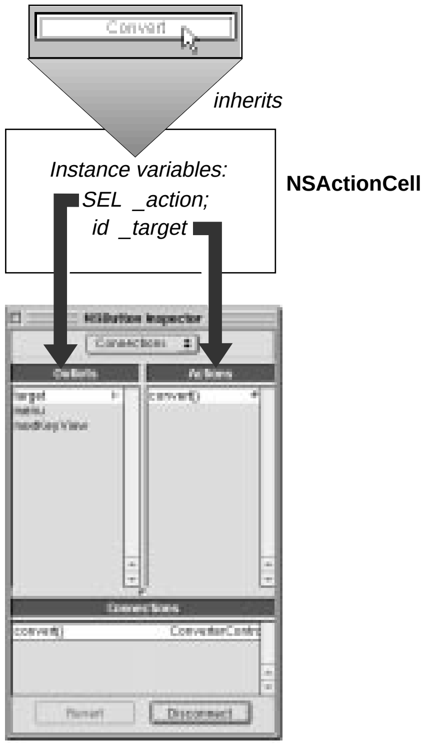
The Target/Action Paradigm

Interface Builder allows you to view and specify connections between a control object and its target in the Connections display of the control's inspector. The relation of target and action in this Inspector might not be apparent. First, target is an outlet of a cell object that identifies the recipient of an action message. So what is a cell object and what does it have to do with a button?

One or more cell objects are always associated with a control object (that is, an object inheriting from NSControl, such as a button). Control objects "drive" the invocation of action methods, but they get the target and action from a cell. This way one control object, such as an NSMatrix, can have different targets and actions for each of its cells, as well as its own target and action. NSActionCell defines the target and action outlets, and most kinds of cells in the Application Kit inherit these outlets.

When a user clicks a button, the button gets the target and action information from its cell. The action is a selector indicating the method to invoke in the target object. The button sends the approriate message to its target, which is typically an instance of a custom class.

Figure 0-5 Target and Action in Interface Builder



The Actions column of the Connections display shows the action methods defined by the class of the target object and known by Interface Builder. Interface Builder identifies action methods because their declarations follow the syntax:

```
- (IBAction)doThis:(id)sender;
```

The return argument can also be void instead of IBAction, but the argument is always sender.

Model-View-Controller Paradigm

A common and useful paradigm for object-oriented applications, particularly business applications, is Model-View-Controller (MVC). Derived from Smalltalk-80, MVC proposes three types of objects in an application, separated by abstract boundaries and communicating with each other across those boundaries.

Model object. This type of object represents special knowledge and expertise. Model objects hold a company's data and define the logic that manipulates that data. For example, a Customer object, common in business applications, is a Model object. It holds data describing the salient facts of a customer and has access to algorithms that access and calculate new data from those facts. A more specialized Model class might be one in a meteorological system called Front; objects of this class would contain the data and intelligence to represent weather fronts. Model objects are not directly displayed. They often are reusable, distributed, persistent, and portable to a variety of platforms.

View object. A View object in the paradigm represents something visible on the user interface (a window, for example, or a button). A View object is "ignorant" of the data it displays. The Application Kit usually provides all the View objects you need: windows, text fields, scroll views, buttons, browsers, and so on. But you might want to create your own View objects to show or represent your data in a novel way (for example, a graph view). You can also group View objects within a window in novel ways specific to an application. View objects, especially those in kits, tend to be very reusable and so provide consistency between applications.

Controller object. Acting as a mediator between Model objects and View objects in an application is a Controller object. There is usually one per application or window. A Controller object communicates data back and forth between the Model objects and the View objects. It also performs all the application-specific chores, such as loading nib files and acting as window and application delegate. Since what a Controller does is very specific to an application, it is generally not reusable even though it often comprises much of an application's code. (This last statement does not mean, however, that Controller objects cannot be reused; with a good design, they can.)

Because of the Controller's central, mediating role, Model objects need not know about the state and events of the user interface, and View objects need not know about the programmatic interfaces of the Model objects. You can make your View and Model objects available to others from a palette in Interface Builder.

Hybrid models. MVC, strictly observed, is not advisable in all circumstances. Sometimes it's best to combine roles. For instance, in a graphics-intensive application, such as an arcade game, you might have several View objects that merge the roles of View and Model. In some applications, especially simple ones, you can combine the roles of Controller and Model; these objects join the special data structures and logic of Model objects with the Controller's hooks to the interface.

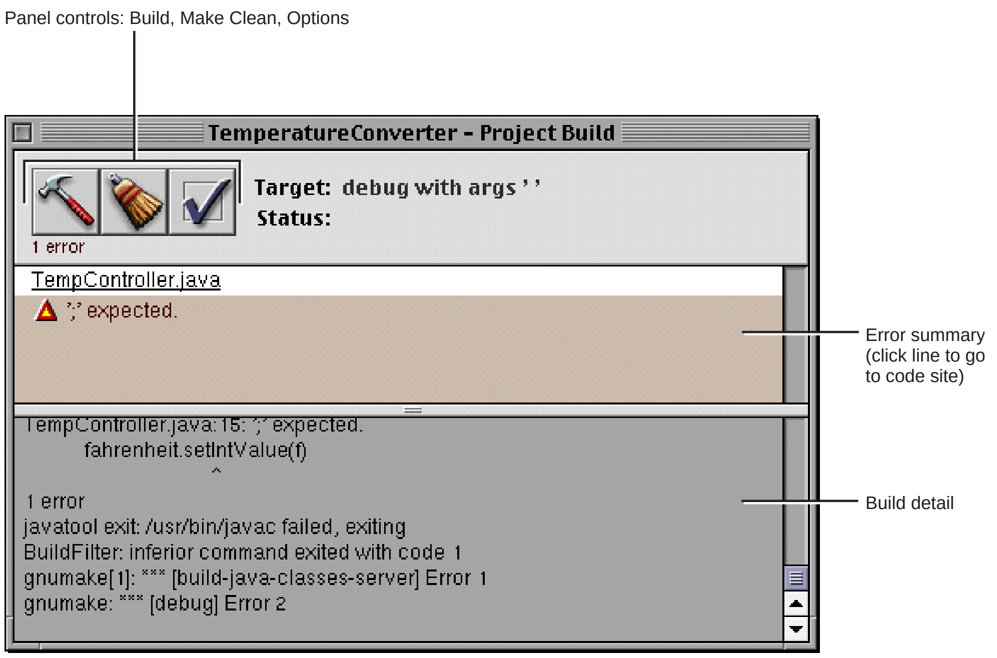
The Build Panel

The Project Build panel has buttons that do the following:

- Initiate the build process.
- Delete the products of the last build("make clean").
- Let you set options for the build.

It also shows the results of the build and takes you to the site of any error in the code when you click the line in the Project Build panel reporting the error.

Figure 0-6 The Build Panel


Panel controls: Build, Make Clean, Options
TemperatureConverter - Project Build
Target: debug with args ' '
Status:
1 error
Error summary (click line to go to code site)
Build detail

TempController.java

';' expected.

```
TempController.java:15: ';' expected.
        fahrenheit.setIntValue(f)
                                 ^
1 error
javatool exit: /usr/bin/javac failed, exiting
BuildFilter: inferior command exited with code 1
gnumake[1]: *** [build-java-classes-server] Error 1
gnumake: *** [debug] Error 2
```