

INSIDE COCOA

Developing Cocoa Java Applications: A Tutorial



Preliminary

7/19/00

Technical Publications

© Apple Computer, Inc. 2000

Apple Computer, Inc.

© 1993-1995, 2000 Apple Computer, Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Computer, Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Computer, Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this book. Apple retains all intellectual property rights associated with the technology described in this book. This book is intended to assist application developers to develop applications only for Apple-labeled or Apple-licensed computers

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, and Macintosh are trademarks of Apple Computer, Inc., registered in the United States and other countries.

Simultaneously published in the United States and Canada

Even though Apple has reviewed this manual, **APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.**

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Chapter 1 Developing Cocoa Java Applications: A Tutorial 7

What You'll Learn in This Tutorial 7

Chapter 2 Building a Simple Application 9

Creating a Project	10
Launch Project Builder	11
Choose the New Project Command	11
Select Project Type	11
Creating the Interface	13
Open the Main Nib File	13
Resize the Window	14
Rename the Window	14
Put Text Fields in the Window	15
Set Attributes of the Text Fields	16
Add Labels for the Text Fields	17
Apply Aqua Layout Guidelines	18
Defining the Controller Class	19
Identify the Class and Its Superclass	19
Specify the Outlets of the Class	20
Specify the Action of the Class	20
Connecting Objects	21
Create an Instance of the Controller Class	21
Connect the Controller to Its Outlets	22
Connect the Action of the Controller	23
Connect the Responders	25
Test the User Interface	26
Implementing the Controller Class	27
Generate the Source Code Files	27

C O N T E N T S

Modify the Source Code Files	28
Implement the convert Method	29
Building and Running the Application	29
Build the Project	30
Launch and Test the Project	30

Chapter 3 Creating a Custom View Class 31

Defining the Custom View Subclass	32
Place and Resize the CustomView Object	32
Specify the Subclass	33
Assign the Class to the CustomView	34
Connecting the View Object	35
Specify a Controller Outlet	35
Connect the Instances	36
Implementing the Custom View	37
Generate the .java File	37
Implement the Constructor	38
Implement the Image-Setting Method	40
Call the Image-Setting Method	41
Completing the Application	41
Add Images to the Project	42
Build the Project	42
Test Drive the Application	42
Creating a Subclass of NSView	43
Define a Custom Subclass of NSView	44
Implement the Code for a Custom NSView	45
Drawing	45
Invalidating the View	46
Event Handling	46
An Example	47

Chapter 4	Debugging Java Applications	51
Preparing to Debug an Application		52
Build for Debugging	52	
Access the Java Debugger	53	
Using the Java Debugger	55	
Set and Manipulate Breakpoints	55	
Step Into and Over Code	57	
Get a Backtrace and Examine a Frame	59	
Examine Objects and Variables	60	
Printing a value	60	
Printing a reference (object)	61	
Printing a reference (class)	61	
Printing an object	62	
Printing the receiver (this)	62	
Debug Multiple Threads	62	
Debugging Java and Objective-C Simultaneously	63	
Set Up For Debugging	63	
Run both gdb and the Java Debugger	64	
Switching Between JavaDebug and gdb	67	
Java Debugger Command Reference	68	
Getting Help	68	
Thread Commands	68	
Stack and Data Inspection	69	
Control Functions	72	
Convenience Functions	74	

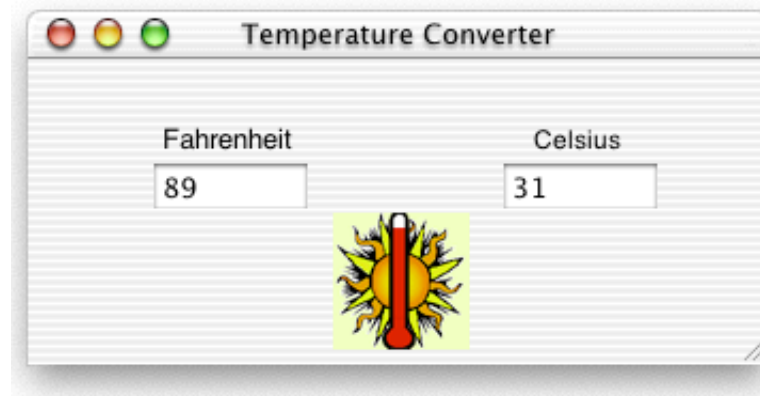
C O N T E N T S

Developing Cocoa Java Applications: A Tutorial

This tutorial introduces the Cocoa application framework of Mac OS X, and teaches you how to leverage Java™ to build robust, object-oriented applications using Apple's Mac OS X tools. Cocoa provides the best way to build modern, multimedia-rich, object-oriented applications for consumers and enterprise customers alike. This tutorial assumes a working knowledge of the Java language and concepts, but does not assume any previous experience with Cocoa, Project Builder, or Interface Builder.

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:

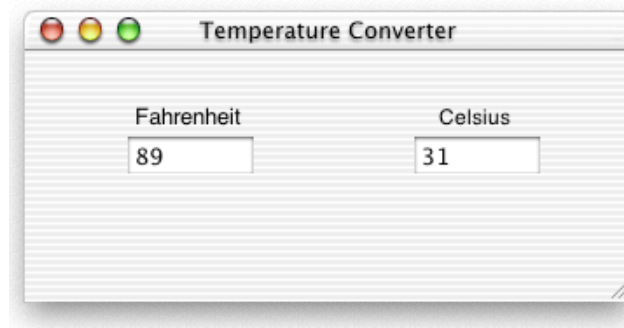
1. **Building a simple application.** Explains how to create a project and a graphical user interface, define a custom controller class, and connect an instance of that class to other objects in the application. It also shows how you must change the source code files generated by Interface Builder to 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 Cocoa Java applications. It also shows how you can debug projects that contain both Java code and Objective-C code.

Building a Simple Application

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 Cocoa application using Java. The tasks in this chapter include:

1. “Creating a Project” (page 10)
2. “Creating the Interface” (page 13)
3. “Implementing the Controller Class” (page 27)
4. “Connecting Objects” (page 21)
5. “Defining the Controller Class” (page 19)
6. “Building and Running the Application” (page 29)

By the end of this chapter you will have created an application called Temperature Converter that looks like this:



Building a Simple Application

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 Cocoa 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

Cocoa projects contain source code files, user interface files (nib files), and application property settings (.plist). Cocoa projects also link to application frameworks (Cocoa.framework, AppKit.framework, and Foundation.framework) and other libraries. Project Builder creates a folder structure for your project, links your project to the system frameworks, and sets the default property list settings depending on the type of project you choose.

Launch Project Builder

Project Builder is the application typically used for creating and managing development projects which use the Cocoa framework. To launch Project Builder:

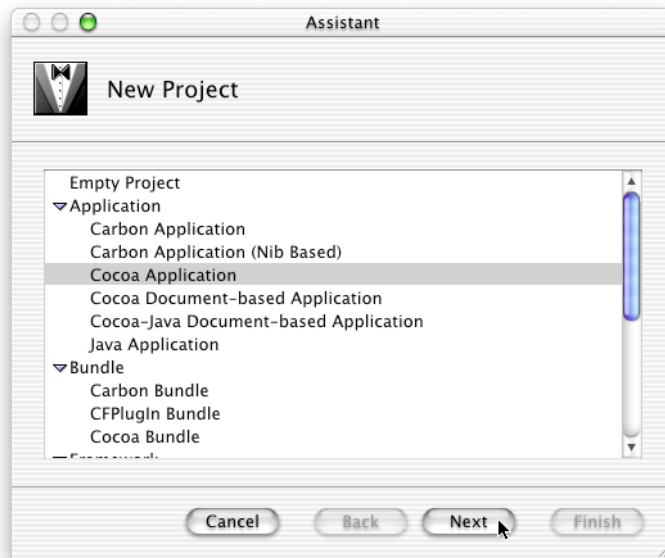
1. Find Project Builder in `/Developer/Applications` and select it.
2. Double-click the icon in the Finder.

Choose the New Project Command

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

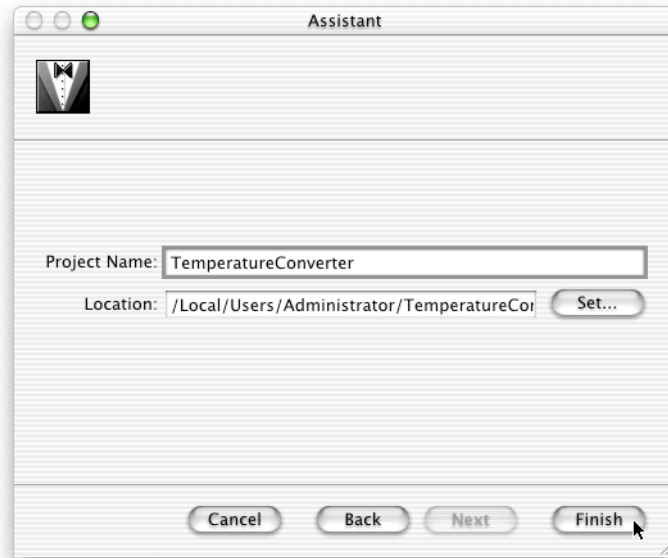
Select Project Type

Project Builder can build many different types of applications, including everything from Cocoa Java applications to Mac OS X kernel extensions and Mac OS X frameworks. For this tutorial, select Cocoa Application and click Next.



Building a Simple Application

1. Using the file-system browser, navigate to the directory where you want your project to be.
2. Type the name of the project in the Project Name field. For the current project, type the name `TemperatureConverter`.
3. Click Finish.



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

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

`English.lproj/`

A directory containing resources localized to your preferred language. In this directory are nib files automatically created for the project. For more information on nib files, see the developer documentation for Interface Builder.

`main.m`

A file, generated for each project, that contains the entry-point code for the application in `main()`. Usually, this file is not edited by the developer.

`TemperatureConverter.pbproj`

This contains information that defines the project, including the information property list (.plist) and project-specific preferences for Project Builder. It too should not be modified. Depending on the version of Mac OS X you are running, `TemperatureConverter.pbproj` may be a file or a folder. If it is a file, you can open your project by double-clicking it. If it is a folder, look for the file `project.pbxproj` inside and double-click it to open your project.

Creating the Interface

The process of developing a Cocoa application begins at the user interface level, rather than at the code level, and allows developers to create and connect objects visually. This allows for rapid application development while providing a genuine object-oriented development environment. Creating the interface includes these steps:

1. “Open the Main Nib File” (page 13)
2. “Resize the Window” (page 14)
3. “Rename the Window” (page 14)
4. “Put Text Fields in the Window” (page 15)
5. “Set Attributes of the Text Fields” (page 16)
6. “Add Labels for the Text Fields” (page 17)

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 main nib file for your project. By default, Project Builder names the first nib file `MainMenu.nib`. Find this file in the Resources category in the Groups and Files panel of the project target.

Building a Simple Application

2. Double-click MainMenu.nib to open Interface Builder.

Important

To learn more about Interface Builder, see the complete Interface Builder reference documentation on the Apple developer web site.

Resize the Window

The window provided for you in the main nib file is too large. To resize a window, drag the lower right 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" (page 32) in the second part of the tutorial for an example of how this is done).

Rename the Window

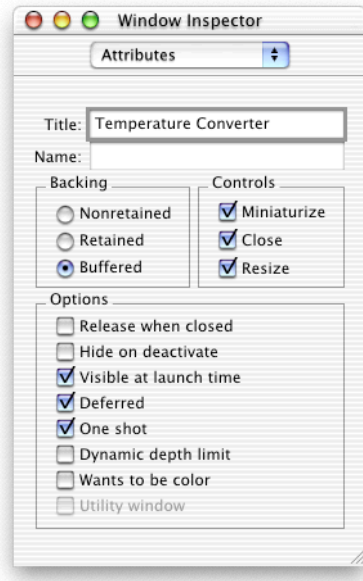
Windows usually carry distinctive titles in their title bars. In Cocoa, each window in a screen is based on an instance of `NSWindow`, which is analogous to the `Frame` class in Java. The title of a window is an attribute of this object. Interface Builder allows you to set the attributes of `NSWindows`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

To set the title of the `TemperatureConverter` window:

1. Select the window by clicking it.
2. Choose Tools > Inspector.

Building a Simple Application

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.”



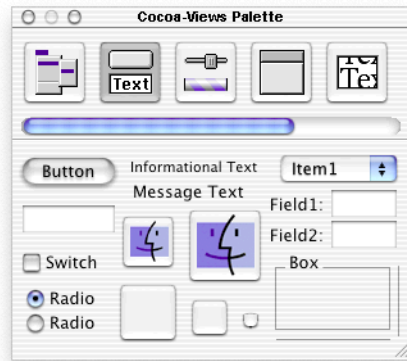
5. Uncheck the Close and Resize checkboxes. These window attributes don't make sense with an application as simple as this.

Don't worry about putting a value into the Name field.

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:



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. 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 Tools > Inspector.
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.

Building a Simple Application

Important

You do not have to set the “Send Action—Only on enter” attribute. When this button is turned on, the text field sends its action message (which will be connected to the `convert()` method) only when the user presses the Enter or Return key while the insertion point is in the field. If what you want is 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.

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. Double-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 > Show Fonts.
 - c. Select “From user's application font” from the “Use Family and Typeface” pop-up list.
 - d. In the Font panel, select 13 under Size.

Building a Simple Application



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 File > Save.

Apply Aqua Layout Guidelines

Interface Builder provides tools to help you build applications that are consistent with the Aqua user interface of Mac OS X. The complete Aqua layout guidelines are available in the document “Adopting the Aqua Interface,” available on the Apple developer web site.

To clean up Temperature Converter’s user interface to comply with the Aqua guidelines, choose Layout > Apply Layout Guidelines to Window. This will apply the correct spacing between user interface elements in the Temperature Converter window.

Defining the Controller Class

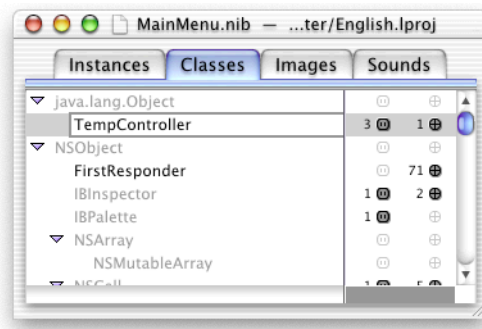
Interface Builder not only lets you construct the user interface of an application from real objects stored on palettes, but lets you partially define a class in terms of its name, its superclass, its outlets, and its actions. This process involves three basic steps:

1. “Identify the Class and Its Superclass” (page 19)
2. “Specify the Outlets of the Class” (page 20)
3. “Specify the Action of the Class” (page 20)

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 MainMenu.nib file window.
2. Highlight `java.lang.Object` in the list of classes.
3. Choose **Classes > Subclass**.
“MyObject” appears in an editable field below `java.lang.Object`.
4. Type `TempController` in place of “MyObject” and press Return.



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.

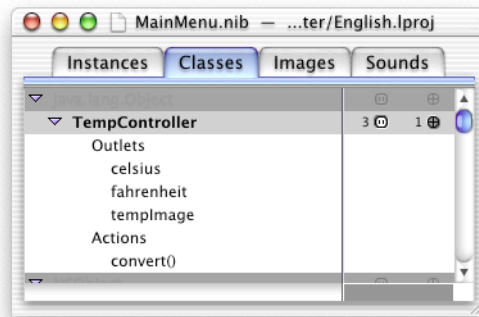
The area under `TempController` expands to include Outlets and Actions.

2. Select Outlets.
3. Choose Classes > Add Outlet.

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 as 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.

Building a Simple Application

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 Classes > Add Action.
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.

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. The process involves these steps:

1. “Create an Instance of the Controller Class” (page 21)
2. “Connect the Controller to Its Outlets” (page 22)
3. “Connect the Action of the Controller” (page 23)
4. “Connect the Responders” (page 25)
5. “Test the User Interface” (page 26)

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.

Building a Simple Application

2. Choose Classes > Instantiate.

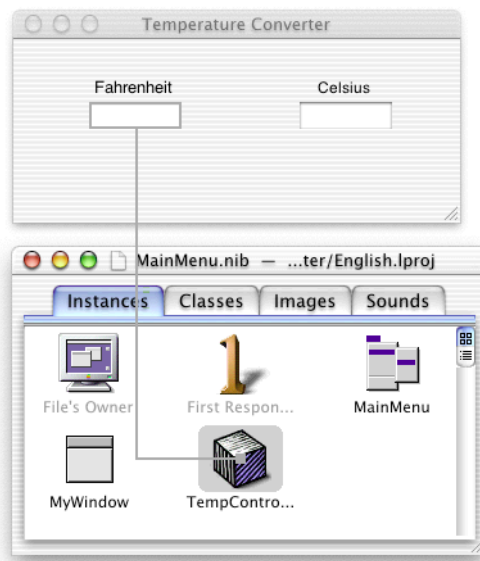
The nib file window automatically changes to the Instances display, and an instance of the TempController class (depicted as a cube) appears in the display.

Connect the Controller to Its Outlets

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

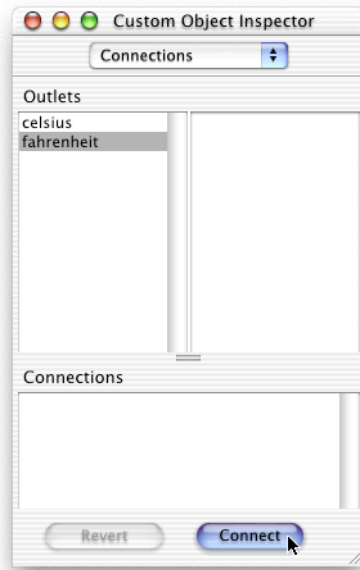
1. Control-drag from the cube representing the custom object to the Fahrenheit text field (the editable field, not the label). A thick black line follows the cursor while you drag.

“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.

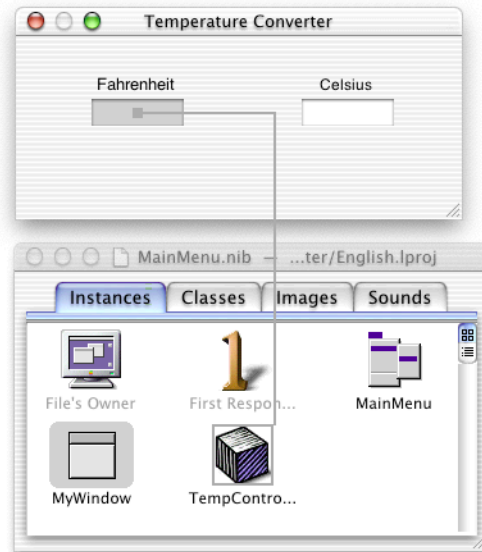
Repeat steps 1 through 4 for the celsius outlet.

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:

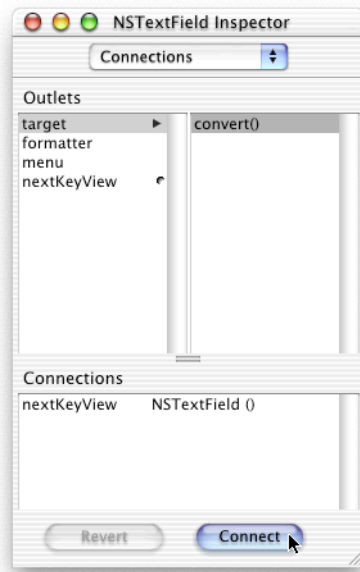
1. Control-drag from the Fahrenheit field (the editable field, not the label) to the cube representing the custom object. A thick black line follows the cursor while you drag.

Building a Simple Application



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.

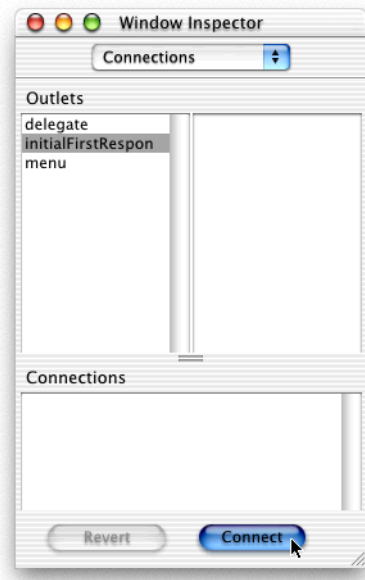
4. Click the **Connect** button.

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.

Test the User Interface

You can now test the user interface you've constructed with Interface Builder. Save the nib file and choose **File > Test Interface**. 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 Fahrenheit 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 Edit > Cut. Click in the other text field and choose Edit > Paste. 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.

1. "Generate the Source Code Files" (page 27)
2. "Modify the Source Code Files" (page 28)
3. "Implement the convert Method" (page 29)

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 Attributes from the Inspector and verify that the Java radio button is selected.
4. Choose Classes > Create Files.
5. Respond to the query "Create TempController.java?" by clicking Yes.
6. Respond to the query "Insert file in project?" by clicking Yes.

Interface Builder creates the file `TempController.java` 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.

Building a Simple Application

Important

In some versions of Mac OS X, Project Builder and Interface Builder are less integrated than they used to be, and you may have to manually import the generated Java files. If you are not presented with the query in point 5 above, navigate to your project directory in a Finder window and move the generated Java file from the English.lproj folder to the root folder of your project. Then in Project Builder, choose “Add Files...” from the Project menu and select the Java file that you just moved. Select the checkbox next to both “Copy into groups folder (if needed)” and “Recursively create groups for added folders.” Future versions of Mac OS X will eliminate these steps by improving the integration of Project Builder and Interface Builder.

Modify the Source Code Files

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

1. Click `tempController.java` in the Groups and Files pane.

The following code is displayed in the code editor:

```
import com.apple.cocoa.application.*;
import com.apple.cocoa.foundation.*;

public class TempController {
    Object celsius;
    Object fahrenheit;
    public void convert(Object sender){
    }
}
```

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

```
import com.apple.cocoa.application.*;
import com.apple.cocoa.foundation.*;

public class TempController {
    NSTextField celsius;
    NSTextField fahrenheit;
    public void convert(NSTextField sender) {
    }
}
```


Building a Simple Application

```
}
```

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`.

Implement the convert Method

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

```
import com.apple.cocoa.application.*;
import com.apple.cocoa.foundation.*;

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 Objective-C and native Java objects in the code. And you can use any Java language element, such as the try/catch exception handler.

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.

Building a Simple Application

1. “Build the Project” (page 30)
2. “Launch and Test the Project” (page 30)

Build the Project

To build the project:

1. Click the Build icon in the project window to build your project.



You can also press Command-B to build your project.

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.

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:

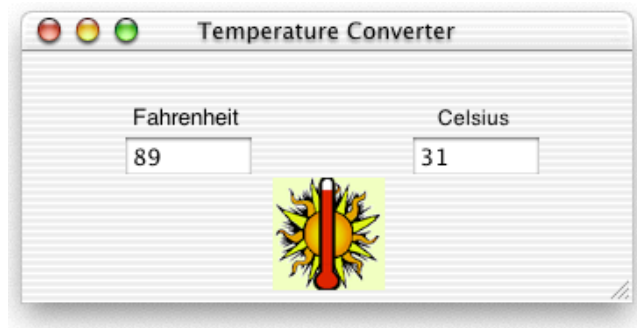
- Locate the file `TemperatureConverter.app` in the project directory. Double-click this file to launch the application.
- Click the Launch button on the project window to run Temperature Converter.



Creating a Custom View Class

This section of the tutorial describes the basic steps for creating a custom view and, more specifically, shows how to create a subclass of an Application Kit class that 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:



Major Tasks:

1. “Defining the Custom View Subclass” (page 32)
2. “Connecting the View Object” (page 35)
3. “Implementing the Custom View” (page 37)
4. “Completing the Application” (page 41)
5. “Creating a Subclass of `NSView`” (page 43)

Creating a Custom View Class

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 Cocoa classes that don’t inherit from `java.lang.Object`.

“Creating a Subclass of `NSView`” (page 43) 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

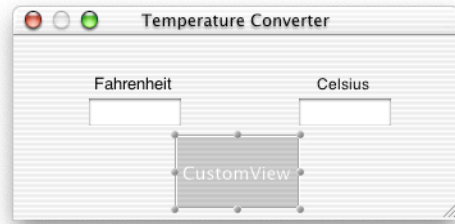
1. “Place and Resize the CustomView Object” (page 32)
2. “Specify the Subclass” (page 33)
3. “Assign the Class to the CustomView” (page 34)

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.

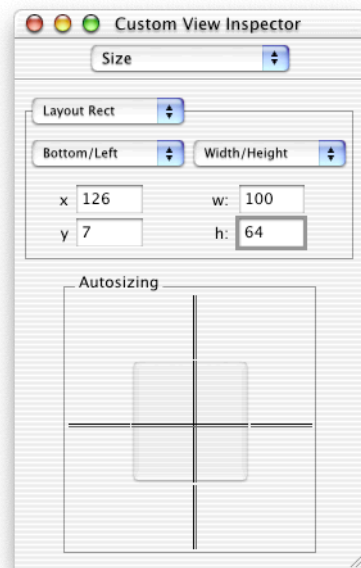
1. Drag the CustomView object from the Views palette and drop it in the window.
Center it in the window beneath the text fields.

Creating a Custom View Class



2. Resize the CustomView using the Size inspector.

Choose Inspector from the Tools menu, select the Size display, and enter 100 for the width (w) field and 64 for the height (h) field. You may have to select Width/Height from the right-most pull-down menu.



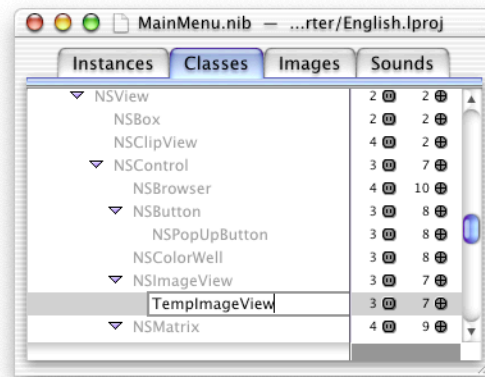
Specify the Subclass

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.

Creating a Custom View 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.



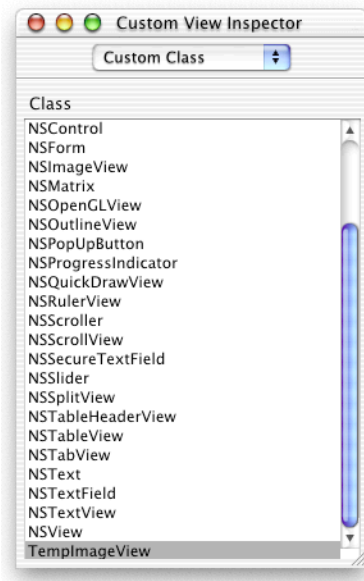
2. Choose Classes > Subclass.
3. Name the class TempImageView.

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

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



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.

1. “Specify a Controller Outlet” (page 35)
2. “Connect the Instances” (page 36)

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.

Creating a Custom View Class

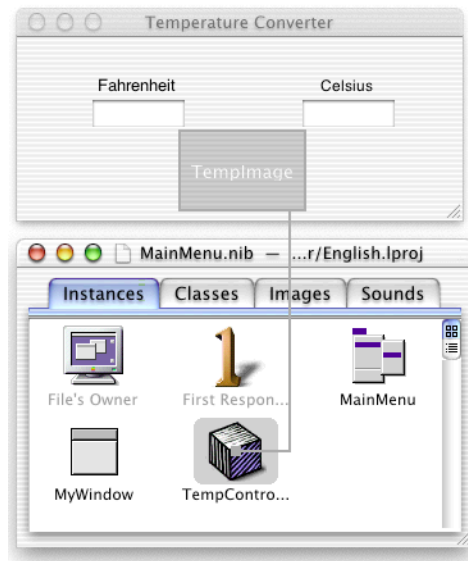
1. Select the TempController class in the Classes display of the nib file window.
2. Click the electrical-outlet icon next to the class.
3. Choose Add Outlet from the Classes menu (or just press Return).
4. Type the name of the outlet: “tempImage”.

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

The AppKit CustomView object would be useless without any programmatic alterations to it, so you must add some Java code to make the view a useful user interface element.

1. “Generate the .java File” (page 37)
2. “Implement the Constructor” (page 38)
3. “Implement the Image-Setting Method” (page 40)
4. “Call the Image-Setting Method” (page 41)

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.

Creating a Custom View Class



4. Choose **Classes > Create Files**.
5. Respond **Yes** to both confirmation prompts. See “Generate the Source Code Files” (page 27) if you need help with this step.

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. Click `TempImageView.java` in Project Builder’s **Groups and Files** pane.
2. Add the instance variables for the three `NSImages` as shown here:

```
/* TempImageView */
import com.apple.cocoa.application.*;
import com.apple.cocoa.foundation.*;

public class TempImageView extends NSImageView {
```

Creating a Custom View Class

```
protected NSImage coldImage; // add this
protected NSImage moderateImage; // add this
protected NSImage hotImage; // add this
```

3. Add this code to load the images, create NSImages, and assign them to instance variables.

```
public TempImageView (NSRect frame) {
    // load images
    super(frame);

    String h = NSBundle mainBundle().pathForResource ("Cold", "tiff");
    if (h != null) {
        coldImage = new NSImage(h, true);
    } else {
        System.err.println ("Image Cold.tiff not found.");
    }

    h = NSBundle mainBundle().pathForResource ("Moderate", "tiff");
    if (h != null) {
        moderateImage = new NSImage(h, true);
    } else {
        System.err.println ("Image Moderate.tiff not found.");
    }

    h = NSBundle mainBundle().pathForResource ("Hot", "tiff");
    if (h != null) {
        hotImage = new NSImage(h, true);
    } else {
        System.err.println ("Image Hot.tiff not found.");
    }

    // Add code from step 4 here.
}
```

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.

Creating a Custom View Class

- To import the images, `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. For instance, you could use the Java try/catch exception handler since Cocoa fully supports Java's JDK.
4. Add this code in place of the last comment in the code above to 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 Cocoa class, especially their arguments and return types. Mac OS X provides a tool to help you, `JavaBrowser`. Located in `/Developer/Applications/`, this tool lists all Cocoa Java classes, their constructors, variables, and methods.

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);
}
```

Creating a Custom View Class

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 of type `TempImageView` in `TempController.java`. So, your instance variable declarations in `TempController.java` should now look like:

```
NSTextField celsius;
NSTextField fahrenheit;
TempImageView tempImage;
```

Completing the Application

You're almost ready to run the application. There are just a few more steps:

1. "Add Images to the Project" (page 42)
2. "Build the Project" (page 42)
3. "Test Drive the Application" (page 42)

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, choose Project > Add Files.
2. Navigate to the following directory:

`Developer/Documentation/Cocoa/JavaTutorial/ApplicationImages`

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 Project Save from the Project menu.

Build the Project

Now you’re ready to build the project. Click the Build button in the project window, or press Command-B to build the project. If there are errors in the code, they will appear in the Build panel display. 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.

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.

Creating a Custom View Class

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 left 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 application 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. There are two major steps:

Creating a Custom View Class

1. “Define a Custom Subclass of NSView” (page 44)
2. “Implement the Code for a Custom NSView” (page 45)

Custom subclasses of NSView are usually 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 Class > Subclass, and name your subclass “TemperatureView”.
4. Add an outlet called “tempImage” to the TempController class so it can communicate with the subclass of NSView you just created. (If you completed the previous section on subclassing NSImageView, just reuse the tempImage outlet you created there).
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 a connection from the TempController instance variable to the CustomView object. (If you completed the previous section on subclassing NSImageView, the connection should still exist.)
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.

Creating a Custom View Class

If you are unsure how to complete any of these steps, refer to the “Defining the Custom View Subclass” (page 32) and “Connecting the View Object” (page 35) 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 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 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). 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.
- `NSAffineTransform` has methods for translating, rotating, and resizing graphical objects, such as those created with `NSBezierPath`.

Creating a Custom View Class

- 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`.)

Creating a Custom View Class

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.

Follow these steps to create the example:

1. Change the contents of `TemperatureView.java` to the code given below.
2. Change the type of the `TempImageView` instance variable in `TempController.java` to type `TemperatureView`.
3. After you run the project with the new code given below, you may want to change the size and/or position of the custom view object in Interface Builder.

```
/* TemperatureView */

import com.apple.cocoa.application.*;
import com.apple.cocoa.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;
```

CHAPTER 3

Creating a Custom View Class

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

    NSRect rect;
    NSColor color;

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

    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) {
```

C H A P T E R 3

Creating a Custom View Class

```
        thickness = 1;
    } else {
        thickness++;
    }
    setNeedsDisplay (true);
}

public boolean acceptsFirstResponder() {
    return true;
}
}
```

C H A P T E R 3

Creating a Custom View Class

Debugging Java Applications

Important

As of this writing, Java debugging in Project Builder is not yet implemented, so this chapter references the previous development environment, Project BuilderWO.

Major Tasks:

1. “Preparing to Debug an Application” (page 52)
2. “Using the Java Debugger” (page 55)
3. “Debugging Java and Objective-C Simultaneously” (page 63)

This part of the tutorial show you the basic steps for debugging Cocoa Java 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.

Project Builder uses the Java Debugger for debugging Cocoa Java 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 Cocoa 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.

Build for Debugging

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

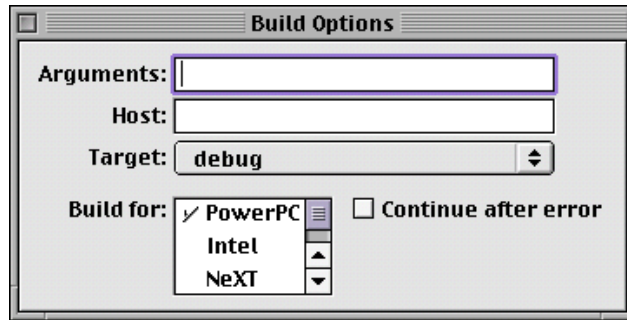


- b. Click the Options button on the Build panel:



2. Set the debug target.

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



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.

Debugging Java Applications

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:

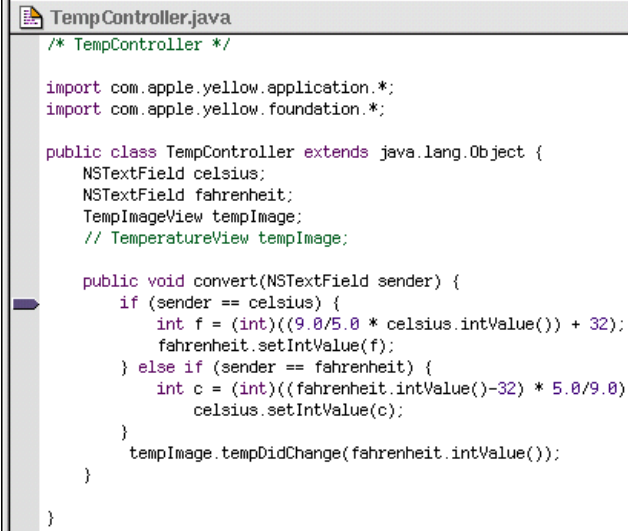
```
JavaDebug>> help
```

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.java
/* 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());
    }
}

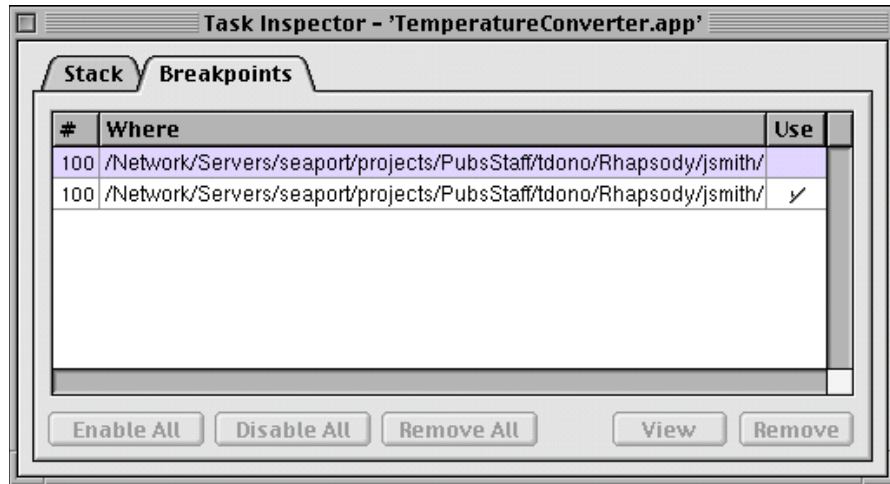
```

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.)



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.

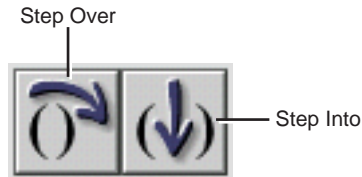
1. Enter a number in a TemperatureConverter field and press Return. Execution stops at the breakpoint you set in the previous exercise; the program counter is indicated by a red pointer:

```

    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);
    }
}

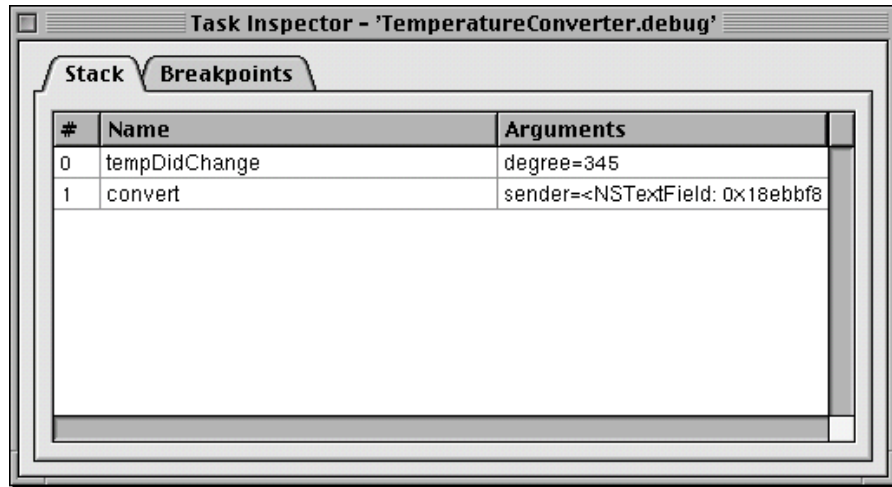
```

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.



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
```

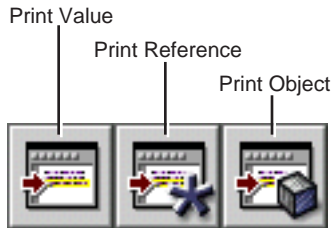
```
[0]TempImageView.tempDidChange (TempImageView:38) pc = 0
[1]TempController.convert (TempController:20) pc = 73
```

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
```


Debugging Java Applications

Printing a reference (object)

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

Printing a reference (class)

```
JavaDebug>> p* NSTextField
NSTextField = 0x3646c0:class(com.apple.cocoa.application.NSTextField) {
    superclass = 0x366408:class(com.apple.cocoa.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
}
```

Debugging Java Applications

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.cocoa.application.NSImage)0x39f670
    protected NSImage moderateImage =
(com.apple.cocoa.application.NSImage)0x39f688
    protected NSImage hotImage =
(com.apple.cocoa.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
```

Debugging Java Applications

```
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 `/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” (page 52) 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 Cocoa 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-macos10), 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 -NSPBDDebug
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” (page 55) 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>>”):

Debugging Java Applications

```
(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 as `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:

CHAPTER 4

Debugging Java Applications

```
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
```

CHAPTER 4

Debugging Java Applications

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:

CHAPTER 4

Debugging Java Applications

```
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” (page 74), 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"
```

CHAPTER 4

Debugging Java Applications

```
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
```

C H A P T E R 4

Debugging Java Applications

```
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.

Debugging Java Applications

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.macos10.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 = "macos10"
```

C H A P T E R 4

Debugging Java Applications

```
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 = "Internal Build"
path.separator = ":"
user.home = "/Local/Users/kend"
```

C H A P T E R 4

Debugging Java Applications