Project Builder for Java

(Legacy)

Java > Tools

2003-10-10

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

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction to Project Builder for Java

Important: The information in this document is obsolete and should not be used for new development.

This document addresses Java development in Mac OS X using Project Builder. Project Builder is part of Apple's integrated development environment.

Important: To run the examples described in this document, you must have installed Java 1.4.1 and the December 2002 (or later) Developer Tools package.

You should read this document if you are a Java developer who is interested in developing applications for Mac OS X or want to port an existing application into Mac OS X.

Organization of This Document

This document has the following chapters and appendixes:

- "Application Development" (page 11) introduces Java development using Project Builder. The chapter explains each of the Java-based templates, which give you a head start when developing a project.
- "Build System" (page 17) addresses the Project Builder build system. It describes build settings, build targets, and build styles.
- "Developing a Tool" (page 33) explains how to use the Java Tool template to develop a text-based Java application. This a good place to start if you're new to Java development in Mac OS X.
- "Developing a Swing Application" (page 41) explains how to use the Java Swing Application template to develop a graphical user interface–based application.
- "Developing a JNI Application" (page 51) provides an overview of the Java JNI Application template, which you can use to develop Java applications that need to interact with native code.
- "Debugging Applications" (page 55) focuses on Project Builder's debugging facilities.
- "Build Settings Reference" (page 61) describes the build settings that you may need to configure in Java applications.

Following the appendixes are a document revision history, and a glossary.

See Also

There are source files and Project Builder projects in the companion files of this document. They are located in `/Developer/ADC Reference Library/documentation/Java/Conceptual/Project_Builder_for_Java/Project_Builder_for_Java_companion.zip`; that directory is called `companion` in the remainder of this document. You can also download the companion files from http://developer.apple.com/documentation/Java/index.html.

For general information about Project Builder, see Project Builder Help. For information on specialized Project Builder customization, see *Customizing Project Builder*, at http://developer.apple.com/documentation/DeveloperTools/index.html.

Application Development

This chapter introduces the development of Java applications using Project Builder. Project Builder provides a development environment in which you can develop, build, and deploy Java applications. In addition, Project Builder has a project template that facilitates the development of applications that use the native Mac OS X environment, that is, applications that have both Java code as well as C or Objective-C code.

Project Builder templates are prebuilt projects that give you a head start in the development of an application. Figure 1-1 shows the New Project pane of the Project Builder Assistant, listing the Java project templates you can use to develop applications. When you want to develop a Swing-based application, for example, you can start with the Swing application template, which provides a fully configured application that follows Apple's guidelines for GUI (graphical user interface) applications. That template is also useful if you're new to Java and Swing and want to see the inner workings of a working application.

Figure 1-1 Project Builder templates for Java development

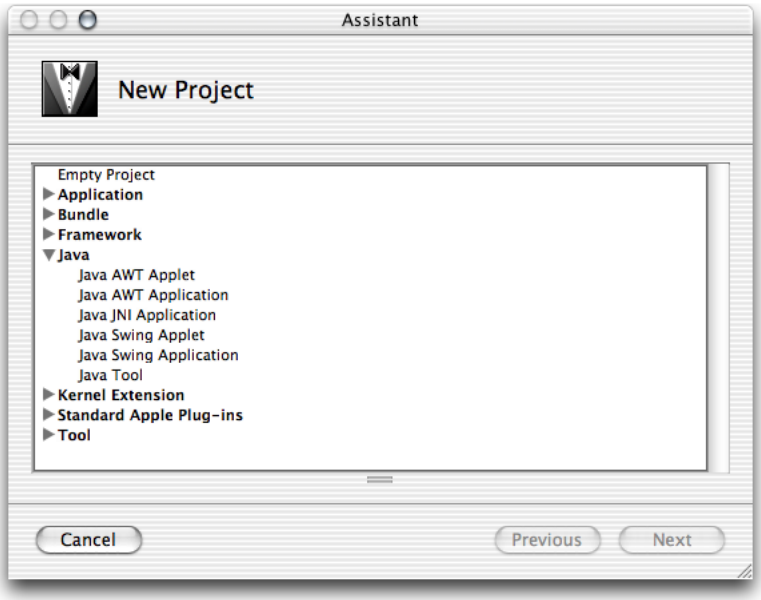


Table 1-1 shows the type of Java applications you can develop with Project Builder and their corresponding project templates.

Table 1-1 Applications types and their corresponding project templates

Application type	Template name
Text-based application	Java Tool
Swing applet	Java Swing Applet

Application type	Template name
Swing application	Java Swing Application
JNI (Java Native Interface) application	Java JNI Application
AWT (Abstract Window Toolkit) applet	Java AWT Applet
AWT application	Java AWT Application

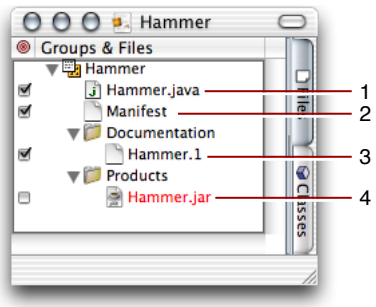
Project Builder has a powerful and flexible build system that facilitates the potentially complex tasks involved in building and deploying products, which include applications, libraries, frameworks, JAR files, and so on. The main elements involved in building products are targets. A project can contain more than one product, each produced by a target. In the case of text-based application projects, such as Java tool projects, the target is a JAR file created by the project's only target.

In general, a target encompasses instructions on how to build a product, which can be an application or a component of one. Build settings are properties that tell Project Builder how to build a product. Build phases are concrete steps Project Builder takes to build a target; for example, compiling source files into object files and linking object files to create an executable file. For more information, see "Targets" (page 19), "Build Settings" (page 17), and "Build Phases" (page 30).

The Tool Template

The Java Tool template provides the files needed to create a simple, text-based application. It includes source files for the class with the `main` method, the JAR manifest, and the man page. Figure 1-2 shows the files that make up a Java tool project.

Figure 1-2 The files of a Java tool project



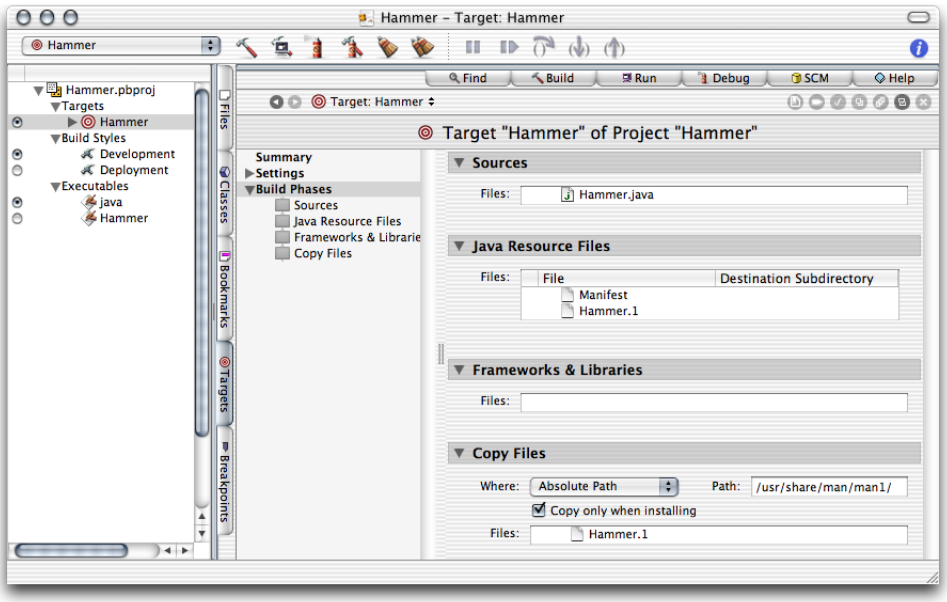
The following list describes the files of a Java tool project named Hammer:

1. `Hammer.java`: Java source file that contains the `main` method. Project Builder names this file after the project.
2. `Manifest`: File that contains information that Project Builder adds to the `MANIFEST.MF` file of the generated JAR file.
3. `Hammer.1`: Source for the man page that documents the tool.

4. `Hammer.jar`: JAR file in which Java class files, the manifest file, and other resources are stored for distribution. This is the product of the project. It's red because it hasn't been produced yet, so the file doesn't exist in the file system.

Figure 1-3 shows the target editor for the Hammer project.

Figure 1-3 Target editor for the Hammer project.



The items under Build Phases in the target editor list the build phases of the Hammer target. The phases are executed from top to bottom when the product is built. That is, the build phases are executed in the following order:

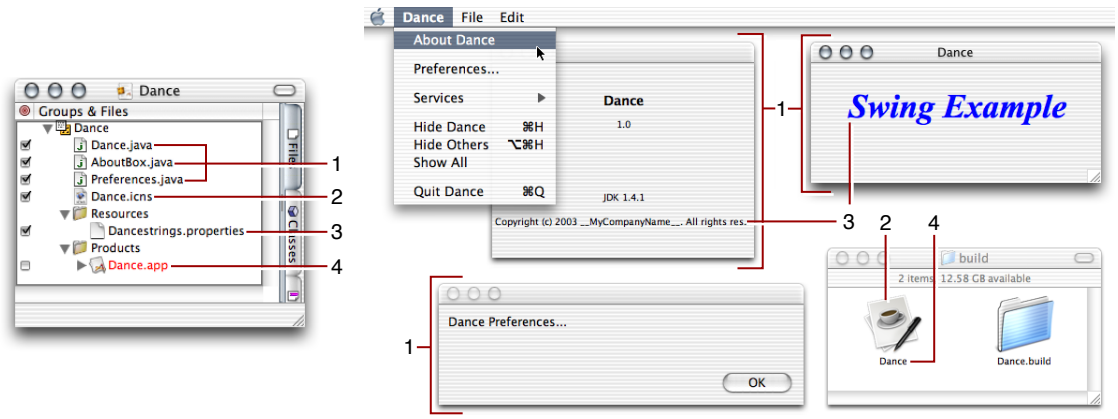
1. **Sources** Determines which Java source files are to be compiled (run through the `javac` compiler).
2. **Java Resource Files** Indicates which files to copy to the root level of the product (the top level of the JAR file).
3. **Frameworks & Libraries** Lists frameworks or libraries to which the Java class files generated in step 1 must link against.
4. **Copy Files** Copies files to specific parts of a product (for example, its resources directory or its plug-ins directory).

The Swing Application Template

The Java Swing Application template provides the files needed to create a desktop application. It includes source files for a controller class (which includes the `main` method) and two JFrames that the user can make visible through menu commands, an icon file, and a properties file. Figure 1-4 shows the files that make up a Java Swing application project.

Figure 1-4 shows the project's files in the Project Builder window, the contents of the project's build folder, and the running application.

Figure 1-4 The files of a Java Swing application project



The following list describes the files in a Java Swing application project named Dance and their relationship to the actual application:

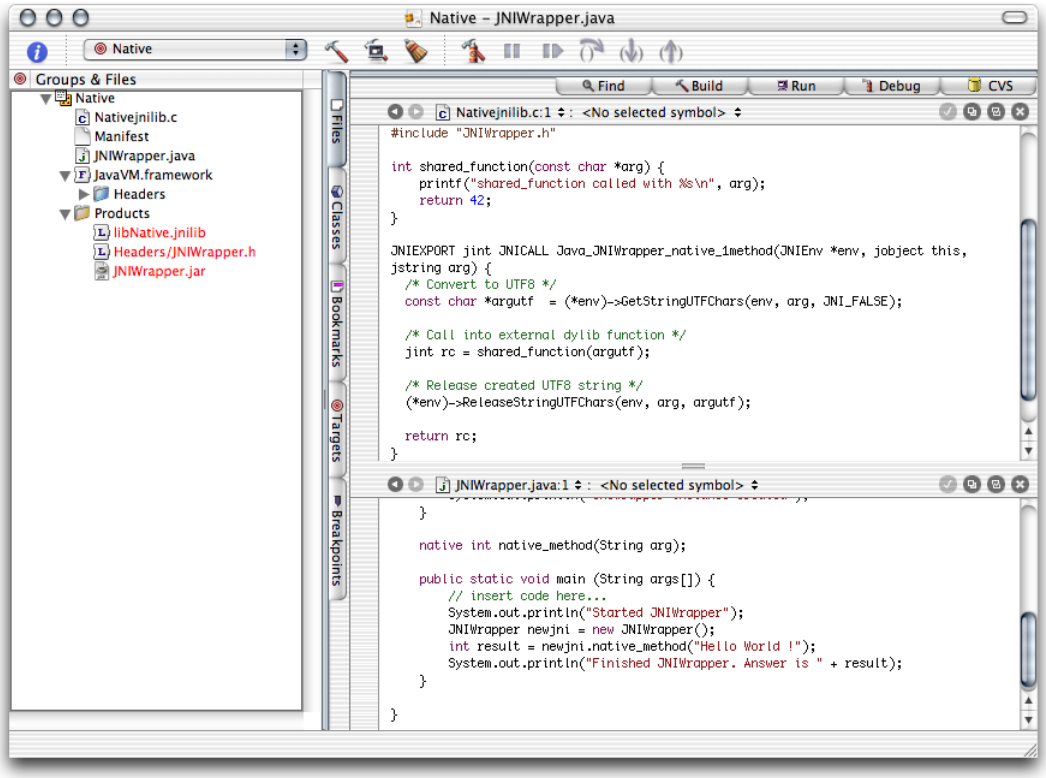
1. `Dance.java`, `AboutBox.java` and `Preferences.java`: Java source files that implement an About box and a preferences dialog.
2. `Dance.icns`: Icon file that contains the icon that the Finder displays for the application package.
3. `Dancestrings.properties`: File that contains the names and values of application properties accessible at runtime. Project Builder places this file inside the JAR file for the application.
4. `Dance.app`: Application package that contains Mac OS X–specific information for the application, as well as the application's JAR file.

The JNI Application Template

The Java Native Interface (JNI) provides a standard interface for communication with native libraries. You may want to use the JNI if you need to interface with native, legacy code from Java applications or when you want to improve the performance of an application by porting certain tasks to native code.

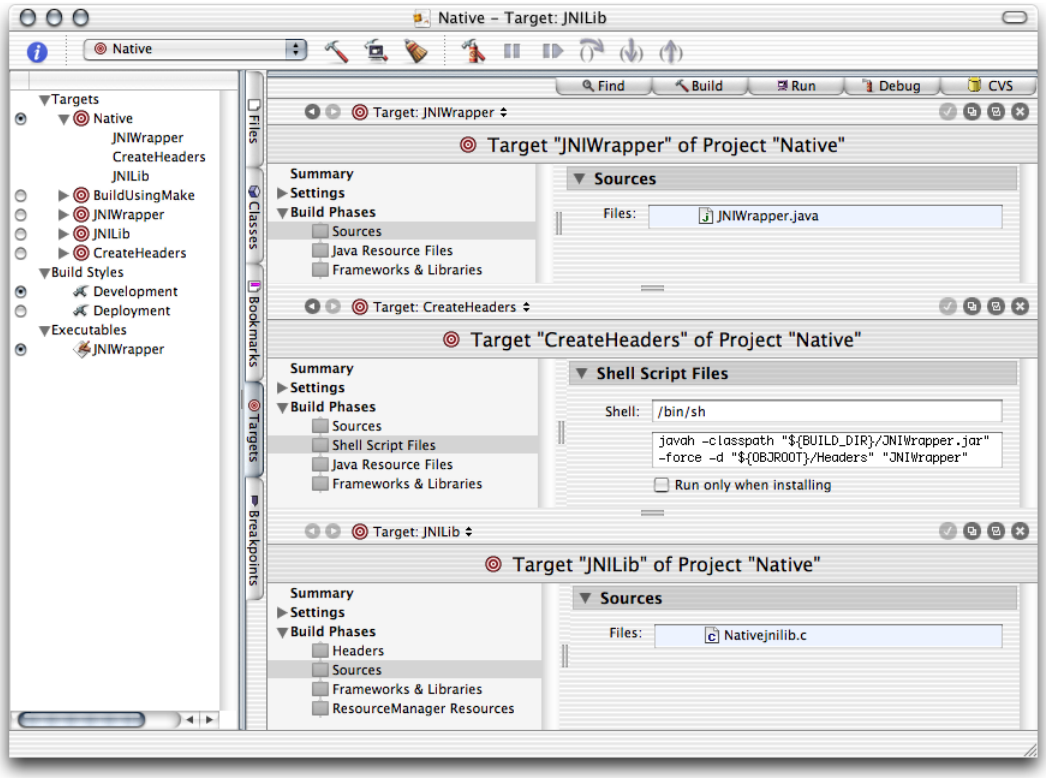
Project Builder provides a template with which you can develop projects that include both native code and Java code. Figure 1-5 shows a project called Pronto created with the JNI Application template.

Figure 1-5 A JNI-based application project



The most interesting part of the Pronto project are its targets. While the previous project types, tool and Swing application, required only one target, a JNI project requires several targets. This is because a JNI project contains three products, a JNI library (which contains the compiled C code), a header to the library, and a JAR file for the Java side of the application. See Figure 1-6.

Figure 1-6 Targets of JNI-based application project



The project has three main targets:

- **JNIWrapper** Compiles the Java source files of the application and archives them in a JAR file. This is the Java application.
- **CreateHeaders** Creates C function prototypes from Java class files in the JAR file generated by the JNIWrapper target.
- **JNILib** Builds the native library by compiling `Prontojnilib.c` and linking it with the Java VM framework (`/System/Library/Frameworks/JavaVM.framework`).

The Native target is an aggregate target. Its purpose is to enclose the JNIWrapper, CreateHeaders, and JNILib targets into one build unit, so that any action performed on it is performed on all the targets it contains. The BuildUsingMake target bypasses the Project Builder build system. It uses `gnumake` (`/usr/bin/gnumake`) to build the application.

You can find detailed information on the JNI at http://java.sun.com/j2se/1.4/docs/guide/jni/.

Build System

This chapter discusses the Project Builder build system, which determines how applications are built. Project Builder uses the Jam software build tool as its build engine. Jam allows Project Builder to easily manage dependencies between a project's elements. It can also take advantage of computers with two or more central processing units (CPUs).

Build Settings

Build settings are similar to Java properties: They store values that Project Builder uses to build products. Project Builder facilitates configuring some build settings through specialized panes (see "Target Information Panes" (page 19)). However, you can set the value of any build setting directly through expert panes. Expert panes show the configuration build settings as a list of key-value pairs. Through these panes you can set the values of build settings for which the more user-friendly specialized panes do not provide a user interface.

The following tables list some of the build settings you may have to use in your projects. "Build Settings Reference" (page 61) has a complete list of Java-related settings. See the Project Builder release notes for a complete list of all settings.

Table 2-1 lists build settings that identify a project and tell Project Builder where to put temporary files generated during product building.

Table 2-1 Project build settings

Build setting	Description
`PROJECT_NAME`	Name of the project. For example, `MyProject`. You should not modify this setting directly.
`SYMROOT`	Base location for built products. For example, `MyProject/build`.
`BUILD_DIR`	Base location for the temporary files generated by a project's targets. For example, `MyProject/build`. You should not modify this setting directly.
`TARGET_BUILD_DIR`	Base location for built products. It's set to `$BUILD_DIR` in development builds (for example, `MyProject/build`), `$INSTALL_DIR` (for example, `/tmp/My-Project.dst/usr/bin`) in deployment builds when the product is installed, and `$BUILD_DIR/UninstalledProducts` when the product is not installed.

Table 2-2 lists build settings that determine where files are placed when you use `pbxbuild` to install a product.

Table 2-2 Deployment build settings

Build setting	Description
DSTROOT	Base location for the installed product. For example, /tmp/MyProject.dst/.
INSTALL_PATH	Location of the installed product. For example usr/bin.
INSTALL_DIR	Fully qualified path for the installed product. By default, it concatenates DSTROOT and INSTALL_PATH. So, with the example values, it evaluates to /tmp/MyProject.dst/usr/bin. You should not modify this setting directly.

Table 2-3 lists build settings that identify a target and tell Project Builder where to put the files it generates.

Table 2-3 Target build settings

Build setting	Description
TARGET_NAME	Name of the target. For example, MyProject. You should not modify this setting directly.
ACTION	The action being performed on a target. Its possible values are build, clean, or install (through pbxbuild). You should not modify this setting directly.
TEMP_DIR	Location for a target's temporary files. For example, MyProject/build/MyProject.build/MyTarget.build.

Table 2-4 lists build settings used to call javac or jikes to compile Java source files.

Table 2-4 Java compiler build settings

Build setting	Description
CLASS_FILE_DIR	Base location for Java class files. For example, MyProject/build/MyProject.build/MyTarget.build/JavaClasses.
JAVA_COMPILER_TARGET_VM_VERSION	Defines the Java virtual machine version that javac compiles Java source files to—for example, 1.4. By default, this setting is undefined.

Table 2-5 lists the build setting that defines the archive of Java class files and the creation of the application package.

Table 2-5 Java application build setting

Build setting	Description
JAVA_MANIFEST_FILE	Path (relative to the project's root directory) to a manifest file to use when archiving Java class files into a JAR file. For example, Manifest.

Targets

Project Builder targets represent a product, such as an application or a framework. A project can produce more than one product. For example, a project can contain Java source files, which are compiled into Java class files by `javac`, and Objective-C source files, which are compiled into object files by `gcc`. Such a project must contain at least two targets, one that compiles the Java sources files and another that compiles the Objective-C source files. The build settings introduced in "Build Settings" (page 17) are what Project Builder uses to determine how to build a product.

Each target has its own set of build setting values; they are autonomous entities within a project. However, you can tell Project Builder that a target depends on one or more additional targets. That way you can guarantee, for example, that when target A needs files produced by target B, target B is executed before target A. In addition, if there's a problem with target B, target A doesn't get executed.

In addition, Project Builder provides the ability to add *aggregate* targets to a project. An aggregate target contains no product-building instructions; instead, it groups other targets. The operations you perform on aggregate targets are carried out on all the targets they enclose.

Each target can contain some or all of the following types of elements:

- **Build settings** The group of build settings that control the build system.
- **Information property list entries** Application package–specific information, such as type, version, icon, and so on.
- **Build phases** Types of tasks to perform on a set of a project's files, such as compile, link, archive, copy, and so on. See "Build Phases" (page 30) for more information.

For more information on targets, see Project Builder Help.

Target Information Panes

Target information panes group information about how a product is built. They contain a user-friendly view of the values of certain build settings. These information panes are grouped in three major groups: Summary, Settings and Info.plist Entries.

Target Summary

The Summary pane shows summary information for a project, including its name, type, and developer comments; it's shown in Figure 2-1.

Figure 2-1 Target Summary pane of the target editor in Project Builder

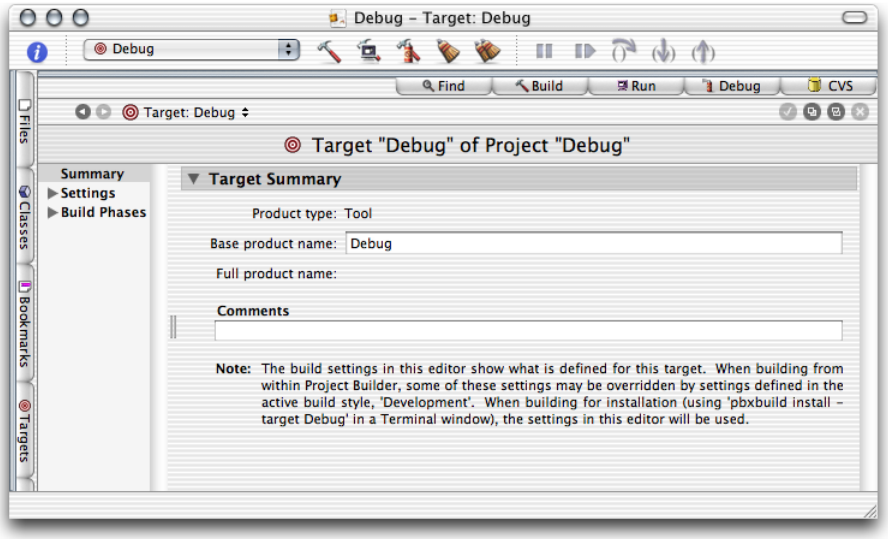


Table 2-6 describes the elements of the Target Summary pane.

Table 2-6 Elements of the Target Summary pane

Element label	Description	Corresponding build setting
Product type	Indicates the type of project. Can be Application, Tool, Framework, and so on.	*None*.
Base product name	Name of the generated product file without an extension.	`PRODUCT_NAME`
Comments	Developer comments about the target.	*None*.

Build Settings

The Build Settings pane groups views of the build settings of a project. It includes two views: Simple View and Expert View. The Simple View provides a easy-to-use user interface to various build settings. The Expert View lists all the build settings. You can use this view when the other views don't provide a way of configuring a particular build setting.

General Settings

The General Settings pane, depicted in Figure 2-2, shows information that pertains to the entire project. Table 2-7 describes its elements.

Figure 2-2 General Settings pane of the target editor in Project Builder

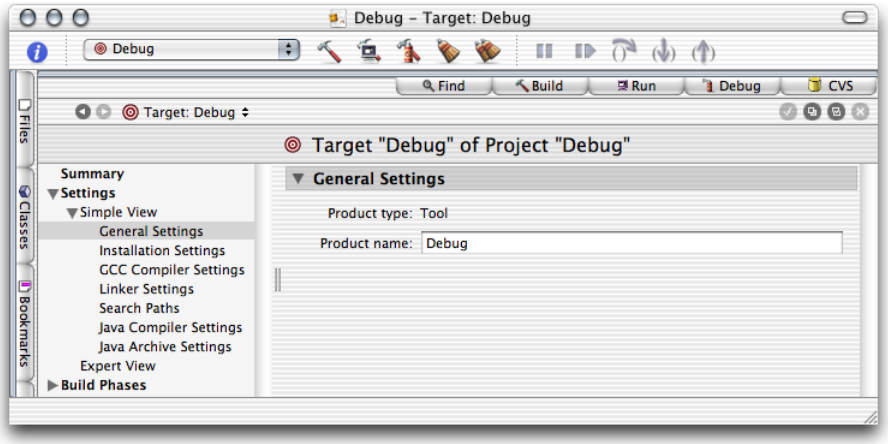


Table 2-7 Elements of the General Settings pane

Element label	Description	Corresponding build setting
Product type	Indicates the type of project. Can be Application, Tool, Framework, and so on.	*None*.
Product name	Name of the generated product file without an extension.	`PRODUCT_NAME`

Installation Settings

The Installation Settings pane, depicted in Figure 2-3, shows installation information for the selected target. Table 2-8 describes its elements.

Figure 2-3 Installation Settings pane of the target editor in Project Builder

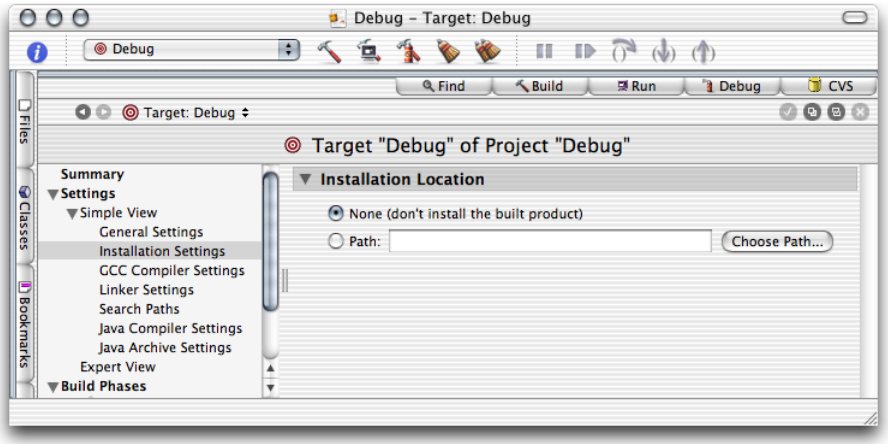


Table 2-8 Elements of the Installation Settings pane

Element label	Description	Corresponding build setting
None	When selected, the product doesn't get installed.	*None*.
Path	When selected, the product gets installed in the directory entered in the text input field.	`INSTALL_PATH`

Search Paths

The Search Paths pane, depicted in Figure 2-4, determines the places Project Builder searches for frameworks, libraries, Java classes, and headers (in the case of a JNI application) to build the selected target. Table 2-9 describes its elements.

Figure 2-4 Search Paths pane of the target editor in Project Builder

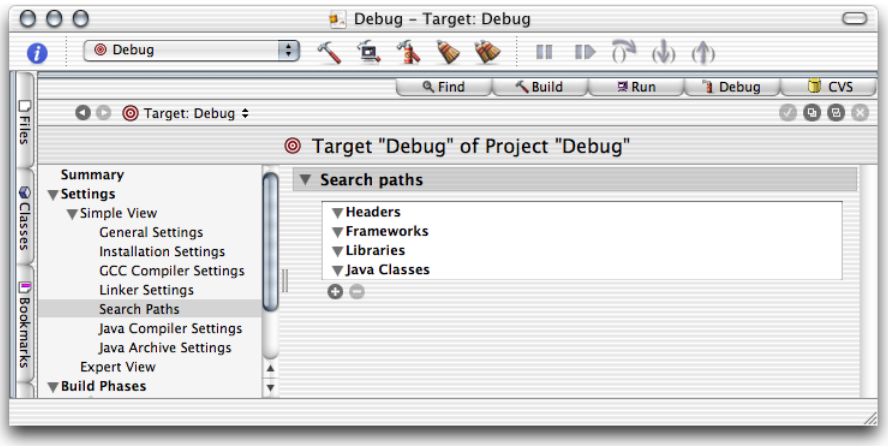


Table 2-9 Elements of the Search Paths pane

Element label	Description	Corresponding build setting
Headers	Search paths for Objective-C header files.	`HEADER_SEARCH_PATHS`
Frameworks	Search paths for frameworks.	`FRAMEWORK_SEARCH_PATHS`
Libraries	Search paths for libraries.	`LIBRARY_SEARCH_PATH`
Java Classes	Search paths for Java class files or JAR files.	`JAVA_CLASS_SEARCH_PATHS`

Java Compiler Settings

The Java Compiler Settings pane, depicted in Figure 2-5, determines some compiler settings for the selected target. Table 2-10 describes its elements.

Figure 2-5 Java Compiler Settings pane of the target editor in Project Builder

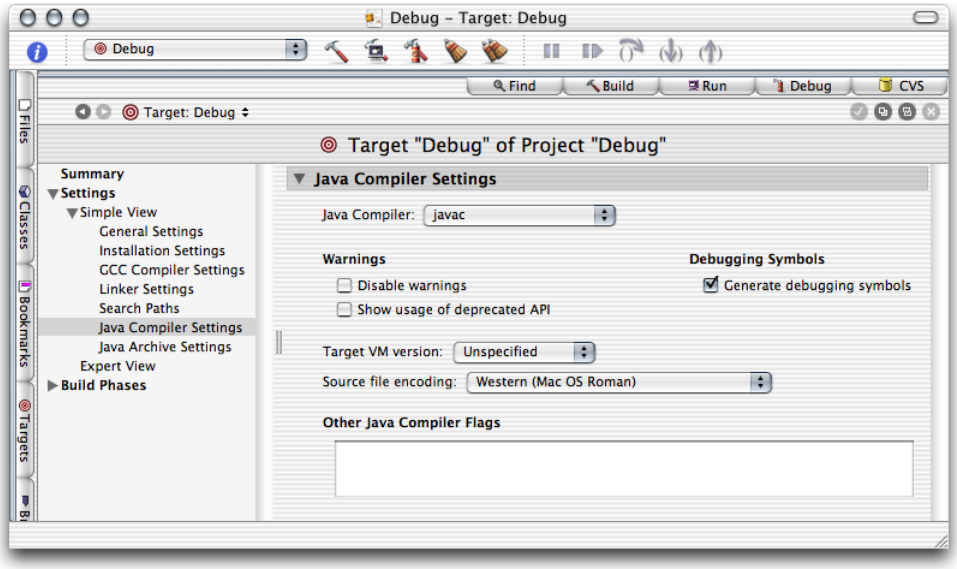


Table 2-10 Elements of the Java Compiler Settings pane

Element label	Description	Corresponding build setting
Java Compiler	Determines the compiler to use to compile Java source files. The options are `javac` and `jikes`.	`JAVA_COMPILER`
Disable warnings	When selected, the compiler doesn't produce warnings.	`JAVA_COMPILER_-DISABLE_WARNINGS`
Show usage of deprecated API	When selected, the compiler warns about deprecated API use.	`JAVA_COMPILER_-DEPRECATED_WARNINGS`
Generate debugging symbols	When selected, the compiler generates debugging symbols.	`JAVA_COMPILER_-DEBUGGING_SYMBOLS`
Target VM version	The virtual machine version the compiler is to produce Java class files for.	`JAVA_COMPILER_-TARGET_VM_VERSION`
Source file encoding	Specifies the character encoding used in all the Java source files that are to be compiled.	`JAVAC_SOURCE_FILE_-ENCODING`
Other Java Compiler Flags	Additional compiler options.	`JAVA_COMPILER_FLAGS`

Java Archive Settings

The Java Archive Settings pane, depicted in Figure 2-6, determines how Java class files in the selected target are archived. Table 2-11 describes its elements.

Figure 2-6 Java Archive Settings pane of the target editor in Project Builder

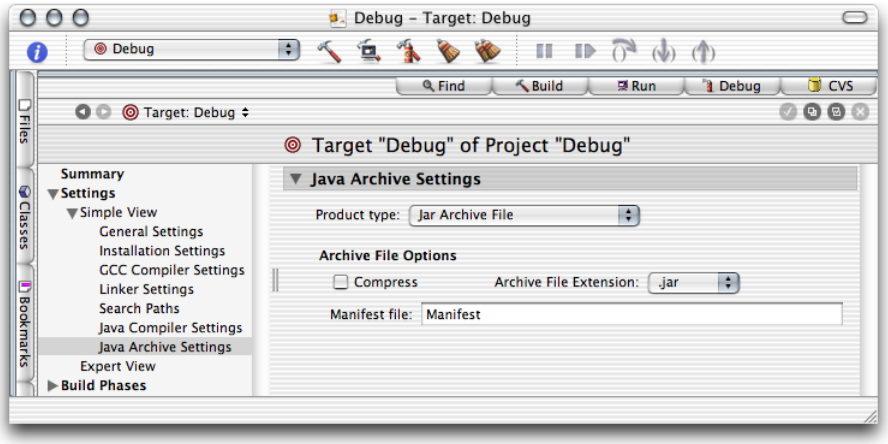


Table 2-11 Elements of the Java Archive Settings pane

Element label	Description	Corresponding build setting
Product type	Determines whether Java class files are archived in a JAR file.	`JAVA_ARCHIVE_CLASSES`
Compress	When unselected, the Java class files are stored in the JAR file, but are not compressed.	`JAVA_ARCHIVE_COMPRESSION`
Archive file extension	The extension to use for the JAR file. The options are `.jar`, `.war`, and `.ear`.	`CLASS_ARCHIVE_SUFFIX`
Manifest file	Name of the supplemental manifest file.	`JAVA_MANIFEST_FILE`

Information Property List Entries

Information property lists (`Info.plist` files) contain information an application can access at runtime. This is similar to Java's system properties. Information property lists, however, specify Mac OS X–specific application details, such as the application type and its icon. In addition, some Java-specific settings are also stored there; for example, the Java VM version that Mac OS X uses to run the application.

The following sections describe the simple views of information property list entries. See *Mac OS X Developer Release Notes: Information Property List* at http://developer.apple.com/releasenotes/index.html for more information about information property lists.

Basic Information

The Basic Information pane, depicted in Figure 2-7, encapsulates identification information about the application package. Table 2-12 describes its elements.

Figure 2-7 Basic Information pane of the target editor in Project Builder

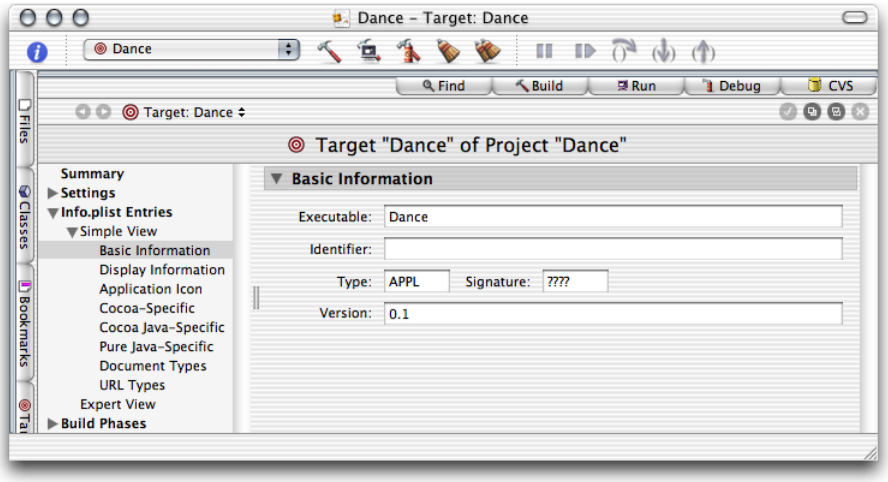


Table 2-12 Elements of the Basic Information pane

Element label	Description	Corresponding Info.plist entry
Executable	Name of the file containing the application's executable code.	`CFBundleExecutable`
Identifier	Package-style name (for example, `com.apple.ProjectBuilder`) used to uniquely identify the application or bundle.	`CFBundleIdentifier`
Type	Four-letter type indicator for the bundle. For example, `APPL` for applications, `FMWK` for frameworks, and so on.	`CFBundlePackageType`
Signature	Four-letter creator code for the bundle.	`CFBundleSignature`
Version	Version number for the bundle. For example, `10.2.3`.	`CFBundleVersion`

Display Information

The Display Information pane, depicted in Figure 2-8, encapsulates display information about the application package file. Table 2-13 describes its elements.

Figure 2-8 Display Information pane of the target editor in Project Builder

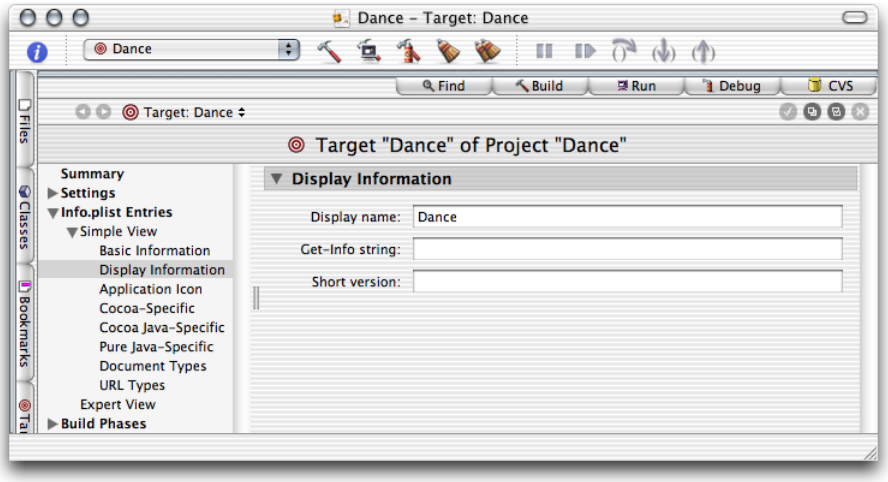


Table 2-13 Elements of the Display Information pane

Element label	Description	Corresponding Info.plist entry
Display name	In application packages, localized name that is displayed in the menu bar.	`CFBundleName`
Get-Info string	Localized string that appears in Info windows or the Inspector in the Finder.	`CFBundleGetInfoString`
Short version	Localized string with bundle-version information. This is the string displayed in Info windows or the Inspector in the Finder when `CFBundleGetInfoString` is undefined.	`CFBundleShortVersion-String`

Application Icon

The Application Icon pane, depicted in Figure 2-9, identifies the icon file to be used for the application package's icon, which is the icon the Finder displays to the user. Table 2-14 describes its elements.

Figure 2-9 Application Icon pane of the target editor in Project Builder

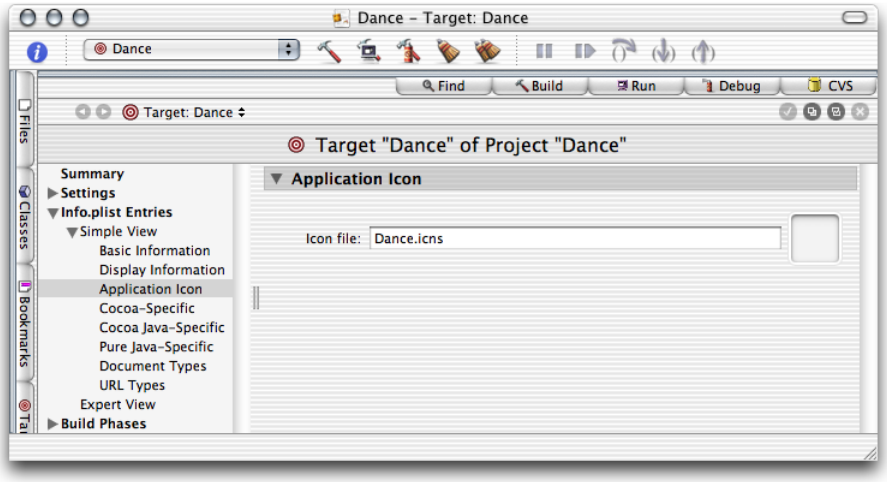


Table 2-14 Elements of the Application Icon pane

Element label	Description	Corresponding Info.plist entry
Icon file	Name of the icon file for the bundle.	`CFBundleIconFile`

Cocoa Java–Specific

The Cocoa Java–Specific pane, depicted in Figure 2-10, contains information specific for Cocoa applications written in Java. Table 2-15 describes its elements.

Figure 2-10 Cocoa Java–Specific pane of the target editor in Project Builder

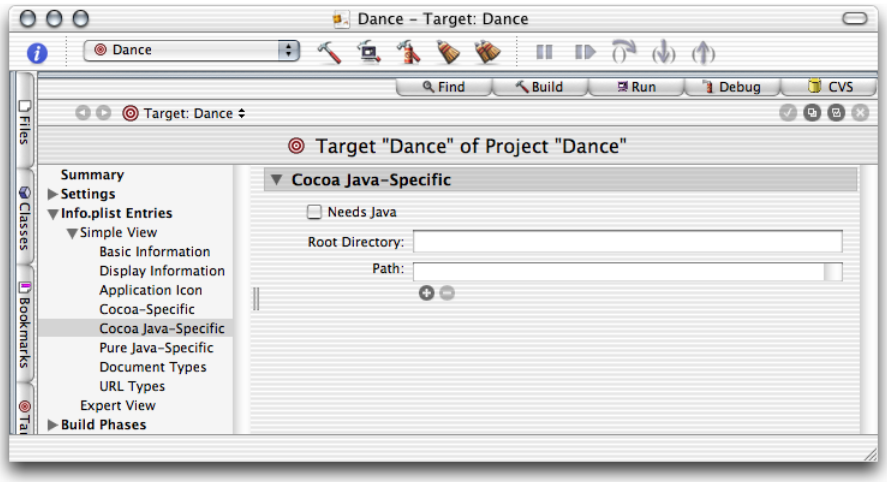


Table 2-15 Elements of the Cocoa Java–Specific pane

Elements label	Description	Corresponding Info.plist entry
Needs Java	When selected, indicates that a Cocoa application needs to instantiate a Java virtual machine.	`NSJavaNeeded`
Root Directory	The directory where the application's JAR files are stored in the application bundle. For example, `Contents/Resources/Java`.	`NSJavaRoot`
Path	List of JAR files contained in the root directory.	`NSJavaPath`

Pure Java–Specific

The Pure Java Specific pane, depicted in Figure 2-11, contains settings that are specific to Pure Java. Table 2-16 describes its elements.

Figure 2-11 Pure Java–Specific pane of the target editor in Project Builder

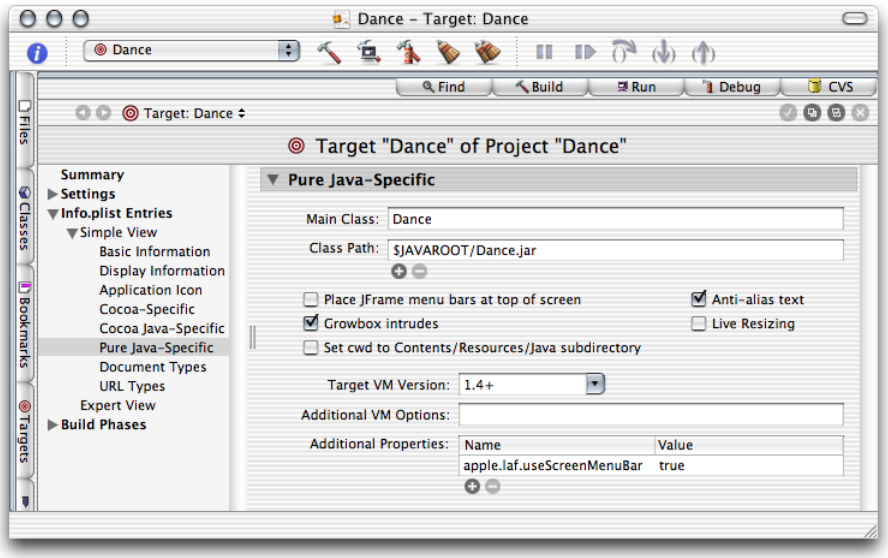


Table 2-16 Elements of the Pure Java–Specific pane

Element label	Description	Corresponding Info.plist entry
Main Class	Fully qualified name of an application's main class.	`Java/MainClass`
Class Path	List of paths to Java class files of JAR files the application uses.	`Java/ClassPath/`

Element label	Description	Corresponding Info.plist entry
Place JFrame menu bars at top of screen	When selected, the application's menu bar follows Mac OS X style: It's placed at the top of the screen instead of within each application window.	`Java/Properties/com.apple.macos.useScreenMenuBa`
Growbox intrudes	When selected, the resize control is part of the window pane. When unselected, a white band is added to the bottom of the window, so that the resize control doesn't intrude in the windows' content.	`Java/Properties/com.apple.mrj.application.growb`
Set cwd to Contents/Resources/Java subdirectory	When selected, the application's working directory is set to the bundle's `Contents/Resources/Java` directory.	`Java/WorkingDirectory`
Anti-alias text	Toggles text anti-aliasing.	`Java/Properties/com.apple.macosx.AntiAliasedTex`
Live resizing	Toggles live resizing of windows.	`Java/Properties/com.apple.mrj.application.live-`
Target VM Version	Version of the Java runtime the application requires. For example, `1.4+`.	`Java/JVMVersion`
Additional VM Options	Command-line options to add to the java invocation. For example, `-Xfuture -Xprof`.	`Java/VMOptions`
Additional Properties	Additional Java system properties, which you can access through `System.getProperty`.	`Java/Properties/`

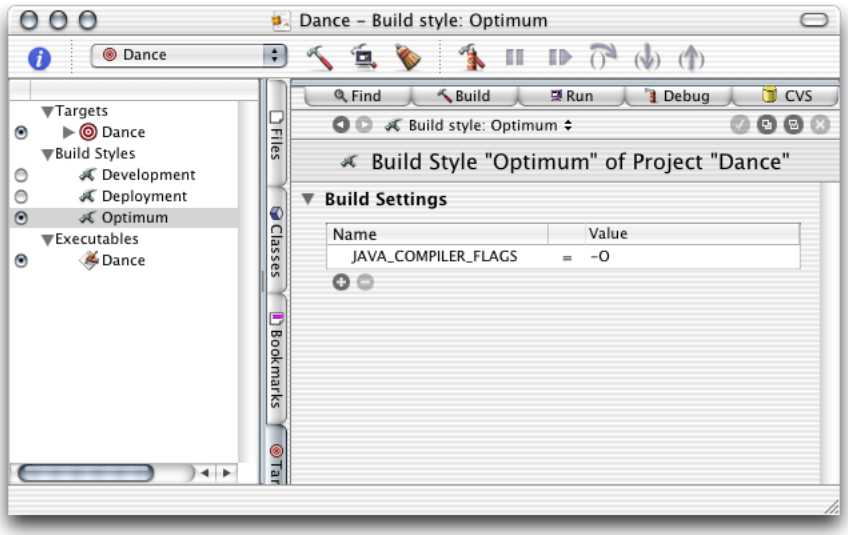
Build Styles

During development, you may want to include debugging information in Java class files, but would rather not include it in the final version of those files. For example, the `JAVA_COMPILER_DEBUGGING_SYMBOLS` build setting determines whether debugging symbols are added to class files. So, a project could have a target called `MyAppDebug` that sets that build setting to `YES` and a target called `MyApp` that sets it to `NO`. However, when you need to set another build setting that affects the building of the application, you would have to make the change in two targets instead of one. To solve this situation, Project Builder includes *build styles*.

Build styles contain build setting configurations that override target build settings. So, instead of having two targets to produce an application, one for debugging and another for your customers, a project would contain one target that builds both types of products and a couple of build styles, one called Development and another named Deployment. The Development build style would contain the `JAVA_COMPILER_DEBUGGING_SYMBOLS = YES` build configuration, while the Deployment build style would have `JAVA_COMPILER_DEBUGGING_SYMBOLS = NO`.

To add a build style to a project, choose Project > New Build Style and name it. Then add the build settings that the build style is to override. For example, the build style shown in Figure 2-12 tells `javac` to optimize code for execution time.

Figure 2-12 Build style definition



Build Phases

Build phases define concrete tasks that Project Builder performs to build a product. These are the build phases you use in Java application projects:

- **Sources** Compiles the selected Java source files using `javac` and puts the generated class files in `$TEMP_DIR/JavaClasses`. It uses the `JavaFileList` file in the target's build directory (the `TEMP_DIR` build setting).
- **Java Resource Files** Copies the selected Java resource files, the `Dancestrings.string` file for example, to `$TEMP_DIR/JavaClasses`.
- **Bundle Resources** Copies the selected bundle resource files, such as the icon file, to the resulting bundle's `Resources` directory.
- **Frameworks & Libraries** Links the class files generated in the Sources build phase with the selected frameworks and libraries, and archives the result in a JAR file (when the `JAVA_ARCHIVE_CLASSES` build setting is set to `YES`).

- **Shell Script Files** Executes a custom shell script. The value of every build setting is accessible in the script using the format `$BUILD_SETTING`, `$(BUILD_SETTING)`, or `${BUILD_SETTING}`. Therefore, you can use shell script phases to perform tasks that the other type of build phases do not support. Further, you can insert shell script phases between other build phases to confirm the value of a build setting.
- **Copy Files** Copies the files indicated in the build phase to a specified location. To select the files to copy, drag them from the Files list into the Files list.

Figure 2-13 is an example of inserting a shell script phase to confirm the value of a build setting. It shows a script that displays the value of the `JAVA_COMPILER_FLAGS` build setting. Figure 2-14 shows the script's output.

Figure 2-13 Build-setting display script

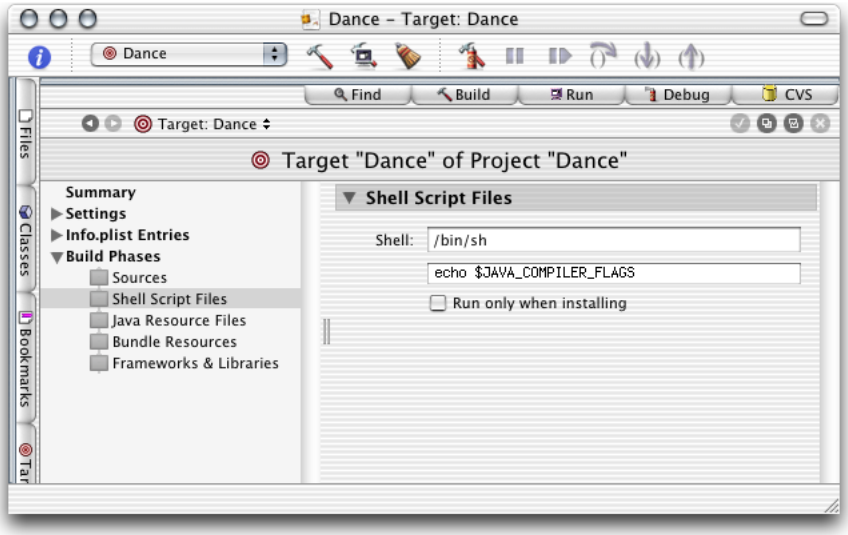
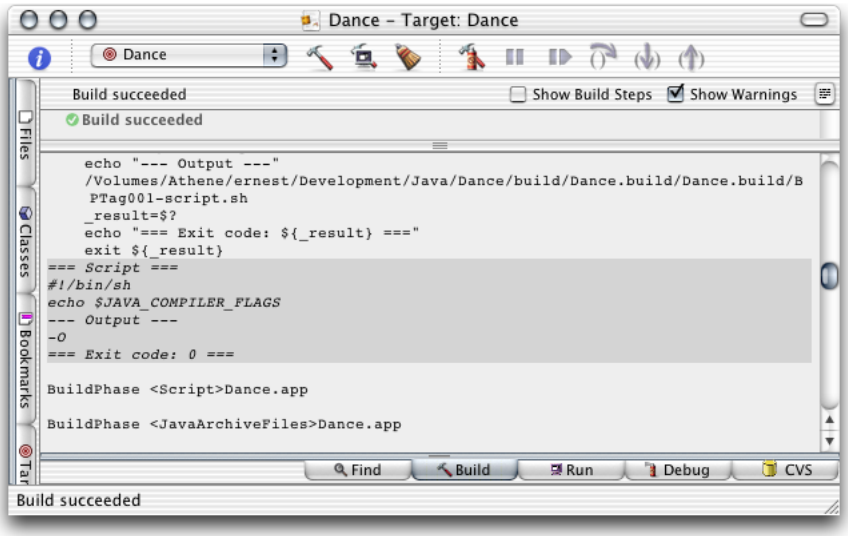


Figure 2-14 Output of a build-setting display script



Developing a Tool

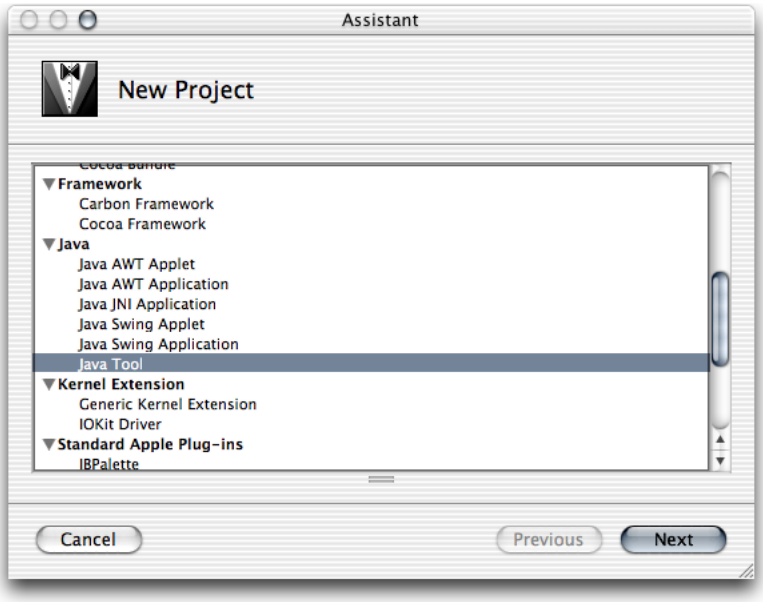
This chapter shows how to develop text-based Java applications or tools in Project Builder using the tool project template. It guides you through the creation of two projects, Hello and Clock. The former one is a "Hello, World" application, while the latter is a simple tool to display the current time, which is included in this document's companion files. See "Introduction to Project Builder for Java" (page 9) for details.

Creating the "Hello, World" Tool

The Java Tool project template provides the prototypical "Hello, World" application. Follow these steps to create your first Java application using Project Builder.

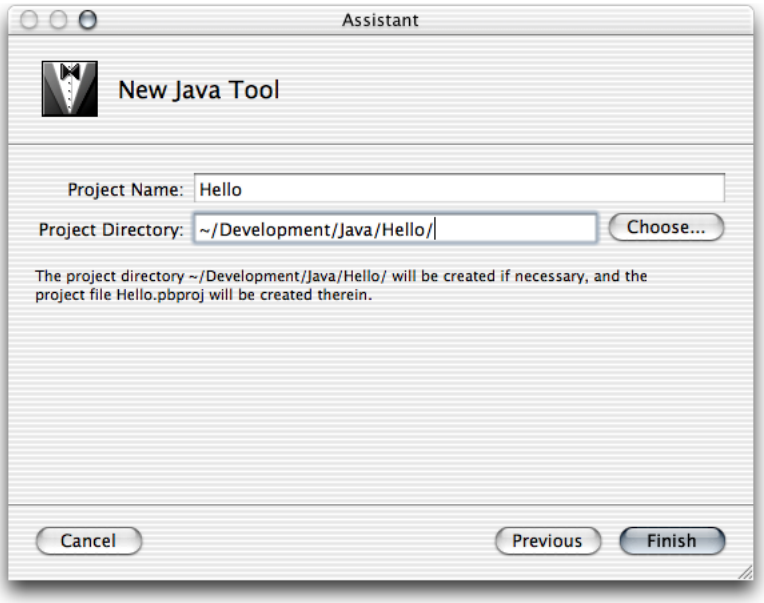
1. Launch Project Builder. It's located in `/Developer/Applications`.
2. Create a Java tool project.

   Choose File > New Project, and select Java Tool under Java in the project-template list of the New Project pane.



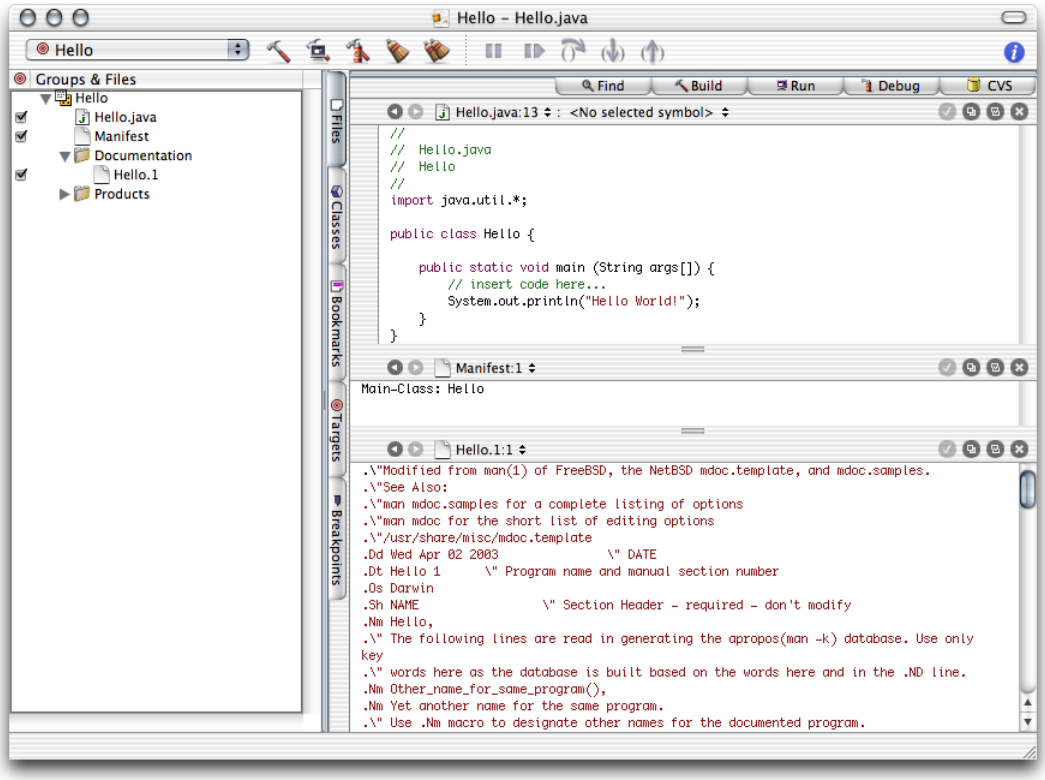
3. Name the project and choose a location for it.

In the New Java Tool pane of the Assistant, enter `Hello` in the Project Name text input field, click Choose, and choose a location for it.



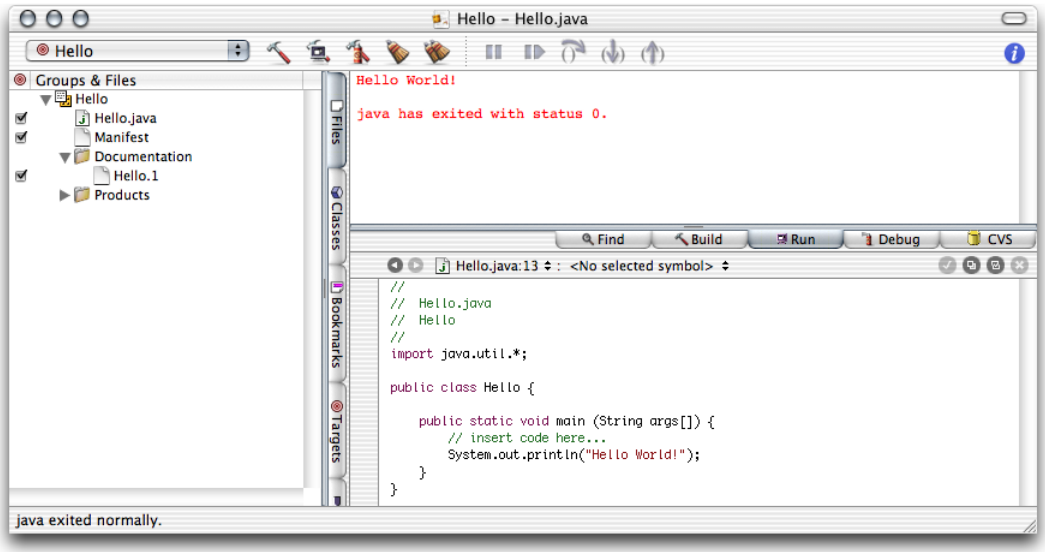
When done, you should see the Project Builder window. Figure 3-1 shows the window with three editor panes, one for each file in the project, the Java source file, the manifest file, and the man page documentation file. The product, `Hello.jar`, is shown in red because it hasn't been built.

Figure 3-1 The Hello project in Project Builder



Build and run the application by choosing Build > Build and Run. Figure 3-2 shows the Run pane of the Project Builder window. The Run pane displays the console output of the application.

Figure 3-2 Project Builder's Run pane showing Hello's console output



Creating the Clock Tool

This section shows how to create the Clock tool. Clock is a text-based application that tells time. It takes an optional command-line argument, the name of the user. You can find the finished product among this document's companion files in `companion/projects/Clock` (see "Introduction to Project Builder for Java" (page 9) for details on companion files).

Follow these instructions to create the Clock tool.

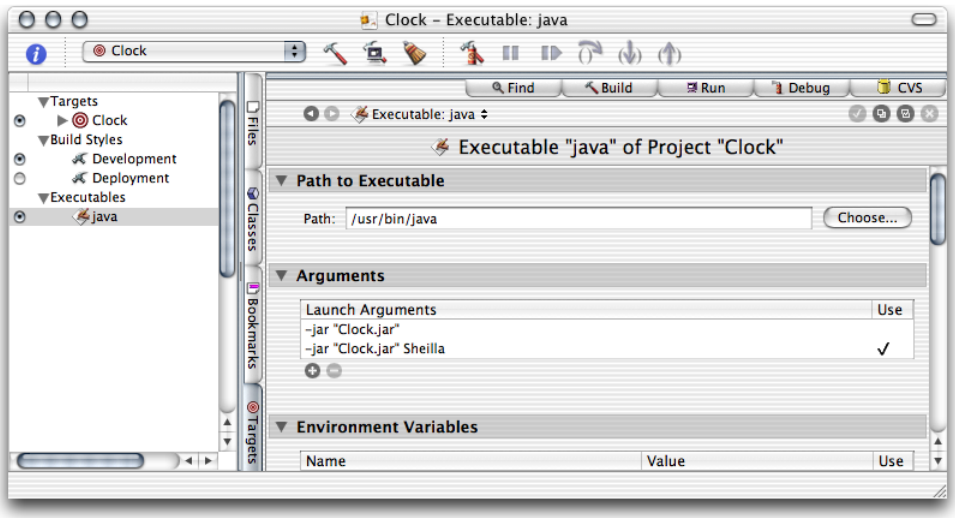
1. Create a Java tool project and name it `Clock`.
2. Edit the `main` method of the Clock class so that it looks like this:

```
public static void main (String args[]) {
    Date date = new Date();

    if (args.length > 0) {
        String user_name = args[0];
        System.out.println("Hello, " + user_name + ". It's " + date);
    }
    else {
        System.out.println("It's " + date);
    }
}
```

3. Add an argument to the application's launch arguments to test it within Project Builder.
    a. Click the Targets tab to display the Targets list.
    b. Click java under Executables in the Targets list.
    c. Click the plus sign (+) in the Arguments pane of the target editor.
    d. Enter `-jar "Clock.jar" Sheilla` in the newly added row of the Launch Arguments list.

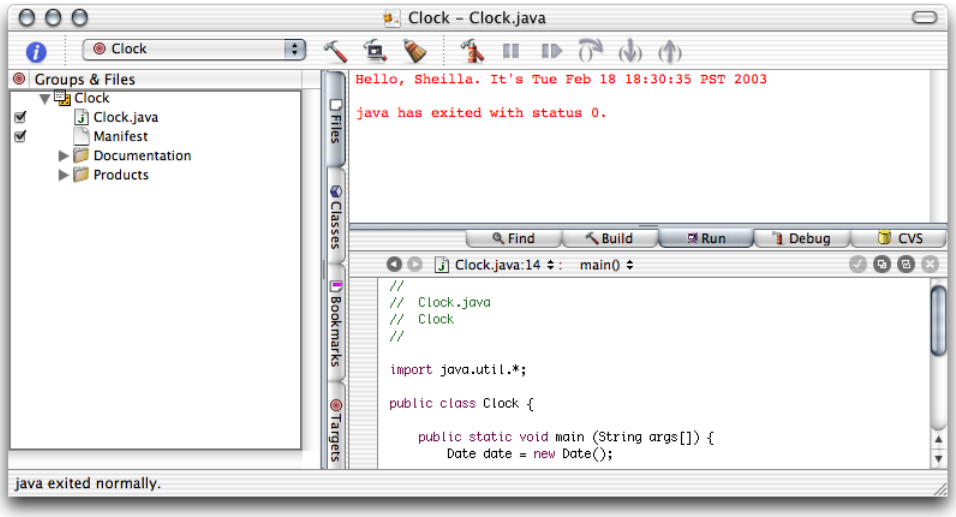
e. Deselect the Use option in the first row by clicking the checkmark in the Use column. The Arguments pane should now look like Figure 3-3.

Figure 3-3 Arguments pane of the executable editor in Project Builder



Build and run the application. You should see its output in Project Builder's Run pane, as shown in Figure 3-4.

Figure 3-4 Output of Clock tool displayed in Project Builder



Installing the Clock Tool

This section shows how to install the Clock tool on a computer. Follow these steps to install Clock on your computer:

1. Determine the location of the installed product by adding the `INSTALL_DIR` build setting to the project and configuring the setting appropriately.

   a. Click the Targets tab to display the Targets list.

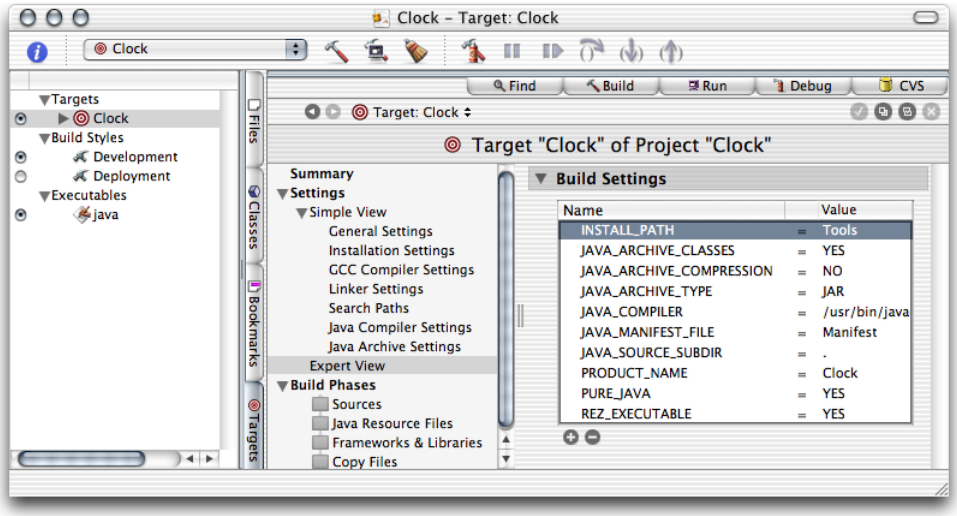
   b. Click the Clock target.

   c. Click Expert View under Settings in the target editor.

   d. Click the plus sign (+) in the Build Settings pane.

   e. In the newly added row, enter `INSTALL_PATH` in the Name column and `Tools` in the Value column. The Expert View pane should look like Figure 3-5.

Figure 3-5 Expert View pane of the target editor in Project Builder



2. Run `pbxbuild` to install the application:

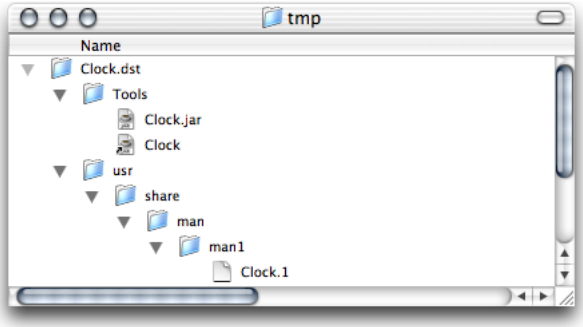
   a. Launch Terminal. It's located in `/Applications/Utilities`.

   b. Execute the following commands:

```
% cd <path_to_Clock_project>
% pbxbuild install -buildstyle Deployment
```

Now, your `/tmp` directory contains the Clock distribution directory (`Clock.dst`), as shown in Figure 3-6.

Figure 3-6 Clock distribution directory in `/tmp`

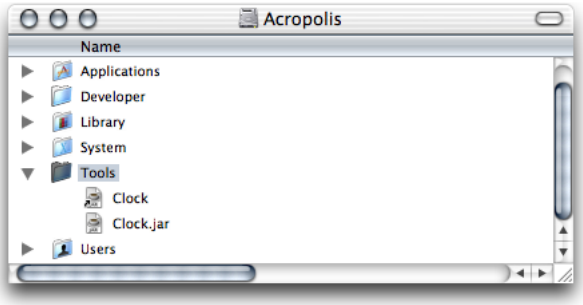


If you want `pbxbuild` to install in the final destination of a product instead of in `/tmp`, use the following commands:

```
sudo pbxbuild clean
sudo pbxbuild install -buildstyle Deployment DSTROOT=/
```

This creates `/Tools` in your root volume if it doesn't already exist and places the application's JAR file there, as shown in Figure 3-7.

Figure 3-7 Clock target directory



To run the application, double-click the JAR file. To view the application's output when you launch it from the Finder, launch Console, located in `/Applications/Utilities`. Figure 3-8 shows Console displaying the output of a Clock session.

Figure 3-8 Output of Clock viewed through Console

Developing a Swing Application

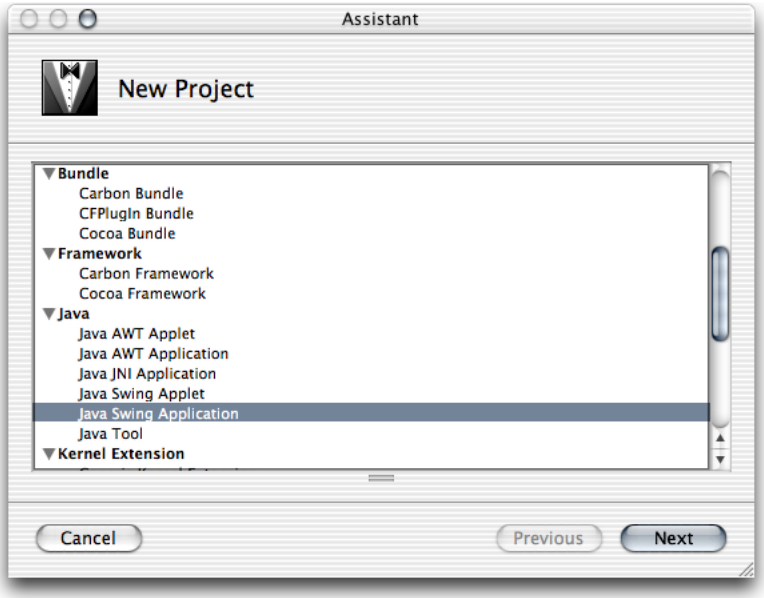
This chapter covers the steps needed to develop Swing applications. First, this chapter guides you through the creation of a simple application, completely based on Project Builder's Swing project template. Second, to show how to port an existing Swing application to Mac OS X, it shows how to create a Swing project based on Sun's File Chooser Demo application and deploy it as a Mac OS X application; the finished project is in `companion/projects/FileChooser`. (See "Introduction to Project Builder for Java" (page 9) for details on this document's companion files.) Finally, this chapter explains how to change the icon the Finder displays for the application from the generic Java application icon. .

Creating the "Hello, Swing" Application

The Swing application template provides another version of the "Hello, World" application. Follow these steps to create a project that demonstrates how a Swing application looks in Mac OS X.

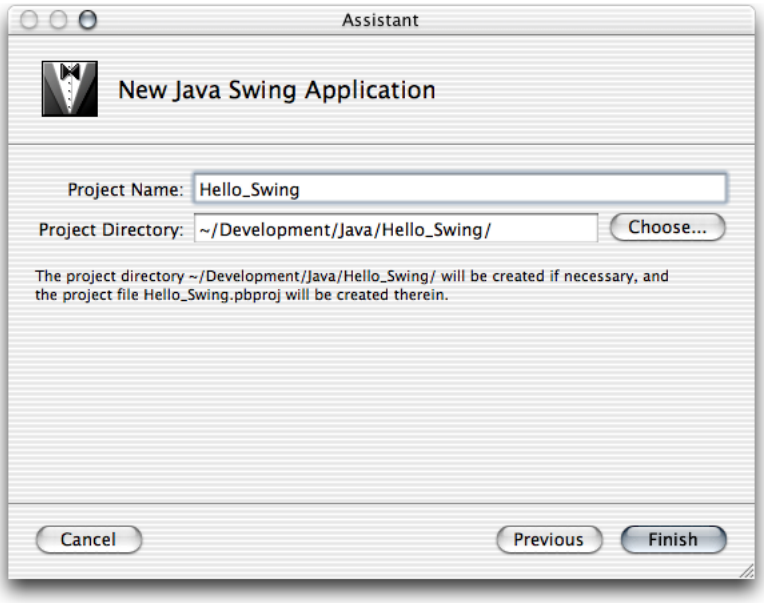
1. Launch Project Builder. It's located in `/Developer/Applications`.
2. Create a Java Swing application project.

   Choose File > New Project, and select Java Swing Application under Java in the project-template list of the New Project pane.



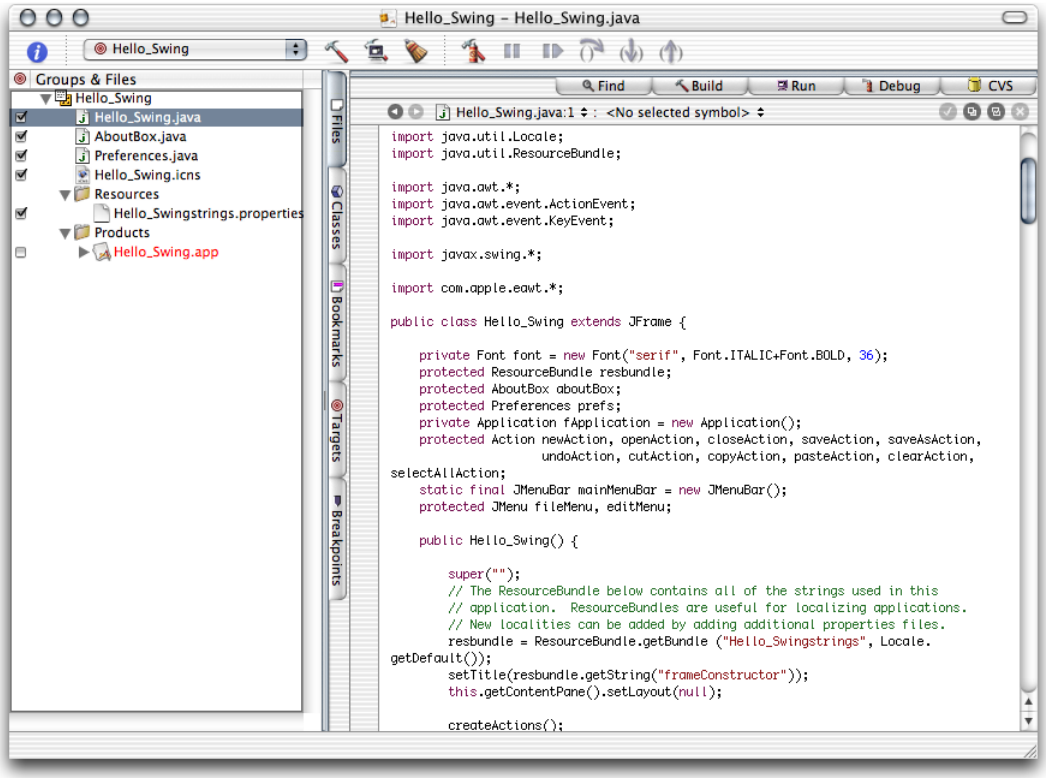
3. Name the project and choose a location for it.

In the New Java Swing Application pane of the Assistant enter `Hello_Swing` in the Project Name text input field, click Choose, and choose a location for the project folder.



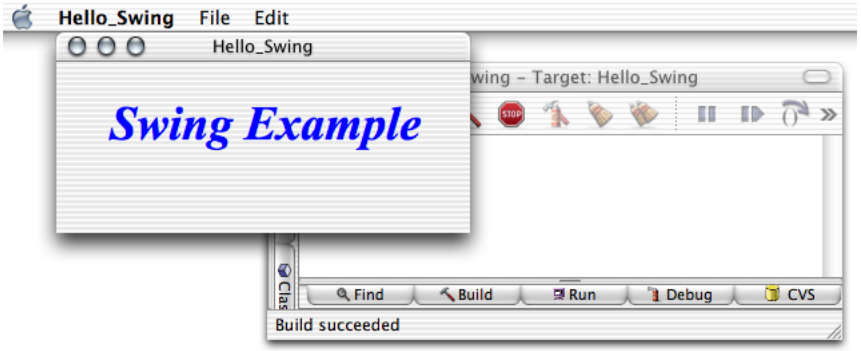
When done, you should see the Project Builder window, shown in Figure 4-1. The product, `Hello_Swing.app`, appears in red because it hasn't been built.

Figure 4-1 The Hello_Swing project in Project Builder's window



Build and run the application by choosing Build > Build and Run. Figure 4-2 shows the running Hello_Swing application.

Figure 4-2 Hello_Swing application in action



Creating the File Chooser Demo

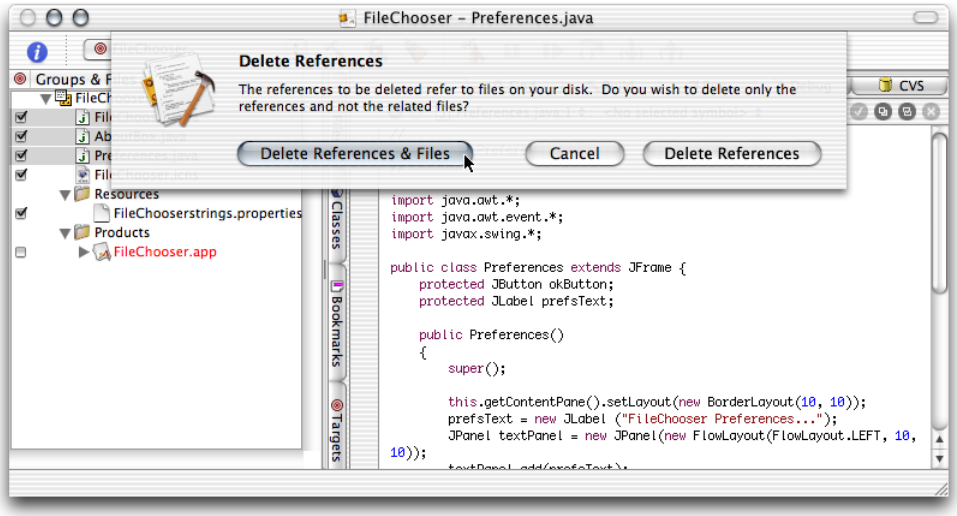
This section explains how to use existing Java source files to create a Swing-based Mac OS X application.

You can download source code that demonstrates the use of the JFileChooser class (`javax.swing`) at http://java.sun.com/docs/books/tutorial/uiswing/components/filechooser.html. You can also use the files included with this document in `companion/source/FileChooser` (see "Introduction to Project Builder for Java" (page 9) for details on companion files).

Perform these steps to create a file-chooser demonstration project.

1. Create a Java Swing application project named `FileChooser`.
2. Remove the standard source files from the project:
   a. Select the `FileChooser.java`, `AboutBox.java`, and `Preferences.java` files in the Files list.
   b. Choose File > Delete or press the Delete key.
   c. Click Delete References & Files in the Delete References dialog, shown in Figure 4-3.

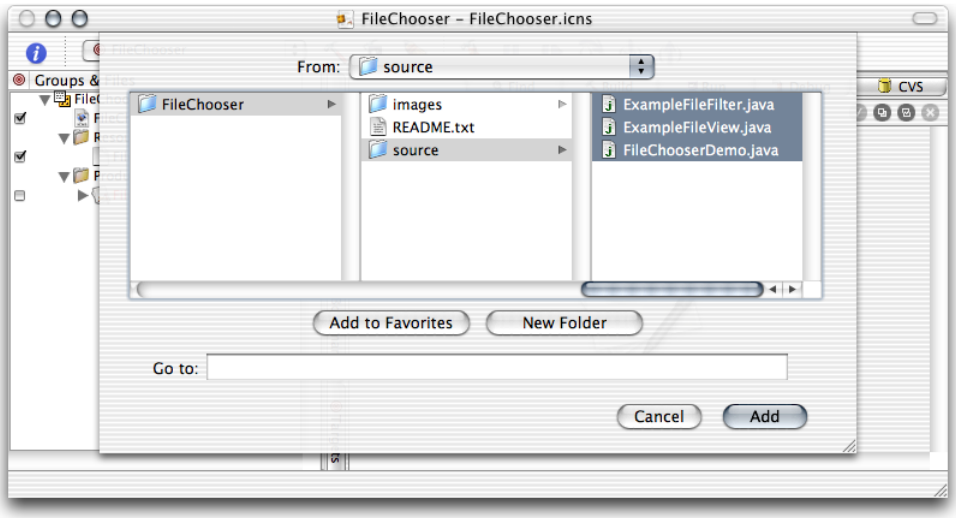
Figure 4-3 Delete References dialog of Project Builder



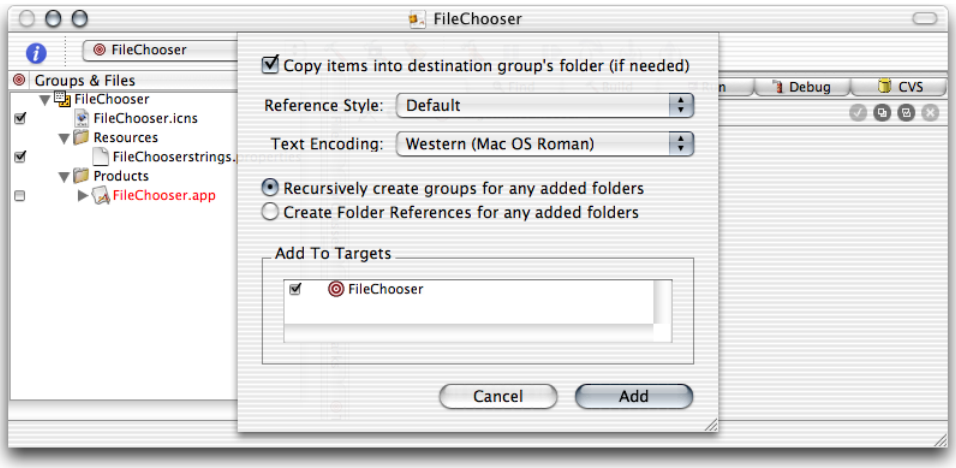
3. Add the source files and image files required for the project:
   a. Choose Project > Add Files.

b. Navigate to where the source files reside, select them, and click Add. Figure 4-4 exemplifies the addition of the file-chooser demonstration files in `companion/source/FileChooser`.

Figure 4-4 Adding source files to a project in Project Builder



c. In the dialog that appears, select "Copy items into destination group's folder" and make sure the FileChooser target is selected in the Add To Targets list.

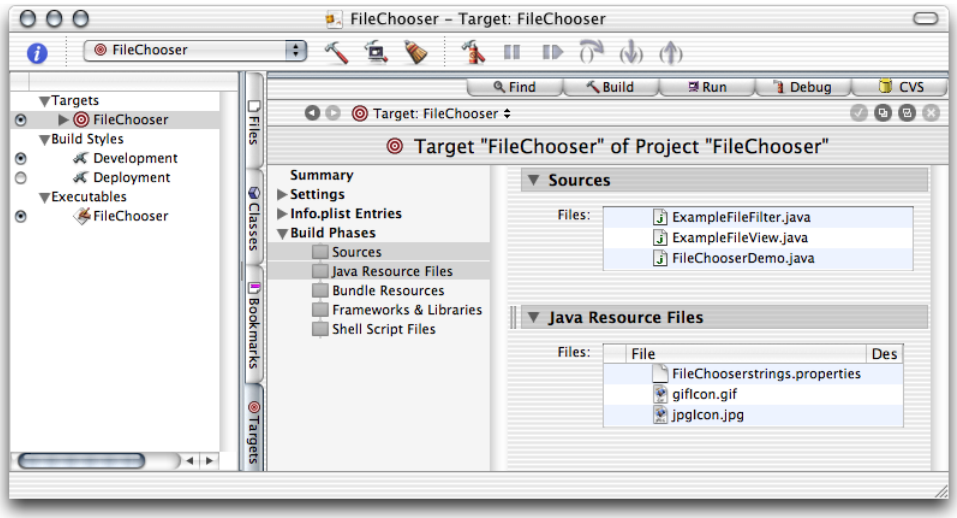


d. Repeat the previous step for the image files.

4. Examine the FileChooser target to verify that the newly added files are assigned to the correct build phases:

a. Click the Targets tab and select the FileChooser target in the Targets list.

b. Select the Sources build phase and the Java Resource Files build phase in the target editor. Make sure the source files and image files you added to the project appear in the Sources pane and the Java Resource Files pane, respectively.



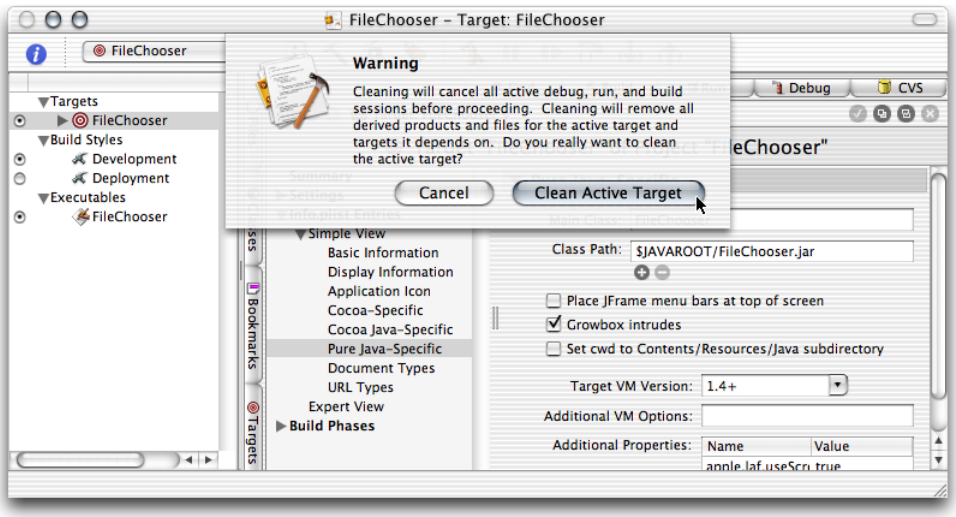
5. Change the name of the main class in the information property list:

a. Select Pure Java–Specific under Simple view under Info.plist Entries in the target editor.

b. Enter `FileChooserDemo` in the Main Class text field.

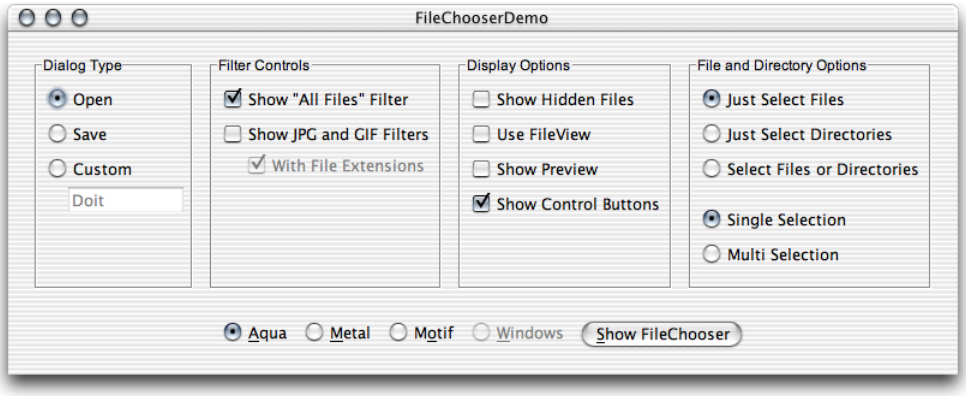
6. Clean the FileChooser target by choosing Build > Clean and click Clean Active Target in the dialog that appears.

Cleaning the target erases any temporary files stored in the target's `build` directory, which may have been left there in previous builds. (If you didn't build the application, you may skip this step.)



Build and run the application. You should see the window shown in Figure 4-5.

Figure 4-5 FileChooser in action

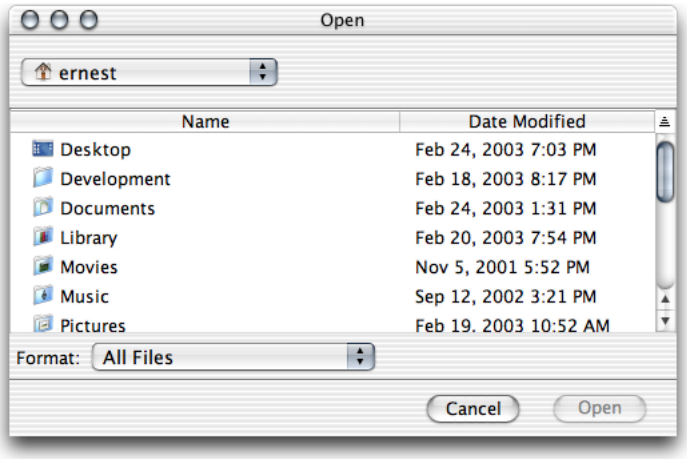


If instead of a running application you get an error message like the following in Project Builder's Run pane, make sure that the name of the application's main class matches the contents of the Main Class entry of the Pure Java–Specific pane of the Info.plist Entries pane in the target editor.

```
[LaunchRunner Error] The main class "FileChooser" could not be found.
[JavaAppLauncher Error] CallStaticVoidMethod() threw an exception
java.lang.NullPointerException
    at apple.launcher.LaunchRunner.run(LaunchRunner.java:85)
    at apple.launcher.LaunchRunner.callMain(LaunchRunner.java:50)
    at
apple.launcher.JavaApplicationLauncher.launch(JavaApplicationLauncher.java:52)
Exception in thread "main"
FileChooser has exited with status 0.
```

When you click the Show FileChooser button of the FileChooserDemo window, you should see a window like the one in Figure 4-6. Of course, the actual look of the window depends on the selections you make in the FileChooserDemo window.

Figure 4-6 Open dialog displayed by FileChooserDemo



Changing an Application's Icon

The Resources folder of an application package holds several types of files, including icon files. The Finder consults the `CFBundleIconFile` information property list entry to determine which of these files to use as the application's icon.

Follow these steps to change the icon of the FileChooser application developed in "Creating the File Chooser Demo" (page 43) from the default icon.

1. Identify the icon file for the new icon.

   You can find an icon file in `/Developer/Applications/Pixie.app/Contents/Resources/Big.icns`.

   In Terminal, execute the following command:

   ```
   cp /Developer/Applications/Pixie.app/Contents/Resources/Big.icns
   <FileChooser_project_directory>
   ```

2. Remove the `FileChooser.icns` file from the FileChooser project:

   a. Select `FileChooser.icns` in the Files list in the Project Builder main window.

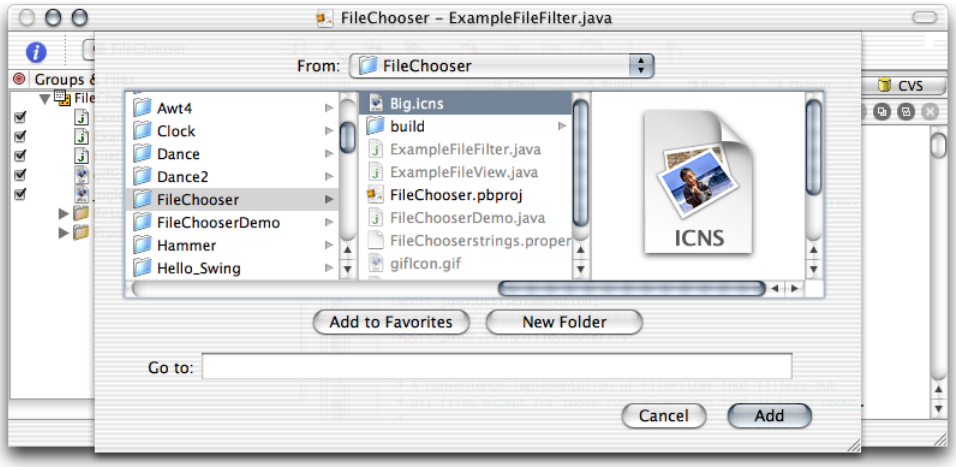
   b. Choose Edit > Delete or press the Delete key.

   c. Click Delete References & Files.

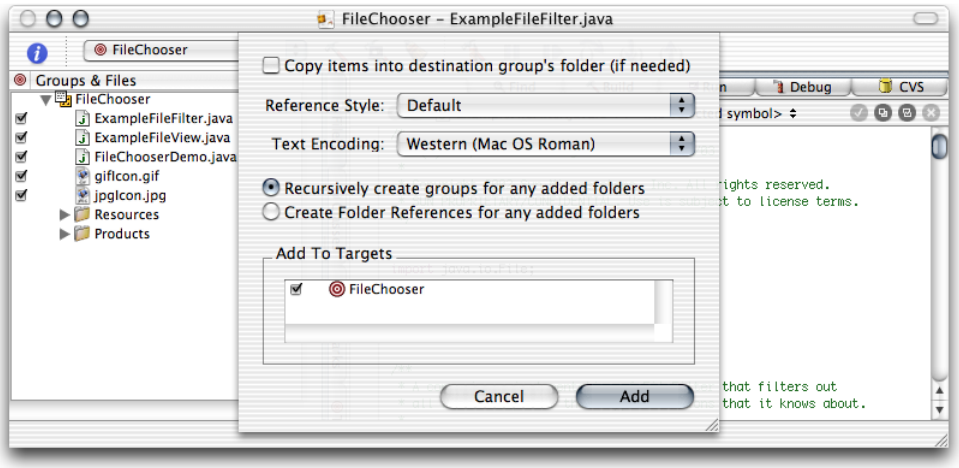
3. Add the icon file for the desired icon to the project:

   a. Choose Project > Add Files.

   b. Select `Big.icns` in the file list and click Add.



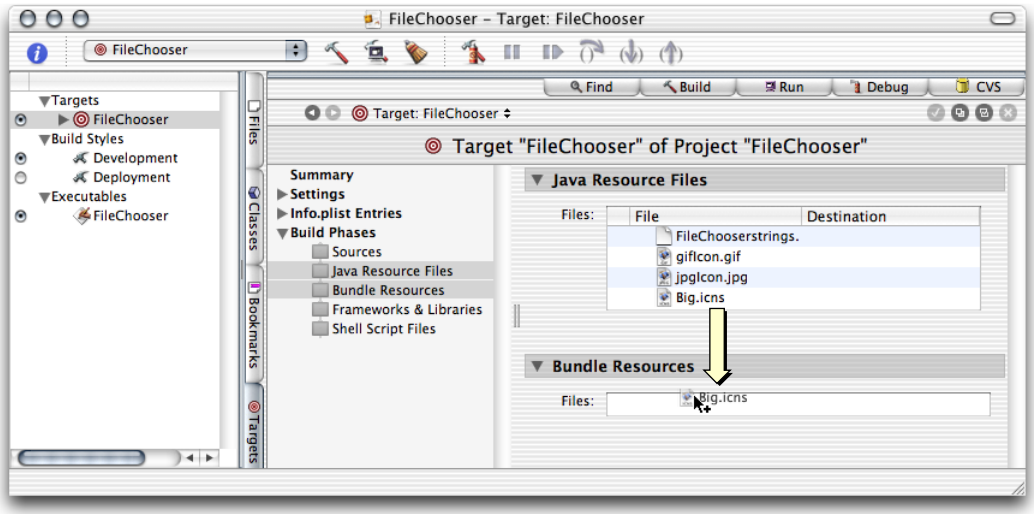
c. In the dialog that appears, make sure "Copy items into destination group's folder" is not selected and click Add.



4. Make sure that the new icon file is assigned to the Bundle Resources build phase and not the Java Resource Files build phase.

a. Select the Java Resource Files build phase and the Bundle Resources build phase in the target editor of the FileChooser target.

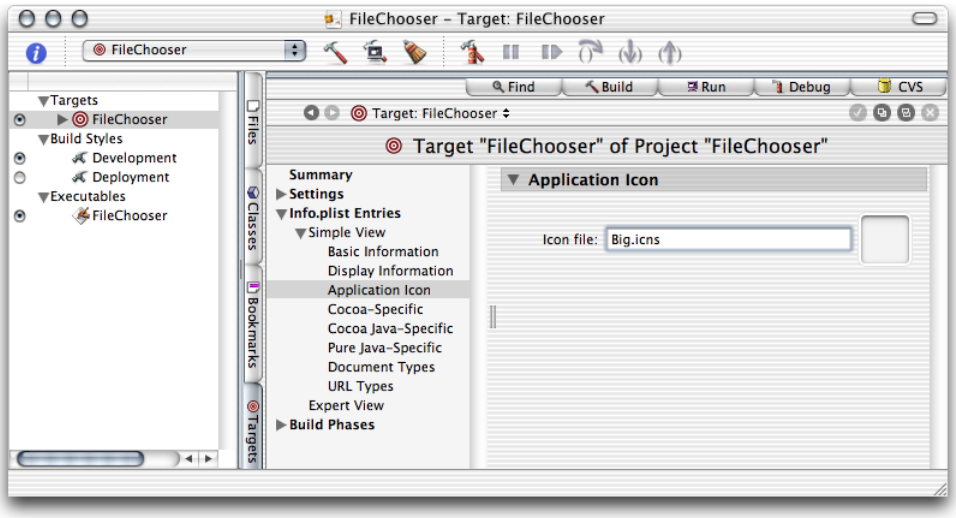
b. Drag `Big.icns` from the Files list of the Java Resource Files pane to the Files list of the Bundle Resources pane.



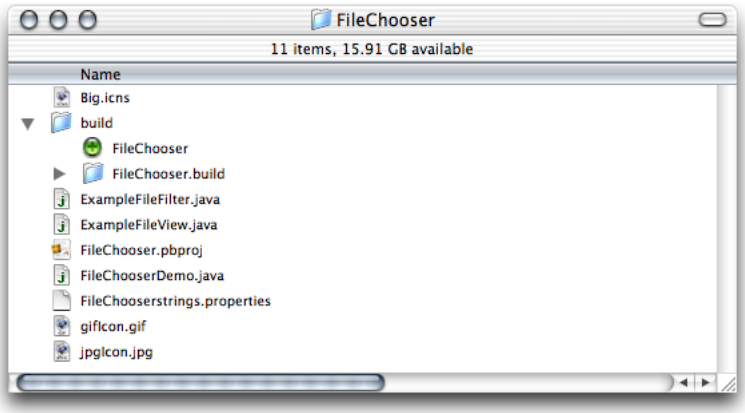
5. Set the name of the icon file of the application.

a. Select Application Icon under Simple View under Info.plist Entries in the target editor.

b. Enter `Big.icns` in the "Icon file" text field of the Application Icon pane.



Clean the project, and build and run the application. The icon for `FileChooser.app` in the build folder of the project should have the icon used by Pixie.



Developing a JNI Application

When you need to leverage existing C or Objective-C code in a Java application or need to improve the performance of an application by executing critical parts natively instead of on the Java virtual machine, you use the Java Native Interface (JNI). The JNI provides a way for Java code to communicate with C-based libraries.

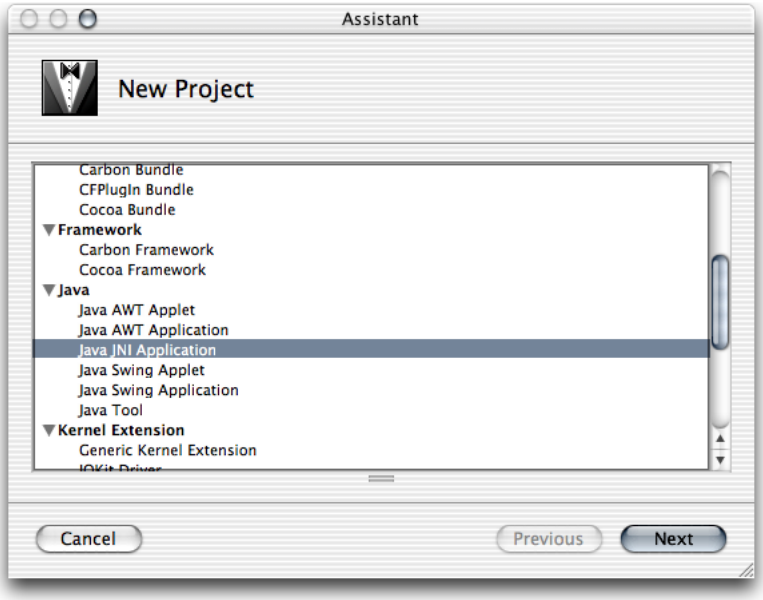
Project Builder provides a template that facilitates the development of JNI-based applications. For an explanation of the elements of that template, including its targets, see "The JNI Application Template" (page 14).

Creating the "Hello, JNI" Application

The JNI application template provides yet another version of a "Hello, World" application. This one, however, joins the flexibility of Java with the high performance of C code to print the famous greeting on the console. Follow these steps to create a JNI-based application.

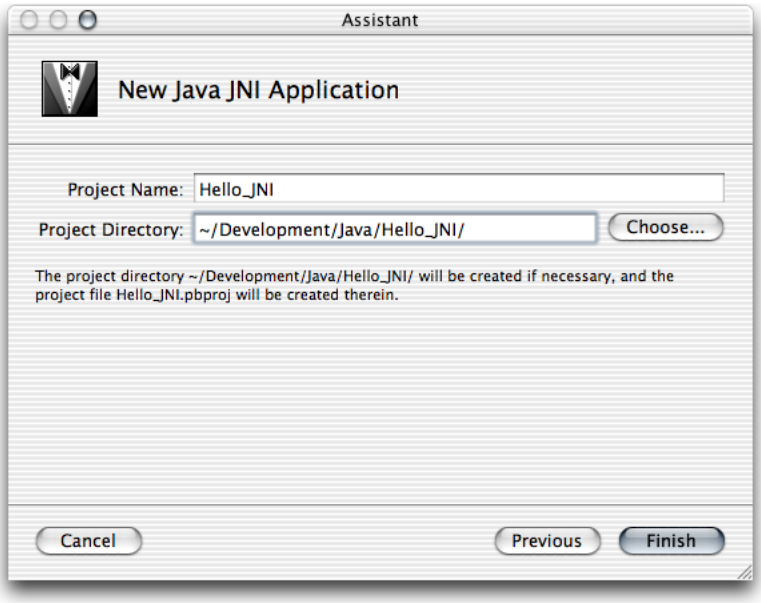
1. Launch Project Builder. It's located in `/Developer/Applications`.
2. Create a Java JNI application project.

   Choose File > New Project, and select Java JNI Application under Java in the template list.



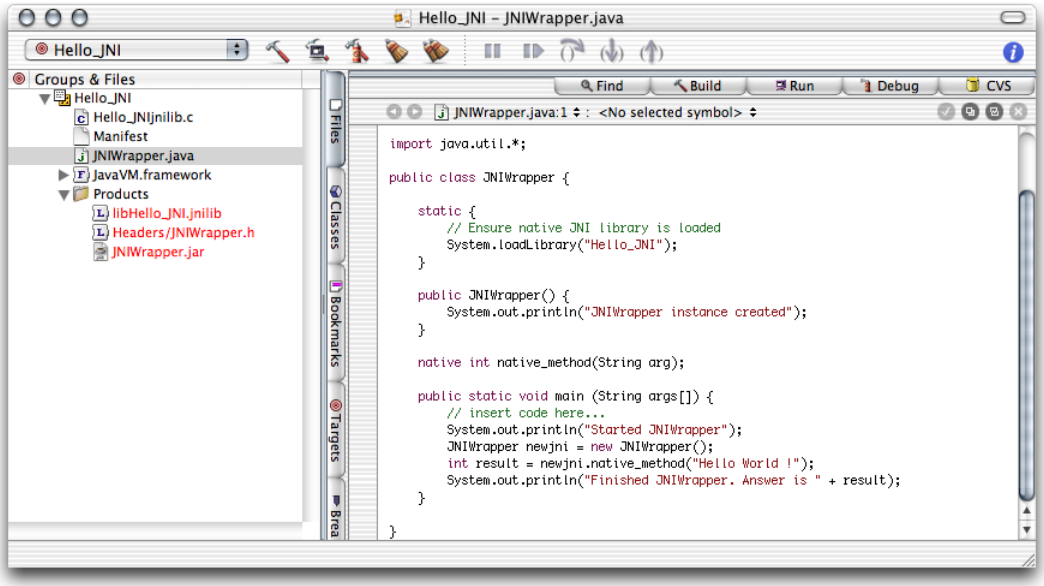
3. Name the project and choose a location for it.

In the New Java JNI Application pane of the Assistant, enter `Hello_JNI` in the Project Name text input field, click Choose, and choose a location for the project folder.



When done, you should see the Project Builder window, shown in Figure 5-1. The files in red are the project's products, which haven't been built.

Figure 5-1 The Leverage project in the Project Builder window



Project Builder generated the source files for the native side and the Java side of the application. They're shown in Listing 5-1 and Listing 5-2.

Listing 5-1 `Leveragejnilib.c` source file in the Leverage project

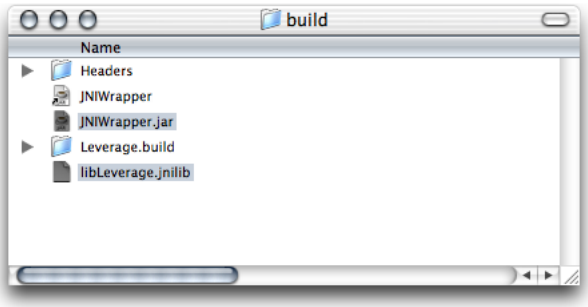
```
#include "JNIWrapper.h"

int shared_function(const char *arg) {
    printf("shared_function called with %s\n", arg);
    return 42;
}

JNIEXPORT jint JNICALL Java_JNIWrapper_native_1method(JNIEnv *env, jobject this,
 jstring arg) {
    /* Convert to UTF8 */
    const char *argutf  = (*env)->GetStringUTFChars(env, arg, JNI_FALSE);

    /* Call into external dylib function */
    jint rc = shared_function(argutf);

    /* Release created UTF8 string. */
    (*env)->ReleaseStringUTFChars(env, arg, argutf);

    return rc;
}
```

Listing 5-2 `JNIWrapper.java` source file in the Leverage project

```
import java.util.*;

public class JNIWrapper {

    static {
        // Ensure native JNI library is loaded.
        System.loadLibrary("Leverage");
    }

    public JNIWrapper() {
        System.out.println("JNIWrapper instance created");
    }

    native int native_method(String arg);

    public static void main (String args[]) {
        System.out.println("Started JNIWrapper");
        JNIWrapper newjni = new JNIWrapper();
        int result = newjni.native_method("Hello World !");
        System.out.println("Finished JNIWrapper. Answer is " + result);
    }

}
```

Now, make sure the Leverage target is selected, and build and run the application. Several files appear in the project's build folder. Because this is a JNI application, in addition to the JAR file containing the Java application, you see a JNI library file, which contains the object file for the native function specified in `Leveragejnilib.c` (Figure 5-2). The Header folder contains the `JNIWrapper.h` file, which is generated by `javah` from the `JNIWrapper.class` file.

Figure 5-2 The build folder of the Leverage project after building the application



JNI-Based Examples

The developer tools package includes several examples of JNI-based applications, including a Cocoa/Java application located in `/Developer/Examples/Java/AppleDemos/CocoaComponent`. Open those projects and examine them to get a glimpse of the power and flexibility that Java and JNI provide.

Debugging Applications

Project Builder provides facilities for debugging Java applications. They allow you to stop the execution of an application at a specific line of code, execute a line of code within a method, step into a method call, step out of a method, or view the contents of variables in any method in the call stack.

This chapter shows how to use Project Builder's debugging facilities to analyze the execution of a small application. It's based on the Debug project included in the companion folder (`companion/projects/Debug`); see "Introduction to Project Builder for Java" (page 9) for details on companion files.

Adding Breakpoints

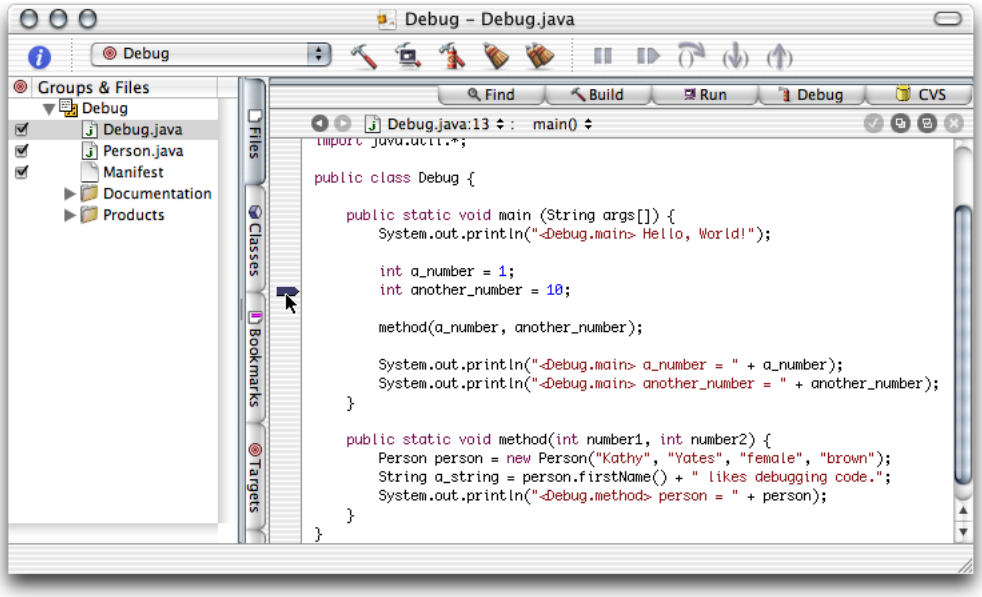
To pause the execution of an application, place a breakpoint marker in the line of code you want execution to stop. Listing 6-1 shows the definition of the Debug class in the Debug project.

Listing 6-1 `Debug.java` file of Debug project

```
import java.util.*;

public class Debug {

    public static void main (String args[]) {
        System.out.println("<Debug.main> Hello, World!");

        int a_number = 1;
        int another_number = 10;                                                    // 1

        method(a_number, another_number);

        System.out.println("<Debug.main> a_number = " + a_number);
        System.out.println("<Debug.main> another_number = " + another_number);
}

    public static void method(int number1, int number2) {
        Person person = new Person("Kathy", "Yates", "female", "brown");
        String a_string = person.firstName() + " likes debugging code.";
        System.out.println("<Debug.method> person = " + person);
    }
}
```

To add a breakpoint to the line numbered 1, click the line's left margin in the editor. You can also set the insertion point in the line and choose Debug > Add Breakpoint at Current Line. Figure 6-1 shows the result.

Figure 6-1 Breakpoint in `Debug.java` file of Debug project



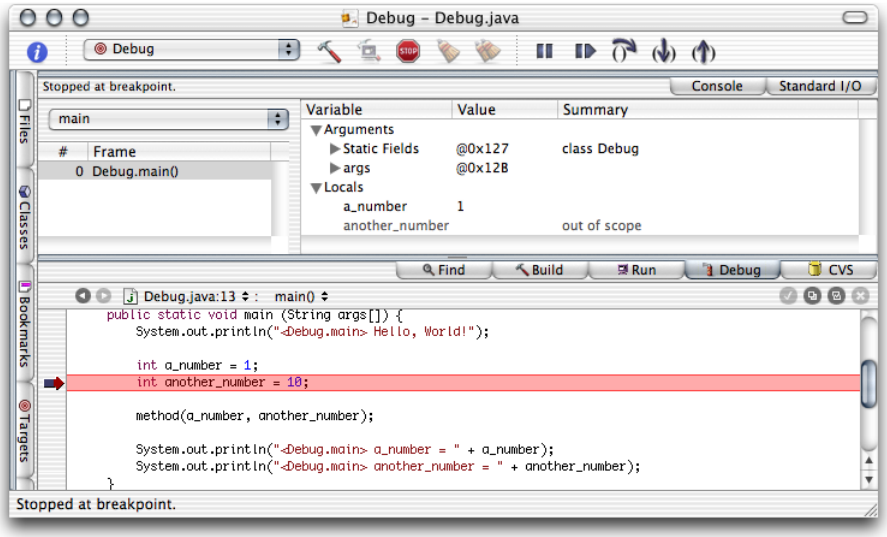
To remove a breakpoint, click the breakpoint marker, drag the marker out of the margin, or choose Debug > Remove Breakpoint at Current Line.

To disable a breakpoint, Command-click the breakpoint marker or choose Debug > Disable Breakpoint at Current Line.

Stepping Through Lines of Code

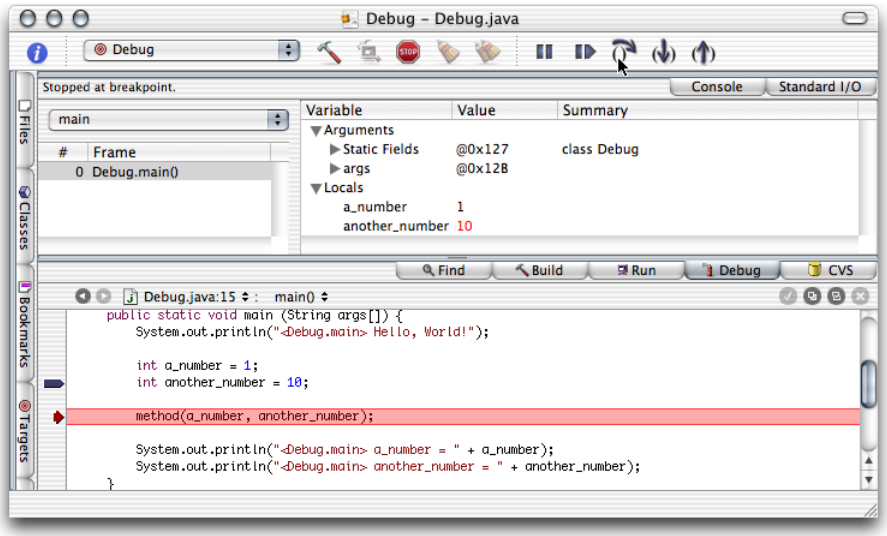
To build and debug the Debug project, choose Build > Build and then choose Debug > Debug Executable, or click the Build and Debug toolbar button. Figure 6-2 shows the result, in which the highlighted line is about to be executed.

Figure 6-2 Debugging an application—stopping



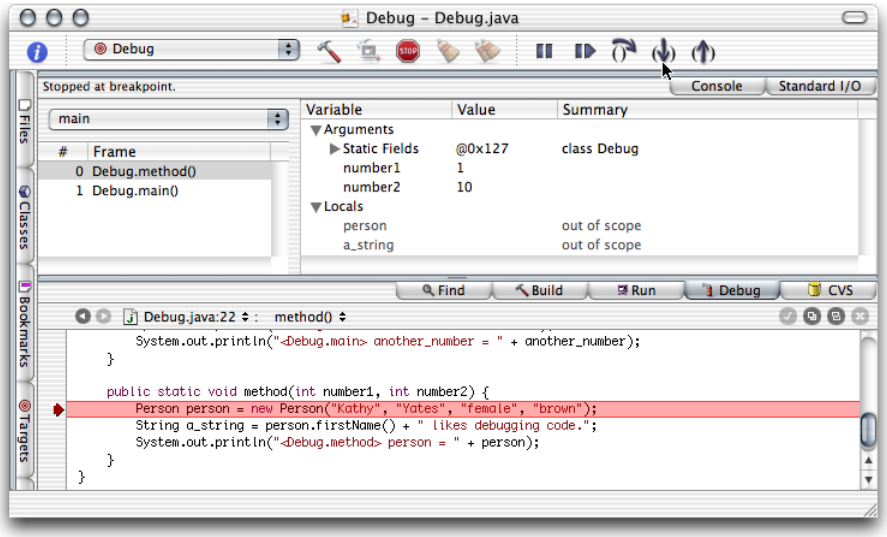
To step to the next line of code choose Debug > Step Over or click the Step Over toolbar button, as shown in Figure 6-3. Because the line executed is not a method call, clicking the Step Into toolbar button would give the same result.

Figure 6-3 Debugging an application—stepping over



To step into a method choose Debug > Step Into or click the Step Into toolbar button, as shown in Figure 6-4.

Figure 6-4 Debugging an application—stepping into a method

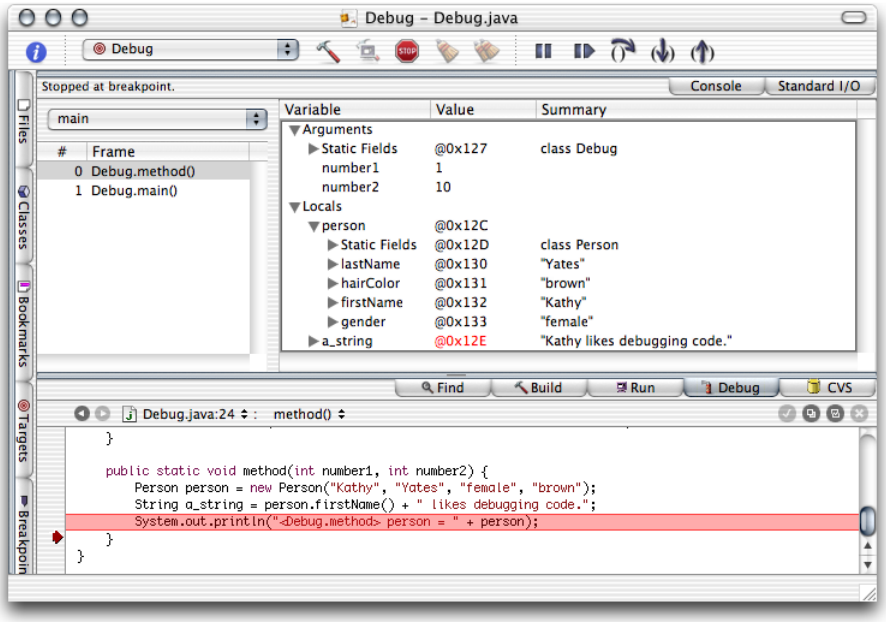


To step out of a method, (that is, to execute the rest of the lines in the current method and return to calling method), choose Debug > Step Out or click the Step Out toolbar button.

Viewing the Debug Information

The pop-up menu to the right of the Files tab (with main chosen) lists threads of execution. The list below it shows the call stack for the chosen thread. The pane to the right of the call stack pane, the variable pane, shows the names of the parameters and variables declared for the currently executing method in the chosen thread. It may also show the arguments used in the method invocation and the values of the local variables. Figure 6-5 shows the call stack of the main thread and parameters and local variables of a method.

Figure 6-5 Debugging an application—viewing variable information



Accessing the Contents of Objects

While you debug code, you may need to see the values of an object's instance variables. Most programmers sprinkle `System.out.println` invocations throughout their code to accomplish this essential task. In Project Builder you can execute an object's `toString` method to get the same effect.

Listing 6-2 shows a partial listing of the Person class. It contains an implementation of the `toString` method.

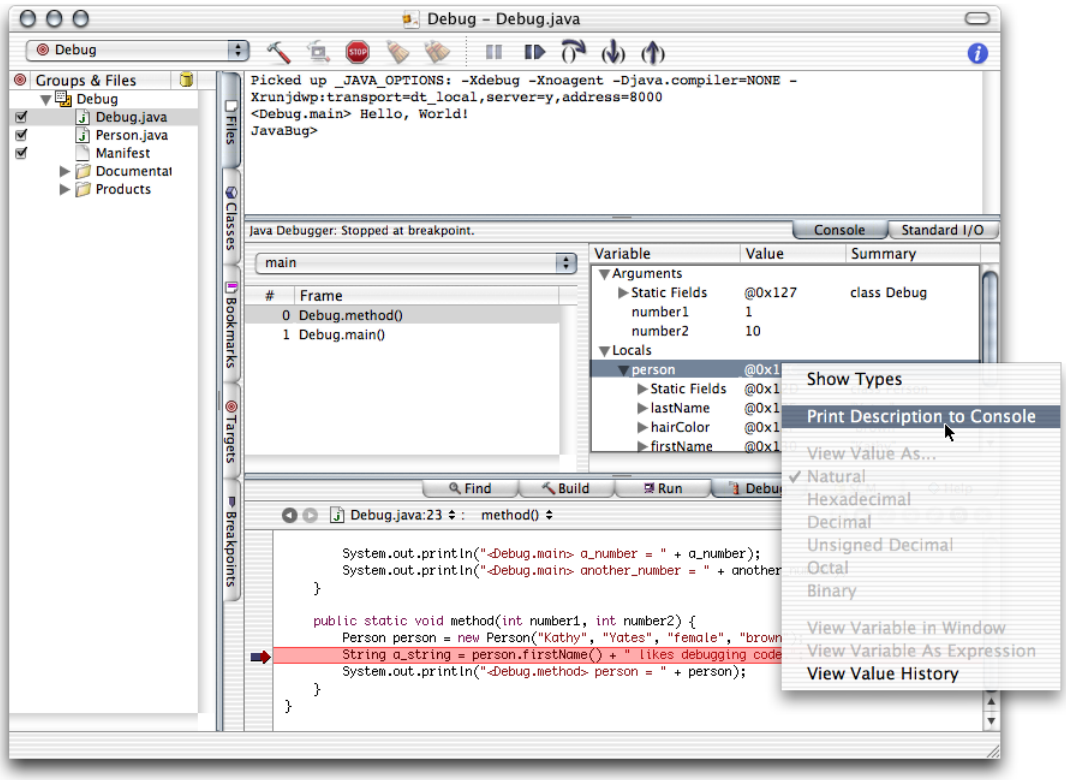
Listing 6-2 `Person.java` file

```
public class Person {
    private String firstName;
    private String lastName;
    private String gender;
    private String hairColor;

    public Person(String firstName, String lastName, String gender, String
hairColor) {
        setFirstName(firstName);
        setLastName(lastName);
        setGender(gender);
        setHairColor(hairColor);
    }

    ...

    public String toString() {
        return "{FirstName: " + firstName() + "},{LastName: " + lastName() +
"},{Gender: " + gender() + "},{HairColor: " + hairColor() + "}";
    }
```

```
}
```

Figure 6-6 depicts a debugging session in which the user chooses the Print Description to Console command through the contextual menu of person in the Variable list of the Debug pane.

Figure 6-6 Debugging an application—viewing an object's contents



Listing 6-3 shows the output generated.

Listing 6-3 Console output after executing Print Description to Console command on a Person object

```
Picked up _JAVA_OPTIONS: -Xdebug -Xnoagent -Djava.compiler=NONE
-Xrunjdwp:transport=dt_local,server=y,address=8000
<Debug.main> Hello, World!

Printing description of person:
"{FirstName: Kathy},{LastName: Yates},{Gender: female},{HairColor: brown}"
JavaBug>
```

Build Settings Reference

This appendix lists some of the build settings you are likely to use in Java-based projects. See the Project Builder release notes for a complete treatment of Project Builder's build settings.

Project Settings Reference

Table A-1 describes build settings that apply to a project as a whole; that is, they apply to all targets in a project.

Table A-1 Project build settings

Build setting	Description
PROJECT_NAME	Name of the project. This setting is read-only.
SYMROOT	Base location for built products. Configured initially as <project_directory>/build.
BUILD_DIR	Base location for the temporary files generated by a project's targets. Default: <project_directory>/build. This setting is read-only.
TARET_BUILD_DIR	The location for products. Set initially to $BUILD_DIR in development builds and $INSTALL_DIR in deployment builds when the product is installed. When the product is not installed, the setting is configured to $BUILD_DIR/UninstalledProducts in development and deployment builds.
BUILT_PRODUCTS_DIR	The base location for all products. Configured initially as $BUILD_DIR.

Deployment Settings Reference

Table A-2 describes build setting that determine the location of an installed product and its permissions.

Table A-2 Deployment build settings

Build setting	Description
DSTROOT	Base location for the installed product. Default: /tmp/$PROJECT_NAME.dst/.
INSTALL_PATH	Location of the installed product. For example, /my_app_path. This setting is undefined by default.

Build setting	Description
INSTALL_DIR	Fully qualified path for the installed product. By default, it concatenates DSTROOT and INSTALL_PATH. This setting is read-only.
SKIP_INSTALL	Determines whether the target's product gets installed. When undefined, which is the default, the target's product gets installed.
DEPLOYMENT_LOCATION	When YES, the product gets installed in its deployment location ($INSTALL_DIR). Otherwise, the product gets installed in $BUILT_PRODUCTS_DIR. This setting is undefined by default.
INSTALL_OWNER	User who owns the generated product. As pbxbuild should be run by root, the owner should be root. This is applied after the product is deployed.
INSTALL_GROUP	Group who owns the generated product. Usually, staff. This is applied after the product is deployed.
INSTALL_MODE_FLAG	The mode that is applied to the product after it's deployed. Default: ugo-w, o+rX.

Target Settings Reference

Table A-3 describes build settings that identify a target and determine the location of source files and of a directory for temporary files created as a product gets built.

Table A-3 Target build settings

Build setting	Description
TARGET_NAME	Name of the target. This setting is read-only.
PRODUCT_NAME	Name of the product the target builds. This setting is read-only.
ACTION	The action being performed on the target. Values: build or clean from Project Builder, install, installhdrs, and installsrc from pbxbuild. When its value is clean, the target's build directory is deleted and no build phases are executed. This setting is read-only.
SRCROOT	The base location of project sources. It's set to the contents of the PWD environment variable when PWD is defined or to the current directory otherwise.
OBJROOT	The base location for intermediate build files. Configured initially as $SRCROOT/build (MyProject/build).
TEMP_DIR	The location of a target's intermediate files. Configured initially as $OBJROOT/$PROJECT_NAME.build/$TARGET_NAME.build.

Java Compiler Settings

Table A-4 describes build settings that determine the flags that are used in the invocation of the Java compiler as well as the location of generated Java class files.

Table A-4 Java compiler build settings

Build setting	Description
CLASS_FILE_DIR	The base location for Java class files. Configured as $TEMP_DIR/JavaClasses. This setting is read-only.
JAVA_COMPILER	The compiler used in Sources (compilation) build phases. Initially configured as /usr/bin/javac.
JAVA_COMPILER_DEBUGGING_SYMBOLS	Determines whether Java classes are compiled with debugging symbols. When NO, debugging symbols are not generated. When undefined or YES, debugging symbols are generated. Initially undefined.
JAVA_COMPILER_DISABLE_WARNINGS	Determines whether the compiler generates warnings. When YES, warnings are not produced. When undefined or NO, warnings are produced. Initially undefined.
JAVA_COMPILER_DEPRECATED_WARNINGS	Determines whether the compiler shows a description of the use of deprecated API (whether the -deprecation command-line option of javac and jikes is used).
JAVA_COMPILER_TARGET_VM_VERSION	Determines the target Java virtual machine for generated class files (javac and jikes -target command-line options).
JAVAC_SOURCE_FILE_ENCODING	Determines the value for the -encoding command-line option of javac and jikes. When undefined, MACINTOSH is used.
JAVA_COMPILER_FLAGS	Use to set compiler options not supported in build settings for javac and jikes. For example, you can set the -extdirs command-line option of javac to include paths to additional JAR files.
JAVAC_DEFAULT_FLAGS	Base javac command-line options to use for javac. When undefined, the options are configured as -J-Xms64m -J-XX:NewSize=4M -J-Dfile.encoding=UTF8. For more information, see *Inside Mac OS X: Java Development on Mac OS X*.
JIKES_DEFAULT_FLAGS	Base jikes command-line options to use for javac. When undefined, the options are configured as +E +OLDCSO.
JAVA_CLASS_SEARCH_PATHS	Space-separated list of paths of required JAR files. This list is added to the -classpath command-line option of the compiler invocation.
OTHER_JAVA_CLASS_PATH	Colon-separated list of additional paths of required JAR files. This list is added to the -classpath command-line option of the compiler invocation.

Build setting	Description
LINKED_CLASS_ARCHIVES	Space-separated list of required JAR files. Initially configured as the combination of $LINKED_CLASS_ARCHIVES and $OTHER_JAVA_-CLASS_PATH. This setting is read-only.

Java Application Settings

Table A-5 describes build settings that determine whether Java class files are archived, how they are archived, and the name of the archive file, among other items.

Table A-5 Java application build settings

Build setting	Description
JAVA_ARCHIVE_CLASSES	Determines the disposition of Java class files generated by the target. This setting can have two values, YES or NO. When YES (the default), Java classes are archived in a JAR file, which is then copied to the product's Contents/Resources/Java directory. When NO, the class files are copied to that directory. You should not change the value of this setting if you plan to distribute your application or tool.
JAVA_ARCHIVE_-COMPRESSION	Determines whether the contents of the archive file are compressed. When YES the contents of the archive are compressed; otherwise, the contents are not compressed. Initially unconfigured.
CLASS_ARCHIVE_SUFFIX	Determines the extension used for the JAR file. Values: .jar, .war, or .ear.
JAVA_MANIFEST_FILE	Project-directory based path to the file used to supplement the default manifest file (MANIFEST.MF) of the JAR file.
JAVA_APP_STUB	Path to the Cocoa application stub that's embedded in a bundle-based Java application to launch the Java application. Configured as /System/Library/Frameworks/JavaVM.framework/Resources/Mac-OS/JavaApplicationStub. This setting is read-only.
DEVELOPMENT_PLIST_-FILE	Path to the development-settings property list file of the product. Initially configured as $SYMROOT/pbdevelopment.plist.

Document Revision History

This table describes the changes to *Project Builder for Java*.

Date	Notes
2003-10-10	Corrected obsolete links.
2003-05-01	Second preliminary version of *Project Builder for Java*.
2003-03-01	Preliminary version of *Project Builder for Java*.

Document Revision History

Glossary

build phase A build phases defines a concrete task that Project Builder performs to build a product.

build setting A build setting is a variable that stores a specific aspect to be used for building a product.

build style Build styles contain build setting configurations that override the configurations of the active target in a project They allow you to make small changes to a target's configuration without having to create a separate target.

information property list A property list that contains essential configuration information for bundles. A file named `Info.plist` (or a platform-specific variant of that filename) contains the information property list and is packaged inside the bundle.

product An element that gets created as part of the process of generating a running application, such as library files and executable files.

target A target is a blueprint for building a product from specified resources in a project. It consists of a list of the necessary files the actions that need to be performed on them to generate a product.

target, aggregate An aggregate target groups other targets; it contains no product-building instructions. The operations you perform on an aggregate targets are carried out on all the targets it encloses.