Java 1.3.1 Development for Mac OS X

(Legacy)

Java > Unsupported

2002-09-01

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

Contents

Figures, Tables, and Listings

About This Book

Important: The information in this document is obsolete and should not be used for new development.

With Mac OS X, Apple delivers a Java implementation that adds value for both developers and end users. Java is fully integrated into Mac OS X, which offers developers easy access to the Aqua user interface, UNIX-based tools and technologies, QuickTime, OpenGL, and Cocoa. This built-in flexibility allows developers to create robust, well-designed applications that offer users a wide range of rich features and functionality.

This book provides an overview of Java development on Mac OS X and discusses the available features. It also provides simple examples on using the development tools available with Mac OS X.

This book is for the Java developer interested in writing Java applications on Mac OS X version 10.2 with Java 2 Standard Edition (J2SE) version 1.3.1. It does not discuss Java 2 Standard Edition version 1.4 on Mac OS X. Information on previous versions of Java on Mac OS X can be found in the Release Notes at http://developer.apple.com/documentation/java. It is primarily geared toward developers of pure Java applications, but it will also be useful for Cocoa Java development and WebObjects Java development.

This is not a tutorial for the Java language. If you are not already proficient in Java, this document will still be helpful to you but it will not teach you about the Java language and J2SE packages. Many resources exist in print and on the Web for learning the Java programming language. If you are new to programming in Java, you may want to start with one of Sun's tutorials available online at http://developer.java.sun.com/developer/onlineTraining/new2java/.

How to Use This Book

Overall, the purpose is document to highlight how Java development in Mac OS X may be different if you are accustomed to Java development on other platforms. It does not attempt to present a conclusive overview of Java itself. It is meant as a supplemental guide to help you save time and development effort. It introduces you to the tools available to you, discusses some details of how Java is implemented, points out potential trouble spots, and provides some reference documentation.

This book has information for many different types of Java developers. To help you determine what information is important for you the following listing provides an overview of the contents of the following chapters:

- "How Java Is Implemented in Mac OS X" (page 11) Presents a broad overview of how Java works in Mac OS X. It also highlights some areas where things differ between Java in Mac OS X and other platforms. This chapter has many important details for anyone doing Java development on Mac OS X. It is a good place to begin if you don't have specific questions, but want an overview of what is available.
- "Deployment Options" (page 19) Reveals the four suggested deployment vehicles for your Java applications on the Mac OS X operating system. If you have a Java application that you want to distribute on Mac OS X, this chapter is important reading. It discusses not only details to be aware of in the standard distribution methods, but also presents some unique to Mac OS X.

- "The Development Environment" (page 29) If you are new to Java development on Mac OS X, this chapter provides important background information on what tools are available to you.
- "Cross-Platform Practices for Great Native Behavior" (page 37) Discusses some considerations you should make while designing your applications so that they will behave just as well on Mac OS X as they do on other platforms.
- "Using Native Features of Mac OS X in Java Applications" (page 47) Explores some of the options available to you in Mac OS X that are not available on other platforms. Some things discussed in this chapter will tie your application to Mac OS X specifically, while other things will help you to add value to your Java applications on Mac OS X without affecting your application's cross-platform compatibility. The information in this chapter may be important if you are designing a new application or modifying an existing one.
- "Project Builder Tutorial" (page 57) If you want to take advantage of the free Project Builder IDE for you Java development, this is a useful place to start.
- "MRJAppBuilder Tutorial" (page 69) This is a tutorial for anyone distributing a Java application on Mac OS X.
- "Mac OS X Java System Properties" (page 75) Provides a listing of the common Java system properties useful in Mac OS X.

Other Resources

This document and other Java documentation for Mac OS X, including the javadoc API reference, is available online at http://developer.apple.com/documentation/java. A subset of this documentation is installed in `/Developer/Documentation/Mac OS X/Java` on a Mac OS X system with the Mac OS X Developer Tools. You can view this documentation through a Web Browser or through Project Builder (from Project Builder's Help menu, choose Developer Help Center).

The main Java technology page http://developer.apple.com/java contains many links to information about Java development in Mac OS X.

The `java-dev` mailing list is a great source of information on a wide range of Java development topics in Mac OS X. You can sign up for this list at http://lists.apple.com.

Sun's Java site, http://java.sun.com is the essential reference point for Java development in general.

Filing and Tracking Bugs

If you find issues with the implementation of Java that are not represented in this document or want to follow the resolution of an issue, you may do so online through Radar, Apple's bug tracking system. To access Radar, you will need an Apple Developer Connection (ADC) account. You can view the ADC membership options, including the free online membership at, http://developer.apple.com/membership/index.html. With an ADC membership, you can file and view bugs at http://bugreport.apple.com. When filing new bugs for Java on Mac OS X, please use Java (new bugs) for the Component and X as the Version.

How Java Is Implemented in Mac OS X

Java is included in every copy of Mac OS X and Mac OS X Server. Mac OS X provides full support for Java 2 Platform, Standard Edition (J2SE), and support for much of Java 2 Platform, Enterprise Edition (J2EE).

Because of the emphasis Apple has placed on integrating Java into the operating system, there are many benefits to developing and deploying Java applications on Mac OS X rather than other platforms. In Mac OS X, Java is not an add-on feature, but an integral part of the operating system. This means that you do not need to deal with installing and configuring the Java Runtime Environment (JRE) or the Java Development Kit (JDK) as you, or the end users of your application, might need to on other platforms. In addition to the Java pieces you are already familiar with, Apple provides extensions and enhancements to these standard Java APIs as well as Java access to some of Apple's other technologies. With Java on Mac OS X, Apple provides all that you expect in a standard Java 2 implementation while providing the extra value that makes it better on the Macintosh for you as a developer and for your customers.

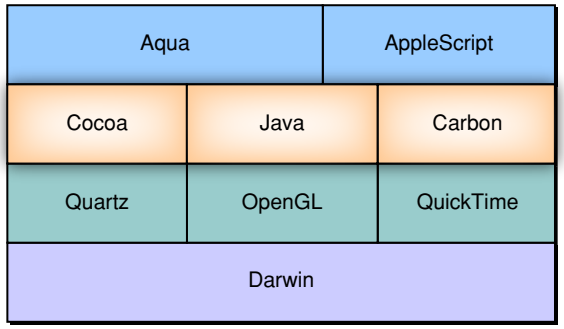
Java Integration in the Operating System

At the core of Mac OS X lives a BSD-style UNIX-based operating system called Darwin. On top of that, Apple has built a collection of fundamental technologies for presenting graphics and sound to users far beyond what has been traditionally available in a UNIX-based operating system. Further collections of tools, event loops, and functions/methods yield high-level APIs for application developers. These include:

- Carbon for traditional Macintosh developers
- Cocoa, an object-oriented application development environment
- Java, an object-oriented environment for building cross-platform applications

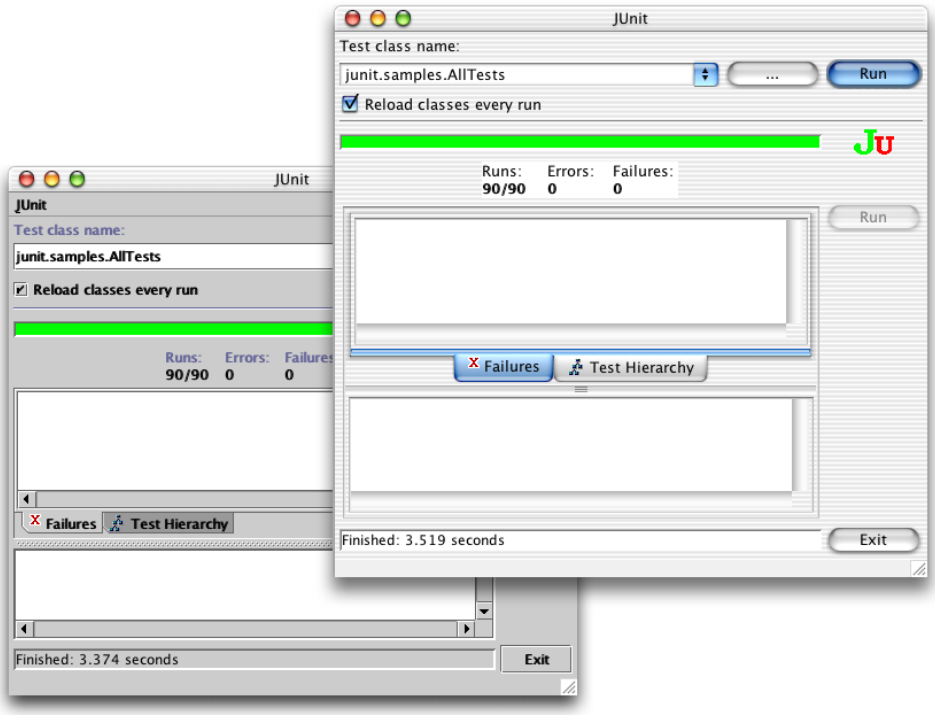
Figure 2-1 (page 11) depicts the conceptual layout of the Mac OS X operating system. A more extensive analysis of the system as a whole is available in *Inside Mac OS X: System Overview*.

Figure 2-1 Architecture of Mac OS X



One of the most obvious advantages of the integration of Java into the operating system can be seen in the user interface of any Swing application. Java on Mac OS X includes a default Aqua look and feel for your Swing applications. You can see from Figure 2-2 (page 12) how this can really make your Java applications look great compared to the default look and feel, Metal.

Figure 2-2 Metal and Aqua look and feel on Mac OS X



Because Mac OS X is a UNIX-based operating system, you have access to BSD tools and technologies useful in software development. You are working with the foundation of a robust, powerful operating system that gives you preemptive multitasking and protected memory.

In Mac OS X you can also take advantage of the operating system's other fundamental features. For example, in Mac OS X you can take advantage of the cross-platform QuickTime API through QuickTime for Java. Apple also provides Java access to the Core Audio framework of Mac OS X, OpenGL, Quartz, the Mac OS X font implementation (with support for OpenType, PostScript Type I and TrueType fonts), the Mac OS X spelling and speech frameworks, and the Foundation and Application Kit frameworks of Cocoa.

What Is Included

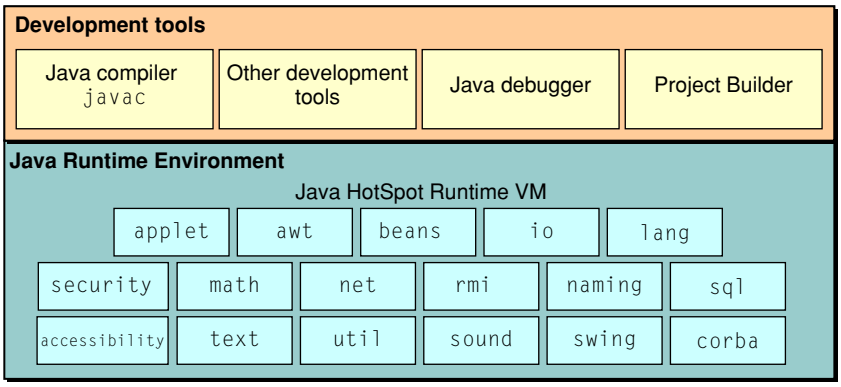
Mac OS X ships with the Java 2 Platform, Standard Edition (J2SE) 1.3.1. Full support of Java 2 means that there is first-class support for Swing, the Collections framework, the Accessibility API, and other APIs from Sun. Support for applets that require Java 2 is also provided. You will also find implementations of the Java2D graphics architecture and the policy-based security model. In addition to the standard parts of J2SE, Mac OS X includes Java Web Start and the security frameworks as well as some packages that were formerly available as part of the Java 2 Enterprise Edition (J2EE) like Remote Method Invocation (RMI) over IIOP, the Java Naming and Directory Interface (JNDI), and the Java Database Connectivity (JDBC) model.

The Java implementation is installed in `/System/Library/Frameworks/JavaVM.framework`. This includes the virtual machine, the command-line tools, the runtime classes, and the API documentation. Additional tools and documentation are included with the free Mac OS X Developer Tools and are installed in `/Developer/Tools` and `/Developer/Documentation`, respectively.

The basic command-line tools are installed in `/System/Library/Frameworks/JavaVM.framework/Commands` with links from `/usr/bin` so that your scripts and tools work on Mac OS X like they would on other platforms.

For the most part, the Java 2 Platform Standard Edition version 1.3.1 works on Mac OS X like it does on other platforms. The standard packages you expect to find are all included in every installation of Mac OS X, as are the basic development tools. With the Mac OS X Developer Tools installed, you have a complete Java Development environment as illustrated in

Figure 2-3 Mac OS X as a Java development platform



Although the `java.sound` packages are included with Java in Mac OS X, you cannot currently use them for sound input. You can work around this by using Mac OS X native audio system. You can find more information about the audio architecture of Mac OS X at http://developer.apple.com/audio. Specific options for providing audio functionality to your Java application in Mac OS X include these:

- Use QuickTime for Java-based services. This works on Mac OS 9, Mac OS X, and Windows.
- Use the Core Audio framework. This API is for Mac OS X only and provides Java classes to access the underlying Core Audio platform services for audio and MIDI in Mac OS X.
- Use JNI to provide a native binding to the Core Audio framework directly.

Java 2D Graphics Implementation

The Mac OS X implementation of the Java2D API is based on Apple's Quartz graphics engine. The Quartz graphics system improves drawing quality over other platforms' Java 2D implementation. Because Java on Mac OS X draws its graphics with Quartz, you might notice that images drawn do not match the Sun Java 2D implementation pixel for pixel, especially when anti-aliasing is on. This is because Quartz's anti-aliasing algorithms for both line art and text are different from Sun's default implementations, so the rendered pixels do not match exactly. Since anti-aliasing is implemented through the operating system itself, it does not hinder graphics performance. Turning it off for your Java application will probably not affect the speed up your code.

In Mac OS X, windows are double-buffered, this includes Java windows. Java on Mac OS X attempts to flush the buffer to the screen often enough to have good drawing behavior without compromising performance. If for some reason you need to force window buffers to be flushed immediately, you may do so with `Toolkit.sync`.

Mac OS X provides hardware graphics acceleration for your Java Swing graphics on computers whose video cards have 16 MB or more of video RAM. If enabled, this technology passes Swing and Java 2D graphics calls directly to the video card. This can result in significant speed increases for your graphics-intensive Java applications. See "Modifying the Default Settings for Hardware Graphics Acceleration" (page 47) for information on changing the default settings of the hardware graphics acceleration.

Anti-aliasing is on by default for text and graphics, but it can be turned off using the properties described in "Mac OS X–Specific Properties" (page 77), or by calling `java.awt.Graphics.setRenderingHint` within your Java application. With anti-aliasing on, drawing shapes over each other may cause different results than in Java 1.1. For example, drawing a white line on top of a black line does not completely erase the line; the compositing rules leave some gray pixels around the edges. Also, drawing text multiple times in the same place causes the partially-covered pixels along the edges to get darker and darker, making the text look smudged.

The Virtual Machine

The Java platform for Mac OS X is based on the Java HotSpot client virtual machine (VM) from the Sun JDK 1.3.1_03 technology train. Because the VM is based on the HotSpot VM, synching is a low overhead operation. The VM maps Java threads directly to Mach threads. The Darwin kernel (XNU) schedules these Mach threads as it would schedule any other threads in the system giving you true preemptive multitasking.

One unique aspect of the Mac OS X Java implementation is its use of a technology to enable shared generation of class files. A single shared Java archive for the Java classes in the system is generated once. This archive is regenerated when you update Java. Since each Java application does not need to generate an independent archive at runtime you save both memory and CPU cycles.

The `-server` option to Java is different in Mac OS X than on other platforms in that it doesn't invoke a different VM. The Java VM is the same Java HotSpot Client VM technology. Using the `-server` flag however, does change a few default settings to values more appropriate to a server type program. For example, a different shared archive generation is used. On Mac OS X Server `-server` is the default. On Mac OS X, `-client` is the default.

Specific guidelines for interacting with the VM are in the following sections.

Performance Profiling Tools

To get the most out of your Java applications, you can take advantage of the performance and profiling tools available to you in Mac OS X. Built-in profiling support is provided in flags you can pass in to the VM at runtime. For basic CPU monitoring and profiling, use the `-Xrunhprof` flag as follows:

- `java -Xrunhprof:cpu=sample`*`yourApplication`*
- `java -Xrunhprof:monitor=y`*`yourApplication`*

For allocation profiling use the `-Xaprof` flag. To profile a single thread use `-Xprof`.

Third-party tools like Borland's Optimizeit and Hewlet Packard's `HPjmeter` are also valuable tools.

Nonstandard Virtual Machine Options

The Mac OS X Java VM includes some nonstandard VM options that you should be aware of. Since these are nonstandard (`-X`) options, you should keep in mind that these are not guaranteed to be available on other VM implementations, and could change from release to release. These options are set by passing a `-X`*optionname* flag to the Java runtime (`java`). You can also see a list of available options by typing `java -X` in a Terminal window. VM options that have important details to note about them are outlined in Table 2-1 (page 15).

Table 2-1 HotSpot VM options

Property	Notes
`-Xdock:name=`*`applicationName`*	Overrides the default application name displayed in the Dock and the menu bar.
`-Xincgc`	This option is disabled in Mac OS X version 10.2.
`-Xms`*`size`*	Sets the initial Java heap size. The maximum heap limit is about 2 GB.
`-Xms`*`size`*	Sets the initial Java heap size. The maximum heap limit is about 2 GB.
`-Xmx`*`size`*	Sets maximum Java heap size. The maximum heap limit is about 2 GB.
`-XX:+UseTLE`	Enables Apple's thread local eden (TLE) allocation. This allows for more scalable allocation for heavily threaded applications, greatly increasing allocation performance. It is on by default on multiprocessor computers.
`-XX:+PrintJavaStackAtFatalState`	Enable Java backtraces to be generated when a crash occurs in native code. Note that this is not guaranteed to work all the time since the crash may destroy some critical data structures.

The Java Native Interface

Java Native Interface (JNI) in Mac OS X works as you would expect it to on other platforms with a couple of important details to remember.

JNI libraries are named with the library name used in the `System.loadLibrary` method of your Java code prefixed by `lib` and suffixed with `.jnilib`. For example, `System.loadLibrary("hello")` loads the library named `libhello.jnilib`.

In building your JNI libraries, you have two options. You can either build them as bundles or as dynamic shared libraries (sometimes called dylibs). If you are concerned about maintaining backward compatibility with previous versions of Java on Mac OS X, you should build as a bundle; otherwise you will probably want to build as a dylib. Dylibs have the added value of being able to be prebound which speeds up the launch time of your application. They are also a little simpler to build if you have multiple libraries to link together.

To build as a dynamic shared library, use the `-dynamiclib` flag. Since your `javah` produced `.h` file includes `jni.h`, you need to make sure you include its source directory. Putting all of that together looks something like this:

```
cc -c -I/System/Library/Frameworks/JavaVM.framework/Headers sourceFile.c

cc -dynamiclib -o libhello.jnilib sourceFile.o -framework JavaVM
```

To build a JNI library as a bundle use the `-bundle` flag:

```
cc -bundle -I/System/Library/Frameworks/JavaVM.framework/Headers -o libName.jnilib
-framework JavaVM sourceFiles
```

For example, if the files `hello.c` and `hola.c` contain the implementations of the native methods to be built into a dynamic shared JNI library, and it needs to be called with `System.loadLibrary`("hello"), you would build the resultant library, `libhello.jnilib`, with this code:

```
cc -c -I/System/Library/Frameworks/JavaVM.framework/Headers hola.c
cc -c -I/System/Library/Frameworks/JavaVM.framework/Headers hello.c
cc -dynamiclib -o libhello.jnilib hola.o hello.o -framework JavaVM
```

Often JNI libraries have interdependencies. For example assume the following:

- `libA.jnilib` contains a function `foo()`
- `libB.jnilib` needs to link against `libA.jnilib` to make use of `foo()`

This is not a problem if you build your JNI libraries as dynamic shared libraries, but if you build them as bundles this does not work since symbols are private to a bundle. If you need to use bundles for backward compatibility, one solution is to put the common functions into a separate dynamic shared library and link that to the bundle. For example:

1. Compile the JNI library:

   ```
   cc -g -I/System/Library/Frameworks/JavaVM.framework/Headers -c -o myJNILib.o
   myJNILib.c
   ```

2. Compile the file with the common functions:

   ```
   cc -g -I/System/Library/Frameworks/JavaVM.framework/Headers -c -o
   CommonFunctions.o CommonFunctions.c
   ```

3. Build the object file for your common functions as a dynamic shared library:

   ```
   cc -dynamiclib -o libCommonFunctions.dylib CommonFunctions.o
   ```

4. Build your JNI library as a bundle and link against the dynamic shared library with your common functions in it:

   ```
   cc -bundle -lCommonFunctions -o libMyJNILib.jnilib myJNILib.o
   ```

Note: Once you have built your JNI libraries, make sure to let Java know where they are. You can do this either by passing in the path with the `-Djava.library.path` option on the command line or in your information property list, or by setting the `DYLD_LIBRARY_PATH` environment variable.

You can also access native code libraries through JDirect. See "JDirect" (page 53) for more information on using this Mac–specific technology.

Environmental Differences

Although the Aqua user interface may belie it, behind the scenes Mac OS X is a UNIX-based operating system. The things you expect to be there are indeed there. At times, however, pieces might be in a slightly different spot than you expect. To make your experience on Mac OS X as smooth as possible, this section looks at where you might expect to find things on Mac OS X and also addresses some of the issues of environment variables.

Note: You can access the command line interface to Mac OS X through the Terminal application in `/Applications/Utilities/`. Terminal launches your user shell. By default this is `tcsh`, an enhanced C shell. `bash`, a Bourne compatible-shell, `zsh`, a `ksh`-like shell, and `csh` are also provided.

Finding JAVAHOME

Many Java applications require you to identify Java's home directory on your system (`JAVAHOME`), especially during installation. On Mac OS X this should be set to `/Library/Java/Home`. You might notice that this is actually a symbolic link to `/System/Library/Frameworks/JavaVM.framework/Home`. This is the real Java home directory, but since it is owned by the system you should not modify it, nor should you expect that users will be able to modify it on their system. This directory will be modified by software updates from Apple so any changes you make there are volatile. The `/Library/Java/Home` symlink is provided as an abstraction so that you can be sure that your applications will continue to work when Apple updates either the Java VM or the operating system itself. `/Library/Java/Home` allows access to the `/bin` subdirectory where command-line tools like `java` and `javac` can be found. These tools are also accessible through `/usr/bin`.

Where to Put Extensions

Java can be extended by adding custom `.jar`, `.zip`, and `.class` files, as well as native JNI libraries, into an extensions directory. On some platforms this is designated by the `java.ext.dir` directory. In Mac OS X, put your extensions in `/Library/Java/Extensions`. Java automatically looks in that directory as it is starting up a VM.

Putting extensions in `/Library/Java/Extensions` loads those extensions for any user on that particular computer. If you want to limit which users can use certain extensions, you could put them in the user's home directory in `~/Library/Java/Extensions`. You may need to make that directory if it does not already exist. It is important to note that extensions in the user's directory are loaded first, so if you have duplicate JAR files in your user's `Library/Java/Extensions` directory and the system's `/Library/Java/Extensions` directory, those in the latter directory may overload classes in the former.

Where to Put Preferences

Some applications store their preferences in the application directory; others store their preferences where the host platform stores its preferences. In Mac OS X, user preferences for an application are stored in the `~/Library/Preferences` directory. System-level preferences, or preferences that should effect all users should be stored in `/Library/Preferences`. The standard preference format in Mac OS X is an XML property list, with a title of the form of a reverse URL followed by the `.plist` extension, for example `com.apple.ProjectBulder.plist`. This directory can be reached from Java code by appending the string `Library/Preferences` to the `user.home` System property.

Setting the Classpath

In Mac OS X, you treat the classpath the way you would in any BSD shell. You can use `setenv classpath`*newClasspathAddition* to add to the classpath for the duration of the current session for that particular Terminal window. You still have to explicitly add a reference to this variable when compiling by typing `javac -classpath %CLASSPATH`*filename*. You can determine the current value of `CLASSPATH` with the command `echo $CLASSPATH`. By default no `CLASSPATH` variable is explicitly set for the shell, although Java knows to look in the various `Library/Java/Extensions` folder for developer-supplied JAR files.

You can avoid retyping the `setenv classpath` command for each Terminal session by automatically adding paths to your classpath shell startup script (`.cshrc` or `.tcshrc` for the default `tcsh` shell). For example, find or create `~/.cshrc`. That file should have a line that looks similar to `setenv CLASSPATH ".:/Users/username/Projects/HelloWorld/:/Users/username/Projects/HelloWorld/HelloWorld.jar"`. In this example, the current directory, the specific directory of a particular project, and a JAR file in that directory are included in the classpath.

Java Output

When you launch a Java application from the command line, standard output goes to the Terminal window. When you launch a Java application by double clicking it, your Java output is displayed in the Console application in `/Applications/Utilities`. Applets that use the Java Plug-in may display output in the Java Console.

Deployment Options

There are basically four different distribution methods for Java applications in Mac OS X. You can distribute your applications as a JAR file, a Mac OS X Java application, a Java Web Start application, or an applet. They each have different benefits as outlined in the following sections.

Distributing Your Application as a JAR File

The most basic distribution method for your Java applications is as a JAR file. It is the simplest way for you as a developer since it requires very little, if any, changes from the JAR files you distribute on other platforms. It does though, have drawbacks for the end user. The major drawback being that Swing applications look out of place with their odd application names in the menu bar and generic Java icons in the dock. Considering this alone, deploying your application from a JAR file is not recommended on Mac OS X if your application has a graphical interface and will be run by general users.

If you do choose to deploy your application from a JAR file on Mac OS X, it is important to remember to include in your JAR file a valid manifest file that declares the class that contains the `main` method. Without one, users will have to resort to launching your application from the command line. This will seem very outdated for users whose platform has let them launch an application simply by double-clicking it for almost twenty years!

If you do not have the class with `main` declared in your JAR file a simple way to fix this is as follows:

1. Unarchive your JAR file into a working directory with some variant of `jar xvf` *myjar*`.jar`
2. In the `META-INF` directory is a `MANIFEST.MF` file. Copy that file and add a line that begins with `Main-Class:` followed by the name of your `main` class, for example, `Main-Class: HelloWorld`.
3. Archive you files again but this time use the `-m` option to `jar` and designate the relative path to the manifest file you just modified, for example, `jar cmf` *`YourModifiedManifestFile`*`.txt`*`YourJARFile`*`.jar*.class`

This is a very basic example that does not take into account more advanced uses of the `jar` program. More detailed information on adding a manifest to a JAR file can be found in the `jar(1)` man page. From Project Builder, choose “Open man page” from the Help menu. (In Terminal, type `man jar` .)

Mac OS X Java Applications

Native Mac OS X applications include more than just the executable code. They also include images, sounds, icons, localizable strings, and other resources that the application may use. They might even include executables for different versions of the operating system. The applications that are visible in the Finder are actually directories that hold the executable code and the relevant resources. This directory structure is

hidden from view in the Finder by the `.app` suffix and a specific bit that is set for that directory. Such a directory is often referred to as a Mac OS X application bundle. More information on Mac OS X application bundles is available in *Inside Mac OS X: System Overview*. Since the J2SE platform includes ways to deal with many of these other things, you might not need the full functionality of a Mac OS X application. There are, however, some things that you gain by wrapping your pure Java JAR file into an application bundle, and it is very simple to do.

Overall, it provides a better user experience for your users, helping your application to integrate more closely with native Mac OS X applications. Specific benefits include these:

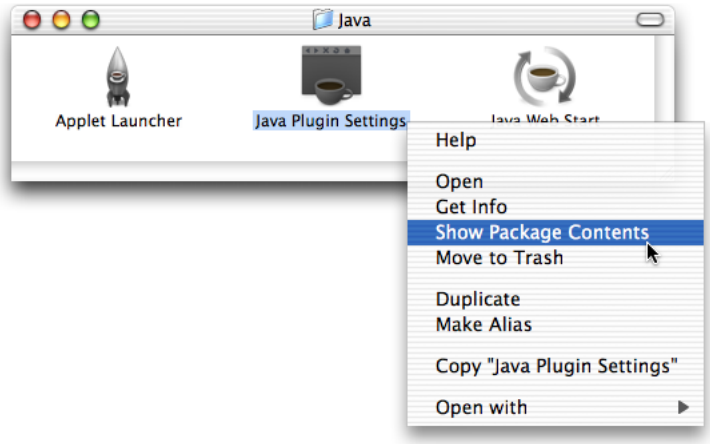
- Users can simply double-click the application to launch it.
- If you add an appropriate icon, it shows the application icon in the Dock, clearly identifying your application. Otherwise, a default Java icon appears in the Dock.
- It lets you set specific system properties that can make your Java application hard to distinguish from a native application.
- You can bind specific document types to your application. This will allow users to launch your application by double clicking a document associated with it.

MRJAppBuilder, discussed in more detail in "MRJAppBuilder" (page 32) and "MRJAppBuilder Tutorial" (page 69), helps you easily bundle your existing Java application as a Mac OS X Java application.

Application Bundles

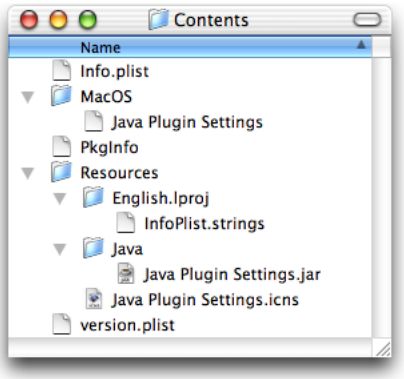
What users see as an application in Mac OS X is actually a bundle of resources to a developer. To get a glimpse inside an application on Mac OS X, you can either explore the directory of resources from the Terminal or from the Finder. Although by default the Finder displays applications as a single object, you can view the true makeup of any application in the Finder. To see what is contained inside a standard application bundle, Control-click any application and choose Show Package Contents as in Figure 3-1 (page 20).

Figure 3-1 Show application bundle contents



You should see something like Figure 3-2 (page 21).

Figure 3-2 Contents of a Java application bundle



Notice some important details:

- Mac OS X expects to see an `Info.plist` file in the Contents folder. In the case of a Java application, this contains some important information that Mac OS X uses to set up the Java runtime environment for your application. More information about these property lists is in "Property List Attributes for Java Applications" (page 21)
- If you have an icon that should be displayed in the Mac OS X Finder, put it in the Resources folder. There is a Mac OS X–specific file type designated by the `.icns` suffix, but most common image types work. To make an icon (`.icns`) file from your images, use the Icon Composer application installed in `/Developer/Applications` with the Mac OS X Developer Tools.
- Your Java code, in either JAR or `.class` files is put into `Resources/Java`. This is launched by a native executable file in the MacOS folder.

There are other files in the application bundle, but the specific ones mentioned are the ones you probably care most about as a Java developer. You can learn more about the other files an application bundle in *Inside Mac OS X: System Overview*.

Property List Attributes for Java Applications

Mac OS X makes extensive use of XML files for various system settings. These are called property lists and have a `.plist` extension. The `Info.plist` file in the Contents folder of a Mac OS X application is one such property list. If you build your Java application in Project Builder or MRJAppBuilder, this file is automatically generated for you. Even if it is built for you, there may be times when you may want to modify it. Since it is a simple XML file, you can modify it with any text editor. Its settings will be read the next time you launch your application from the Finder. An example property list for a Java application is shown in Listing 3-1 (page 21).

Listing 3-1 `Info.plist` file for simple Java application

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>CFBundleDevelopmentRegion</key>
```

```
    <string>English</string>
    <key>CFBundleExecutable</key>
    <string>SampleApp</string>
    <key>CFBundleIconFile</key>
    <string>SampleApp.icns</string>
    <key>CFBundleInfoDictionaryVersion</key>
    <string>6.0</string>
    <key>CFBundlePackageType</key>
    <string>APPL</string>
    <key>CFBundleSignature</key>
    <string>????</string>
    <key>CFBundleVersion</key>
    <string>0.1</string>
    <key>Java</key>
    <dict>
        <key>ClassPath</key>
        <string>$JAVAROOT/SampleApp.jar</string>
        <key>MainClass</key>
        <string>SampleApp</string>
        <key>Properties</key>
        <dict>
            <key>com.apple.macos.useScreenMenuBar</key>
            <string>true</string>
            <key>com.apple.mrj.application.apple.menu.about.name</key>
            <string>SampleApp</string>
        </dict>
    </dict>
</dict>
</plist>
```

The property list is divided into hierarchical dictionaries. The top-level dictionary contains the information that the operating system needs to properly launch the application. The keys in this section are prefixed by CFBundle and are usually self explanatory. Where they are not, see the documentation in *Mac OS X Developer Release Notes: Information Property List* and *Inside Mac OS X: System Overview*.

At the end of the CFBundle keys, a `Java` key designates the beginning of a Java dictionary. The two top-level keys in the Java dictionary are required in the property list of a Java application bundle. They are defined as follows:

- `MainClass`, corresponds to the `com.apple.mrj.application.main` system property. The string value for this key should specify the fully qualified class name for the class containing you application's `main` method.
- `ClassPath`, corresponds to the `com.apple.mrj.application.classpath` system property. The string value for this key should specify the fully qualified path to the directories where your class files are or to your JAR files.

Aside from those two required keys, there are some optional keys that you can add here. They are described in "Mac OS X Application Properties" (page 76).

The `Properties` sub dictionary of the Java dictionary contains keys that you pass to your Java applications from the command line with the `-D` option. For example, if you want to run this same application from the command line, you pass in the two keys, `com.apple.macos.usescreenmenubar` and `com.apple.mrj.application.apple.menu.about.name` as follows:

```
java -Dcom.apple.macos.useScreenMenuBar=true
-Dcom.apple.mrj.application.apple.menu.about.name=SampleApp SampleApp
```

The keys in the `Properties` dictionary include both Mac OS X–specific options as well as general Java options. Mac OS X–specific keys and values that you may add to this dictionary of the property list are specified in "Mac OS X Application Properties" (page 76).

If you examine an application built with MRJAppBuilder, you might notice that some of the keys seem to be missing from the `Properties` dictionary. MRJAppBuilder uses a legacy style non-XML property list named MRJApp.properties in the `Contents/Resources` folder of an application bundle. This property list contains the same flags and system properties that you would normally find in Properties sub-dictionary (of the Java dictionary) in an Info.plist. In general, this list behaves like the Info.plist with a few exceptions:

- Arguments to `main()` may not contain embedded spaces. You can work around this by passing in such arguments to the Info.plist, though.
- The `APP_PACKAGE` property is not expanded when used in `com.apple.mrj.application` parameters.
- Additional command-line arguments designated in `MRJApp.properties` are ignored if the application is launched from the command line.

Setting the Java Runtime Properties for an Application Bundle

Having seen the structure of the Mac OS X information property list should help you to debug your application and modify existing applications, but how do you generate that list in the first place? You could build application bundles by hand in the Terminal, write the `Info.plist` file yourself in a text editor, and designate the file as an application with the `SetFile` command-line tool. There are two much simpler solutions: Project Builder and MRJAppBuilder.

If you build a Java AWT or Swing application from one of Project Builder's templates, Project Builder automatically generates a default `Info.plist` file. You may either modify this by hand or directly from Project Builder as outlined in "Modifying the Application Parameters" (page 64). If you want to turn your preexisting Java application into a Mac OS X Java Application, MRJAppBuilder is the tool for you. It allows you to take existing `.class` or JAR files and wrap the information around them that Mac OS X expects to find when launching a native application, including setting up the runtime properties of your Java application. A simple tutorial for MRJAppBuilder is available in "MRJAppBuilder Tutorial" (page 69). "Project Builder Tutorial" (page 57) provides a tutorial on using Project Builder to build a Mac OS X native application.

Setting Runtime Properties in Project Builder

When you begin a new Java Swing or Java AWT application project in Project Builder, the `ClassPath` and `MainClass` properties are generated automatically. The `ClassPath` property is updated as you add and remove `.class` and `.jar` files. If you change the name of your `main` class or want to append values for any of the other properties available in "Mac OS X Java System Properties" (page 75), you may do so by editing the target settings. Choose Edit Active Target from the Project menu. In the resulting window (or pane depending your settings) you can modify the settings in a more user-friendly manner than by editing the property list by hand. Some of the most common properties are set with a checkbox. Other properties are set in the Additional Properties and Additional VM Options fields.

Setting Runtime Properties in MRJAppBuilder

The Java Properties pane of MRJAppBuilder allows you to set Java runtime properties. The main and classpath fields are not editable. These values are derived from your settings in the Application pane. A default set of properties is already provided but it is simple to add more:

1. Click the Add button.
2. In the property field, put the name of the system property as specified in "Mac OS X Java System Properties" (page 75).
3. Fill in the appropriate value for that property.
4. The Description field can be left blank; its information will not be saved.

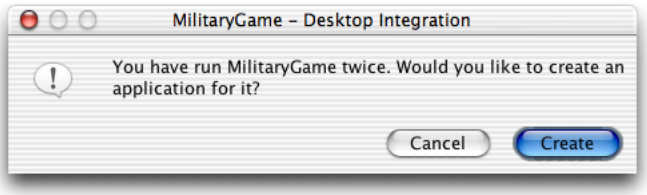
When you build your application, you will see the specified settings in the `Info.plist` and `MRJApp.properties` files.

Java Web Start

Java Web Start provides yet another way you can distribute your Java applications on Mac OS X. Although not a standard part of Java 1.3.1, Mac OS X does provide Java Web Start with the default installation of the operating system. This is an implementation of the Java Network Launching Protocol & API (JNLP) Specification, v1.0.1. This means that if you choose to build JNLP-aware applications, Mac OS X users do not need to do anything to take advantage of them. They have access to your applications through the Web browser and the Java Web Start application (installed in `/Applications/Utilities/Java`).

By default, if a user launches a Java Web Start application more than twice from the same URL, they are prompted to save the application as a standard Mac OS X application, as shown in Figure 3-3 (page 24). They are also prompted on where they want to save your application. The application is still a Java Web Start application, with all the benefits that offer, but it now easier for users to run your application since they do not have to launch a Web browser or the Java Web Start application.

Figure 3-3 Java Web Start integration



The desktop integration setting can be changed in the Preferences of the Java Web Start application.

There are only a few details to be aware of in how the Mac OS X implementation of Java Web Start differs from the Windows and Solaris versions:

- It does not support downloading of additional Java Runtime Environments (JREs). Mac OS X includes J2SE 1.3.1, so if your application specifically requires JRE 1.2 or 1.4, it will not work. Specifications for versions numbers that can expand to include 1.3.1 will work though, for example *1.2+* or *1.3+*.
- It is s not necessary to set up proxy information explicitly in the Web Start application. Java Web Start in Mac OS X automatically picks up your proxy settings from the Network preference pane.
- Java Web Start caches its data in the user's `/Library/Caches/Java Web Start` directory.

Applets

Applets have always been one of the most common deployment methods for Java. Mac OS X provides a robust environment for applet development through the use of Sun's Java Plug-in. Applets use the same 1.3.1 VM that Java applications use. Their behavior should be similar to the behavior of Sun's Java Plug-in. Of course as with all applet deployment, the host platform's support for Java is only part of the story. How Web browsers interpret your HTML code to launch the applet is also key. This section gives you some relevant information for deploying your applets in Mac OS X.

Accessing Mac OS X–Specific Properties From Applets

Mac OS X includes some specific system properties that you might want to take advantage of in your applets. Except for the `com.apple.macos.useScreenMenuBar`, unsigned applets cannot access these properties. If you want to use any of the properties discussed in "Mac OS X Java System Properties" (page 75), you can grant permission to access them by adding a line to your system-wide `java.policy` file located at `/Library/Java/Home/lib/security/`.

The line should be of the form:

`java.util.PropertyPermission` *systemPropertyName*, `read`;

Java Applet Plug-in

In previous versions of Mac OS X, applications that displayed applets, such as Web browsers, used the Mac OS X Java Embedding framework to embed the Java applets in the native application. This framework uses Sun's reference `appletviewer` class not Sun's Java Plug-in architecture. When this framework is used to display an applet, users miss out on the additional features of Sun's Java Plug-in. Additionally applet behavior is dependent on how the browser maker uses the Java Embedding framework.

In Mac OS X version 10.2, browser developers may easily display applets with the Java Applet Plug-in architecture (`Java Applet.plugin`) without writing browser–specific code to use the Java Embedding framework. Mac OS X's Java Applet Plug-in takes full advantage of Sun's Java Plug-in. What this means to you is that if you want to deploy your Java applets on Mac OS X is that you should modify your HTML code so that your applets are be run through the new Java Applet Plug-in architecture and not the Java Embedding framework.

Taking Advantage of the Java Applet Plug-in with HTML

Although Sun encourages the use of the `<APPLET>` tag for all applets, this does not always give you the recommended behavior. For example, in some browsers, `<APPLET>` does not give you the full functionality of Sun's Java Plug-in, while the `<OBJECT>` tag, which is mapped to the mime type `application/x-java-applet`, does. This is illustrated in Figure 3-4 (page 26).

Figure 3-4 Effect of HTML tags in Mac OS X 10.2

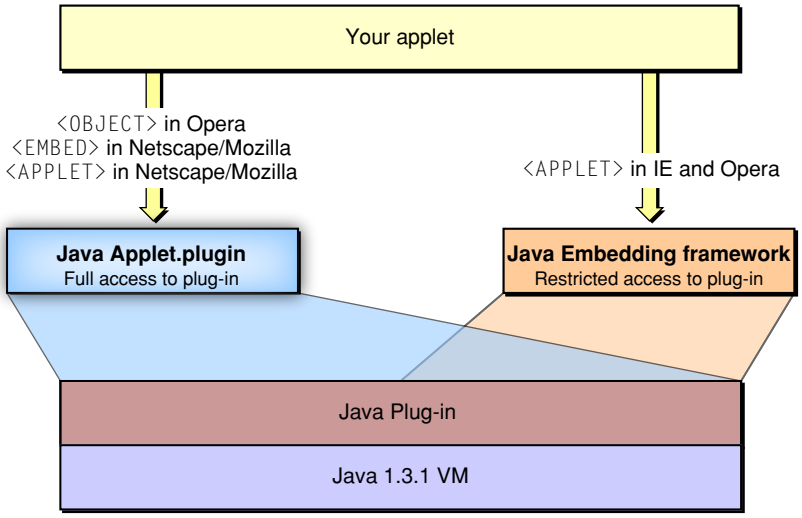


Figure 3-4 (page 26) shows the effect of the <APPLET> tag in comparison to the <EMBED> tag. You can see that the suggested <APPLET> tag has the desired results only in Mozilla/Netscape. To work around the different interpretations of the <APPLET> tag, you have a few options. You can just use the tag listed in Listing 3-1 (page 21) that invokes the Java Plug-in or you can use Sun's HTML converter to get HTML that works in any browser. The converter is available at http://java.sun.com/products/plugin/1.3/docs/html-conv.html. Without changing anything, you should see the same applet behavior that you saw in previous versions of Mac OS X.

Table 3-1 Interpretation of HTML tags in common Mac OS X browsers

Browser	<APPLET>	<OBJECT>	<EMBED>
Internet Explorer 5.2.x (default browser in Mac OS X 10.2)	Java Embedding framework		
Netscape/Mozilla	Treated as an <EMBED> tag, which by default maps to the `application/x-java-applet` mime type.		Java Plug-in
OmniWeb			Java Plug-in
Opera		Java Plug-in	
Chimera	Treated as an <EMBED> tag.		Java Plug-in

Benefits of Using the Java Applet Plug-in

Using the Java Applet Plug-in provides a better experience for people that use your applets. Changes have been made in the areas discussed below.

JAR Caching

You can now designate that you want to store certain JAR files for repeated use. If you developed for Mac OS 9, notice that this is similar to JAR caching on MRJ 2.2.x. The cache is stored in the users home folder in `Library/Caches/Java`. To take advantage of JAR file caching, you may need to modify your HTML with the following tags:

`<PARAM NAME = "cache_option" VALUE="plugin">`
: Turns on caching.

`<PARAM NAME = "cache_archive" VALUE="`*a*`.jar,`*b*`.jar,`*c*`.jar">`
: This is an optional tag used to specify the list of JAR files you want to cache.

  JAR files in `cache_archive` are searched first, then the JAR files designated with the `ARCHIVE` tag are used

`<PARAM NAME = "cache_version" VALUE="`*1.2.0.1*, *2.1.1.2*, *1.1.2.7*`">`
: This is an optional tag used to specify the version number of the JAR files designated with `cache_archive`. Each value corresponds to the respective JAR files designated with `cache_archive`. If the version value is newer than what is cached, the JAR file in the cache is updated. If this tag is omitted, the plug-in checks the server to see if there is a newer version available and caches that version.

JAR file caching in Mac OS X conforms to the Java 1.3.1_03 standard. It does not conform to the Java 1.4 standard. This means that there are certain things you should keep in mind:

- JAR files specified with the `ARCHIVE` tag are not cached.
- The `cache_version_ex` parameter is not supported.
- There is no JAR file indexing support.
- For better forward compatibility, it is suggested that you use the `cache_archive` and `cache_version` parameters.

Signed JAR Files

If a user decides to always trust your JAR file, a certificate is stored in the user's home folder in `Library/Preferences/Java Plugin certificates 1.3.1`.

Users can view their JAR file certificates in the Java Plugin Settings application.

The Java Console

The Java Console provides a way to log and trace the behavior of your applet while it is running. It can give you interactive thread information and allow you to force garbage collection. The Java Console is the display medium for `System.outSystem.err` for applets. The Java Console is visible if you select the option Show Java Console in the Java Plugin Settings utility.

Java Plugin Settings Application

The Java Plugin Settings application allows users to fine-tune how applets behave in Mac OS X. It behaves on Mac OS X as it does on other platforms. It is important to recognize that settings you make while testing may not be the settings that users have on their computers. These settings are stored per user in `~/Library/Preferences/com.apple.java.plugin.properties131`.

The Development Environment

Mac OS X provides a very robust environment for Java development. This chapter outlines some of the tools available to you in Mac OS X. It also discusses some things that may be different in the development environment from other platforms that you may have used for Java development.

Java Development Tools

There are three basic types of Java development tools available to you in Mac OS X:

- There are the standard command-line Java tools that you normally associate with the Java Development Kit (JDK).
- Apple provides additional command-line and GUI-based tools free with the Mac OS X Developer Tools.
- There are many third-party tools, both open source and commercial, that you can use for Java development on Mac OS X.

Standard JDK Tools

Most of the same tools that you would expect to find on Linux, Solaris, or Windows with the Java Development Kit (JDK) installed are included with Mac OS X by default. There are no extra installation steps required. Most of the standard JDK tools are command-line tools. The Java command-line tools are accessible from `/usr/bin` and `/Library/Java/Home/bin`. These tools are included with the default installation of Mac OS X:

- basic Java tools
  - `java`— runtime and virtual machine
  - `javac`—compiler
  - `javah`—C header and stub file generator
  - `javap`—class file disassembler
  - `jdb`—debugger
  - `jar`—archive tool
  - `javadoc`—API documentation generator
  - `appletviewer`—applet viewer
  - `extcheck`—JAR conflict detection utility
- Remote Method Invocation (RMI) tools
  - `rmic`—RMI stub compiler

- ❏ `rmid`—RMI activation system daemon
- ❏ `rmiregistry`—remote object registry
- ❏ `serialver`—tool to obtain class serial version

- internationalization tool
- ❏ `native2acsii`—text converter to Unicode

- security tools
- ❏ `jarsigner`—JAR signing and verification tool
- ❏ `keytool`—key and certificate management tool
- ❏ `policytool`—tool to manage Java policy files

- Java Interface Definition Language (IDL) and RMI over Internet Inter-ORB Protocol (IIOP) tools
- ❏ `idlj`—IDL-to-Java compiler
- ❏ `tnameserv`—Java IDL name server starter script

Documentation on using these tools is available in the online manual (`man`) pages. These are available in Project Builder under the Help menu as well as in the shell itself.

Displaying a Java Stack Trace

When debugging your Java applications, you may want to display a stack trace. If you launched your application from the command line, CTRL-\ generates a `SIGQUIT` signal and prints the current stack trace in your Terminal window. If you launch your Java application from the Finder, you just need to find its process ID (PID) and deliver a `SIGQUIT` signal to that process. You see the stack trace in the Console application. For example, first launch Applet Launcher, Terminal, and Console (all are in `Applications/Utilities`). If you want to generate a stack trace for Applet Launcher, in Terminal type:

```
ps -auxwww | grep "Applet Launcher"
```

You would see the results as something like:

```
bgerfen 1490   0.0  2.3   255700  17912  ??  S     2:39PM   0:05.76
/Applications/Utilities/Java/Applet Launcher.app/Contents/MacOS/Applet Launcher
 -psn_0_5898241
bgerfen 1507   0.0  0.0     1116      4 std  R+    2:41PM   0:00.00 grep -i
Applet Launcher
```

Once you have the appropriate PID, kill it. For this example this looks like:

```
kill -QUIT 1490
```

Console displays the stack trace.

Note: If you deliver a signal to a process that is running in `gdb`, `gdb` breaks on the signal. You can just continue on (with the c command) since you are actually not interested in the signal itself, but in the output the signal handler gives you.

You may also want to include monitor status information in the stack trace. If you are running your application from the command line, just run your program with the `-XX:+JavaMonitorsInStackTrace` flag. You can also make a `.hotspotrc` file in your home directory with this line in it:

```
+JavaMonitorsInStackTrace
```

Since this file is parsed every time a Java VM starts up, it is in effect for applications run from the command line, double-clickable applications run from the Finder, and even Java applications embedded within other applications.

If you have a reproducible crash, you might also want to find the stack trace for the native code by running your application in `gdb`. After a crash use this command:

```
thread apply all bt
```

Other Command-Line Tools

Having a UNIX-based core at the heart of the operating system provides you, as a Java developer, access not only to Java tools, but also a host of general UNIX-based development tools.

A look in `/usr/bin` shows many tools that make Java development on Mac OS X very comfortable if you are already accustomed to a UNIX-based operating system. (In the Finder, choose "Go to Folder" from the Go menu.) You will find `emacs`, `gdb`, `make`, `pico`, `vi`, and `perl` among others. If you do not see all of these, you may need to install the Mac OS X Developer Tools. See "Where to Get the Tools" (page 34) for more information.

If you are looking for Mac OS X ports of other command-line tools, look first in the Darwin CVS repository available at http://developer.apple.com/darwin or http://www.opendarwin.org. You can also look at the main repository of the tool you are trying to obtain. For basic information on porting your favorite non-Java tools to Mac OS X, see *Inside Mac OS X: UNIX Porting Guide*.

GUI-Based Tools

Among the GUI-based tools are three that are especially important for Java development: Project Builder, MRJAppBuilder, and Applet Launcher.

Project Builder

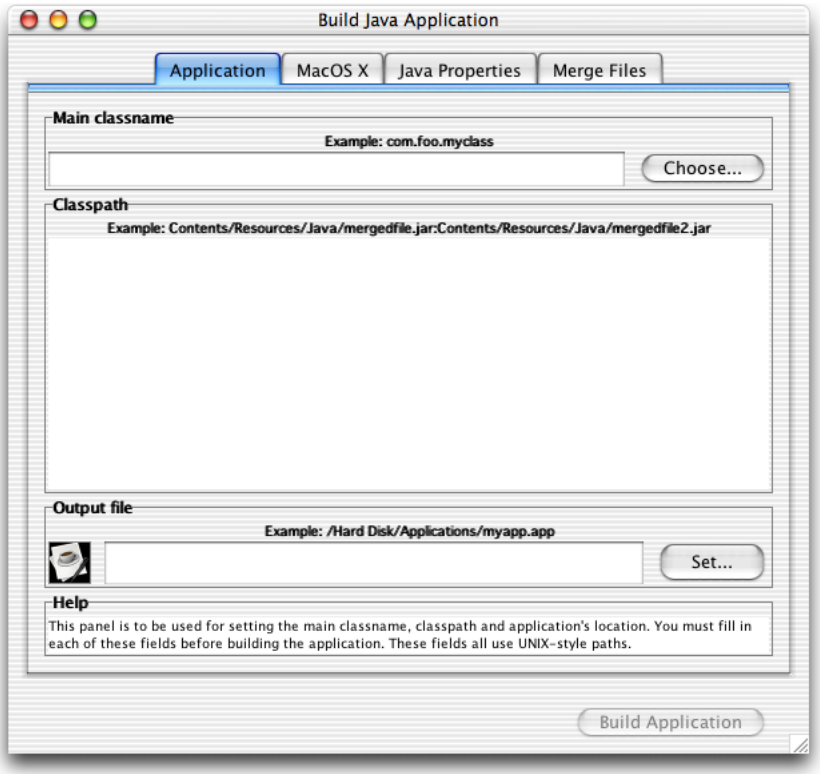
Project Builder is a complete integrated developmente environment (IDE) that allows you to edit, compile, debug, and package your Java applications. A tutorial of how to build a simple Java application is included in "Project Builder Tutorial" (page 57). Additional details of using Project Builder to build Java applications are included in the section on compiling Java files in Project Builder Help.

MRJAppBuilder

MRJAppBuilder is a utility for packaging already-compiled Java applications to run as Mac OS X applications. MRJAppBuilder constructs applications in the same application bundle format as other Mac OS X applications. It is very simple to use and allows you, with minimal work, to make your Java application launch like native Mac OS X applications. You can even easily set options like the Dock icon and the application name that appears in the menu bar without modifying your Java source code. Additionally you can use MRJAppBuilder to set runtime flags that you might want to use only in Mac OS X. In order to use MRJAppBuilder, you need only know the main class name of your application and have access to the `.class` files.

The interface to MRJAppBuilder is simple and readily apparent when you open the application. As illustrated in Figure 4-1 (page 32), there are four tabbed panes in the MRJAppBuilder window.

Figure 4-1 MRJAppBuilder



The information in the Application pane is all that you need to make a Mac OS X application. All three fields are required. The "Main classname" field is where you enter the name of the class that contains `main`. This field represents the value of the property `com.apple.mrj.application.main`. Remember to include a full package name. For example, `com.myCompany.myMainClass`.

The Classpath field allows you to modify the classpath. This is automatically set to point to the JAR file that you're bundling into your application. If you want to use JAR or `.class` files that will not be included in the resulting application bundle, add a classpath entry of this form:

```
$APP_PACKAGE/../MyJARFile.jar
```

`$APP_PACKAGE` is a special path string that represents the application bundle directory. The last required field is the "Output file" field. This is where the resulting application bundle will be built. An optional setting in this pane is the application icon. Click the icon in the "Output file" section to bring up a file chooser dialog for selecting an `.icns` file.

Settings in Mac OS X, Java Properties, and Merge Files panes are optional. The Mac OS X pane allows you to set values specific to the Mac OS X application bundle format. If you do not specify `CFBundleExecutable` or `CFBundleName`, they are set based on the name of the output file you choose.

The Java Properties pane lets you set specific runtime properties, which can include standard Java properties as well as the specific Mac OS X system properties discussed in "Mac OS X Application Properties" (page 76). See "Setting Runtime Properties in MRJAppBuilder" (page 23) for more information on the Java Properties pane.

The Merge Files pane provides a means for adding files to the application bundle, such as `.zip` or JAR files. Each item added to the merge list is copied into the application's `Contents/Resources/Java` directory. Each item you add to the merge list gets automatically added to the classpath.

When you have finished making settings, click Build Application. MRJAppBuilder does not provide an import mechanism. If you build an application and want to change certain settings, you need to make a new application with MRJAppBuilder or modify the `Info.plist` and `MRJApp.properties` files by hand. Information on these files can be found in "Application Bundles" (page 20).

Applet Launcher

Applet Launcher (in `/Applications/Utilities/Java`) allows you to run applets without the overhead of launching a Web browser. It provides a graphical interface to Sun's Java Plug-in. If you want to test the performance of your applet with the `sun.applet.AppletViewer` class, you should use the `appletviewer` command-line tool (`/usr/bin/`). You can enter the path to an applet using its fully-qualified URL, and then press the Launch button. For example, entering the following URL launches the ArcTest applet:

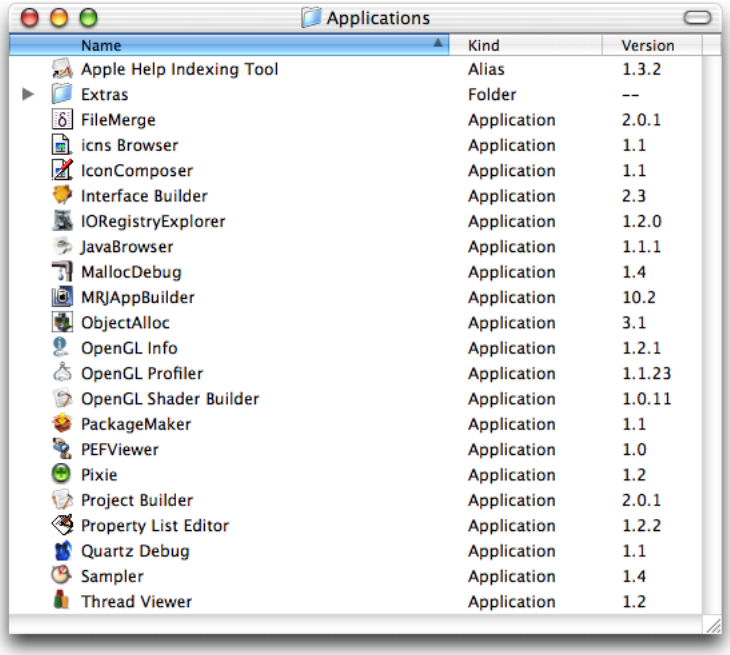
file:///Developer/Examples/Java/Applets/ArcTest/example1.html

Performance and behavior settings for applets may be adjusted in the Java Plugin Settings application installed in `/Applications/Utilities/Java`.

Other Tools

The Mac OS X Developer Tools also provide many tools useful for any kind of development, not just Java. For example Package Maker, File Merge, and Icon Composer are just a few examples. Figure 4-2 (page 34) shows all of the tools that get installed into `/Developer/Applications`.

Figure 4-2 Tools in `/Developer/Applications`



In addition to the Apple-supplied tools, since you have both a UNIX foundation and a full implementation of Java 2 Standard Edition, you may use many third party tools in Mac OS X. IDEs like Borland's JBuilder, Sun's ONE Studio, and Metrowerks' CodeWarrior are all available. There are many text editors, as well as other tools for specific functionality like `ant`, JUnit, and OptimizeIt. If you are interested in Java 2 Platform, Enterprise Edition (J2EE) development, Pramati and Jboss both offer J2EE development environments for Mac OS X. Some of the most important pieces of J2EE, like Enterprise Java Beans and Java Server Pages (JSP), are supported in Apple's WebObjects product (http://www.apple.com/webobjects). Zentek's iJADE product line is one solution for Java 2 Platform, Micro Edition (J2ME) development in Mac OS X.

Where to Get the Tools

The standard JDK tools are installed with a default user installation of Mac OS X. If your computer has Mac OS X installed but does not appear to have tools like `javac`, you can remedy this by making sure that the BSD packages are installed:

1. In the Welcome to Mac OS X folder of the Mac OS X Install Disk 1, double-click the Install Mac OS X icon.
2. In the window that opens, click the Restart button.
3. When your computer has restarted, follow the onscreen prompts until you get to the Installation Type phase. (This is indicated on the left side of the Install Mac OS X window.)
4. Select the Customize button.
5. Select BSD Subsytem.Though you may select other packages to add to your computer, for basic command-line Java development, you do not need any other packages. (The Base System and Essential System Software packages are always included.)

6. Proceed with the installation by following the onscreen prompts. Once you have restarted your computer, you should find the command-line Java tools installed in `/usr/bin`.

The Mac OS X Developer Tools CD comes with Mac OS X and is provided with new Macintosh computers. If you do not have a current copy of the Mac OS X Developer Tools, you may download them from the Apple Developer Connection Web site at http://connect.apple.com. Even if you do not want to use the Apple provided GUI-based tools, you should install the Mac OS X Developer Tools since they contains some important tools not installed by default.

Note: The Developer Tools available through the Apple Developer Connection online may be a more recent version than the Developer Tools CD that came with your copy of Mac OS X. It is advisable to check the website first. Compatibility information between releases is discussed in the release notes which are available online at http://developer.apple.com/releasenotes.

Cross-Platform Practices for Great Native Behavior

If you are developing Java applications that you plan to deploy on multiple platforms, there are some things that you should keep in mind to support Mac OS X as well as the other platforms. This chapter discusses some basic points to consider and small changes you can make to your Java code to help it run well on any platform, but are especially important to consider to make your application look like a Mac OS X application. In general, these changes do not require you to use any APIs or properties specific to Mac OS X. This chapter mainly discusses user interface issues.

The Aqua Look and Feel

Before discussing cross-platform coding practices, it is important to lay the foundation of what is available in Mac OS X. The default interface to the operating system is called Aqua. Although other interfaces can be displayed in Mac OS X, there is great benefit in usability by adhering closely to the native interface. To that end, Apple has worked hard to make your Java applications appear and behave as much like native applications as possible. Taking advantage of the basic look of the Aqua user interface is very simple. It requires you to do nothing other than write clean Java code. In Mac OS X, the default pluggable look and feel (PLAF) for Swing applications, `com.apple.mrj.swing.MacLookAndFeel`, gives an Aqua appearance to your Swing applications. So if you don't explicitly change your code to invoke another look and feel, Swing applications look like native applications by default.

In general it is good practice to avoid explicitly setting the PLAF in your Java code.This makes your application fit in much better on any platform. In the case of Mac OS X, this is especially true. If do you need to change the look and feel of your application to test it for different platforms, it is better to set the `swing.defaultlaf` Java property for your application at runtime by passing in `-Dswing.defaultlaf=`*yourLookAndFeel* when launching the application. For development purposes, you may want to temporarily change Mac OS X's default look and feel. You can do this by modifying the `swing.properties` file in `/Library/Java/Home/lib`.

You don't need to use AWT components for the native look of Aqua because Swing provides it. You can enjoy further benefits in performance and predictable graphics behavior by not mixing the heavyweight and lightweight components of AWT and Aqua.

Behavior is part of a user interface as well as appearance. Automatic adoption of the Aqua appearance is a first step, but there are many details that still are in your hands when trying to provide the best Aqua experience. *Inside Mac OS X: Aqua Human Interface Guidelines* is the definitive guide for how applications should appear and behave in Mac OS X and why they should behave that way. If you have a decision to make that is not specified anywhere else, that document is a great source of answers to user interface questions. If you find areas where your pure Java applications do not take advantage of these guidelines, please let Apple know.

There are numerous details that go into designing a first-class application on any platform. Although it may be impossible to build a perfect application, there are a few topics that are hot issues to users of a particular platform. As a Java developer, a few areas to keep in mind as you design are discussed in the following sections.

Placing and Painting Components

The complicated nestings and interrelations of containers and components in Swing interfaces can sometimes make it difficult to remember that the development platform is not always the deployment platform. The different look and feel designs available can provide widely variable appearances and behaviors. This section brings to the foreground some details that while helping your application to work well on any platform, are especially important in Mac OS X.

Layout Managers

Explicitly setting the x and y coordinates is dangerous when you consider the multiple platforms and look and feel designs that the application may run under. The results of a "well-designed" application in one environment may be disastrous in another, with components painting on top of each other and running off the edge of a container, among other things. It is generally unsafe to assume that placing buttons and controls at explicit coordinates is portable. Use the AWT layout managers to solve this problem. The layout managers use abstracted location constants and determine the exact placement of these controls for a specific environment. Layout managers take into account the sizes of each individual component while maintaining their placement relative to one another within the container.

Sizing Components

In general, do not set component sizes explicitly. Each look and feel has font styles and sizes. These font sizes will affect the required size of the component containing the text. Moving explicitly sized components to a new look and feel with a larger font size can cause problems. The safest means of keeping your components a proper, minimal size in a portable manner is to simply use `yourComponent.setSize(yourComponent.getPreferredSize())`.

Most layout managers and containers respect a components preferred size, usually making this call unnecessary. As your interface becomes more complicated however, you may find this call handy for containers with many child components.

Coloring Components

Because a given look and feel tends to have universal coloring and styling for most, if not all of its controls, developers may be tempted to create custom components that match the look and feel of standard user interface classes. This is perfectly legal, but adds maintenance and portability costs. It is easy to set an explicit color that you think works well with the current look and feel. Changing to a different look and feel though may surprise you with an occasional non-standard component. To ensure that your custom control matches standard components, query the `UIManager` class for the desired colors. An example of this is a custom Window object that contains some standard lightweight components but wants to paint its uncovered background to match that of the rest of the applications containers and windows. To do this, you can call

```
myPanel.setBackground(UIManager.getColor("window"))
```

This returns the color appropriate for the current look and feel. The other advantage of using such standard methods is that they provide more specialized backgrounds that are not easily reconstructed, such as the striped background used for Aqua containers and windows.

Windows and Dialogs

Window coordinates and insets are compatible with the JDK. In a nutshell: Window bounds refer to the outside of the window's frame; the coordinates of the window put (0,0) at the top left of the title bar (not at the top left of the content region as in MRJ on Mac OS 9). The `getInsets` method returns the amount by which content needs to be inset in the window to avoid the window border. This should affect only applications that are doing precision positioning of windows (especially full-screen windows), or those that bypass layout managers to do their own hard-coded component positioning.

Windows behave differently in Mac OS X than they do on other platforms. For example an application can be open without having any windows. Windows minimize to the Dock, and windows with variable content always have scroll bars. This section highlights the windows details you should be aware of and discusses how to deal with window behavior in Mac OS X.

Use of the Multiple Document Interface

The multiple document interface (MDI) model of the `javax.swing.JDesktopPane` class can provide a confusing user experience in Mac OS X. Windows minimized in a JDesktopPane move around as the JDesktopPane changes size. In the JDesktopPane, windows minimize to the bottom of the pane while independent windows minimize to the Dock. Furthermore, JDesktopPane restricts users from moving windows where they want. They are forced to deal with two different scopes of windows, those within the JDesktopPane and the JDesktopPane itself. Normally Mac OS X users interact with applications through numerous free-floating, independent windows and a single menu bar at the top of the screen. Users can intersperse these windows with other application windows (from the same application or other applications) anywhere they want in their view, which includes the entire desktop. Users are not constrained visually to one area of the screen when using a particular application. When building cross-platform applications with multiple windows, it is generally a good idea to avoid using `javax.swing.JDesktopPane`s.

There are times when there is not a simple way to solve window-related problems other than using a JDesktopPane. For example, you might have a Java application that requires a floating toolbar-like entity, referred to in Swing as an internal utility window", that needs to always remain in the foreground regardless of which document is active. Although Java currently has no means of providing this other than by using JDesktopPane, for new development you may want to consider designing a more platform-neutral user interface with a single dynamic container, similar to applications like JBuilder or LimeWire. If you are bringing an existing MDI-based application to the Macintosh from another platform and do not want to refactor the code, Mac OS X does support the MDI as specified in the J2SE 1.3.1 specification.

Windows With Scroll Bars (Using JScrollPanes)

In Mac OS X, scrollable document windows have a scroll bar regardless of whether or not there is enough content in the window to require scrolling. The scroller itself is present only when the size of the content exceeds the viewable area of the window. This prevents users from perceiving that the viewable area is changing size. By default, a Swing JFrame has no scroll bars, regardless of how it is resized. The easiest way to provide scrollable content in a frame is to place your frame's components inside a JScrollPane, which can then be added to the parent frame. In the default behavior of JScrollPane however, scrollbars only appear if they content in the pane exceeds the size of the window. If you are using a JScrollPane in your application, you can set the JScrollPane's scroll bar policy to always display the scroll bars, even when the content is smaller than the viewable size of the window. An example is shown in Listing 5-1 (page 40).

Listing 5-1 Setting JScrollBar policies to be more like Aqua

```
JScrollPane jsp = new JScrollPane();
jsp.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
jsp.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
```

With this setting the scroll bars are solid—with scrollers appearing when the contents are larger than the viewable area. You might want to do this conditionally based on the host platform since the default policy, `AS_NEEDED`, may more closely resemble other platforms native behavior.

File Choosing Dialogs

The `java.awt.FileDialog` and `javax.swing.JFileChooser` classes are the two main mechanisms to create quick and easy access to the file system for Java applications. Although each has its advantages, `java.awt.FileDialog` provides considerable benefit in making your application behave more like a native application. The difference between the two is especially evident in Mac OS X as Figure 5-1 (page 40) and Figure 5-2 (page 41) show.

Figure 5-1 `java.awt.FileDialog`

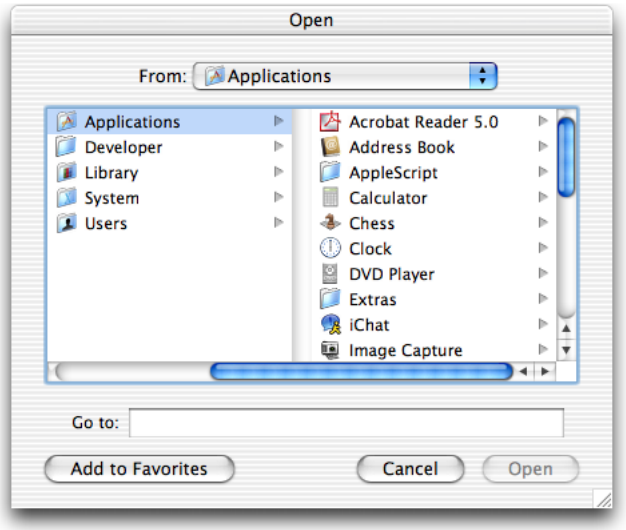
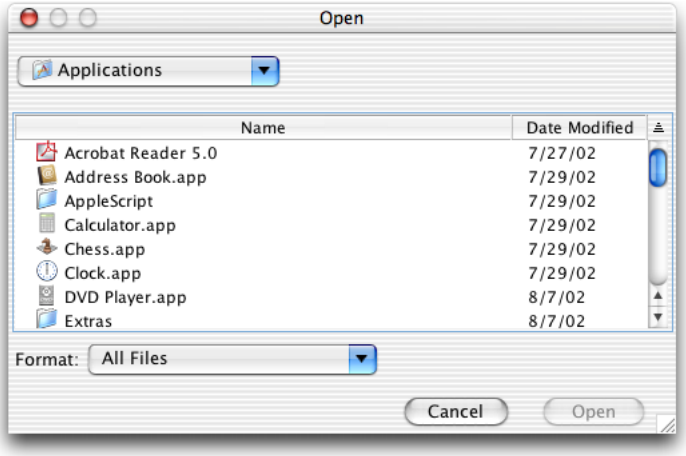


Figure 5-2 `javax.swing.jFileChooser`



The column-view style of browsing is adopted automatically by the AWT FileDialog while the Swing JFileChooser uses a navigation style different from that of native Mac OS X applications. Unless you need the functional advantages of JFileChooser you probably want to use `java.awt.FileDialog`. Since the FileDialog is modal and draws on top of other visible components, which is not the usual consequence of mixing Swing and AWT components.

Note: `javax.swing.JFileChooser` only supports traversal of Mac OS X file aliases in the default file view. If you replace it, aliases are not traversable. This limitation does not affect soft links, only Mac OS X aliases.

Dealing With Bundles in Mac OS X

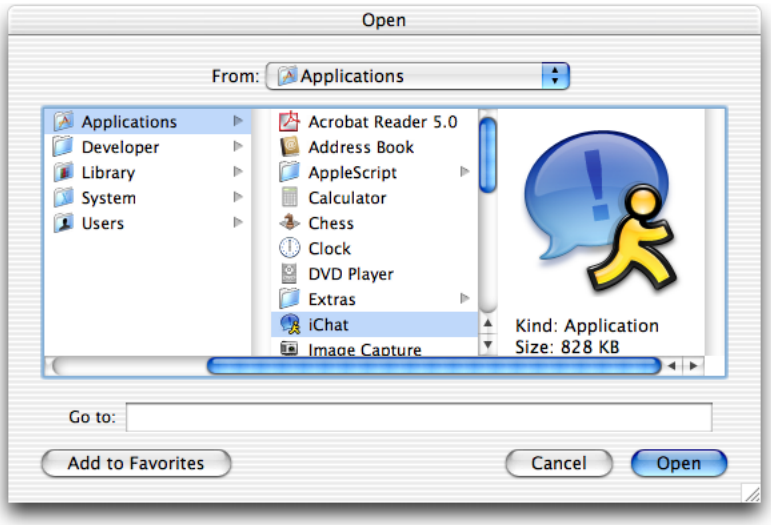
As described in "Application Bundles" (page 20), a native Mac OS X application is packaged as a bundle, which appears in the Finder as a single object but is actually a directory that contains all the resources of an application. Packages are also bundles that can be viewed as single objects or as directories. Java applications using JFileChooser or FileDialog recognize bundles as directories and allow inappropriate navigation. If you are building an application for developers, you might want to display the actual, on-disk contents of the directory, but if you are building an application for an end user, you do not want to display the bundle contents. Apple provides properties for both dialog classes that allow you the option of controlling how application and package bundles are displayed.

`java.awt.FileDialog` can be set to display application and package bundles as non navigable using the `com.apple.macos.use-file-dialog-packages` runtime system property. For example, to turn off navigation, run your application as follows:

```
java -Dcom.apple.macos.use-file-dialog-packages=true yourApplication
```

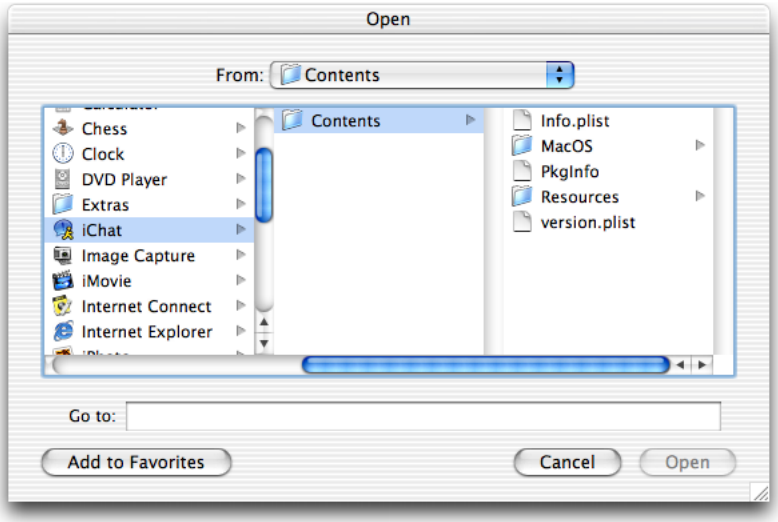
The resulting file dialog resembles Figure 5-3 (page 42). Notice that you cannot navigate into the application bundle.

Figure 5-3 Application displayed as an atomic object



The default setting of `com.apple.macos.use-file-dialog-packages`, `false`, leaves the contents of application and package bundles navigable by users as seen in Figure 5-4 (page 42).

Figure 5-4 Application displayed as a directory



You can set this behavior in your application bundle using the technique described in "Setting the Java Runtime Properties for an Application Bundle" (page 23). This allows you to alter your AWT dialogs for Mac OS X without any code change. If your application requires unique instances to behave differently, you can set the property to `true` or `false` at runtime as necessary using `System.setProperty`.

The `FileDialog.setFile` method does not work as expected in Mac OS X, so you can not specify which directory the dialog opens in.

`javax.swing.JFileChooser` offers more flexibility than `java.awt.FileDialog`. If you are already using it in your code, there are two properties that let you specify whether or not application and package bundles reveal their contents. Modifying JFileChooser's behavior does require you to modify your source code. The properties `JFileChooser.appBundleIsTraversable` and `JFileChooser.packageIsTraversable` take values of `never` or `always`. If you want to hide the contents of both types of bundles, set `JFileChooser.packageIsTraversable` to `never`. If you want to hide the contents of application bundles but show the contents of package bundles, set `JFileChooser.appBundleIsTraversable` to `never`. Generally, you should set `JFileChooser.packageIsTraversable` to `never`.

You can set these properties globally for all instances of JFileChooser in your application with the `UIManager.put` method. You can set them on a per-instance basis via the `putClientProperty` instance method inherited from `javax.swing.JComponent`.

Note: These properties have an effect only when using the Swing's Aqua look and feel in Mac OS X.

Menus

One difficulty in cross-platform Java user interface development is dealing with menus. The appearance of menu items tends to vary between platforms, as does how you handle meta keys described in menus. Unfortunately, many Java programmers write their applications with only the current development platform in mind and explicitly specify the appropriate modifier or trigger in their code. This poses problems in porting applications from platform to platform, as well as risking misinterpretation of what the platform's appropriate triggers are. Elegant and portable solutions to creating the appropriate actions on any given platform do exist and are covered in the following sections.

Menu Shortcuts

Keyboard shortcuts for invoking menu actions are often set with an explicit `javax.swing.KeyStroke` specification. This becomes complicated when moving to a new platform with a different modifier key because new KeyStrokes need to be conditionally created based on the current client platform. The solution to this problem is to use `java.awt.Toolkit.getMenuShortcutKeyMask` to ask the system for the appropriate key rather than defining it yourself.

When calling this method, the current platform's Toolkit implementation returns the proper mask for you. This single call checks for the current platform and then guesses which key is correct. In the case of adding a Copy item to a menu, this means that you can replace the complication of Listing 5-2 (page 43) with the simplicity of Listing 5-3 (page 44).

Listing 5-2 Explicitly setting KeyStrokes based on the host platform

```
JMenuItem jmi = new JMenuItem("Copy");
    String vers = System.getProperty("os.name").toLowerCase();
    if (s.indexOf("windows") != -1) {
      jmi.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_C, Event.CTRL_MASK));
    } else if (s.indexOf("mac") != -1) {
      jmi.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_C, Event.META_MASK));
    }
```

Listing 5-3 Using `getMenuShortcutKeyMask` to set meta keys

```
JMenuItem jmi = new JMenuItem("Copy");
jmi.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_C,
    Toolkit.getDefaultToolkit().getMenuShortcutKeyMask()));
```

Most common actions in Mac OS X have a keyboard equivalent that uses the Command key, previously known as the Apple key, as a modifier. There may be additional modifier keys like Shift, Option, or Control, but the Command key is the primary key that alerts an application that what follows is a command, not regular input. Not all platforms provided this consistency in user interface behavior.

Different keys may prompt an application to begin listening for commands. The code presented in Listing 5-3 (page 44) provides only one generalized mask, so you still may need to make conditional settings depending on your intended host platform. The fact that Mac OS X keyboard equivalents use Command makes your life a bit easier if you're bringing your application to Mac OS X. To make sure you are not overriding any of the keyboard commands Macintosh users have been accustomed to for over twenty years, see *Inside Mac OS X: Aqua Human Interface Guidelines* for the definitive list of the most common and reserved shortcuts.

Some platforms support menu item mnemonics or single-key shortcuts to menus and their contents using the Alt key. Mnemonics for these shortcuts are highlighted with a single underlined letter in the menu or item's name. This is a foreign concept to Mac OS X users. Although it is supported as a part of Java, the suggested way to handle the identification of keyboard shortcuts in Mac OS X is by clearly identifying all of the required keys. For an example, look at the File menu in the Finder. When writing your Java applications, it is suggested that you apply Swing mnemonics in a platform-sensitive manner in your code if possible, such as using a single `setMnemonics()` method that is conditionally called when constructing your interface.

Note: Since the `ALT_MASK` modifier evaluates to the Option key on the Mac, Control-Alt masks set for Windows become Command-Option masks if you use `getMenuShortcutKeyMask` in conjunction with `ALT_MASK`.

Menu Item Icons

Like mnemonics, menu item icons are also available and functional via Swing in Mac OS X. They are not a standard part of the Aqua interface, although some applications do display them—most notably the Finder in the Go menu. You may want to consider applying these icons conditionally based on platform. Whether or not you choose to display menu item icons in Mac OS X, you should be aware that Aqua does specify a specific set of special characters to be used in menus. See the information on using special characters in menus in *Inside Mac OS X: Aqua Human Interface Guidelines*

Contextual Menus

There is no problem supporting contextual menus in your Java applications on Mac OS X—they are fully supported. There are slight differences in terminology though. Java calls them popup menus while Aqua calls them contextual menus. More important is how they are triggered on different platforms. On Mac OS X, they are triggered by a Control-click. (By default, the second button of a two-button mouse maps to Control-click in Mac OS X.) In Windows, the right mouse button is the standard trigger for contextual menus.

These are two very different cases, which could result in fragmented and conditional code. One important aspect of both triggers is shared, the mouse click. To ensure that your program is interpreting the proper contextual menu trigger, it is again a good idea to ask the AWT to do the interpreting for you with `java.awt.event.MouseEvent.isPopupTrigger`.

The method is defined in `java.awt.event.MouseEvent` because you need to activate the contextual menu through a `java.awt.event.MouseListener` on a given component when a mouse event on that component is detected. The important thing to determine is how and when to detect the proper event. In Mac OS X, the pop up trigger is set on `MOUSE_PRESSED`. In Windows it is set on `MOUSE_RELEASED`. For portability, both cases should be considered as shown in Listing 5-4 (page 45).

Listing 5-4 Using `isPopupTrigger` to detect contextual menu activation

```
JLabel label = new JLabel("I have a pop up menu!");

label.addMouseListener(new MouseAdapter(){
    public void mousePressed(MouseEvent e) {
       evaluatePopup(e);
    }

    public void mouseReleased(MouseEvent e) {
       evaluatePopup(e);
    }

    private void evaluatePopup(MouseEvent e) {
       if (e.isPopupTrigger()) {
          // show the pop up menu...
       }
    }
});
```

Event Handling

Two notes on event handling may be useful to your in your development and deployment of applications in Mac OS X:

- Many of the Swing and AWT components are implemented with native code from the Carbon API of Mac OS X. Although this should not affect your Java code, it might be important information in debugging applications that do not appear to behave correctly, especially in regard to the runtime handling of events. If you do find an issue where the event handling is not as specified by the Java 2 specification, please file a bug as explained in "Filing and Tracking Bugs" (page 10).
- Although mouse-down events from additional buttons on multi button USB mice are delivered correctly, mouse-up and mouse-drag events involving the additional buttons may not be delivered with the correct modifiers. There may be ambiguity between various button masks and meta key masks. Use the utility functions as a workaround (for example, `java.awt.event.InputEvent.isMetaDown`) instead of accessing the modifiers directly.

Using Native Features of Mac OS X in Java Applications

If you are targeting a Java application for Mac OS X deployment, this chapter offers a collection of things you might need to know to help your application perform the best it can in Mac OS X.

Modifying the Default Settings for Hardware Graphics Acceleration

When your Swing applications start up, the Java implementation in Mac OS X determines which video card is installed in your system and compares this against a list of known unsupported video cards. If your card does not appear in, or is commented out of this list, then hardware graphics acceleration is invoked. If you do nothing at all, your Java applications take advantage of hardware acceleration if that computer has a compatible video card. You may find in testing your application in Mac OS X that you do not want hardware acceleration on or you might find certain video cards that you wish to turn off hardware acceleration for. If these cards are not ones that Apple automatically turns off hardware acceleration for, you need to explicitly turn them off with the `com.apple.hwaccelexclude` property using the naming conventions described in "Video Cards Designation Strings" (page 48). For example, to turn off hardware acceleration on just the ATI Radeon 8500 but leave it on for all other cards, you could use the following command in Terminal:

```
java -Dcom.apple.hwaccel=true -Dcom.apple.hwaccelexclude=ATIRadeon8500_67108864
 -jar App.jar
```

By using the `com.apple.hwaccelexclude` property, you have complete control of which cards are ignored. If you choose to turn off hardware acceleration for any specific cards, make sure that you also turn it off for those that are uncommented by default in the `hwexcludelist.properties` file as well. Otherwise you will turn it off for the video card you are concerned with, but it will be on with cards that Apple initially had it turned off for.

Taking this into account, a better example of turning off hardware acceleration for the ATI Radeon 8500 would also turn it off for the 8 MB ATI Rage 128 card in the first generation iBooks and Titanium PowerBooks:

```
java -Dcom.apple.hwaccel=true
-Dcom.apple.hwaccelexclude=ATIRadeon8500_67108864,ATIRage128_8388608 -jar App.jar
```

The `com.apple.hwaccel=true` is not necessary in Mac OS X version 10.2, but it is suggested that you use it along with the -Dcom.apple.hwaccelexclude flag.

To turn off hardware acceleration completely, pass in the command-line flag `com.apple.hwaccel=false`. For example:

```
java -Dcom.apple.hwaccel=false -jar App.jar
```

To turn on hardware acceleration for all video cards, pass in the command-line flag `com.apple.hwaccelexclude` with no arguments as follows:

```
java -Dcom.apple.hwaccel=true -Dcom.apple.hwaccelexclude= -jar App.jar
```

You may incorporate any of these command-line options into your Mac OS X applications bundles using MRJAppBuilder or Project Builder.

Advanced Options

While testing your Java applications, you may want to see what effect Mac OS X's Java hardware graphics acceleration has on your code without having to restart your application each time you make a change. By passing in the system property `com.apple.usedebugkeys=true` and you can use function keys combinations to turn hardware acceleration on and off. There are two sets of key combinations you can use:

- Command-F7 to turn on hardware acceleration gracefully. This is equivalent to the default behavior except that you may toggle it on and off while the application is running. Option-F7 turns it off.
- Command-F8 forces hardware acceleration on. This differs from Command-F7 in that it forces everything to be hardware acceleration without applying any of the logic that Command-F7 would use. This is for testing only, and you may see unpredictable results. Option-F8 turns it off.

Note: Once you have forced hardware acceleration on with Command-F8, you cannot use Command-F7 to turn it on without restarting your application.

The `com.apple.forcehwaccel` runtime property allows you to force hardware acceleration on or off without looking at the list of suggested video cards. This is similar to using Command-F8 with `com.apple.usedebugkeys` as noted above except that with `com.apple.forcehwaccel`, the state is explicitly determined at runtime. Available values are `true` and `false`.

Another runtime property, `com.apple.hwaccelnogrowbox`, is provided so that you can determine if hardware acceleration is on or off just by looking at a window. By setting this to `true`, the grow box (called a resize control in Aqua) normally found in the bottom right corner is not displayed if hardware acceleration is on. It will be visible if hardware acceleration is not on. Setting this to `false`, gives you the default behavior that shows the grow box regardless of whether hardware acceleration is on or off.

Video Cards Designation Strings

Java hardware graphics acceleration is supported on most of the default video cards in Apple computers, provided they have 16 MB or more of on-card video memory. For the purpose of Java graphics hardware acceleration, each of these cards has been assigned a distinct identifying string. There are a few ways of determining these strings:

- If you are unsure of the type of video card in a computer, use the `hwaccel_info_tool` available online at http://connect.apple.com. Instructions on using this command-line tool are provided in a man page installed with the tool.
- If you know what type of video card is in your computer, you can determine the appropriate string by looking in the `hwaccelexclude.properties` file in `/System/Library/Frameworks/JavaVM.framework/Versions/1.3.1/Home/lib` or by usingTable 6-1 (page 49).

Table 6-1 Video-card designation strings

Video card model	Memory	String
ATI Rage Mobility 128	8 MB	`ATIRage128_8388608`
ATI Rage 128	16 MB	`ATIRage128_16777216`
ATI Radeon	16 MB	`ATIRadeon_16777216`
ATI Rage 128	32 MB	`ATIRage128_33554432`
ATI Radeon 7500	32 MB	`ATIRadeon_33554432`
ATI Radeon 8500	32 MB	`ATIRadeon8500_67108864`
NVidia GeForce2	32 MB	`NVidia11_33554432`
NVidia GeForce3	64 MB	`NVidia20_67108864`
NVidia GeForce4MX	64 MB	`NVidia11_67108864`
NVidia GeForce4TNT	64 MB	`NVidia20_134217728`

Specifying a Name and Icon for Command-Line Applications

By default, Java processes that are launched from a shell do not show up in the Dock or the menu bar until they show a window. If they never show a window, they never appear in the Dock. This allows server-type processes to run behind the scenes with no user-visible manifestation.

If you are running an application from the command line that opens a window, the class name of the application's main class appears as the application name in the Dock and application menu. You may specify a more appropriate name to display by using this command-line option:

`-Xdock:name=`*`applicationName`*

You can also specify an icon to replace the generic Java icon with this option:

`-Xdock:icon=`*`pathToIconFile`*

The icon file must be a Mac OS X icon file—the same type you would use for any other Mac OS X application. If either the application name or path to the icon file has spaces in it, wrap it in double quotes.

Using the Macintosh Menu Bar

In Swing, an application's menu bar is applied on a per-frame (window) basis. Similar to the Windows model, the menu bar appears directly under the frame's title bar. This is different from the Macintosh model, where the application has a single menu bar that controls all of the application's windows. To solve the basic problem, a runtime property is provided:

```
com.apple.macos.useScreenMenuBar
```

This property can have a value of `true` or `false`. If undefined, the standard Java behavior equivalent to a value of `false` is used. When read by the Java runtime at application startup, a given JFrame's JMenuBar is placed at the top of the screen, where a Macintosh user would expect it to be. Since this is a simple runtime property that must be used by the host VM, setting it in your application has no effect on other platforms since they do not even check for it.

Note that a JMenuBar attached to a JDialog does not appear at the top of the screen as expected when setting this property, but rather inside the dialog as if the property were not set. If your JDialog does not have a menu bar, the default parent JFrame's menu bar is displayed when the JDialog is brought into focus.

If you find the need to attach menus to a dialog window, you may want to consider making the window a JFrame instead of a JDialog. A dialog should be informational or present the user with a simple decision, not provide complex choices. A window with enough functionality to necessitate a menu bar may be better as a JFrame.

Unfortunately, this solves only part of the problem—the placement of the menu bar. The fundamental discrepancy between the idea of a single menu bar per application of the Macintosh and a menu bar per window of Java, still exists. In other words, setting this property causes the menu to appear at the top of the screen, but only when the specific window it was assigned to is in focus. If your application has multiple windows, and a window other than the one holding the menu bar is focused, the menu bar vanishes. The Aqua guidelines state that the menu bar should always be visible in an application; even an insignificant window such as an alert dialog should still show the menu bar (though you may want to disable the menus).

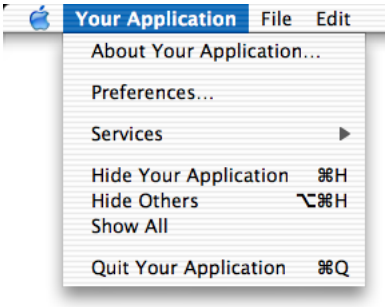
The Window Menu

One of the suggestions in *Inside Mac OS X: Aqua Human Interface Guidelines* is that all Mac OS X applications should provide a Window menu to keep track of all currently open windows. A Window menu should contain a list of currently active (visible) windows, with the corresponding menu item checked if a given window is currently in the foreground. Likewise, selection of a given Window menu item should result in the corresponding window being brought to the front. New windows should be added to the menu, and closed windows should be removed. The ordering of the menu items is typically the order in which the windows appeared. *Inside Mac OS X: Aqua Human Interface Guidelines* has more specific guidance on the Window menu.

The Application Menu

Any Java application that uses AWT/Swing, or is packaged in a double-clickable application bundle, is automatically launched with an application menu similar to native applications on Mac OS X. This application menu, by default, contains the full name of the main class as the title. This name can be changed using the `com.apple.mrj.application.apple.menu.about.name` application property or the `-Xdock:name` command-line property. According to the Aqua guidelines, the name you specify for the application menu should be no longer than 16 characters. Figure 6-1 (page 51) shows an application menu.

Figure 6-1 Application menu for a Java application in Mac OS X



The next step to customizing your application menu is to have your own handling code called when an item in the application menu is chosen. Apple has provided a means for this through interfaces in the `com.apple.mrj` package. Each interface has a special callback method that is called when the appropriate application menu item is chosen. The following callback interfaces for the application menu are available:

- com.apple.MRJAboutHandler, to handle the About menu item.
- com.apple.MRJPrefsHandler, to handle the Preferences menu item.
- com.apple.MRJQuitHandler, to trigger final clean-up logic when the Quit menu item is chosen. By default, that Quit menu item just calls `System.exit`. You might actually want to do more than just exit.

To handle a given application menu item:

1. Implement the appropriate handler interface.
2. Define the appropriate `handler` method in your implementation: `handleAbout`, `handlePrefs`, or `handleQuit`.
3. Register your handler using the appropriate static methods in the `com.apple.mrj.MRJApplicationUtils` class: either `registerAboutHandler`, `registerPrefsHandler`, or `registerQuitHandler`.

You can see examples of these implementations in a default Swing application project in Project Builder.

If your application is to be deployed on other platforms, where Preferences, Quit, and About are accessed elsewhere on the menu bar (in a File or Edit menu, for example), you may want to make this placement conditional based on the operating system of the host platform. This is preferable to just adding a second instance of each of these menu items for Mac OS X. This minor modification can go a long way to making your Java application feel more like a native application on Mac OS X.

More MRJ Handlers

In addition to the interfaces provided for handling application menu items, Java on Mac OS X provides two other handlers:

- com.apple.MRJOpenApplicationHandler, which lets you respond to an Open Application Apple event.

- com.apple.MRJOpenDocumentHandler, that responds to double-clicking a supported document or the dragging of a document onto your application's icon.

These handler interfaces are intended to enhance a Java application's behavior in the Mac OS X Finder and are used in the same manner as the application menu interfaces described above.

Localizing Packaged Java applications on Mac OS X

To run correctly in non-English locales, Java applications bundles need to have the appropriate localized folders inside the application package. This is true even if the Java application handles its localization through Java `ResourceBundles`. Specifically, you need to have a folder named with the locale name and the `.lproj` suffix present in the application's `Resources` folder for any locale that you wish to use. For example, you need a `Japanese.lproj` folder inside `YourApplication.pkg/Contents/Resources/` in order for Japanese localization to work correctly. The folder itself can be empty, but it must be present for Mac OS X to set the locale correctly when the application launches. Otherwise Mac OS X launches your application with the English US locale.

The Bundle Services documentation and *Inside Mac OS X: System Overview*, both available from http://developer.apple.com/documentation, provide more details about the application bundle format.

QuickTime for Java

QuickTime for Java provides a cross-platform API that allows Java developers to build multimedia components, including streaming audio and video, into applications and applets for both Macintosh and Windows. More information on QuickTime for Java is available online at http://developer.apple.com/quicktime/qtjava/.

Java Core Audio Packages

Mac OS X includes a very robust sound subsystem. Hooks for using this system directly from Java are provided in the Java Core Audio packages. More information on the Mac OS X Core Audio framework can be found online at http://developer.apple.com/audio.

Java Spelling and Speech Frameworks

Two very useful frameworks, Spelling and Speech, are also available for you to use in Mac OS X. These are implemented as Java Beans. The source code, examples, and documentation are available at http://developer.apple.com/java.

JDirect

In addition to the standard Java Native Interface, Mac OS X also includes JDirect3. JDirect3 allows access to preexisting native code libraries from Java without you needing to explicitly use JNI. It builds JNI stubs for you on the fly.

JDirect3 is not a standard part of J2SE, and it is important to keep in mind that it ties your code to Mac OS X. JDirect3 is not recommended for future new development since changes in future releases of Mac OS X will yield your code unusable.

JDirect1, which has been deprecated since 1997, is not supported in Mac OS X. If you have JDirect2 code, it requires minor modifications to run under JDirect3. If you are porting old JDirect2 code to JDirect3, there are a few important details to keep in mind as the following sections discuss.

Human Interface Toolbox Synchronization

You must synchronize all Human Interface Toolbox calls. Since Java threads are native Mach threads and can preempt Toolbox calls made by the host application or by other Java threads otherwise. The Java implementation itself is based on the Carbon threading model in Mac OS X. This threading model is neither reentrant or thread-safe. Since reentrant calls can corrupt memory or crash your application, it is important that only one thread be inside Carbon at any time.

This is the correct way to call the Toolbox from Java on Mac OS X:

```
import com.apple.mrj.macos.carbon.CarbonLock;
try {
    CarbonLock.acquire();

//Carbon call here

} finally {
    CarbonLock.release();
}
```

This way the Carbon lock handling can easily be reversed for JDirect callbacks.

Be sure to use `CarbonLock` around all Carbon calls. Since threads are preemptive on Mac OS X, you should hold the Carbon lock for the shortest amount of time possible.

Make sure not to do anything that might throw an exception in the `finally` block that calls `CarbonLock.release`. Structure your code so that the Carbon locking is separate from other finally blocks.

Debugging Features for JDirect

To help debug JDirect on Mac OS X, you can define a shell variable, `JDIRECT_VERBOSE` to write verbose JDirect loading info to `stderr`. This works only when launching Java code from the command line, not when double-clicking a bundled application in the Finder. You can launch any double-clickable application from the command line by specifying the path to the application's executable.

For example, to set the shell variable, and then launch an application called Foo.app, you would do the following in `csh`:

```
setenv JDIRECT_VERBOSE

/path/Foo.app/Contents/MacOS/Foo
```

MethodClosureUPP Not Supported

The class `MethodClosureUPP` was created for use in Mac OS 9 and is not supported in Mac OS X. If you are using Carbon callbacks and want code that can run in Mac OS 9 and Mac OS X, you need to have a helper class that creates a new `MethodClosureUPP` when in Mac OS 9 or a new `MethodClosure` when in Mac OS X.

JDirect Access to Bundles

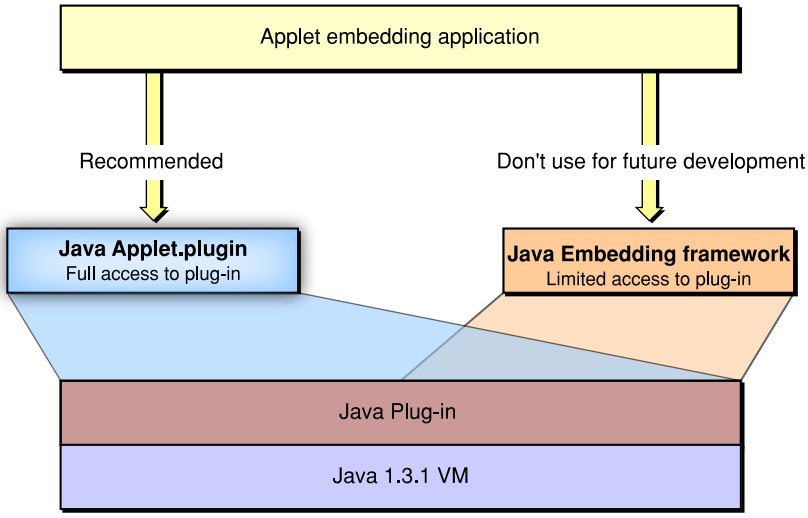
Normally, your `JDirect_MacOSX` string is a full path to a dylib, which is the standard packaging for shared Mach-O binaries in Mac OS X. You can also specify a full path to a bundle. The bundle path name must end in `.bundle` and be a properly constructed directory. A bundle can use both Mach-O binaries and CFM binaries. The Bundle Services documentation in *Inside Mac OS X: System Overview* has more details.

Embedding Applets in Native Applications

If you are a Mac OS X developer that needs to embed Java applets inside of your native application, use the Java Applet Plug-in (`Java Applet.plugin`) architecture of Mac OS X. In previous versions of Mac OS X, you might have used Java Embedding framework. Though this framework still works, it is not recommended for future development. It may change significantly in future versions of Java on Mac OS X.

As illustrated in Figure 6-2 (page 55), using the Java Applet Plug-in architecture will allow applets displayed in you application to take full advantage of Sun's Java Plug-in; the Java Embedding framework provides only a subset of that functionality. The Java Applet Plug-in therefore provides both developmental benefits for developers as well as an enhanced user experience for applets displayed in your application.

Figure 6-2 Java Applet Plug-in versus Java Embedding Framework



The Java Applet Plug-in architecture allows your native code access to Sun's Java Plug-in which in turn lets you access a virtual machine. Since it is a standard Netscape 4.0 style plug-in, you may also use it to provide Java support inside native applications. `Java Applet.plugin` is a native Mach-O library. It is a standard Mac OS X bundle. It exports the same symbols that the UNIX Netscape plug-in exports. Complete details on integrating a Netscape-style plug-in is beyond the scope of this document. .

Cocoa Java

In Mac OS X, Cocoa applications can be written in Java as well as their native Objective-C. Cocoa Java applications will behave just like a native applications but will not be portable to other platforms.

If you are using Cocoa Java for your user interface, it is important that you do not try to mix Cocoa Java with standard Java components. Since many of these components are Carbon-based, you will run into many problems. In particular, you should avoid trying to mix the following with Cocoa Java:

- Swing/AWT components, including Java events
- QuickTime for Java components
- JDirect calls

The main reason for this is that Carbon and Cocoa use different run loops, which conflict with one another if mixed in the same application.

More information on Cocoa Java is available at http://developer.apple.com/documentation/cocoa.

Project Builder Tutorial

Apple provides a complete set of development tools free with Mac OS X. This set includes compilers, a debugger, a complete integrated development environment (IDE), as well as many other tools and resources. Apple's IDE, Project Builder, allows you to edit, compile, debug, and package your Java applications. This chapter provides a simple tutorial on using Project Builder to build a Swing-based Java application and an applet.

Building a Java Application With Project Builder

To follow this example, you need to have the Mac OS X Developer Tools installed. Project Builder is installed in `/Developer/Applications`. The sourcecode files you need are `ExampleFileFilter.java`, `ExampleFileView.java`, and `FileChooserDemo.java`; they are installed in `/Developer/Examples/Java/JFC/FileChooserDemo`. If you do not see these files you need, install the Mac OS X Developer Tools. If you do a custom install, make sure that the Developer Example and Developer Tools Software packages are installed. For this example you will build Sun's FileChooserDemo that lets you see how different Java look and feel designs display file browsers.

The Makeup of a Project Builder Project

Open Project Builder. From the Project Builder File menu choose New Project.

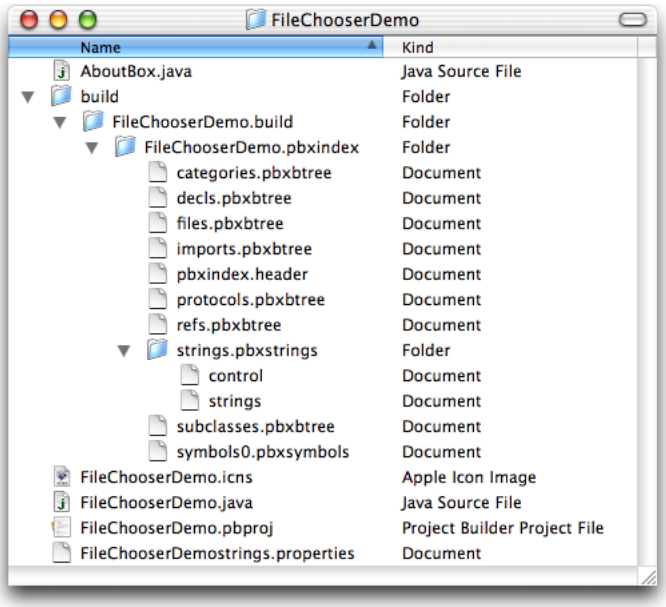
You see that you can use Project Builder to create AppleScript, Carbon, Cocoa, and Java applications as well as bundles, tools, frameworks, kernel extensions, and plug-ins. In the Java section, there is support for creating several types of Java projects including tools, applets, and applications. Select Java Swing Application by double-clicking it.

Enter `FileChooserDemo` in the Project Name field and place the project in a location that is convenient for you by either entering a pathname or by clicking the Choose button and navigating using the browser. In either case, the Location field displays the location of your project.

Click Finish. Project Builder generates the appropriate files and places them inside the directory that you specified.

A look at this project's directory in the Finder reveals the files that Project Builder creates when you make a new project as shown in Figure A-1 (page 58).

Figure A-1 Project folder contents



The build directory contains the results of building your project. The file `FileChoserDemo.icns` contains the icons for your application. This file is the icon set that Project Builder assigns by default to Java applications if you do not specify another `.icns` file to use. `FileChooserDemo.pbproj` is the Project Builder project file. It contains generated meta data about your project, and Project Builder maintains this file. `FileChooserDemostring.properties` maintains a list of localizable strings. There are also two Java source files.

To further explore the interactions of these parts of the FileChooserDemo project, make Project Builder active again.

In the Files list you see the same basic layout as in the Finder. With such a simple project, it might appear that the Files list is just a mirror of the file-system layout. It is important to note that this is not the case. Project Builder keeps track of the files in your project by means of references. It does not have to actually copy the file to the project folder. This allows you to set up your project in the Files list in a manner that streamlines development without having to be concerned with how the files are structured on disk. You could, for example, have source code in a directory different from that of your Project Builder project.

Building a Java Project

Before making any changes to your new project, verify that it builds a valid Java application. Choose Build and Run from the Build menu. You can also click the Build and Run button (the one with the hammer and computer monitor) in the toolbar or simply press Command-R. Your application should build and then launch a simple Hello World application. It's window should look like the one shown in Figure A-2 (page 59).

Figure A-2 Result of building a default project

Notice the generic Java icon in the Dock for this application.

Figure A-3 Generic Java icon

Also notice that the application, though a pure Java application, takes on the Aqua look and feel of a Mac OS X application, right down to the genie effect when you minimize the window.

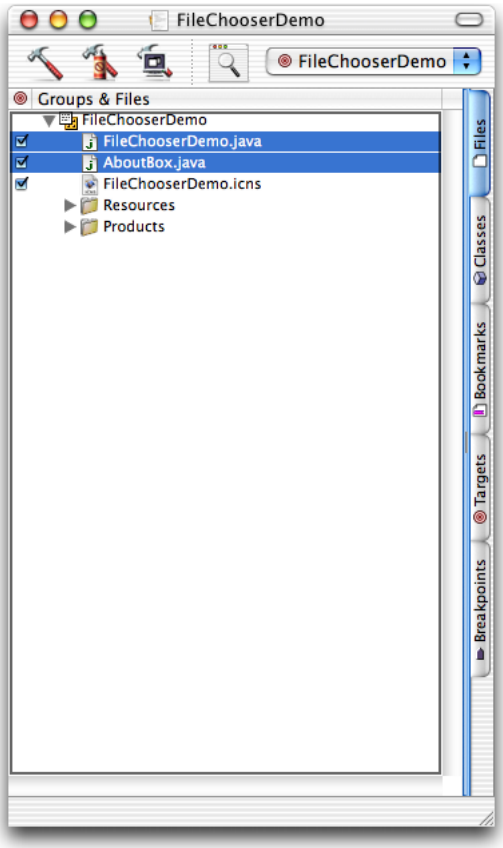
Project Builder has not only compiled the `.java` source files, but also wrapped the resulting Java application as a Mac OS X application. In the Finder you can see the double-clickable Mac OS X application inside your project folder in the `Build` directory. In building the Mac OS X application, Project Builder has set certain system properties. For example, by default Project Builder passes the flag `-Dcom.apple.macosx.useScreenMenuBar=true` to your application when it runs. This puts the menu bar at the top of the screen where it is in native Macintosh applications.

Adding Your Source Files

Building a default application is nice, but how do you get your code to build in Project Builder? So far you have only built and run the default files provided in the Java Swing application template. Now you are going to add modify the template to get a feel for how you can modify the default Project Builder templates with your own Java source code.

If the FileChooserDemo application is still running, quit it. You can do this either by clicking the Stop button in the Project Builder toolbar or by pressing Command-Q in the application itself. Before adding the correct source files, you need to remove the two source files that Project Builder gave you by default. In the Files list, you see `FileChooserDemo.java` and `AboutBox.java` as indicated in Figure A-4 (page 60).

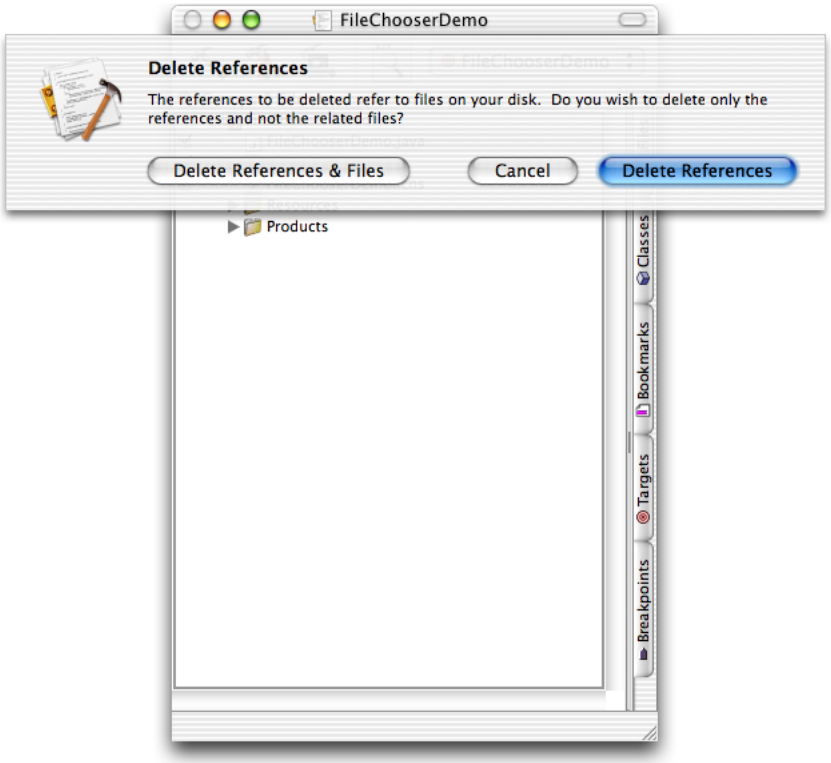
Figure A-4 Default files in a new Java Swing application



`FileChooserDemo.java` contains the `main` method invoked when the application runs. You will replace this file with the `FileChooserDemo.java` source file in `/Developer/Examples/Java`. The use of MRJAboutBox and MRJQuitHandler in `FileChooserDemo.java` allows your Java application to more closely resemble a native Mac OS X application. Their use, however, is beyond the scope of this example. There is another file, `AboutBox.java`, which contains the AboutBox class that is used in conjunction with the MRJAboutHandler.

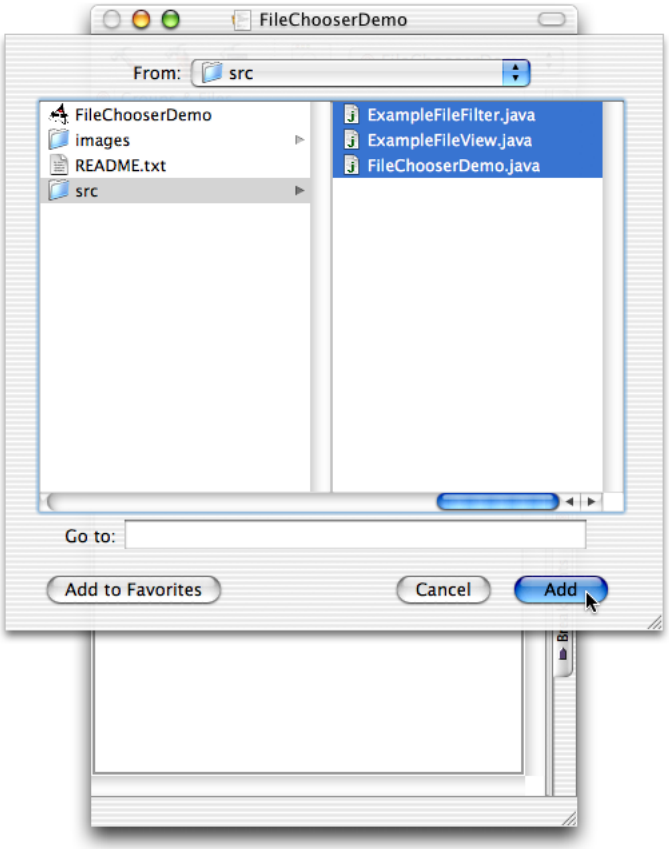
To get rid of the old `FileChooserDemo.java` file, select it in the Files list and choose Delete from Project Builder's Edit menu. In the alert that appears, click Delete References & Files. This removes the reference to the file in Project Builder and deletes the file from your project directory. Delete `AboutBox.java` in the same manner.

Figure A-5 Delete References alert



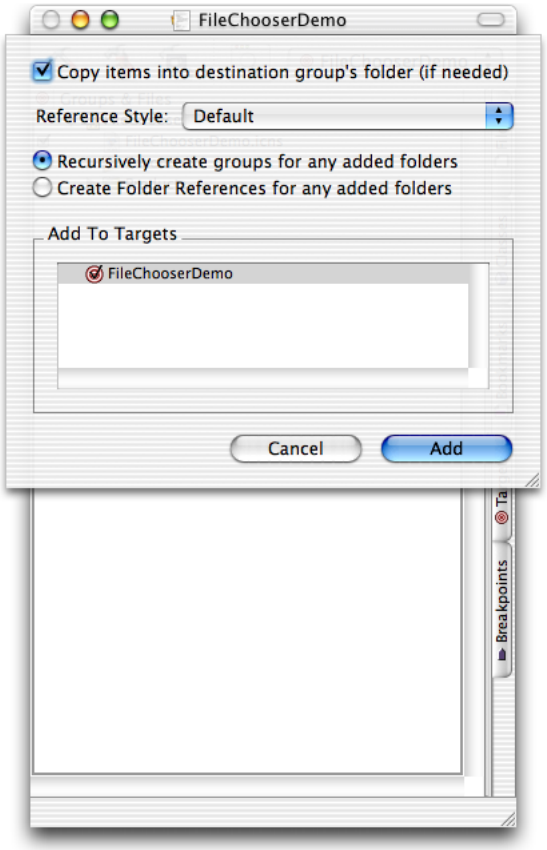
You now have a Java project with no source files. From the Project menu, choose Add Files. Navigate to `/Developer/Examples/Java/JFC/FileChooserDemo/src` either in the browser or by typing the path in the "Go to" field. Select `ExampleFileFilter.java`, `ExampleFileView.java`, and `FileChooserDemo.java`. Click Add.

Figure A-6 Selecting files to add to a Java project



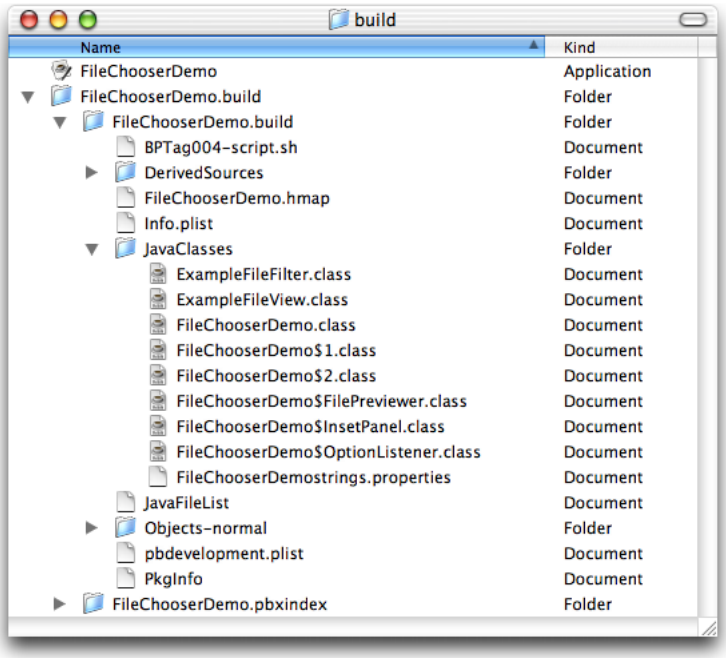
In the next dialog, you need to decide whether to copy the files themselves into the project folder or just make references to them. Do you want a local version that you can alter without affecting other projects or do you want to maintain a single copy of that source file? In this case the former is appropriate so select the option "Copy items into destination group's folder." Leave the Reference Style pup-up menu set to Default. Since you are not adding any folders, your choice for the folder reference radio buttons does not matter. Click Add.

Figure A-7 Copy items into project folder option



Once you have the appropriate files in your project, you can compile the Java code and build the Mac OS X application. Since you removed some files and added others since your last build, it is a good idea to clean the active target before building. If you don't clean the active target, you might get build errors. From the Build menu select Clean. Dismiss the resulting warning by clicking Clean Active Target. Now you are ready to build. Click the Build button (with the hammer icon) in the Project Builder toolbar or choose Build from the Build menu. Look at the results of this process by opening your project folder in the Finder.

Figure A-8 Contents of a built application



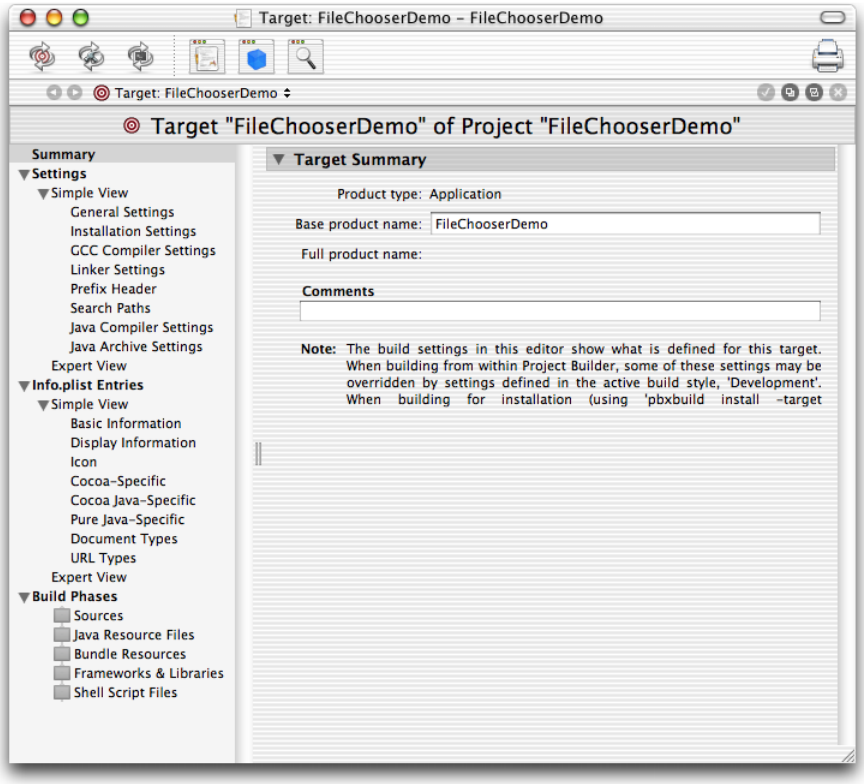
At the top level of the project directory are the source files and the `FileChooserDemo.icns` file that were there before. The `build` directory now contains the class files and the Mac OS X double-clickable application, FileChooserDemo. You can see how everything was built by looking at the messages in Project Builder's Build pane. When Project Builder builds a Java application, the source code is compiled, archived as a JAR file, and then converted to a Mac OS X application.

Modifying the Application Parameters

Although Project Builder did compile the Java source files and used `jar` to archive them together, it also put together the Mac OS X application bundle. In doing this it evaluates certain settings in Project Builder to construct the property lists that Java on Mac OS X uses at runtime. (see "Property List Attributes for Java Applications" (page 21)). For example, Project Builder determined the name FileChooserDemo to display in the title bar and determined the main class to run when your application is double-clicked in the Finder. Some properties don't get set automatically though and some retain default values unless you specify otherwise. For example if you launch the FileChooserDemo application, you can see that the default Java icon is still in the Dock. This icon is added to your application bundle if you don't add your own icon.

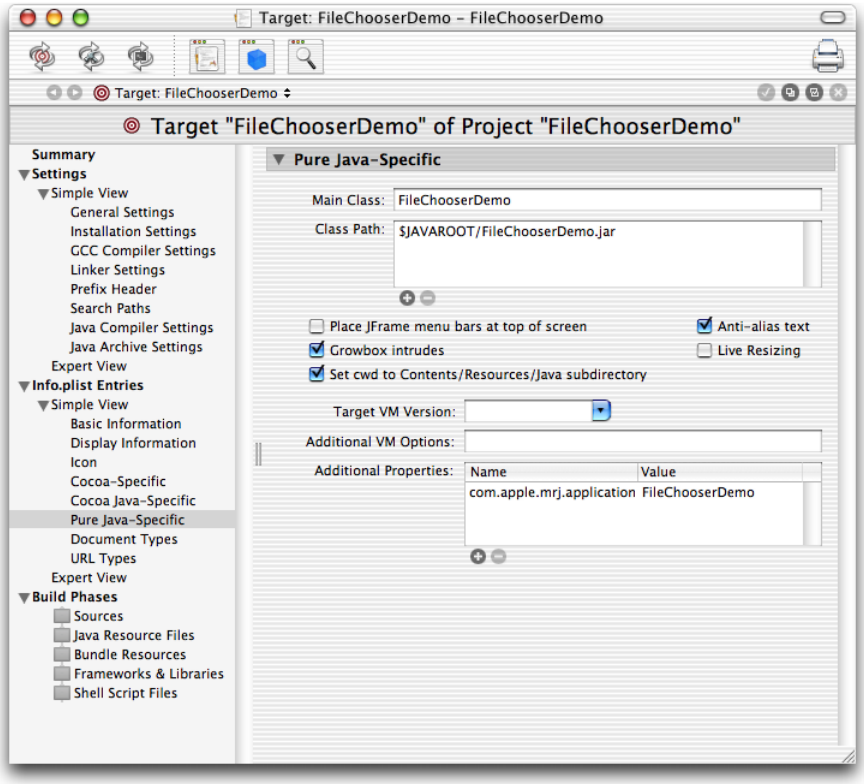
It is simple to modify the generic Java runtime properties of your application or to add specific Mac OS X properties from Project Builder. In your FileChooserDemo Project window, click the Targets tab. Select the FileChooserDemo target in the top left window. It should be the only target there. (If you are using Project Builder's multi window user interface, double click the target.) In the resulting window or pane you can modify many different aspects of that target as Figure A-9 (page 65) shows.

Figure A-9 Target settings



There are many settings here that you can modify, but the ones you will probably modify the most include those in Pure Java-Specific. To see how you can modify these, select the Pure Java-Specific section. Project Builder should display a window that looks like the one in Figure A-10 (page 66).

Figure A-10 Pure Java-Specific settings



This pane shows the default settings. To see the effect of changing settings here, add a command-line setting to the Additional Properties list, then clean and rebuild the application. For example:

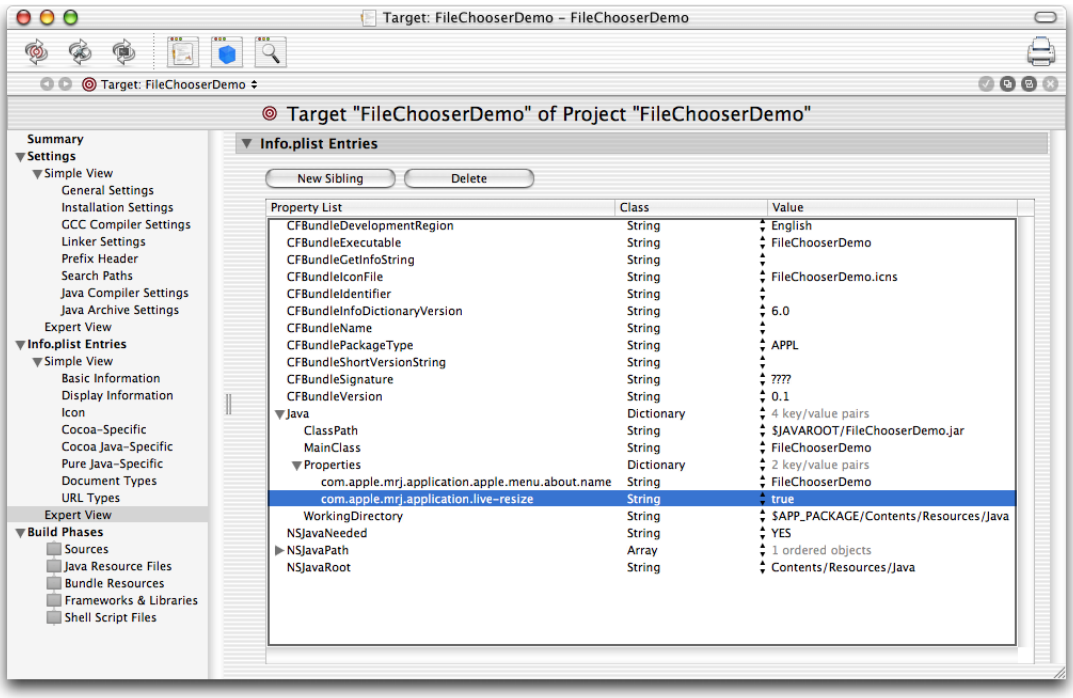
1. Click the plus (+) button under Additional Properties.
2. Replace the highlighted `new`.property with `com.apple.mrj.application.live-resize`.
3. Replace `value` with `true`.
4. Choose Clean from the Build menu. In the sheet select Clean Active Target.
5. From the Build menu, select Build and Run.

You can see in the resulting application that if you resize the window it now attempts to fill in the contents as the window is resizing, instead of just resizing an outline of the window. You can also see why this system property is off by default! (For an application with a simpler window, turning on this system property might be beneficial.)

You can add any properties in the Pure Java Specific section that you might pass in at the command line. Those that you pass in with the `-D` flag should go in the Additional Properties section. Those that you pass in with `-X` should go into the Additional VM Options section.

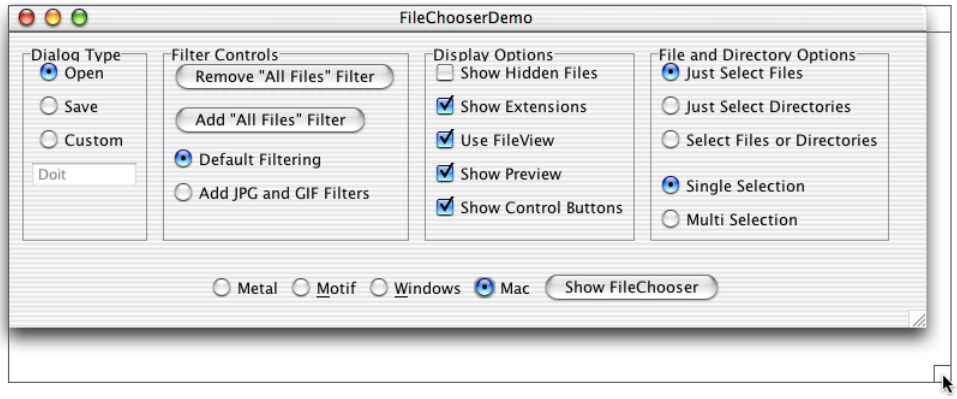
Having tried an option that didn't make your application better, you should remove it. In removing it though, try using the Expert View indicated in Figure A-11 (page 67). Just select the offending property, and delete it.

Figure A-11 Expert View



Now if you clean, build, and run the application, you should see that the original behavior has returned. This is seen in Figure A-12. Whether modifying the settings in the Expert View or Simple View, you are really just modifying the property list entries as discussed in "Property List Attributes for Java Applications" (page 21).

Figure A-12 Default setting for live resizing



With the basic steps presented here, you have the foundation to build your own Java applications in Project Builder and take advantage of the value it adds by allowing users to run your pure Java application just like they would any other Mac OS X application.

Building Applets With Project Builder

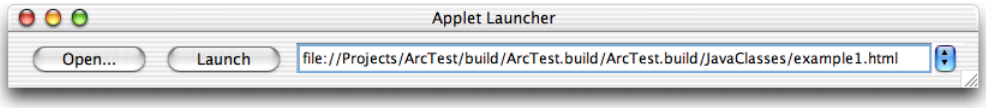
Building applets with Project Builder is very similar to building applications. Since applets are hosted within another application, there is no need to convert them to Mac OS X applications like you did with the Pure Java application. Having built an application in Project Builder, building your applet should be very straightforward. As with the application, the simplest way to start is with one of Project Builder's templates. By adding your own files and removing the default files, you can quickly get your own applets built in Project Builder.

Build one of the applets in `/Developer/Examples/Java/Applets`. For example:

1. Open a new Project Builder project. Use the Java AWT Applet template. Name it `ArcTest`.
2. Delete the `ArcTest.java` and `example1.html` files.
3. Add in `ArcTest.java` and `example1.html` from `/Developer/Examples/Java/Applets/ArcTest` like you did in "Adding Your Source Files" (page 59).
4. Clean and Build the applet.

Once you have built your applet, you can test it by loading the HTML file into a browser or into the Applet Launcher application found in `/Applications/Utilities/Java`. As illustrated in Figure A-13 (page 68), just put in the path to the HTML file that Project Builder generated. This file is in your project folder in `build/ArdTest.build/ArcTest.build/JavaClasses`.

Figure A-13 Applet Launcher



By following this short tutorial, you should now see that Project Builder gives you a simple-to-use development environment for pure Java development, while also helping you to make your Java applications fit seamlessly into Mac OS X.

MRJAppBuilder Tutorial

Although command-line Java applications are great for development, when you want to distribute your application, you want the user to be able to launch it just like any other Mac OS X application—without a trip to the command line. MRJAppBuilder allows you to take your existing Java `.class` or `.jar` files and wrap them into a Mac OS X application bundle. This chapter provides a simple tutorial that you can work through to help you understand this process.

To follow this example, you need to have the Mac OS X Developer Tools installed. MRJAppBuilder is installed in `/Developer/Applications`. The sourcecode files you need are `ExampleFileFilter.java`, `ExampleFileView.java`, and `FileChooserDemo.java`; they are installed in `/Developer/Examples/Java/JFC`/FileChooserDemo/src. If you do not see these files, install the Mac OS X Developer Tools. If you do a custom install, make sure that the Developer Example and Developer Tools Software packages are installed. This example will take a JAR file and show you how to bundle it as a Mac OS X application.

Building a Basic Application

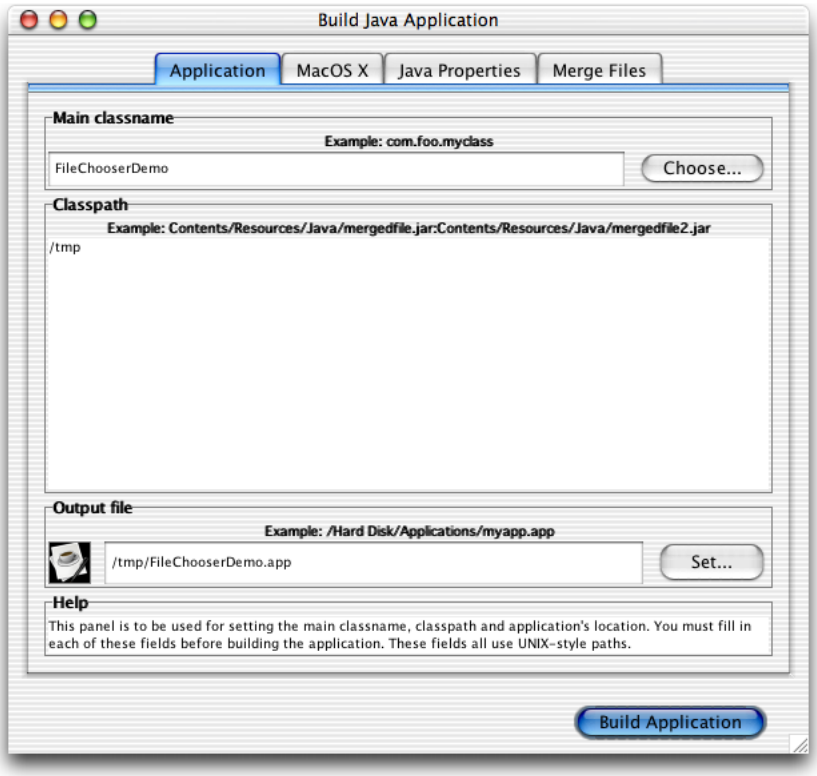
MRJAppBuilder works with either stand-alone class files or class files in a JAR file. Since most of the source code in `/Developer/Examples/Java` is not compiled, the first step is to compile a selection to obtain the class files.

In this example, you will use the FileChooserDemo application. From Terminal, compile the three Java files in `/Developer/Examples/Java/JFC/FileChooserDemo/src` with `javac -d /tmp /Developer/Examples/Java/JFC/FileChooserDemo/src/*.java`. This gives you the required class files.

Once you have the class files, open the MRJAppBuilder application in `/Developer/Applications`. When the application opens, you are presented with the Application pane. The required fields for building a valid application are all present in this pane.

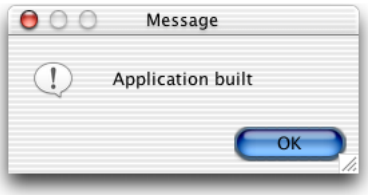
In this example set the main classname to FileChooserDemo. This is the class that contains the `main` method. Set the classpath to `/tmp`. In this example, there are only class files in the classpath. It might be appropriate in a more complicated application for the classpath to include image, font, or sound files as well. Set the output file to whatever location is convenient to use for testing your application. This field should include the fully qualified intended location and name of the resultant application, for example `/tmp/FileChooserDemo.app`. Be sure to append the `.app` suffix to the name you choose for this application. The result should be similar to Figure B-1 (page 70).

Figure B-1 MRJAppBuilder Application pane



With these three fields filled in, you are ready to build the application. Click the Build Application button. You are informed that your build was successful.

Figure B-2 A successful build



Navigating in the Finder to the directory you specified in the "Output file" field reveals a double-clickable application. (Hint: In Finder, choose Go to Folder from the Go menu.)

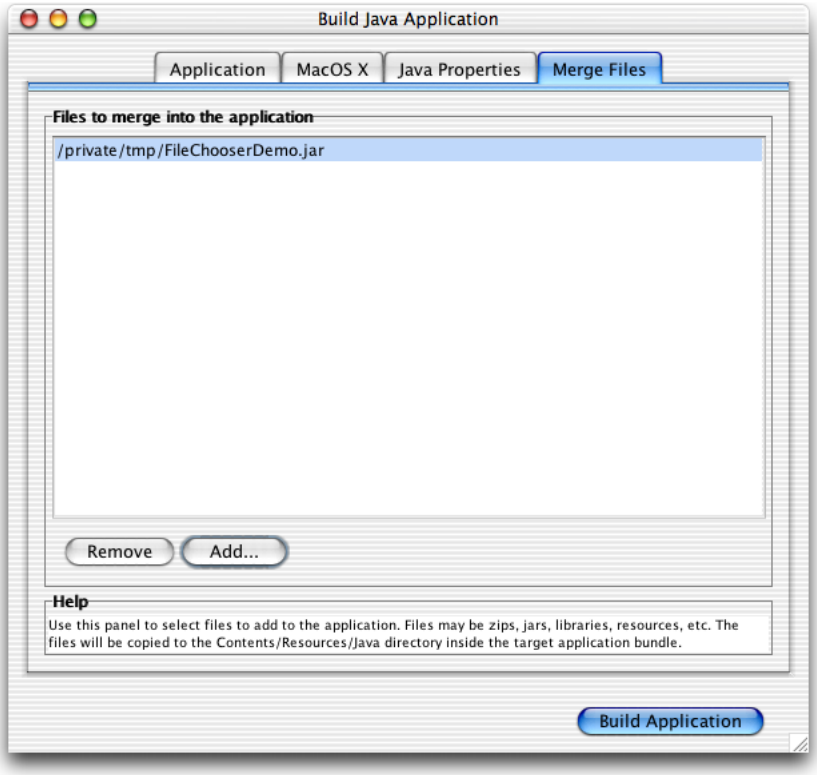
Building a More Robust Application

The example in "Building a Basic Application" (page 69) worked fine, but notice that you set the path to a specific directory, `/tmp`. Unless your users happen to have the appropriate class files installed in `/tmp`, this application won't work on their computers. How do you get around this? That leads to another pane in MRJAppBuilder. Before using it though you need to prepare the appropriate files. In this example, navigate to your `/tmp` directory and make a JAR file from the class files you put there as follows:

```
jar cf FileChooserDemo.jar *.class
```

You should still have MRJAppBuilder open. If not go ahead and open it and set it up the way it was before. Click the tab labeled Merge Files. It gives you the option to add files. Click the Add button and choose the JAR file you just made. The result should look like Figure B-3 (page 71).

Figure B-3 Merge Files pane



The JAR file is copied into the `/Contents/Resources/Java` directory of the resulting application bundle. Now go back to the Application pane. Notice that the path to this JAR file was automatically added to the `/tmp` path you had there before. (You can get rid of that reference to `/tmp` now if you haven't already.) You now have a Mac OS X application that a user can install by simply dropping in the Finder without having to deal with installing anything else or being concerned with where the application is installed.

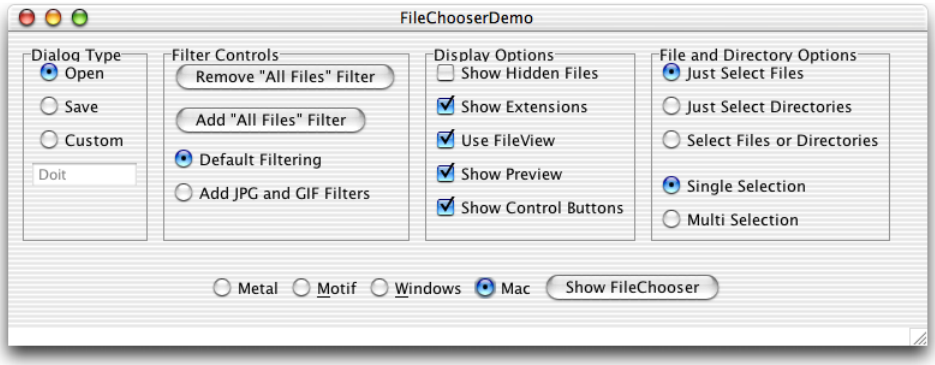
Making Your Application More Mac-like

So far you have built a basic application that has some features that Mac users expect. For example they can install it with a simple drag and drop. If you launch that application, you will notice something missing. The icon in the Dock is a generic Java icon. You can fix that easily enough in MRJAppBuilder. For the sake of this example, just copy an icon from another application, in this case the icon from the prebuilt version of the FileChooserDemo application. To do this, select the generic icon in the Application pane. It is near the bottom of the pane, in the "Output file" section. If you click it, it opens a file chooser. Navigate to the file named `JavaApp.icns` in `/Developer/Examples/Java/JFC/FileChooserDemo/FileChooserDemo.app/Contents/Resources` and click Select. (You will probably need to change the Format pop-up menu to All Files from the default Icon Files to navigate there.) Now if you build the application, you should see that it displays the new icon in the Dock.

Java Properties Pane

An icon is only the first step. There are a few other things you can do to make a well-written Java application hard to distinguish from a native Mac OS X application. MRJAppBuilder provides an interface for making a lot of these changes in the Java Properties pane. Before going there, launch the FileChooserDemo application that you just built. Notice that the window has a white bar at the bottom of the window as shown in Figure B-4 (page 72).

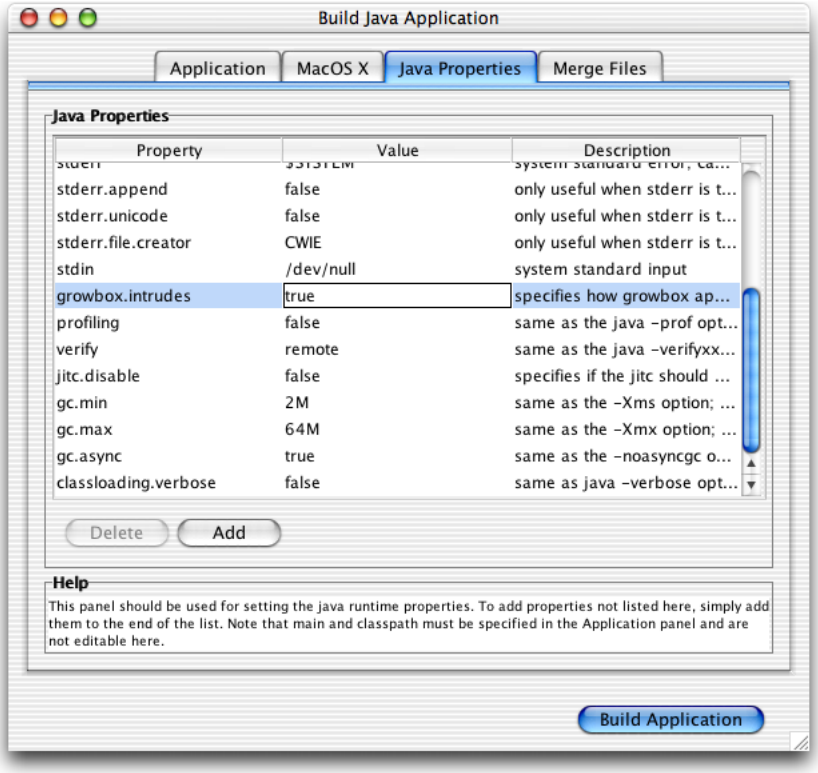
Figure B-4 FileChooserDemo application with relics



This is because by default, MRJAppBuilder sets certain system properties. In this case, `com.apple.mrj.application.growbox.intrudes` is set to `false`. The result is that an extra 15 pixels are added to the window. You can correct this in the Java Properties pane.

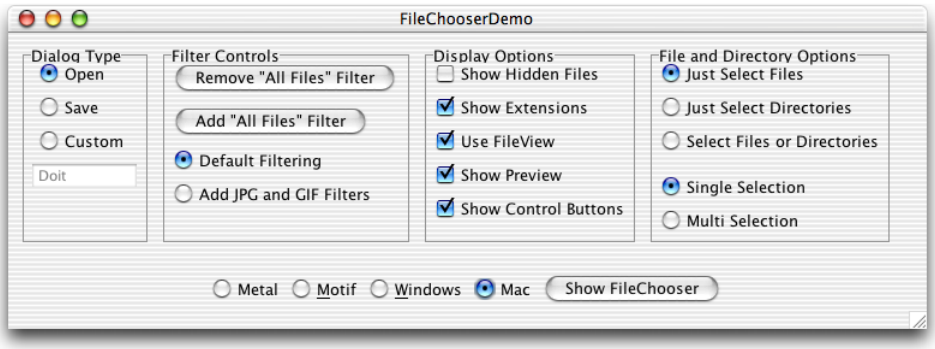
Click the Java Properties tab. This pane contains properties that are passed to the application when it is run. Of special note here is the Parameters property. This is where you can specify any command-line parameters that need to be passed to your application when it is run. To see the effect of modifying the Java properties, change the value of the `growbox.intrudes` property from `false` to `true` as in Figure B-5 (page 73).

Figure B-5 Modifying the `growbox.intrudes` property



Quit the previously built FileChooserDemo if it is still running, and build and run the new version. Notice that the relics no longer surround the window when you switch between the different interface styles.

Figure B-6 `com.apple.mrj.application.growbox.intrudes=true`



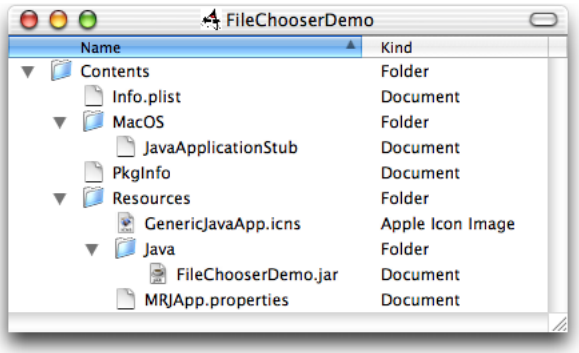
You can change the value of the other properties by clicking the appropriate value field and making the desired change. See Appendix C, "Mac OS X Java System Properties," (page 75) for a listing and description of some Mac OS X system properties.

Mac OS X Pane

The Mac OS X pane allows you to set attributes of the Mac OS X application bundle. Features like the application icon and name can be set here. For information on these keys and how to use them, see *Mac OS X Technology Overview* . These are not explored in this tutorial.

In this tutorial, you have seen how to wrap your JAR files into a native Mac OS X application, and you have seen how to modify the parameters of that application to build an application that fits in with the native applications in Mac OS X. There was no magic going on behind the scenes. MRJAppBuilder is a very simple application that builds an application bundle directory structure, determines runtime options to be passed to the Java virtual machine when the application is invoked, and sets some Mac OS X application properties. To get a glimpse inside an application in Mac OS X, you can either explore the directory of the `.app` from the Terminal or from the Finder. To see what is contained inside an application bundle generated by MRJAppBuilder, Control-click an MRJAppBuilder-generated application. You could use the FileChooserDemo application or even MRJAppBuilder itself. Inside is a structure similar to Figure B-7 (page 74)

Figure B-7 Application bundle contents



Once you have built an application bundle with MRJAppBuilder, you might want to fine tune the settings in the `Info.plist` or the `MRJApp.properties` files by hand. Any text editor will do and you won't need to set up all of the fields in MRJAppBuilder each time you want to make changes.

Mac OS X Java System Properties

This appendix lists the Mac OS X–specific system properties that you should be aware of.

Java Virtual Machine Properties

You can use `System.getProperties().list(System.out)` to obtain a complete list of system properties. The most useful ones are included here, as well as brief descriptions.

The properties in Table C-1 (page 75) all return a value of type `String`. They can be used in this manner:

```
if (System.getProperty("os.version").equals("10.2"))
 System.out.println("Mac OS X version 10.2 Jaguar");
```

Table C-1 JVM properties

Property	Value	Notes
`java.version`	1.3.1	
`java.vm.version`	1.3.1_03-69	
`line.separator`	'\n'	This differs from MRJ in Mac OS 9, which used `\r`, but is consistent with UNIX-based Java implementations and with the BSD and Cocoa frameworks.
`mrj.version`	3.3	Testing for the existence of this property is the general way to detect whether you are using an Apple-developed JVM. To detect specific features like JDirect or MRJToolkit, it is better to use reflection to check for the existence of the specific classes they use.
`os.name`	Mac OS X	This value was `Darwin` in developmental versions of Mac OS X. The MRJ on Mac OS 9 used `Mac OS`. It is suggested that you use `System.getProperty("os.name").startsWith("Mac OS")` if you need to test whether your code is running on any version of Mac OS.
`os.version`	10.2	This is the version number available in the About This Mac dialog. Developmental versions of Java on Mac OS X used the Darwin kernel number (1.1, 1.2, and so forth)

Mac OS X Application Properties

It is simple to wrap your pure Java application inside of a natively executable Mac OS X application bundle. This is discussed in "Mac OS X Java Applications" (page 19). In so doing, there are two properties that must be defined. These are listed in Table C-2 (page 76). If you are using Project Builder or MRJAppBuilder, you do not need to do anything to define them since both of those applications take care of this for you.

Table C-2 required Mac OS X application properties

Property	`Info.plist` key	Description
`com.apple.mrj.application.main`	`MainClass`	The fully qualified class name of the class containing the application's `main` method.
`com.apple.mrj.application.classpath`	`ClassPath`	The paths of all required directories or JAR files.

In addition, you can pass in the options listed in Table C-3 (page 76).

Table C-3 Application launch properties

Property	Info.plist Key	Description
`com.apple.mrj.application.parameters`	`Arguments`	Space-separated list of arguments that are parsed to build the `String[]` passed to `main`.
`com.apple.mrj.application.workingdirectory`	`WorkingDirectory`	Sets the current working directory for the application. By default the current working directory is set to the parent directory of the application bundle. The `APP_PACKAGE` variable may be used to refer to the root of the application bundle.
`com.apple.mrj.application.vm.options`	`VMOptions`	Space-separated list of options for the Java virtual machine. These are the `-X` and `-XX` options without the `-X(X)` prefix.

Note the following about these properties:

- None of these properties are set by default.
- If you are adding properties to an `Info.plist`, they should go into the Java dictionary.
- When adding properties to an `Info.plist`, do not use the fully qualified package name. Instead use the `Info.plist` key value designated in the appropriate table.
- In setting the values for any of these keys, you may take advantage of two variables unique to application bundles:

  `APP_PACKAGE`
  The root directory of the application bundle.

  `JAVAROOT`
  This is not set by default. If you do set it in the `Info.plist` file, you may use `$JAVAROOT` to resolve its value.

Note: These variables are used only by Mac OS X for launching your application. Do not use them in your Java code.

Mac OS X–Specific Properties

The properties listed in Table C-4 (page 77) are all properties that are found only on Mac OS X. Overall they help your applications to fit in more cleanly with the rest of the Mac OS X environment. They can be set three ways:

- In the `Info.plist` or `MRJApp.properties` files through Project Builder or MRJAppBuilder respectively. You can modify these files later with a text editor.
- In your `main` class as arguments to `System.setProperty`.
- From the command-line using the `-D` flag as follows:

```
java -Dcom.apple.macosx.useScreenMenuBar=trueYourSwingApp
```

Table C-4 System properties related to the graphical user interface

Property	Default Value	Notes
`com.apple.hwaccel`	`true`	Turns on hardware graphics acceleration for the not commented out of `/Library/Java/Home/lib/hwexclude.prop` set to `false`, hardware acceleration is turned of used in conjunction with the `com.apple.hwacc` property.
`com.apple.hwexclude`	`none`	When specific video card designation strings are with this property, hardware graphics accelerati turned on for the respective video cards. It is be other video cards. When this property is set, `/Library/Java/Home/lib/hwexclude.prop` ignored.

Property	Default Value	Notes
com.apple.macos.use-file-dialog-packages	false	When set to true, causes java.awt.FileDial application packages as if they were files and doc navigation into them.
com.apple.macos.useScreenMenuBar	false	Puts Swing menus in the Mac OS X menu bar if Aqua look and feel. Java applications created wi Builder have this set to true. Note that JMenuB JDialogs are not moved to the Mac OS X menu b
com.apple.macos.useSmallTabs	none	If defined, and set to true, tab controls in Swing a more closely resemble the Metal look and feel. I false, the tabs assume a larger size more simila default Aqua controls.
com.apple.macosx.AntiAliasedTextOn	true	Use anti-aliasing when rendering text.
com.apple.macosx.AntiAliasedGraphicsOn	true	Use anti-aliasing when rendering graphics.
com.apple.mrj.application.apple.menu.about.name	None	If defined, an About command is added to the t application menu and can be detected by regist com.apple.mrj.AboutHandler. Java applicati with Project Builder have this set to initial name project.
com.apple.mrj.application.growbox.intrudes	true	Resizable window's growbox (resize control) intr AWT content. If turned off, the bottom of the wi pushed down 15 pixels.
com.apple.mrj.application.live-resize	false	Enables live resizing of windows.

Document Revision History

This table describes the changes to *Java 1.3.1 Development for Mac OS X*.

Date	Notes
2003-04-01	Changed the title from *Java Development on Mac OS X: Java 2 Platform, Standard Edition Version 1.3.1* to *Java 1.3.1 Development for Mac OS X*.
2002-09-01	Format completely revised. Changed target emphasis from Mac OS 9 Java developers to Java developers coming from other platforms. Updated to include new features introduced in Java 1.3.1, including the `Java.applet.plugin` and information about hardware acceleration.
2002-07-01	Updated for Mac OS X version 10.2. Modified tutorials to work with new operating system and corrected some typographical errors.
2001-12-01	Document originally released with a focus on describing what is different in Java development from Mac OS 9 to Mac OS X.

Document Revision History

Glossary

application bundle The executable code and related resources as they are packaged into a prescribed directory hierarchy.

Apple menu A menu that provides items that are available to users at all times, regardless of which application is active. It is the leftmost menu in the menu bar.

application menu A menu that contains items that apply to the application as a whole, rather than to a specific document or other window. The application menu is immediately to the right of the Apple menu.

Apple Developer Connection The primary source for technical and business resources and information for anyone developing for Apple's software and hardware platforms anywhere in the world. It includes programs, products, and services and a website filled with up-to-date technical documentation for existing and emerging Apple technologies. The Apple Developer Connection is at http://www.apple.com/developer/.

AppleScript A scripting language used to control the actions of the computer and the applications that run on it.

Aqua The graphical user interface for Mac OS X.

bundle A directory in the file system that stores executable code and the software resources related to that code. Applications, plug-ins, and frameworks are types of bundles. Except for frameworks, bundles are file packages, presented by the Finder as a single file.

BSD Berkeley Software Distribution. Formerly known as the Berkeley version of UNIX, BSD is now simply called the BSD operating system. The BSD portion of Mac OS X is based on 4 FreeBSD 4.4, a "flavor" of 4.4 BSD.

Carbon An application environment for Mac OS X that features a set of programming interfaces derived from earlier versions of the Mac OS. The Carbon API has been modified to work properly with Mac OS X, especially with the foundation of the operating system, the kernel environment. Carbon applications can run in Mac OS X, Mac OS 9, and all versions of Mac OS 8 later than Mac OS 8.1.

Cocoa An advanced object-oriented development platform for Mac OS X. Cocoa is a set of frameworks with programming interfaces in both Java and Objective-C. It is based on the integration of OPENSTEP, Apple technologies, and Java.

Classic An application environment for Mac OS X that lets you run non-Carbon legacy Mac OS software. It supports programs built for both Power PC and 68K chip architectures and is fully integrated with the Finder and the other application environments.

Darwin Another name for the Mac OS X core operating system. The Darwin kernel is equivalent to the Mac OS X kernel plus the BSD libraries and commands essential to the BSD Commands environment. Darwin is Open Source technology.

dmg A Mac OS X disk image file.

dock An area on the edge of the screen that holds applications, documents, minimized windows, folders, storage devices, and links to websites. It is customizable by users to allow them to easily organize and quickly access their most used resources.

Finder The system application that acts as the primary interface for file-system interaction.

HFS (Hierarchical File System) The Mac OS Standard file-system format, used to represent a collection of files as a hierarchy of directories (folders), each of which may contain either files or folders themselves. HFS is a two-fork volume format.

HFS+ The Mac OS Extended file-system format. This file-system format was introduced as part of Mac OS 8.1, adding support for filenames longer than 31 characters, Unicode representation of file and directory names, and efficient operation on very large disks. HFS+ is a multiple-fork volume format.

JDirect A Mac OS–specific technology that allows you to access native code without building Java Native Interface libraries.

Mach The lowest level of the Mac OS X kernel. Mach provides such basic services and abstractions as threads, tasks, ports, interprocess communication (IPC), scheduling, physical and virtual address space management, virtual memory, and timers.

Mach-O The executable format of Mach object files. This is the default executable format in Mac OS X.

MRJAppBuilder An application used to bundle cross platform Java applications into native Mac OS X Java applications. It is included with the Mac OS X Developer Tools in `/Developer/Applications`.

`.pkg` **file** A Mac OS X Installer file. May be grouped together into a meta package (`.mpkg`).

Objective-C An object-oriented programming language based on standard C and a runtime system that implements the dynamic functions of the language. Objective-C's few extensions to the C language are mostly based on Smalltalk, one of the first object-oriented programming languages. Objective-C is available in the Cocoa application environment.

OpenGL An industry-wide standard for developing portable 3D graphics applications.

plist See property list (page 82).

Project Builder Apple's graphical integrated development environment. It is available free with the Mac OS X Developer Tools package.

property list A structured, textual representation of data that uses the Extensible Markup Language (XML) as the structuring medium. Elements of a property list represent data of certain types, such as arrays, dictionaries, and strings.

Quartz Quartz is a powerful graphics system that delivers a rich imaging model, on-the-fly rendering, anti-aliasing, and compositing of PostScript graphics. Quartz also implements the windowing system for Mac OS X and provides low-level services such as event handling and cursor management. It also offers facilities for rendering and printing that use PDF as an internal model for graphics representation.

QuickTime QuickTime is a powerful, cross-platform, multimedia technology for manipulating, enhancing, and storing video, sound, animation, graphics, text, music, and even 360-degree virtual reality. It also allows you to stream digital video where the data stream can be either live or stored.

window menu A menu that contains commands for managing document windows. The menu lists an application's open document windows, including minimized windows, in the order in which they were opened.

XNU The Mac OS X kernel. It combines functionality of Mach and BSD as well as the driver model, the I/O Kit.