Java Development Guide for Mac OS X

2006-05-23

Contents

Figures, Tables, and Listings

Introduction to Java Development Guide for Mac OS X

Java 2 Platform, Standard Edition (J2SE) for Mac OS X offers a Java environment featuring a high level of integration with Mac OS X. This integration brings together the Java platform's versatility and Mac OS X's advanced technologies to offer users a wider selection of applications and developers a first-class development and deployment platform.

Mac OS X v.10.4 includes Java 1.4.2 right out of the box and offers J2SE 5.0 as a Software Update. Combined, these Java distributions open up the entire Mac user base to Java application and applet developers, and conversely, the world of Java applications to Mac OS X users.

While Java's promise of "write once, run anywhere" is true on Mac OS X, there are a number of things you should do to make your application's user experience fall in line with various conventions and behaviors that Mac users have come to expect from their applications.This document attempts to highlight items so you can spend your time writing applications, not trying to figure out why something doesn't work the way you think it should.

Who Should Read This Document?

This document is for the Java developer interested in writing Java applications for Mac OS X version 10.4 with J2SE 5.0 or Java 1.4.2. This document is primarily for developers of pure Java applications, but it may also be useful for WebObjects and Cocoa Java development. It focuses primarily on J2SE 5.0 developers targeting J2SE 5.0 Release 4 on Mac OS X v.10.4, but is also applicable to Java 1.4.2 development on Mac OS X. When a difference between Apple's implementations of J2SE 5.0 and Java 1.4.2 is present, J2SE 5.0 is assumed to be the default and Java 1.4.2 differences are noted. This reflects how J2SE 5.0 is default version of Java on Mac OS X v.10.4 as of J2SE Release 4.

Note: J2SE 5.0 Release 4 is available for Mac OS X v.10.4 via Software Update (available via the Apple Menu or the System Preferences application) or as a manual download from http://www.apple.com/support/downloads/j2se50release4ppc.html (PowerPC) or http://www.apple.com/support/downloads/j2se50release4intel.html (Intel).

Information on Java development for previous versions of Mac OS X using Java 1.3.1 is available in a separate document, *Java 1.3.1 Development for Mac OS X*.

This is not a tutorial for the Java language. This document assumes you have a basic understanding of Java development. 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://java.sun.com/learning/new2java/.

Organization of This Document

This guide contains the following articles:

- "Overview of Java for Mac OS X" (page 11) describes the Java platforms available on Mac OS X.
- "Apple Developer Tools for Java" (page 17) introduces you to Apple suite of developer tools.
- "Java Deployment Options for Mac OS X" (page 21) dicusses how you can distribute your Java application on Mac OS X.
- "User Interface Toolkits for Java" (page 31) shows you the different user interface elements common in Mac OS X.
- "Core Java APIs on Mac OS X" (page 41) discusses the how core Java APIs vary on Mac OS X.
- "The Java VM for Mac OS X " (page 45) describes important information about Mac OS X's Java virtual machine.
- "Mac OS X Integration for Java" (page 51) provides you with some handy tips for making your Java application act and feel more like a native Mac OS X application.

See Also

General information about Mac OS X, including more on many of the topics discussed in this document can be found in *Mac OS X Technology Overview*.

Answers to frequently asked questions about Java for Mac OS X are addressed in the Java FAQ.

General information on previous versions of Java for Mac OS X can be found in the Java Release Notes.

This document and other Java documentation for Mac OS X, including the Javadoc API reference, is available in the Java Reference Library. A subset of this documentation is installed in `/Developer/ADC Reference Library/documentation/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 Xcode (from Xcode's Help menu, choose *Documentation* and then click *Java*).

The main Apple website for Java technology, http://developer.apple.com/java/, contains 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 web 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 covered in this document or you want to follow the resolution of an issue, you may do so online through Radar, Apple's bug tracking system. To access Radar, you 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/. With an ADC membership, you can file and view bugs at http://bugreport.apple.com/. When filing new bugs for Java in Mac OS X, please use `Java (new bugs)` for Component and `X` as Version.

Overview of Java for Mac OS X

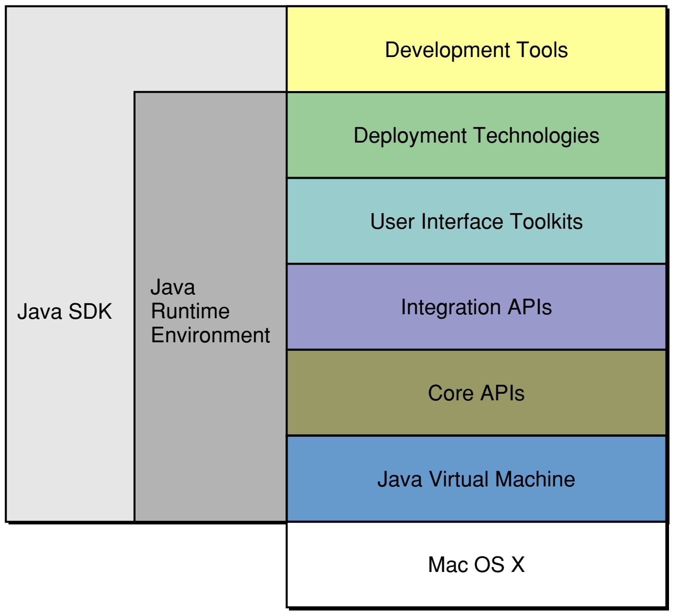
This article provides a broad overview of how Java fits into Mac OS X. It is suggested background information for anyone new to Java development for Mac OS X.

Java and Mac OS X

The complete Java implementation in Mac OS X includes the components you would normally associate with the Java Runtime Environment (JRE) as well as the Java Software Development Kit (SDK). To get the full benefits of the SDK, install the Java Developer Tools, available as part of the Xcode Tools and at http://developer.apple.com/ . More details about the Java SDK in Mac OS X are provided in "Java Deployment Options for Mac OS X" (page 21).

As Figure 1 (page 11) illustrates, the individual components of the JRE and SDK are all built on the native Mac OS X operating system environment.

Figure 1 J2SE platform for Mac OS X



The following sections give a high-level overview of how Java for Mac OS X is different from Java for other platforms. Subsequent chapters delve into the individual layers shown in Figure 1 to discuss the differences of each component of the JRE and SDK in more detail. Illustrations at the beginning of each chapter show what is within the components, which gives you an idea of what is covered in that chapter.

Java, Built In

"Write once, run anywhere" is true only if Java is everywhere. With Mac OS X, you know the Java Runtime Environment (JRE) is there for your Java applications—the Java runtime is built into the operating system. This means that when developing Java applications for deployment on Mac OS X, you know that Java is already installed and configured to work with your customer's operating system. This assurance is good for you as a developer.

The fact that Java is in one of the three high-level APIs for application development, along with Cocoa and Carbon, benefits both you and your customers. It means that with just a little work on your part, Java applications can be nearly indistinguishable from native applications. Information on how to achieve this is provided in "Mac OS X Integration for Java" (page 51). Users don't need to learn different behaviors for Java applications—moreover they shouldn't even know that applications are Java applications.

By building Java as a part of Mac OS X, Apple is able to provide:

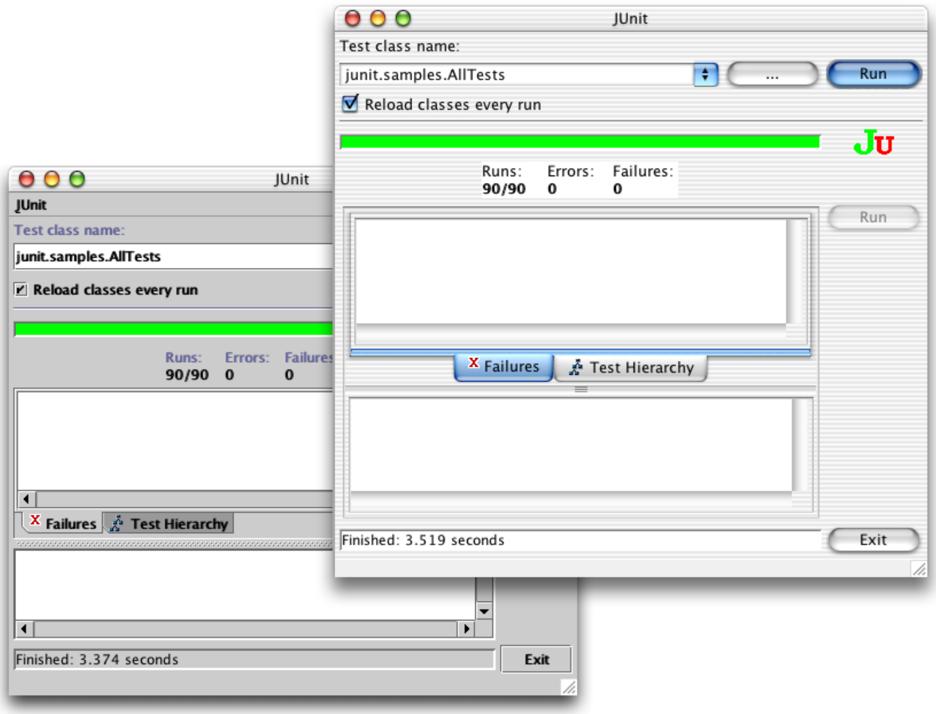
- A Java 1.4.2 implementation with every installation of Mac OS X v.10.4
- A J2SE 5.0 implementation distributed as a Software Update or manual download from http://www.apple.com/support/downloads/j2se50release4ppc.html (PowerPC) or http://www.apple.com/support/downloads/j2se50release4intel.html (Intel)

Note: There is no redistribution license for Java in Mac OS X. If your customers need J2SE 5.0 and they do not have it, they should get it directly from Apple via Software Update or the Apple Support page at http://www.apple.com/support/.

The Aqua User Interface

Anyone who has run a GUI-based Java application in Mac OS X is bound to notice one of the most striking differences between Java on the Macintosh and Java elsewhere. Figure 2 (page 13) shows this distinction by showing the Metal look and feel in Mac OS X, which is essentially the way the user interface looks on other platforms, and the Aqua look and feel.

Figure 2 Standard Java Metal and Apple's Aqua look and feel in Mac OS X



By default, Java Swing applications in Mac OS X use the Aqua look and feel (LAF). Although this is the default LAF, it is not required; the standard Java Metal LAF is also available. While the use of the Aqua LAF is encouraged for Swing applications, different design philosophies inherent in an application might make the Aqua LAF inappropriate. To use the Metal LAF, modify your code to include `UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel")`. Further details on the Aqua LAF are provided in "User Interface Toolkits for Java" (page 31).

Note: Despite the fact that the standard Java appearance is called *Metal*, don't confuse it with Apple's *Textured* look and feel (often referred to as *Metal*), used in applications like iTunes and Safari.

Finding Your Way Around

One of the first things newcomers to Java development in Mac OS X face is figuring out where everything is on the platform. This section outlines some basic things to remember and offers some guidelines to follow when trying to figure out where things are in the Mac OS X filesystem.

Since Java is built into the operating system, it is implemented as a Mac OS X framework. For more information on frameworks, see *Framework Programming Guide*. The code that makes the Java implementations in Mac OS X work can be found in `/System/Library/Frameworks/JavaVM.framework/`. That directory contains one directory, `/Versions/`, and some symbolic links to directories inside the `Versions` directory. The layout of the `JavaVM.framework` directory is designed to accommodate design decisions from previous versions of Java as well as to support future versions of Java. As of J2SE 5.0 Release 4, the `CurrentJDK` symlink points to the `1.5.0` directory. This is where the code that actually implements J2SE 5.0 resides. If you haven't installed J2SE 5.0, the `CurrentJDK` symlink points towards the `1.4.2` directory that contains Apple's Java 1.4.2 implementation for Mac OS X.

Although the exact uses of the files within the `1.5.0` directory are interesting from the perspective of how Java is implemented in Mac OS X, the only directories that you should be concerned with are the `Commands`, `Home`, and `Headers` directories. Even for these, there are restrictions on how you use their contents in the proper way. You should never count on specific paths to anything in these directories for code that you ship to customers. The directories are laid out to make things work well with the underlying operating system and the Java virtual machine. You should also never write anything into these directories on a user's system. Although you can see what is there, consider the contents of `/System/Library/Frameworks/JavaVM.framework/` as read only, and recognize that the contents may change with updates to Java or the operating system.

There are times that you want to install JAR files or JNI libraries into the Java home directory or would expect to link against a header file there. How do you do that if you are not supposed to write into the `JavaVM.framework` directory or link specifically against paths in there? Apple provides a way to do this that should be stable across Java or operating system updates in `/Library/Java/`.

JAVAHOME

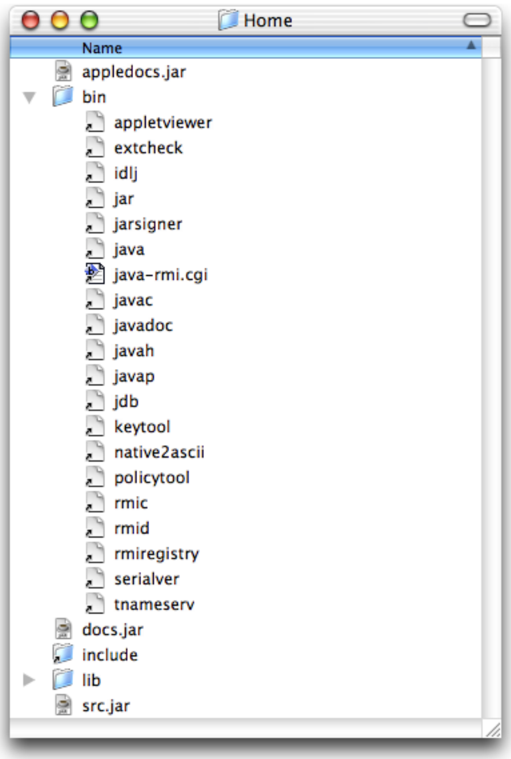
Some applications look for Java's home directory (`JAVA_HOME`) on the users system, especially during installation. If you need to explicitly set this, in a shell script or an installer, set it to `/Library/Java/Home/`. Setting it to the target of that symbolic link can result in a broken application for your customers. Programatically you can use `System.getProperty("java.home")`, as you would expect.

`/Library/Java/Home/` also contains the `/bin/` subdirectory where symbolic links to command-line tools like `java` and `javac` can be found. These tools are also accessible through `/usr/bin/`.

Note: Since the links in `/usr/bin/` point to the tools for J2SE 5.0, to invoke Java 1.4.2 tools you must use the full path. For example, to run the Java 1.4.2 version of `java` use `/System/Library/Frameworks/JavaVM.framework/Versions/1.4.2/Commands/java`.

Figure 3 (page 15) shows the contents of `/Library/Java/Home/`.

Figure 3 /Library/Java/Home


Home
Name
appledocs.jar
bin
appletviewer
extcheck
idlj
jar
jarsigner
java
java-rmi.cgi
javac
javadoc
javah
javap
jdb
keytool
native2ascii
policytool
rmic
rmid
rmiregistry
serialver
tnameserv
docs.jar
include
lib
src.jar

Java 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` system property. 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 every user on that particular computer. If you want to limit which users can use certain extensions, you can put them in the `/Library/Java/Extensions/` directory inside the appropriate users' home directories. By default that folder does not exist, so you may need to make it.

Output From Java Programs

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 display output in the Java Console if the console has been turned on in either the Java 1.4.2 Plugin Settings or Java Preferences applications (See "Other Tools" (page 19) for information on these applications.).

HFS+

The default filesystem of Mac OS X, HFS+ (Mac OS Extended format), is case-insensitive but case-preserving. Although it preserves the case of files written to it, it does not recognize the difference between uppercase and lowercase. You should make sure that no files in the same directory have names that differ only by case. For example, having a file named `mybigimage.tiff` and `MyBigImage.tiff` in the same directory can create unpredictable results. Note that while most UNIX-based operating systems are case-sensitive, Windows is case-insensitive (and not case-preserving) so this is a general guideline for any cross-platform Java development.

Note: Mac OS X v.10.4 allows HFS+ volumes that are fully case-sensitive. Since this is only an option that is chosen at install time and that the traditional behavior described above is the default, don't assume that case-sensitivity can be relied upon.

Details about how HFS+ relates to font encoding can be found in "Font Encoding" (page 35).

Apple Developer Tools for Java

This article provides a broad overview of Apple's tools for Java development. It covers Apple's Xcode IDE, the Jar Bundler application, and methods for obtaining and viewing documentation.

Java Tools on Mac OS X

The Java development tools in Mac OS X are similar to the tools you find in other UNIX-based Java development implementations. The command-line tools that Sun provides as part of the Java SDK for Linux and Solaris work the same in Mac OS X as they do on those platforms. There are only a few significant distinctions between the standard Java development tools in Mac OS X and those found on other UNIX-based platforms:

- The installed location of the command-line tools is different in Mac OS X. The tools are installed with the rest of `JavaVM.framework` in `/System/Library/Frameworks/`. Symbolic links are provided from `/usr/bin/` to these tools. For more information on overall differences in where Java components are in Mac OS X, see "Finding Your Way Around" (page 13).
- `Tools.jar` does not exist. Scripts that look for this file to find the SDK tools need to be rewritten.

Apple provides additional tools for developing and distributing Java applications on Mac OS X. The following sections discuss Xcode, Jar Bundler, and other tools specific to Mac OS X.

Xcode Tools

Apple provides a full suite of general developer tools with Mac OS X. This suite of tools, the Xcode Tools, is free but not installed by default. The tools are available on the Mac OS v.10.4 DVD, and installers are included in `/Applications/Installers/Developer Tools/` on new computers. The most current version of the developer tools is also available online at the Apple Developer Connection (ADC) Member Site http://connect.apple.com/.

Get the Current Tools

New features and bug fixes for the Mac OS X Developer Tools are released throughout the year. Even if you already have the Xcode Tools installed, you should check the Member Site (http://connect.apple.com/) for the most up-to-date version. You need to be enrolled as an ADC member to access that site. If you do not have an ADC membership, you can enroll for various levels of membership, including a free online membership that allows you access to the Member Site, at http://developer.apple.com/membership/.

The Xcode Tools are available from the *Download Software* link. There are two components to download that together give you the full Java development environment for Mac OS X. The Mac OS X section contains the base Xcode Tools. Download and install the most current released version available. There are also a Java–specific updates to the base developer tools, the J2SE 5.0 Developer Tools, which is available in the *Java* section. Download and install these as well.

With the Xcode Tools and the Java Developer Tools, you have a full-featured development environment including:

- Command-line tools installed in `/Developer/Tools/`
- Graphical tools installed in `/Developer/Applications/`
- Sample code installed in `/Developer/Examples/`
- Documentation installed in `/Developer/ADC Reference Library/`

Xcode

The core component of the Mac OS X development environment is Xcode. Xcode is a complete integrated development environment (IDE) that allows you to edit, compile, debug, and package Mac OS X applications written in multiple languages. Even if you do not intend to use it for your primary Java development, become familiar with Xcode. Downloadable sample code and the sample code installed in `/Developer/Examples/Java/` are both usually provided as Xcode projects. Additionally, there are some elements of documentation viewing that are available only through Xcode.

For more on using Xcode for Java development, see *Xcode 2 User Guide*.

Using Xcode with J2SE 5.0

The Java templates in Xcode are setup for Java 1.4.2. To use J2SE 5.0 instead, modify these settings:

Target Settings:
Double click the target to edit and provide `/System/Library/Frameworks/JavaVM.framework/Versions/1.5/Commands/javac` as the value for the `JAVA_COMPILER` build setting. Change the `Target VM Version` and `Source Version` in the Java Compiler Setting to use `1.5`.

Executable Settings:
Double click the executable named `java` and enter `/System/Library/Frameworks/JavaVM.framework/Versions/1.5/Commands/java` as the Executable Path in the *General* tab of Executable info.

Applet Development:

Double click the executable named `appletviewer` and enter `/System/Library/Frameworks/JavaVM.framework/Versions/1.5/Commands/appletviewer` as the Executable Path in the *General* tab of Executable info.

Jar Bundler

Jar Bundler is an application that turns pure Java applications into applications that can be launched just like native Mac OS X applications. Although the Terminal application is a part of every installation of Mac OS X, many Mac OS X users never use it. To prevent your users from having to use Terminal for your Java applications, you should wrap your application as a Mac OS X application bundle (See "Mac OS X Application Bundles" (page 22)). Jar Bundler allows you to do this very easily. It also provides a simple interface for you to set system properties that make your applications perform their best in Mac OS X.

Jar Bundler is installed in `/Developer/Applications/Java Tools/` with the Java Developer Tools. If Jar Bundler is not on your system and you see an application named MRJAppBuilder, you need to install the Java 1.4.2 Developer Tools Update. Jar Bundler replaces MRJAppBuilder and can be used for both J2SE 5.0 and Java 1.4.2 applications.

More information on Jar Bundler is available in *Jar Bundler User Guide*.

Applet Launcher

Applet Launcher (in `/Developer/Applications/Java Tools/`) provides a graphical interface to Sun's Java Plug-in. Applet Launcher loads an applet from an HTML page. For example, entering the following URL launches the ArcTest applet:

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

Applet Launcher is useful for seeing how your applets perform in J2SE 5.0 on Mac OS X. Performance and behavior settings for applets may be adjusted in the Java Preferences application installed in `/Applications/Utilities/Java/J2SE 5.0/`.

Note: Applet Launcher uses the J2SE 5.0 VM after J2SE 5.0 Release 4 is installed. To test applets with Java 1.4.2, change the Java Plug-in VM in Java Preferences and load the applet in Safari.

Other Tools

In addition to Applet Launcher, `/Applications/Utilities/Java/` contains these Java-related tools that you might find useful when testing your application:

- *Input Method HotKey* to set the keyboard combination that invokes the input method dialog in applications with multiple input methods
- *Java Preferences* for specifying settings for all Java applications and plug-ins and J2SE 5.0 applets
- *Java 1.4.2 Plugin Settings* for specifying settings for Java 1.4.2 applets

- *Java Web Start*, to allow you launch and modify settings for JNLP-aware Java Web Start applications

In addition to Xcode and Jar Bundler, `/Developer/Applications/` contains some applications that you can use for Java development though they are not Java-specific. *PackageMaker*, *FileMerge*, and *Icon Composer* are a few examples that you might consider using.

Having a UNIX-based core at the heart of the operating system provides you with a host of general UNIX-based development tools as well. A look in `/usr/bin/` shows many tools that make Java development in Mac OS X very comfortable if you are already accustomed to a UNIX-based operating system. (In the Finder, choose Go > Go to Folder... and type in `/usr/bin/`.) You will find `emacs`, `make`, `pico`, `perl`, and `vi` among others.

You can find additional development tools in the Darwin and OpenDarwin CVS repositories available at http://developer.apple.com/darwin/ and http://www.opendarwin.org/ respectively. For basic information on porting your favorite non-Java tools to Mac OS X, see *Porting UNIX/Linux Applications to Mac OS X*.

Developer Documentation

Documentation for Java development in Mac OS X is provided both online and locally with the installation of the Xcode Tools. The most current version of the documentation is available from the Java Reference Library on the Apple Developer Connection website. A snapshot of this documentation is also installed on your computer. This documentation is mainly HTML based, so you can view it in your choice of browser by launching the main navigation page at `/Developer/ADC Reference Library/documentation/Java/index.html`. Man pages for the command-line tools are accessible from the command line `man` program and through Xcode's Help menu.

Note that Apple does not attempt to provide a full Java documentation suite online or with the Xcode Tools. Sun supplies very thorough documentation available online at http://java.sun.com/reference/docs/. Apple's documentation aims to augment Sun's documentation for Java development issues specific to Mac OS X and to document Mac OS X–specific features of Java. Your primary source for general Java documentation is Sun's Java documentation web site.

Providing Documentation Feedback

If you find errors in the Java documentation or would like to request either feature or content enhancements, you can file bugs at http://bugreport.apple.com/. When filing documentation bugs for Java in Mac OS X, please use `Java Documentation (developer)` for Component and `X` as Version.

You may also send email feedback to javadevdoc@apple.com or use the feedback links at the bottom of ADC Reference Library documents and Tech Notes.

Java Deployment Options for Mac OS X

When deploying Java applications in Mac OS X, you have access to the Java Plug-in and Java Web Start as you do on other platforms. There are also two additional deployment technologies in Mac OS X. You may deploy applications as double-clickable JAR files or as native Mac OS X application bundles. This chapter discusses these four deployment technologies. Note that there are three different Java implementations currently available on Mac OS X, Java 1.4.2 and J2SE 5.0. Make sure that you are using the correct Java implementation for your application. Information on what you need to do to specify the correct Java implementation is included in the following sections.

Double-Clickable JAR Files

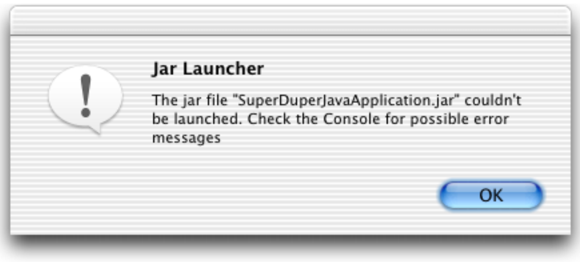
The simplest way to deploy an application is to distribute it as a JAR file. The biggest advantage of this technique is that it requires very little, if any, changes from the JAR files you distribute on other platforms. This technique, however, has drawbacks for your users. Applications distributed as JAR files are given a default Java applications icon, instead of one specific to that application, and do not allow you to easily specify runtime options without doing so either programatically or from a shell script. If your application has a graphical interface and will be run by general users, you should consider wrapping the JAR file as a Mac OS X application bundle. You can find more information on how to do this in "Mac OS X Application Bundles" (page 22).

Double-clickable JAR files launch with J2SE 5.0. If a JAR file needs to be launched in Java 1.4.2, wrap the JAR file as a Mac OS X application bundle using Jar Bundler.

Note: You can double-click on a Java class file to launch your application, but this is not a recommended method for application deployment.

If you choose to deploy your application from a JAR file in Mac OS X, the manifest file must specify which class contains the `main` method.Without this, the JAR file is not double-clickable and users see an error message like the one shown in Figure 1 (page 22).

Figure 1 Jar Launcher error



Jar Launcher

The jar file "SuperDuperJavaApplication.jar" couldn't be launched. Check the Console for possible error messages

If you have a JAR file that does not already have the main class specified in the manifest, you can remedy this as follows:

1. Unarchive your JAR file into a working directory with some variant of `jar xvf` *myjar*`.jar`
2. In the resultant `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, a default manifest file in Mac OS X looks like this:

   ```
   Manifest-Version: 1.0
   Created-By: 1.4.2_07 (Apple Computer, Inc.)
   ```

   With the addition of the main class designation it looks like:

   ```
   Manifest-Version: 1.0
   Created-By: 1.4.2_07 (Apple Computer, Inc.)
   Main-Class: YourAppsMainClass
   ```

3. Archive your 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.

Mac OS X Application Bundles

Native Mac OS X applications are more than just executable files. Although a user sees a single icon in the Finder, an application is actually an entire directory that includes images, sounds, icons, documentation, localizable strings, and other resources that the application may use in addition to the executable file itself. The application bundle simplifies application development in many ways for developers. The Finder, which displays an application bundle as a single item, retains simplicity for users, including the ability to just drag and drop one item to install an application.

This section discusses Mac OS X application bundles as they relate to deploying Java applications. More general information on Mac OS X application bundles is available in *Bundle Programming Guide*.

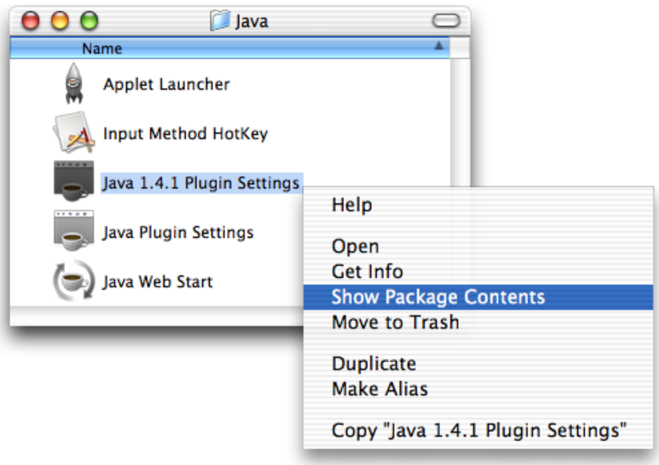
When deploying Java applications in Mac OS X, consider making your Java application into a Mac OS X application bundle. It is easy to do and offers many benefits:

- Users can simply double-click the application to launch it.
- If you add an appropriate icon, it shows the application icon in the Dock and in the menu bar, clearly identifying your application. (Otherwise, a default Java icon appears in the Dock.)
- It lets you easily set Mac OS X–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 lets users launch your application by double-clicking a document associated with it.

The Contents of an Application Bundle

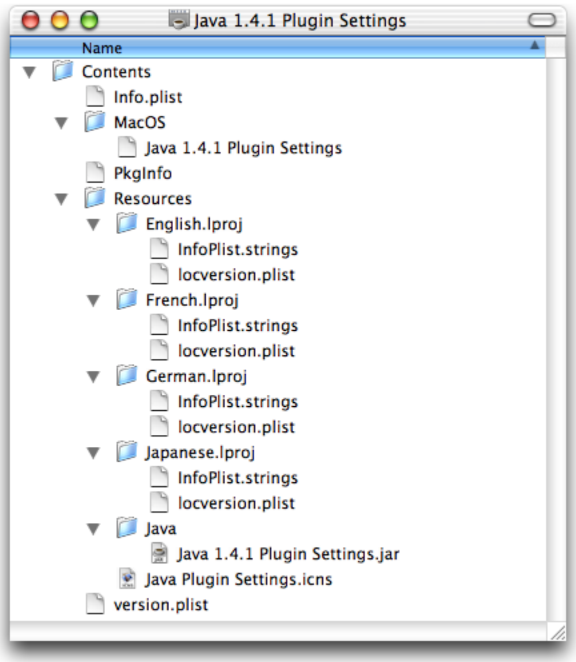
The application bundle directory structure is hidden from view in the Finder by the `.app` suffix and a specific attribute, the bundle bit, that is set for that directory. (See *Runtime Configuration Guidelines* for more information on Finder attributes.) The combination of these two things makes the directory a bundle. To get a glimpse inside an application bundle, you can explore the directory of resources from Terminal or from the Finder. Although by default the Finder displays applications as a single object, you can see inside by Control-clicking (or right-clicking if you have a multiple-button mouse) an application icon and choosing Show Package Contents as shown in Figure 2 (page 23).

Figure 2 Show application bundle contents



You should see something similar to the directory structure shown in Figure 3 (page 24).

Figure 3 Contents of a Java application bundle



```
Contents
    Info.plist
    MacOS
        Java 1.4.1 Plugin Settings
    PkgInfo
    Resources
        English.lproj
            InfoPlist.strings
            locversion.plist
        French.lproj
            InfoPlist.strings
            locversion.plist
        German.lproj
            InfoPlist.strings
            locversion.plist
        Japanese.lproj
            InfoPlist.strings
            locversion.plist
        Java
            Java 1.4.1 Plugin Settings.jar
        Java Plugin Settings.icns
    version.plist
```

Applications bundles for Java applications should have the following components:

- 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 Java Dictionary Info.plist Keys.
- A file named `PkgInfo` should also be in the Contents folder. This is a simple text file that contains the string `APPL` optionally followed directly by a four letter creator code. If an application does not have a registered creator code, the string `APPL????` should be used. You may register your application with Apple's creator code database on the ADC Creator Code Registration site at http://developer.apple.com/datatype/.
- The application's icon that is displayed in the Dock and the Finder should be 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/Utilities/`.
- The Java code itself, in either `jar` or `.class` files, in `Resources/Java/`.
- A native executable file in the `MacOS` folder that launches the Java VM.
- Optional localized versions of strings may be included in folders designated by the `.lproj` suffix. The example in Figure 3 (page 24) includes localizable strings for four different languages. See "Additional Considerations for Non-English Applications" (page 27) for more information on localized application bundles.

There are other files in the application bundle, but these are the ones that you should have in a Java application bundle. You can learn more about the other files in an application bundle, as well as more information about some of these items, in *Framework Programming Guide*.

A Java Application's Info.plist File

Mac OS X makes use of XML files for various system settings. The most common type of XML document used is the property list. Property lists have a `.plist` extension. The `Info.plist` file in the `Contents` folder of a Mac OS X application is a property list.

The `Info.plist` file lets you fine-tune how well your application is presented in Mac OS X. With slight tweaking of some of the information in this file, you can make your application virtually indistinguishable from a native application in Mac OS X, which is important for making an application that users appreciate and demand.

If you build your Java application in Xcode or Jar Bundler, the `Info.plist` file is automatically generated for you. If you are building application bundles through a shell or Ant script, you need to generate this file yourself. Even if it is built for you, you may want to modify it. Since it is a simple XML file, you can modify it with any text editor.

Listing 1 (page 25) shows an `Info.plist` for a Java application that has been wrapped as an application bundle.

Listing 1 Example Info.plist

```
<?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>Java 1.4.2 Plugin Settings</string>
    <key>CFBundleGetInfoString</key>
    <string>2.1.1 (for Java 1.4.2), Copyright 2004 Apple Computer, Inc. All
Rights Reserved.</string>
    <key>CFBundleIconFile</key>
    <string>Java Plugin Settings.icns</string>
    <key>CFBundleIdentifier</key>
    <string>com.apple.java.PluginSettings.142</string>
    <key>CFBundleInfoDictionaryVersion</key>
    <string>6.0</string>
    <key>CFBundleName</key>
    <string>Java 1.4.2 Plugin Settings</string>
    <key>CFBundlePackageType</key>
    <string>APPL</string>
    <key>CFBundleShortVersionString</key>
    <string>2.1.1</string>
    <key>CFBundleSignature</key>
    <string>????</string>
    <key>CFBundleVersion</key>
    <string>2.1.1</string>
    <key>Java</key>
    <dict>
        <key>JVMVersion</key>
        <string>1.4*</string>
        <key>MainClass</key>
        <string>sun.plugin.panel.ControlPanel</string>
        <key>VMOptions</key>
        <string>-Xbootclasspath/p:/System/Library/Frameworks/
JavaVM.framework/Versions/1.4.2/Home/lib/jaws.jar:/System/Library/
Frameworks/JavaVM.framework/Versions/1.4.2/Home/lib/netscape.jar</string>
    </dict>
    <key>LSHasLocalizedDisplayName</key>
    <true/>
    <key>NSHumanReadableCopyright</key>
    <string>Copyright 2003 Apple Computer, Inc., All Rights Reserved.</string>
    <key>NSJavaPath</key>
    <array/>
</dict>
</plist>
```

A property list file is divided into hierarchical sections called dictionaries. These are designated with the `dict` key. 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 *Runtime Configuration Guidelines*.

At the end of the CFBundle keys, this example includes a `Java` key designating the beginning of a Java dictionary. At least two keys should be in this dictionary; the `MainClass` is required and the `JVMVersion` key is highly recommended. A listing of all the available keys and Java version values for the Java dictionary is provided in Java Dictionary Info.plist Keys.

If you examine an older Java application distributed as an application bundle, some of the keys may be missing from the `Properties` dictionary. Java application bundles used to include the Java-specific information distributed between an `Info.plist` file and another file, `MRJApp.properties` in `Contents/Resources/` in the application bundle. If you are updating an existing application bundle, you should move the information from the `MRJApp.properties` file into the appropriate key in the Java dictionary in the `Info.plist` file.

Making a Java Application Bundle

There are three ways to make a Java application bundle:

- With Xcode
- With Jar Bundler
- From the command line

If you build a new Java AWT or Swing application, use one of Xcode's templates and Xcode automatically generates an application bundle complete with a default `Info.plist` file. You can fine-tune the behavior of this application by modifying the `Info.plist` file directly in the Products group of the Files pane or by modifying the settings in the Info.plist Entries section of the Targets pane. For more information on using Xcode for Java development, see Xcode Help (available from the Help menu in Xcode) and *Xcode 2 User Guide*.

If you want to turn your existing Java application into a Mac OS X Java application, use the Jar Bundler application available in `/Developer/Tools/`. It allows you to take existing `.class` or `jar` files and wrap them as application bundles. Information about Jar Bundler, including a tutorial, is provided in *Jar Bundler User Guide*.

To build a valid application bundle from the command-line, for example in a shell script or an Ant file, you need to follow these steps:

1. Set up the correct directory hierarchy. The top level directory should be named with the name of your application with the suffix `.app`.

   There should be a Contents directory at the root of the application bundle. It should contain a `MacOS` directory and a `Resources` directory. A Java directory should be inside of the Resources directory.

   The directory layout should look like this.

   ```
   YourApplicationName.app/
       Contents/
           MacOS/
           Resources/
               Java/
   ```

2. Copy the `JavaApplicationStub` file from `/System/Library/Frameworks/JavaVM.framework/Versions/Current/Resources/MacOS/` into the `MacOS` directory of your application bundle.

3. Make an `Info.plist` file in the `Contents` directory of your application bundle. You can start with an example like that given in Listing 1 (page 25) and modify it or generate a completely new one from scratch. Note that the application bundle does not launch unless you have set the correct attributes in this property list, especially the `MainClass` key.

4. Make a `PkgInfo` file in the `Contents` directory. It should be a plain text file. If you have not registered a creator code with ADC, the contents should be `APPL????`. If you have registered a creator code replace the `????` with your creator code.

5. Put your application's icon file into the `Contents/Resources/` directory. (For testing purposes, you can copy the generic Java application icon from `/Developer/Applications/Jar Bundler.app/Contents/Resources/`.)

6. Copy your Java `.jar` or `.class` files into `/Contents/Resources/Java/`.

7. Set the bundle bit Finder attribute with `SetFile`, found in `/Developer/Tools/`. For example, `/Developer/Tools/SetFile -a B YourApplicationName.app`.

After these steps, you should have a double-clickable application bundle wrapped around your Java application.

Additional Considerations for Non-English Applications

To run correctly in locales other than US English, Java application bundles must have a localized folder for each appropriate language inside the application bundle. Even if the Java application handles its localization through Java `ResourceBundles`, the folder itself must be there for the operating system to set the locale correctly when the application launches. Otherwise Mac OS X launches your application with the US English locale.

Put a folder named with the locale name and the `.lproj` suffix in the application's Resources folder for any locale that you wish to use. For example if you include a Japanese and French version of your application, include a `Japanese.lproj` folder and a `French.lproj` folder in `YourApplicationName.app/Contents/Resources/`. The folder itself can be empty, but it must be present.

Bundle Programming Guide provides more detail about the application bundle format.

Distributing Application Bundles

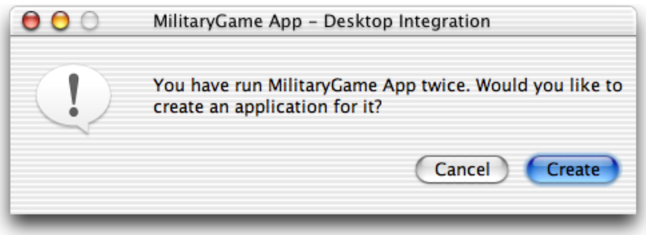
The recommended way to distribute application bundles is as a compressed disk image. This gives users the ease of a drag-and-drop installation. Put your application bundle along with any relevant documentation on a disk image with Disk Utility, compress it, and distribute it. Disk Utility is available in `/Applications/Utilities/`. You can further simplify the installation process for your application by making the disk image Internet enabled. For information on how to do this see Distributing Software with Internet Enabled Disk Images.

Java Web Start

As a part of Java 1.4.2 and J2SE 5.0, Mac OS X also supports deploying your application as a Java Web Start application. Java Web Start is an implementation of the Java Network Launching Protocol & API (JNLP) specification, which means that if you make your application JNLP-aware, Mac OS X users do not need to do anything to use it. 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 4 (page 28). 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 offers, but it is 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 4 Java Web Start integration


MilitaryGame App – Desktop Integration

You have run MilitaryGame App twice. Would you like to create an application for it?

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

You need to be aware of only a few details about 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 v.10.4 includes Java 1.4.2 and offers J2SE 5.0 as a Software Update. If your application requires J2SE 5.0, you need direct your customers to Apple to obtain it. The version keys that are valid for Java Web Start applications are the same as those for Mac OS X application bundles and are listed in Java Dictionary Info.plist Keys.

- It is not necessary to set up proxy information explicitly in the Web Start application. Java Web Start in Mac OS X automatically picks up the proxy settings from the Network pane in System Preferences.
- Java Web Start caches its data in the user's `/Library/Caches/Java Web Start/` directory.

The Java Plug-in

Java 1.4.2 and J2SE 5.0 for Mac OS X include the Java Plug-in for you to deploy applets in browsers and other Java embedding applications. The only browsers that currently support the Java 1.4.2 and J2SE 5.0 Java Plug-in use the Web Kit API. Other browsers on Mac OS X currently use either a Java 1.3.1 Java Plug-in or the legacy Java Embedding framework. The Java Embedding framework supports Java 1.3.1 by implementing the `appletviewer` class and therefore does not provide the added benefits that the Java Plug-in provides like JAR caching, handling of signed JAR files, and the Java console.

When testing your applications, you can determine which version of the Java Plug-in the application is using with the `GetOpenProperties` applet available from Sun at http://java.sun.com/docs/books/tutorial/deployment/applet/. These are the Java implementations used by current browsers:

Safari 2.0
: Java 1.4.2 and J2SE 5.0 Plug-in

Internet Explorer 5.2
: Java 1.3.1 Embedding framework

Netscape 7.2/ Mozilla 1.7
: Java 1.3.1 Plug-in

The Applet Launcher application in `/Applications/Utilities/Java/` also uses the J2SE 5.0 Plug-in, but it is not a full-featured browser – it is more of a development tool. For more information on Applet Launcher see " Applet Launcher" (page 19).

" Using Xcode with J2SE 5.0" (page 18) includes information on using Applet Launcher with J2SE 5.0.

For all of the common browsers, the `<APPLET>` tag has proven to be less problematic than the `<OBJECT>` and `<EMBED>` tags.

User Interface Toolkits for Java

This article discusses how the Mac OS X implementation of the Swing, AWT, accessibility, and sound user interface toolkits differ from other platforms. Although there is some additional functionality in Mac OS X, for the most part these toolkits work as you would expect them to on other platforms. This chapter does not discuss user interface design issues that you should consider in Mac OS X. For that information see "Making User Interface Decisions" (page 51).

Swing

In Mac OS X, Swing uses `apple.laf.AquaLookAndFeel` as the default look and feel (LAF). Swing attempts to be platform neutral, but there are some parts of it that are at odds with the Aqua user interface. Apple has tried to bridge the gap with a common ground that provides both developers and users an experience that is not foreign. This section discusses a few details where the Aqua LAF differs from the default implementation on other platforms.

While testing your application, you might want to test it on the standard Java Metal LAF as well as Aqua. To run your application with Metal, pass the `-Dswing.defaultlaf=javax.swing.plaf.metal.MetalLookAndFeel` flag to `java`.

JMenuBars

Macintosh users expect to find their menus in the same spot – no matter what window they have open – at the top of the screen in the menu bar. In the default Metal LAF, as well as the Windows LAF, menus are applied on a per-frame basis inside the window under the title bar.

To get menus out of the window and into the menu bar you need only to set a single system property:

```
apple.laf.useScreenMenuBar
```

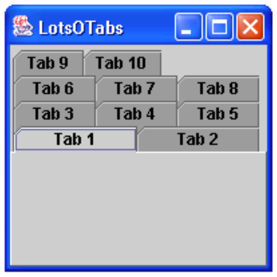
This property can have a value of `true` or `false`. By default it is `false`, which means menus are in the window instead of the menu bar. When set to `true`, the Java runtime moves any given JFrame's JMenuBar to the top of the screen, where Macintosh users expect it. Since this is just a simple runtime property that only the Mac OS X VM looks for, there is no harm in putting it into your cross-platform code base.

Note that this setting does not work for JDialogs that have JMenus. A dialog should be informational or present the user with a simple decision, not provide complex choices. If users are performing actions in a dialog, it is not really a dialog and you should consider a JFrame instead of a JDialog.

JTabbedPanes

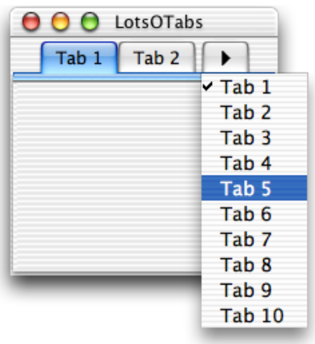
Another example of a difference in Mac OS X is with JTabbedPanes. With other LAFs, if you have JTabbedPane with too many tabs to fit in the parent window – they just get stacked on top of each other as shown in Figure 1 (page 32).

Figure 1 A tabbed pane with multiple tabs on another platform



In the Aqua user interface of Mac OS X, tab controls are never stacked. The Aqua LAF implementation of multiple tabs includes a special tab on the right that exposes a pull-down menu to navigate to the tabbed panes not visible. This behavior, shown in Figure 2 (page 32) allows you to program your application just as you would on any other platform, but provides users an experience that is more consistent with Mac OS X.

Figure 2 A tabbed pane with multiple tabs in Mac OS X

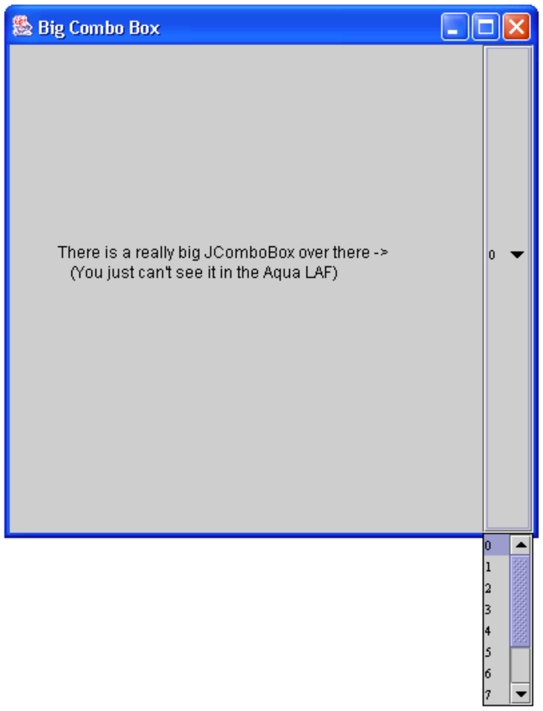


One other thing to keep in mind about JTabbedPanes in the Aqua look and feel is that they have a standard size. If you put an image in a tab, the image is scaled to fit the tab instead of the tab to the image.

Component Sizing

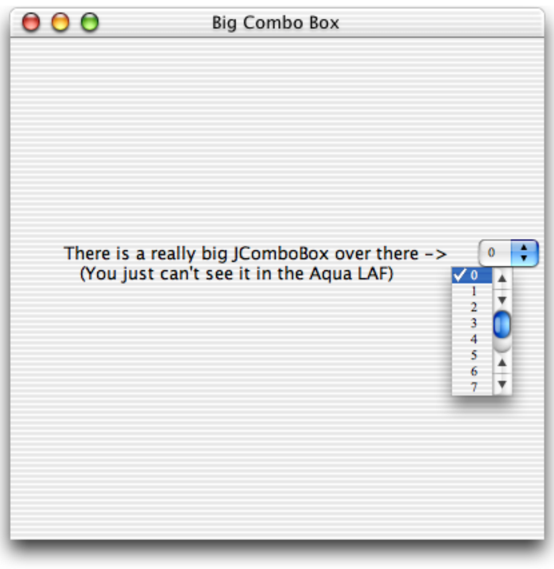
Aqua has very well-defined guidelines for the size of its widgets. Swing, on the other hand, does not. The Aqua LAF tries to find a common ground. Since any combo box larger than twenty pixels would look out of place in Mac OS X, that is all that is displayed, even if the actual size of the combo box is bigger. For example, Figure 3 (page 33) shows a very large JComboBox in Windows XP. Note that the drop-down scrolling list appears at the bottom of the button. The same code yields quite a different look in Mac OS X, as can be seen in Figure 4 (page 34). The visible button is sized to that of a standard Aqua combo box. The drop-down list appears at the bottom of the visible button. The entire area that is active on other platforms is still active, but the button itself doesn't appear as large.

Figure 3 Oversize JComboBox in Windows



There is a really big JComboBox over there ->
(You just can't see it in the Aqua LAF)

Figure 4 Oversize JComboBox in the Aqua LAF


Big Combo Box
There is a really big JComboBox over there ->
(You just can't see it in the Aqua LAF)
0
0
1
2
3
4
5
6
7

Note that some other components have similar sizing adjustments to align with the standards set in *Apple Human Interface Guidelines* for example, scroller and sliders. The JComboBox example is an extreme example. Most are not as large, but this gives you an idea of how the Aqua LAF handles this type of situation.

Buttons

There are basically three button types in Mac OS X:

- Push buttons which are rounded rectangles with text labels on them.
- Radio buttons which are in sets of two to seven circular buttons. They are for making mutually exclusive, but related choices.
- Bevel buttons which can display text, an icon, or a picture that can be either a standard push button or have a menu attached.

  Bevel buttons normally have rounded corners. When displayed in a toolbar or when sizing constraints are tight, the corners are squared off.

To be consistent with these button types and their defined use in Mac OS X, there are a some nuances of Swing buttons that you should be aware of:

- JButtons with images in them are rendered as bevel buttons by default.
- A default JButton that contains only text is usually rendered as a push button. (Over a certain height, it is rendered as a bevel button, since Aqua push buttons are limited in their height.)
- JButtons in a toolbar are rendered as bevel buttons with square, not rounded edges.

In addition to these default characteristics which are dependent on height and the contents of the button, you can also explicitly set the type of button with `JButton.buttontype` which accepts `toolbar`, `icon`, or `text`. `toolbar` gives you square bevel button, `icon` gives you a rounded bevel button, and `text` gives you a push button. Keep the Apple Human Interface Guidelines in mind if you explicitly set a button type.

AWT

By its nature, AWT is very different on every platform and there are a few high level things to keep in mind about AWT in Mac OS X. These details are explored in the following sections.

Accelerator Key

The value of the accelerator key can be determined by calling `Toolkit.getDefaultToolkit().getMenuShortcutKeyMask()`. This is further discussed in "Accelerators (Keyboard Shortcuts)" (page 53).

Font Encoding

The default font encoding in Mac OS X is MacRoman. The default font encoding on some other platforms is ISO-Latin-1 or WinLatin-1. These are subsets of UTF-8 which means that files or filenames can be turned into UTF-8 by just turning a byte into a char. Programs that assume this behavior cause problems in Mac OS X.

The simplest way to work around this problem is to specify a font encoding explicitly rather than assuming one.

If you do not specify a font encoding explicitly, recognize that:

- In the conversion from Unicode to MacRoman you may lose information.
- Filenames are not stored on disk in the default font encoding, but in UTF-8. Usually this isn't a problem, since most files are handled in Java as `java.io.File`s, though it is good to be aware of.
- Although filenames are stored on disk as UTF-8, they are stored decomposed. This means certain characters, like e-acute (é) are store as two characters, "e", followed by "´" (acute accent). The default HFS+ filesystem of Mac OS X enforces this behavior. SMB enforces composed Unicode characters. UFS and NFS do not specify whether filenames are stored composed or decomposed, so they can do either.

Minimum Window Size

Mac OS X does not specify a default minimum size for windows. To avoid a 0 by 0 (0x0) pixel window being opened, the following default minimum sizes for Java windows are enforced:

- Borderless windows have a minimum size of 1 by 1 (1x1).

- Windows with a title bar have a minimum size of 128 by 32 (128x32).

Full-Screen Exclusive Mode

In J2SE 5.0 and Java 1.4, `java.awt.GraphicsDevice` includes methods for controlling the full screen of a client computer through Java. In addition to these standard tools, Mac OS X provides a few system properties that may be useful for development of full-screen Java applications. These are discussed in Java System Properties.

Accessibility

With some other platforms, the use of the Java Accessibility API requires the use of a native bridge. This is not necessary in Mac OS X. The code needed to bridge the Accessibility API to the native operating system is built in. Users can configure the accessibility features of Mac OS X through the Universal Access pane of System Preferences. If you are using the Accessibility API, your application can use devices that the user has configured there.

Beginning with Mac OS X v10.4, a screen reader call VoiceOver is included with the operating system. Your Java application automatically utilizes this technology.

Security

In Mac OS X v10.4, Java applications that utilize Kerberos automaticially access the system credentials cache and tickets.

Apple also includes a cryptographic service provider based on the Java Cryptography Architecture. Currently, the following algorithms are supported:

- Mac: MD5, SHA1
- Message Digest: MD5, SHA1
- Secure Random: YarrowPRNG

Java on Mac OS X v10.4 features an implementation of KeyStore that uses the Mac OS X Keychain as its permanent store. You can get an instance of this implementation by using code like this:

```
keyStore = KeyStore.getInstance("KeychainStore", "Apple");
```

See the reference documentation on `java.security.KeyStore` for more usage information.

Sound

Java sound input is supported only at a frame rate of 44100 (in PCM encoding, mono or stereo, 8 or 16 bits per sample) by the Core Audio framework. If you need a different frame rate, you can easily resample in your application.

By default, the Java sound engine in Mac OS X uses the `midsize` sound bank from http://java.sun.com/products/java-media/sound/soundbanks.html.

Input Methods

Mac OS X supports Java input methods. The utility application Input Method Hot Key, installed in `/Applications/Utilities/Java/`, allows you to configure a trigger for input methods. You can download sample input methods from http://java.sun.com/products/jfc/tsc/articles/InputMethod/inputmethod.html.

Java 2D

As with Java on other platforms, the Java2D API takes advantage of the native platform to provide behavior that is as close as possible to the behavior of a native application. In Mac OS X, the Java2D API is based on Apple's Quartz graphics engine. (See http://developer.apple.com/graphicsimaging/quartz/ for more information.) This results in some general differences between Java in Mac OS X and on some other platforms.

In Mac OS X, Java windows are double-buffered. The Java implementation itself 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`.

Normally Quartz displays text anti-aliased. Therefore by default, Java2D renders text in the Aqua LAF for Swing with `KEY_ANTIALIASING` set to `VALUE_ANTIALIAS_ON`. It can be turned off using the properties described in Java System Properties, or by calling `java.awt.Graphics.setRenderingHint` within your Java application. Note that in applets, anti-aliasing is turned off be default.

Since anti-aliasing is handled by Quartz, whether it is on or off does not noticeably affect the graphics performance of your Java code. One reason for turning it off might be to try to get a closer pixel-for-pixel match of your images and text with what you are using on other platforms. Recognize that if you turn off anti-aliasing haphazardly in your application, it can make it look out of place in the Mac OS X environment, and even with anti-aliasing off, you still will not see an exact pixel-for-pixel match across platforms.

On any platform, anti-aliased text and graphics look different from aliased images or text. If the native platform supports and encourages it, you should design your code so that you do not depend on an exact pixel-for-pixel match between platforms regardless of whether or not anti-aliasing is on. Quartz, for example, takes advantage of both colors and transparency for its anti-aliasing instead of just color; differences between it and other platforms are therefore very distinct. Note the differences illustrated in Figure 5 (page 38) and Figure 6 (page 38).

Figure 5 Anti-aliased text in Mac OS X

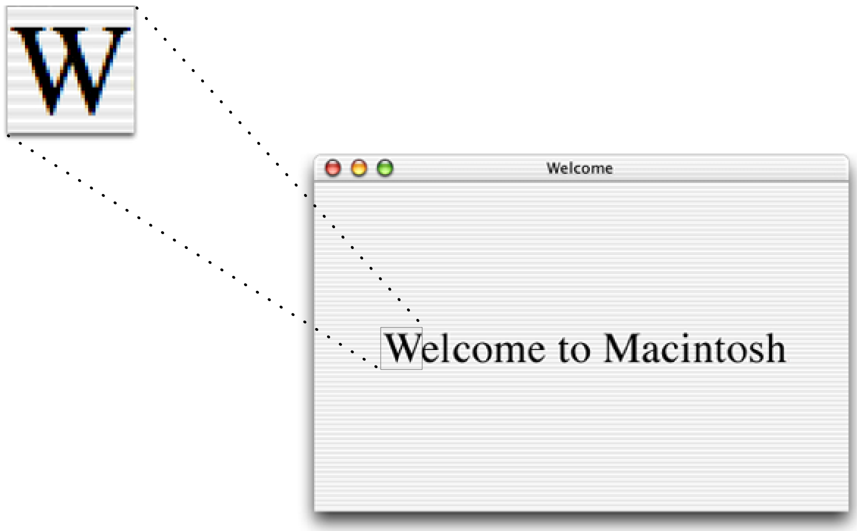
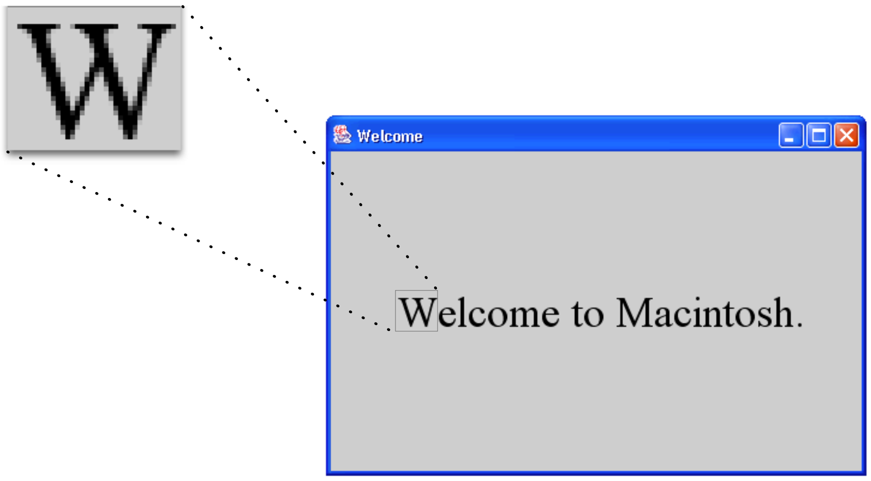
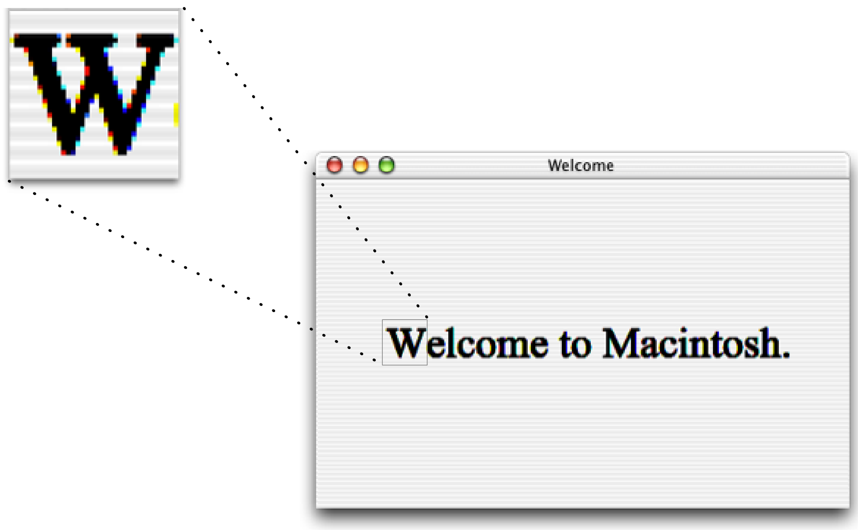


Figure 6 Anti-aliased text in Windows XP



The algorithms used to determine anti-aliasing take many variables into account which means you cannot erase a line by just drawing over it with a line of a different color. For example, drawing a white line on top of a black line does not completely erase the line; the compositing rules leave some pixels around the edges. You end up with a blurred outline of your original image. 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 as can be seen in Figure 7 (page 39). Similarly, do not count on using XOR mode to repaint images. Whenever you are using anti-aliasing, as you often do by default in Mac OS X, repaint the graphics context if you need to replace text or an image.

Figure 7 Painting the same image multiple times



Core Java APIs on Mac OS X

In general, the Core Java APIs behave as you would expect them to on other platforms so most of them are not discussed in this chapter. There are a couple of details concerning Preferences that you should be aware of, as discussed in "Other Tools" (page 19). Basic information on using JNI in Mac OS X is provided in "Input Methods" (page 37).

Networking

Mac OS X v.10.3 and beyond supports IPv6 (Internet Protocol version 6). Because J2SE 5.0 and Java 1.4.2 use IPv6 on platforms that support it, the default networking stack in Mac OS X is the IPv6 stack. You can make Java use the IPv4 stack by setting the `java.net.preferIPv4Stack` system property to `true`.

Preferences

The Preferences API is fully supported in Mac OS X, but there are two details you should be aware of to provide the best experience to users:

- The preferences files generated by the Preferences API are named `com.apple.java.util.prefs`. The user's preferences file is stored in `/Library/Preferences/` in their home directory (`~/Library/Preferences/`). The system preferences are stored in `/Library/Preferences/`.
- To be consistent with the Mac OS X user experience, your preferences should be available from the application menu. The `com.apple.eawt.Application` class provides a mechanism for doing this. See Java 1.4 API: Apple Extensions and J2SE 5.0 Apple Extensions Reference for more information.

JNI

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`.

If you are developing a Cocoa Java application, you need to load your JNI library using a different mechanism. If your library is called `hello.jnilib`, you should call `System.load(NSBundle.mainBundle().pathForResource("hello", "jnilib", "Java"));` Note that this assumes that your library is located in `Resources/Java/`.

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 Mac OS X version 10.0, you should build them as a bundle; otherwise you probably want to build them 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 easier to build if you have multiple libraries to link together.

To build as a dynamic shared library, use the `-dynamiclib` flag. Since your `.h` file produced by `javah` 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 that will 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()`.

Such an interdependency is not a problem if you build your JNI libraries as dynamic shared libraries, but if you build them as bundles it 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
   ```

A complete example of calling a dynamic shared library from a bundle, including both a makefile and an Xcode project, can be found in the *MyFirstJNIProject* sample code. More details on JNI can be found in Tech Note TN2147: JNI Development on Mac OS X.

Note: When building JNI libraries, you need to explicitly designate the path to the `jni.h`. This is in `/System/Library/Frameworks/JavaVM.framework/Headers/`, not `/usr/include/` as on some other platforms.

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 or by putting them in `/Library/Java/Extensions/`.

The Java VM for Mac OS X

The foundation of any Java implementation is the Java virtual machine (VM). The Java implementation for Mac OS X includes the Java HotSpot VM runtime and the Java HotSpot client VM from Sun. The VM options available with the Java VM in Mac OS X vary slightly from those available on other platforms. The available options are presented in Java Virtual Machine Options.

Basic Properties of the Java VM

Table 1 lists the basic properties of the Java VM in Mac OS X. You can use `System.getProperties().list(System.out)` to obtain a complete list of system properties.

Table 1 JVM properties

Property	Sample Value	Notes
`java.version`	1.4.2	Mac OS X version 10.1 and earlier ship with earlier versions of Java. Use this property to test for the minimal version your application requires.
`java.vm.version`	1.4.2_07	
`file.seperator`	'/'	Note that this is a change from Mac OS 9.
`line.separator`	'\n'	This is consistent with UNIX-based Java implementations, but different from Mac OS 9 and Windows.
`os.name`	Mac OS X	Make sure to check for `Mac OS X`, not just `Mac OS` because `Mac OS` returns `true` for Mac OS 9 (and earlier) which did not even have a Java 2 VM.
`os.version`	10.4	Java 1.4 runs only in Mac OS X version 10.2 or later.

Note : The `mrj.version` system property is still exposed by the VM in Java 1.4.2. Although you may still use this to determine if you are running in the Mac OS, for forward compatibility consider using the `os.name` property to determine if you are running in the Mac OS since this property may go away in future attempts to further synchronize the Apple source with the source from Sun.

Mac OS X Java Shared Archive

To help increase the speed of application startup and to reduce the memory footprint of Java applications, the Java VM in Mac OS X makes use of a Java shared archive (JSA). The JSA contains the preprocessed internal HotSpot representations of common standard Java classes that would otherwise be found and processed from the standard `classes.jar` file. Since this data doesn't change, it can be shared across processes by mapping the JSA file directly into shared memory. The JSA file is created upon first system boot after installation of Java. Since each Java application does not need to generate an independent archive at runtime, less memory and fewer CPU cycles are needed. There is nothing you need to do programatically to take advantage of these advances. A brief description of this technology is provided here.

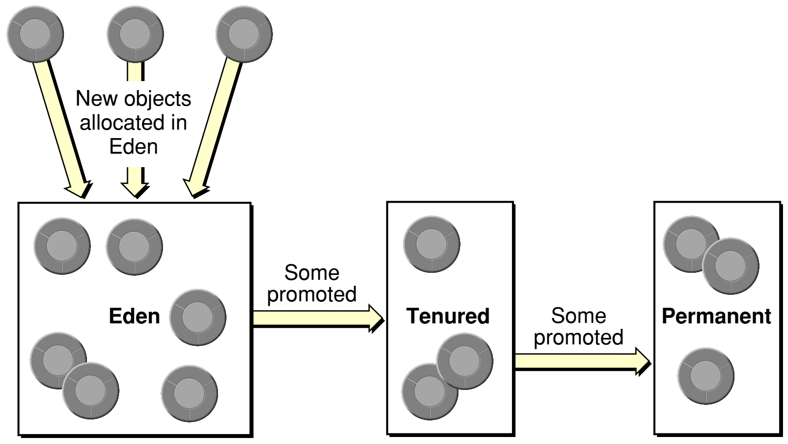
Generational Garbage Collection

Garbage collection (GC) is the means whereby the Java VM gets rid of objects that are no longer in use. Java provides an environment to the programmer where objects and the memory that they use are automatically reclaimed without programmer intervention. This runtime facility, GC, has been under development for decades. The basic ideas is that there is a set of root objects that reference other objects which in turn reference other objects and so on. If an object cannot be reached from one of these root objects, it is declared garbage and its memory space is reclaimed. Searching all objects has always been expensive. About twenty years ago the idea of generational garbage collection was introduced.

Generational GC takes advantage of the fact that most objects die young. That is, most objects are only in use for a short amount of time, a string buffer, for example. In generational GC, a section of memory is set aside where new objects are created. After this space is filled, the objects are copied to another section of memory. Each of these sections is referred to as a generation. The HotSpot VM keeps track of these objects so that it can find the connections back to the root objects when GC occurs.

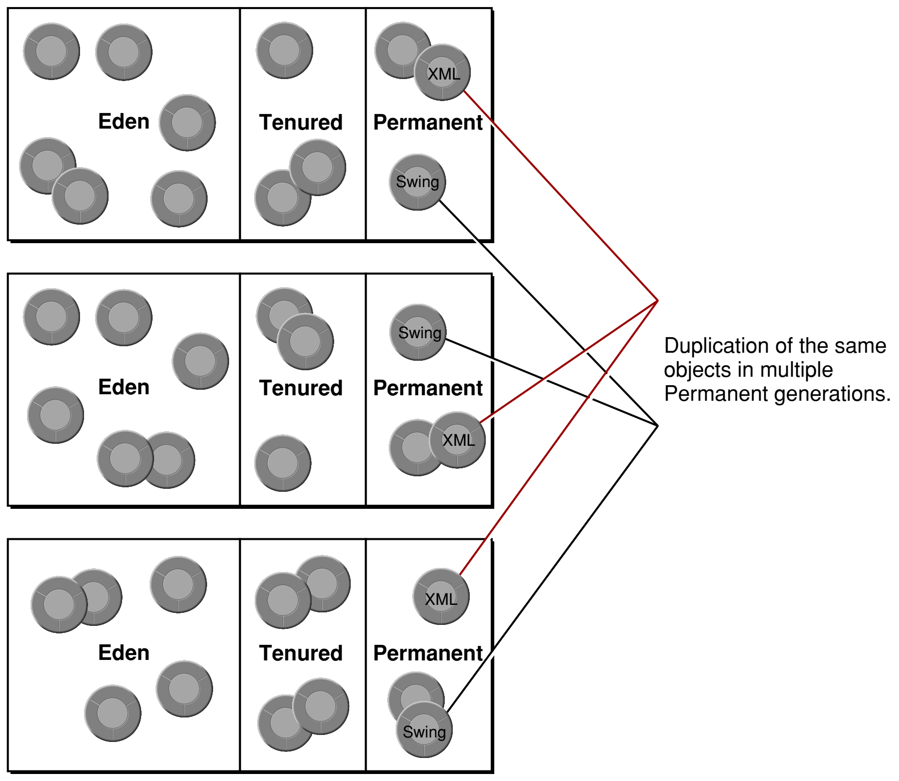
When a GC is run, objects still in use may move up to a more tenured generation. Most generations have their own strategy for culling out objects no longer in use without having to search through all of the memory space. The oldest generation, however, has no such strategy. This default scenario is shown in Figure 1 (page 47) .

Figure 1 Default generational garbage collection



One problem with this is that when garbage collection is run on the permanent generation it takes a while to go through all of that memory space. Moreover, over the course of time, multiple application instances will end up with many of the same objects in their respective permanent generation. If a user is running multiple Java applications, then the permanent generation of one often has the same resources as another which results in wasted memory. Figure 2 (page 47) illustrates the duplication of resources.

Figure 2 Duplication of resources in the Permanent generation



More information on the Java HotSpot VM garbage collection is available online in Sun's Technical White Paper, *The Java HotSpot Virtual Machine, v1.4.1* available at http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/Java_HSpot_WP_v1.4.1_1002_1.html and in *Tuning Garbage Collection with the 1.3.1 Java™ Virtual Machine* available at http://java.sun.com/docs/hotspot/gc/index.html .

The Advantages of the Java Shared Archive

To get around the problem of wasted memory in the permanent generation, Apple implements a Java Shared Archive technology. This technology is similar to the concept of a shared library in C-based languages. Instead of each Java application needing to keep a separate copy of resources that usually end up in the permanent generation, they are able to share a single copy of these resources which adds a new immortal generation. There are some distinctions between this immortal generation and the other generations:

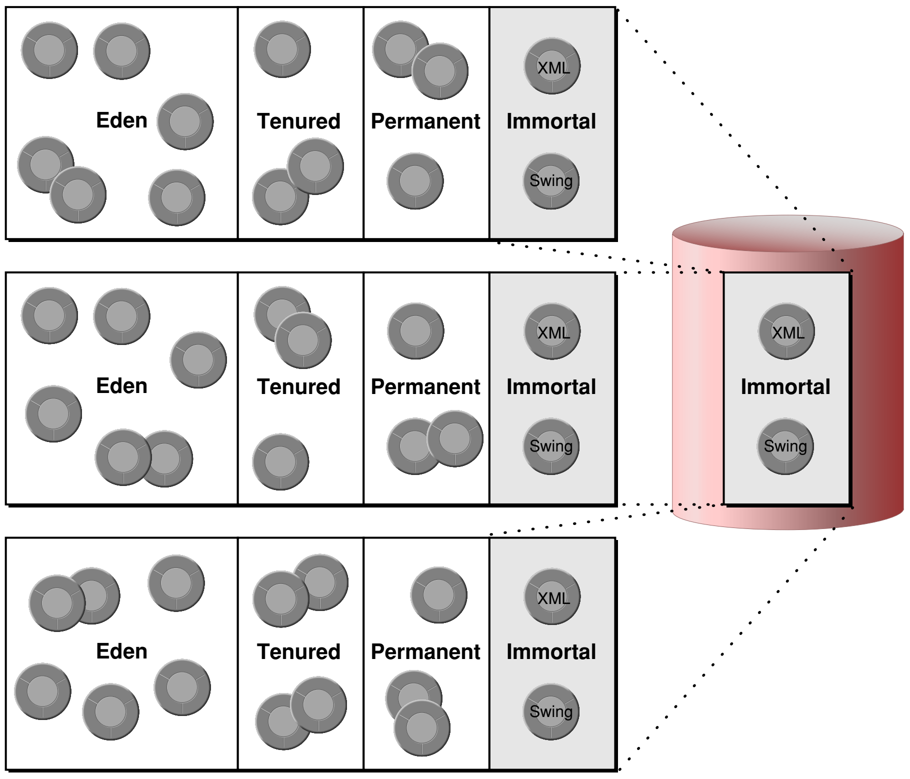
- Resources are explicitly placed there, they are not promoted to that generation.
- It is never garbage collected.
- A single immortal generation is shared by multiple applications.

When Mac OS X starts up, the Java VM starts in a special mode to build (or update) an archive of resources that would likely end up in the `Permanent` generation.

After this archive file has been appropriately assembled, the VM shuts down until the user runs a Java application. When a user starts a Java application, the VM can bootstrap off of those resources already on disk, mapping them into the same memory space as the Java heap. Using the archive file speeds up application launch time.

Shorter launch time is a benefit for a single Java application running in Mac OS X. When multiple applications are running, a shared archive also uses less memory. Subsequent applications do not need to allocate space for the resources in the Java Shared Archive; the space has already been allocated on disk. They just need to map to it. As Figure 3 (page 49) shows, the result is similar to a shared library. The permanent generation tends to be smaller since many of the objects that would be in the permanent generation are available in the Immortal generation.

Figure 3 The Java shared archive implementation of the immortal generation



Apple's Java Shared Archive implementation in Mac OS X is fully compatible with the Java HotSpot specification and has been presented to Sun for further use within Java.

Note : The Java Shared Archive is used only if the default boot class is used. The use of `java.endorsed.dirs` or otherwise modifying the default boot class path prevents your application from using the Java Shared Archive.

In summary, the VM uses the generational heap to allocate internal data structures used to implement Java classes. These classes don't change over time and don't ever need to be garbage collected. The Mac OS X Java VM extends HotSpot's notion of generations to include a new generation for these immortal objects. Along with memory savings, time is also conserved since this generation never needs to be searched for dead objects.

Mac OS X Integration for Java

The more your application fits in with the native environment, the less users have to learn unique behaviors for using your application. A great application looks and feels like an extension of the host platform. This chapter discusses a few details that can help you make your application look and feel like it is an integral part of Mac OS X.

Making User Interface Decisions

J2SE's cross-platform design demands a lot of flexibility in the user interface to accommodate multiple operating systems. The Aqua user interface, on the other hand, is streamlined to provide the absolute best user experience on just one platform. This section helps you to make decisions that let your Java applications approach the appearance and performance characteristics of native applications. In fact, following some of the suggestions presented here can probably help make your application appear and perform more like native applications on other platforms as well. The topics covered here represent just a small subset of design decision topics, but they are high-visibility issues often present in Java applications. The complete guidelines for the Aqua user interface can be found in *Apple Human Interface Guidelines*.

Menus

The appearance and behavior of menu items varies across platforms. Unfortunately, many Java programmers write their applications with only one platform in mind. This section discusses some common areas to improve how your Java menus are displayed and perform in Mac OS X.

The Menu Bar

"JMenuBars" (page 31) discusses how to get the menu out of JFrames and into the Mac OS X menu bar. Having menus in the menu bar is highly encouraged, but it does not automatically provide the native experience of Mac OS X menus for two reasons:

- In Mac OS X an application's menu bar is always visible when an application is the active application, whether or not any windows are currently open.

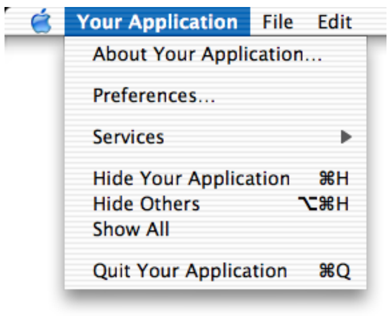
- The menu bar contains all of the menus that an application might use. If a menu has no functionality with the foremost window, the title for that menu is dimmed. Also, the titles of menus that have only menu items that apply to the content of windows are dimmed if no windows are open. Menus do not appear and disappear based on which window is foremost.

In short, the menu bar is always present and, except that some items may be dimmed at times, always looks the same. Removing the menus from your windows and putting them in the menu bar is a great first step that retains cross-platform compatibility. Depending on how your application is designed, getting these other characteristics may require rethinking how you display menus with your code. One solution is to use a menu factory with an off screen JFrame that is always foremost when your application is active.

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 in 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 `-Xdock:name` command-line property, or it can be set in the information property list file for your application as the `CFBundleName` value. For more on Info.plist files, see "A Java Application's Info.plist File" (page 25). According to the Aqua guidelines, the name you specify for the application menu should be no longer than 16 characters. Figure 1 (page 52) shows an application menu.

Figure 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 certain items in the application menu are chosen. Apple provides functionality for this in the `com.apple.eawt` package. The Application and ApplicationAdaptor classes provide a way to handle the Preferences, About, and Quit items.

For more information see Java 1.4 API: Apple Extensions and J2SE 5.0 Apple Extensions Reference. Examples of how to use these can also be found in a default Swing application project in Xcode. Just open a new project in Xcode by selecting Java Swing Application in the New Project Assistant. The resulting project uses all of these handlers.

If your application is to be deployed on other platforms, where Preferences, Quit, and About are elsewhere on the menu bar (in a File or Edit menu, for example), you may want to make this placement conditional based on the host platform's operating system. Conditional placement 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 in Mac OS X.

The Window Menu

Apple Human Interface Guidelines suggests 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 windows, with a checkmark next to the active window. 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 are opened. *Apple Human Interface Guidelines* has more specific guidance on the Window menu.

Accelerators (Keyboard Shortcuts)

Do not set menu item accelerators with an explicit `javax.swing.KeyStroke` specification. Modifier keys vary from platform to platform. Instead, use `java.awt.Tookit.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, using `getMenuShortcutKeyMask` means that you can replace the complexity of "The Contents of an Application Bundle" (page 23) with the simplicity of Listing 2 (page 53).

Listing 1 Explicitly setting accelerators 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 2 Using getMenuShortcutKeyMask to set modifier keys

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

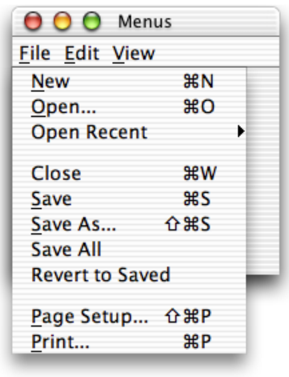
The default modifier key in Mac OS X is the Command key. There may be additional modifier keys like Shift, Option, or Control, but the Command key is the primary key that alerts an application that a command, not regular input follows. Note that not all platforms provide this consistency in user interface behavior. When assigning keyboard shortcuts to items for menu items, make sure that you are not overriding any of the keyboard commands Macintosh users are accustomed to. See *Apple Human Interface Guidelines* for the definitive list of the most common and reserved keyboard shortcuts (keyboard equivalents).

Mnemonics

JMenuItems inherit the concept of mnemonics from JAbstractButton. In the context of menus, mnemonics are shortcuts to menus and their contents using a modifier key in conjunction with a single letter. When you set a mnemonic in a JMenuItem, Java underscores the mnemonic letter in the title of the `JMenuItem` or `JMenu`. The result looks something like that shown in Figure 2 (page 54).

Note: Figure 2 (page 54) is provided only to illustrate mnemonics. Although the figure illustrates the menu bar inside of the window, you should put your menus in the normal Mac OS X menu bar with the `apple.laf.useScreenMenuBar` system property, where mnemonics are never drawn. See Java System Properties.

Figure 2 Mnemonics in Mac OS X



This does not fit in with the Aqua guidelines for multiple reasons. Among them:

- It is extraneous information. The shortcut is already defined to the right of the menu item.
- It is imprecise. Note in this example that Save and Save As both have the letter S underlined.
- It clutters the interface.

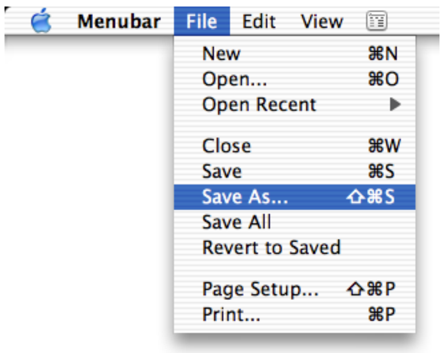
The problem is partially handled for you automatically if you use the `apple.laf.useScreenMenuBar` system property. A better solution though is for you to not to use mnemonics for Mac OS X. If you want mnemonics on another platform, just include a single `setMnemonics()` method that is conditionally called (based on the platform) when constructing your interface.

How then do you get the functionality of mnemonics without using Java's mnemonics? If you have defined a keystroke sequence in the `setAccelerator` method for a menu item, that key sequence is automatically entered into your menus. For example, Listing 3 (page 54) sets an accelerator of Command-Shift-P for a Page Setup menu. Figure 3 (page 55) shows the result of this code along with similar settings for the other items. Note that the symbols representing the Command key and the Shift key are automatically included.

Listing 3 Setting an accelerator

```
JMenuItem saveAsItem = new JMenuItem("Save As...");
    saveAsItem.setAccelerator(
        KeyStroke.getKeyStroke(KeyEvent.VK_S,
(java.awt.event.InputEvent.SHIFT_MASK |
(Toolkit.getDefaultToolkit().getMenuShortcutKeyMask())));
    saveAsItem.addActionListener(new ActionListener(  ) {
      public void actionPerformed(ActionEvent e) { System.out.println("Save
As...") ; }
    });
    fileMenu.add(saveAsItem);
```

Figure 3 A File menu



Though not quite the same as mnemonics, note that Mac OS X provides keyboard and assistive-device navigation to the menus. Preferences for these features are set in the Keyboard and Universal Access panes of System Preferences.

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

Menu Item Icons and Special Characters

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.

Aqua specifies a specific set of special characters to be used in menus. See the information on using special characters in menus in *Apple Human Interface Guidelines*

Contextual Menus

Contextual menus, which are called pop-up menus in Java, are fully supported. In Mac OS X, they are triggered by a Control-click or a right-click. Note that Control-click and right-click are a different type of `MOUSE-EVENT`, even through they have the same result. In Windows, the right mouse button is the standard trigger for contextual menus.

The different triggers 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 "A Java Application's Info.plist File" (page 25).

Listing 4 Using ˜ 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...
       }
    }
});
```

When designing contextual menus, keep in mind that a contextual menu should never be the only way a user can access something. Contextual menus provide convenient access to often-used commands associated with an item, not the primary or sole access.

Components

There are several key concepts to keep mind when designing your user interface. The following sections cover these.

Laying Out and Sizing Components

Do not explicitly set the x and y coordinates when placing components; instead use the AWT layout managers. 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.

In general, do not set component sizes explicitly. Each look and feel has its own font styles and sizes. These font sizes 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 way to make your components a proper size in a portable manner is to change to or add another layout manager, or to set the component's minimum and maximum size to its preferred size. The `setSize()` and `getPreferredSize()` methods are useful when following the latter approach.

Most layout managers and containers respect a component's 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, you 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 application's 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

Mac OS X window coordinates and insets are compatible with the Java SDK. 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. 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 performing precision positioning of windows, especially full-screen windows.

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 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 visually constrained to one area of the screen when using a particular application. When building applications for Mac OS X, try 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 1.4 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. The default behavior of JScrollPane, however, is that scroll bars appear only if the 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 "Making a Java Application Bundle" (page 26).

Listing 5 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 content is 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 4 (page 58) and Figure 5 (page 59) show.

Figure 4 Dialog created with java.awt.FileDialog

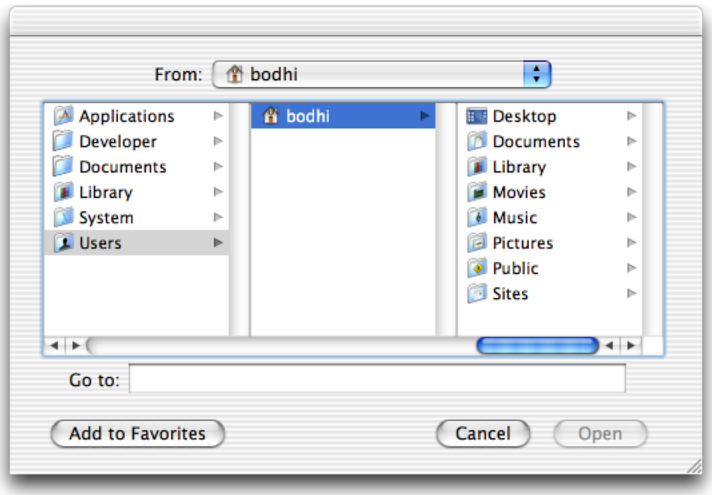
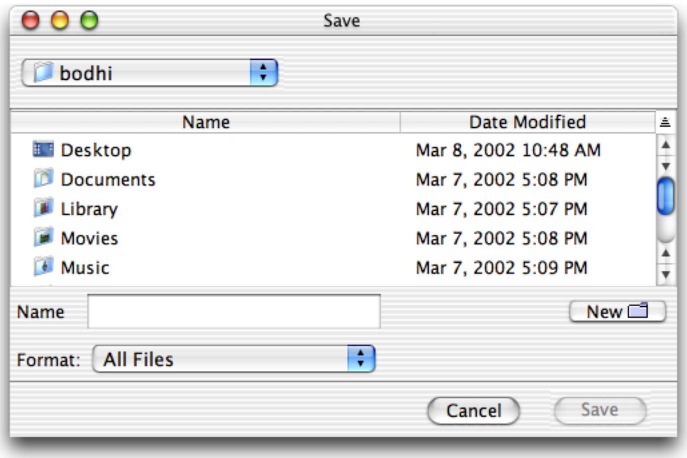


Figure 5 Dialog created with javax.swing.jFileChooser



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, there is not the usual consequence of mixing Swing and AWT components.

When using the AWT FileDialog, you may want a user to select a directory instead of a file. In this case, use the `apple.awt.fileDialogForDirectories` property with the `setProperty.invoke()` method on your FileDialog instance.

Window Modified Indicator

On Mac OS X, it is common for windows to mark their close widget if the window's content has changed since it was last saved. Using this makes your application behave more like a native application and conform to users expectations.

To use a window modified indicator, you need to use the Swing property `windowModified`. It can be set on any subclass of `JComponent` using the `putClientProperty()` method. The value of the property is either `Boolean.TRUE` or `Boolean.FALSE`.

For more on using the Window Modified indicator in your application, review Technical Q&A QA1146: Illustrating document window changes in Swing.

AppleScript

Mac OS X uses Apple events for interprocess communication. Apple events are high-level semantic events that an application can send to itself or other applications. AppleScript allows you to script actions based on these events. Without including native code in your Java application you may let users take some level of control of your application through AppleScript. Java applications can become AppleScript-aware in two ways:

- If you implement the Application and ApplicationAdaptor classes available in the `com.apple.eawt` package. By implementing the event handlers in the ApplicationAdaptor class, your application can generate and handle basic events like Print and Open. Information on these two classes is available in Java 1.4 API: Apple Extensions and J2SE 5.0 Apple Extensions Reference.
- Through the GUI scripting made available in the System Events application. If you use System Events, you can write AppleScript scripts to choose menu items, click buttons, enter text into text fields, and generally control the GUI of your Java applications.

For more on AppleScript, see *Getting Started with AppleScript*.

System Properties

The Mac OS X–specific system properties let you easily add native behavior to your Java applications with a minimum of work on your part. A complete list of supported Mac OS X system properties, including how to use them, is available in Java System Properties.

Document Revision History

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

Date	Notes
2006-05-23	Updated content to include information for J2SE 5.0 Release 4 for Mac OS X.
2006-01-10	Fixed typos throughout the document.
2005-10-04	Fixed various errors and inconsistencies.
2005-04-29	Updated content to include information for J2SE 5.0 Release 1 for Mac OS X. Document renamed Java Development Guide for Mac OS X.
	Updated with information about Java on Mac OS X v10.4.
2004-11-02	Minor revisions and corrections throughout the document.
2004-08-31	Revised for Java 1.4.2. Updated links to reflect documentation changes.
2003-06-23	Removed appendices. They are now available as separate documents. Minor corrections in the overview chapter. Spelling and grammatical errors fixed.
2003-05-15	Revised for Java 1.4.1. Most sections are completely new to reflect the completely new Java implementation. Only the user experience information has been retained although it has been updated as well. Structure of the document was modified dramatically to align with Sun's presentation of the Java 2 platform.
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.