Java Development Guide for Mac OS X

Java

2008-10-15

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

Contents

Introduction 7

Who Should Read This Document? 7
Organization of This Document 7
See Also 8
Filing and Tracking Bugs 8

Overview of Java for Mac OS X 9

Java and Mac OS X 9
Java, Built In 9
32-Bit and 64-Bit Java 10
The Aqua User Interface 10
Finding Your Way Around 12
    The Java Home Directory 12
    Java Extensions 12
    Output from Java Programs 13
The File System 13

Apple Developer Tools for Java 15

JDK Tools in Mac OS X 15
Java IDEs 15
Xcode Tools 16
    Get the Current Tools 16
    Xcode 16
    Jar Bundler 18
    Applet Launcher 18
Other Tools 18
Developer Documentation 19
    Providing Documentation Feedback 19

Java Deployment Options for Mac OS X 21

Java Web Start 21
Mac OS X Application Bundles 22
    The Contents of an Application Bundle 22
    A Java Application's Information Property List File 23
    Making a Java Application Bundle 24
    Localizing Java Applications 25
    Distributing Application Bundles 26
    Making a Mac OS X Java Application Bundle on other Platforms 26

Double-Clickable JAR Files 26
The Java Plug-in 27

Mac OS X Integration for Java 29

Making User Interface Decisions 29
Working with Menus 29
Designing for Component Layout, Size, and Color 33
Working with Windows and Dialogs 34
Apple Events and AppleScript 37
System Properties 37

User Interface Toolkits for Java 39

Swing 39
Menu Bars (JMenuBar) 39
Tabbed Panes (JTabbedPane) 40
Component Sizing 40
Buttons 41
Abstract Window Toolkit (AWT) 42
Character Encoding 43
Accessibility 43
Security 43
Sound 44
Input Methods 44
Java 2D 44
Resolution Independence 45

Core Java APIs and the Java Runtime on Mac OS X 47

Networking 47
Preferences 47
JNI 47
The Java Runtime 49

Document Revision History 51

Figures, Tables, and Listings

Overview of Java for Mac OS X 9

Figure 1 Apple's Aqua look and feel and the standard Java cross-platform look and feel in Mac OS X 11

Apple Developer Tools for Java 15

Figure 1 The Xcode Organizer 17

Java Deployment Options for Mac OS X 21

Figure 1 Contents of a Java application bundle 23
Figure 2 Jar Launcher error 26
Listing 1 A Sample JNLP file 21

Mac OS X Integration for Java 29

Figure 1 Application menu for a Java application in Mac OS X 30
Figure 2 A File menu 32
Figure 3 Dialog created with `java.awt.FileDialog` 35
Figure 4 Dialog created with `javax.swing.JFileChooser` 36
Listing 1 Explicitly setting accelerators based on the host platform 31
Listing 2 Using `getMenuShortcutKeyMask` to set modifier keys 31
Listing 3 Setting an accelerator 31
Listing 4 Detecting contextual-menu activation 33
Listing 5 Setting `JScrollBar` policies to be more like those of Aqua 35
Listing 6 Invoking AppleScript with the `javax.script` API 37

User Interface Toolkits for Java 39

Figure 1 Tabbed panes with multiple tabs in Mac OS X and Windows 40
Figure 2 An oversize `JComboBox` component in Windows 41
Figure 3 An oversize `JComboBox` component in the Aqua LAF 41

Core Java APIs and the Java Runtime on Mac OS X 47

Table 1 JVM properties 49

Introduction

The Java Platform, Standard Edition (or Java SE, formerly known as J2SE) for Mac OS X provides a Java environment that is highly integrated 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 version 10.5 includes J2SE 5.0 right out of the box and provides Java SE 6 as a free 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 ensure that your application's user experience adheres to conventions and behaviors that Mac users have come to expect from their applications. This document seeks to highlight these methods so you can spend your time writing applications instead of troubleshooting.

Who Should Read This Document?

This document is for the Java developer interested in writing Java applications in Mac OS X v10.5 with J2SE 5.0 or Java SE 6. This document is primarily for developers of pure Java applications, but it may also be useful for WebObjects development.

This is not a tutorial for the Java language. This document assumes you have a basic understanding of Java development and Java development environments. 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 9) describes the Java platforms available on Mac OS X.
- "Apple Developer Tools for Java" (page 15) introduces you to the Apple suite of developer tools, along with recommended tools from other manufacturers.
- "Java Deployment Options for Mac OS X" (page 21) discusses how you can distribute your Java application on Mac OS X.
- "Mac OS X Integration for Java" (page 29) provides you with some handy tips for making your Java application act and feel more like a native Mac OS X application.
- "User Interface Toolkits for Java" (page 39) shows you the different user interface elements common in Mac OS X.

- "Core Java APIs and the Java Runtime on Mac OS X" (page 47) discusses the how core Java APIs vary on Mac OS X.

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/Documentation/DocSets/` 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 website, 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 SE Runtime Environment (JRE) as well as the Java SE Development Kit (JDK). More details about JDK in Mac OS X are provided in "Java Deployment Options for Mac OS X" (page 21).

The following sections give a high-level overview of how Java for Mac OS X is different from Java for other platforms.

Java, Built In

"Write once, run anywhere" is true only if Java is everywhere. With Mac OS X, you know the 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.

Java is the only high-level framework on Mac OS X besides Cocoa that provides a graphical toolkit for building applications. 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 29). Users don't need to learn different behaviors for Java applications—in fact, they shouldn't even know that applications are Java applications.

Apple provides multiple versions of Java built into Mac OS X to offer your customers the widest range of compatibility with your applications. You are encouraged to create applications that target the oldest possible Java version but launch with the newest available version. In this way, you accommodate the largest possible audience, and at the same time take advantage of the speedups and operating system integration that later versions afford.

Note: There is no redistribution license for Java in Mac OS X. If your customers need a specific update of the Java runtime 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/.

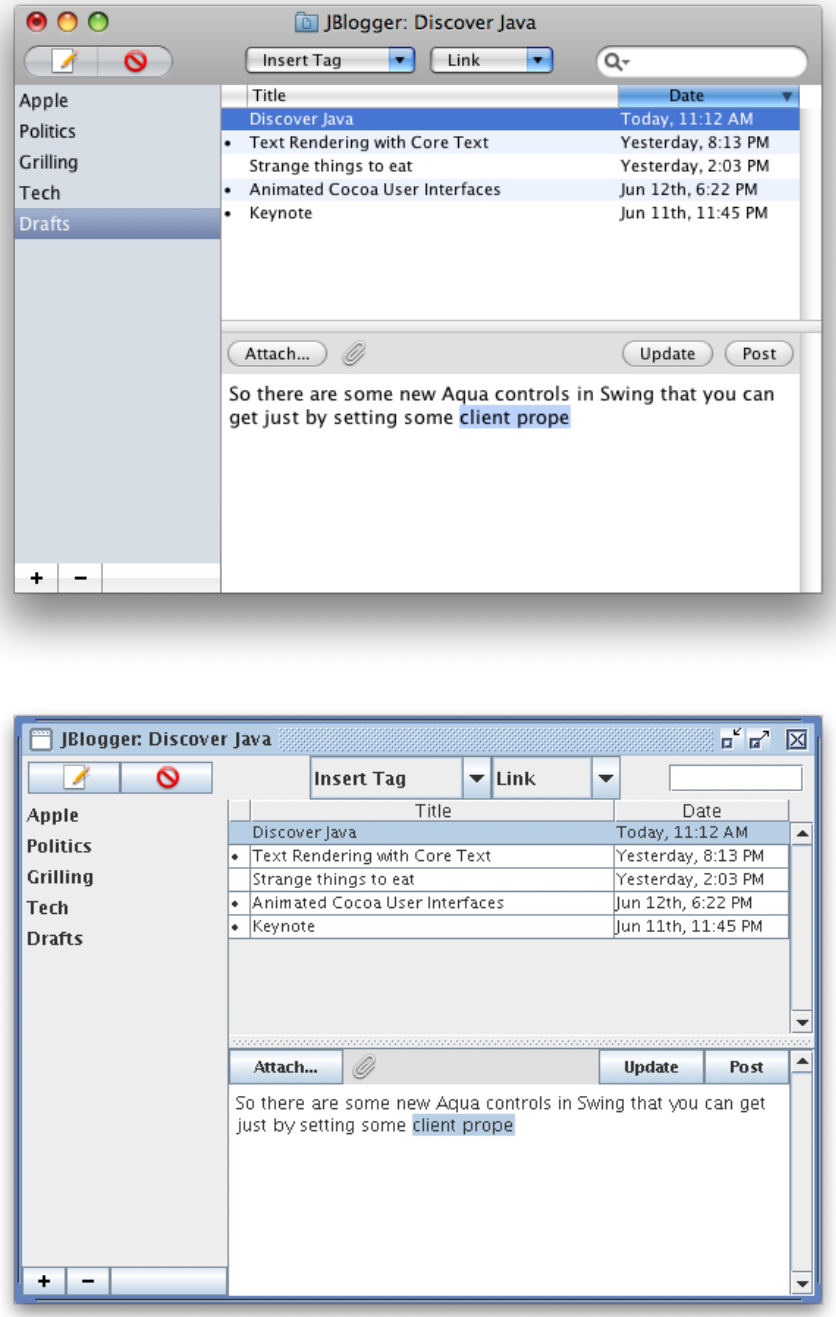
32-Bit and 64-Bit Java

Mac OS X v10.5 includes both a 32-bit and a 64-bit version of J2SE 5.0, along with a 64-bit version of Java SE 6. It is important to note that certain Apple APIs, such as QuickTime for Java (QTJ), are compatible only with 32-bit versions of Java. Similarly, considerations should be made when writing Java Native Interface (JNI) libraries, because the architecture of the library must correspond to the version of the code you are interfacing with.

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 Mac OS X and Java elsewhere. Figure 1 shows this distinction by showing the cross-platform 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 1 Apple's Aqua look and feel and the standard Java cross-platform look and feel in Mac OS X



By default, 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 cross-platform 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 cross-platform LAF, modify your code to include `UIManager.setLookAndFeel(UIManager.getCrossPlatformLookAndFeelClassname())`. Further details on the Aqua LAF are provided in "User Interface Toolkits for Java" (page 39).

Finding Your Way Around

One of the first hurdles newcomers to Java development on 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 navigating 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. By default, the `CurrentJDK` symlink points to the `1.5.0` directory. This is where the code that actually implements J2SE 5.0 resides.

Although the purposes of the files within the `JavaVM.framework` directory are interesting from the perspective of how Java is implemented in Mac OS X, you should consider the contents of the directory opaque for both you and your customers. Additionally, do not rely on a particular path within the `JavaVM.framework` directory in any code that you ship to customers, because the directory's contents will change with updates to Java and the operating system.

The Java Home Directory

Some applications look for Java's home directory (`$JAVA_HOME`) on the user's system, especially during installation. If you need to set this explicitly in a shell script or an installer, set it to `/Library/Java/Home/`. Setting it to the target of this symbolic link can result in a broken application for your customers down the road, when Apple ships a software update that changes the default version of Java, or when the user moves the application to another version of Mac OS X which has a different default version of Java. Programatically you can use `System.getProperty("java.home")`, as you would expect.

`/Library/Java/Home/` also contains the `bin/` subdirectory where command-line tools like `java` and `javac` are found. These tools match the default version of Java for the system as defined by Apple. Additionally, the Java tools available on the default path in `/usr/bin` will dynamically target the top preferred version of Java that the user has chosen for applications in the Java Preferences application.

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 the Java Virtual Machine.

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 the Java Preferences application (see "Other Tools" (page 18) for information on Java Preferences.).

The File System

The default file system 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.png` and `MyBigImage.png` in the same directory can create unpredictable results. Note that while most UNIX-based operating systems are case-sensitive, Windows is case-insensitive so this is a general guideline for any cross-platform Java development.

Note: Mac OS X versions 10.4 and 10.5 allow HFS+ volumes that are fully case-sensitive. Since this is only an option that is chosen at install time and the traditional behavior described above is the default, do not rely on case-sensitivity.

Details about how HFS+ relates to character encoding can be found in "Character Encoding" (page 43).

Apple Developer Tools for Java

This article provides a broad overview of recommended tools for Java development. It covers integrated development environments (IDEs) from other manufacturers, Apple's own Xcode IDE, the Jar Bundler application, and methods for obtaining and viewing documentation.

JDK Tools in Mac OS X

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

- The installed location of the JDK command-line tools is different in Mac OS X. These tools are installed with the rest of `JavaVM.framework` in `/System/Library/Frameworks/`. The Java tools provided in the default path in `/usr/bin/` will execute the version of Java the user has selected as their preferred version for applications in Java Preferences. For more on Java Preferences, see "Other Tools" (page 18). For more information on overall differences in where Java components are in Mac OS X, see "Finding Your Way Around" (page 12).
- `tools.jar` does not exist. Classes usually located here are instead included in `classes.jar`. Scripts that rely on the existence of `tools.jar` need to be rewritten accordingly.

Java IDEs

Java development on any platform often benefits from the use of an Integrated Development Environment (IDE), which provides a more fluid workflow between writing, compiling, running, debugging, and packaging Java code than a simple text editor and the command line. Different IDEs offer unique features and are often suited for different kinds of Java development. These IDEs are industry leaders and offer substantial support for Mac OS X:

- Eclipse IDE for Java Developers (http://www.eclipse.org) is a free download.
- Netbeans IDE (http://www.netbeans.org) is a free download.
- Jetbrains IntelliJ IDEA (http://www.jetbrains.com/idea/) requires a license for continued use after a trial period.
- Xcode (http://developer.apple.com/tools/xcode/) is a free download with a free account from the Apple Developer Connection.

If you are developing a JNI library or intend to have your application communicate with Cocoa, you should plan to use the Xcode Tools for those portions of your development. For more information on JNI development, see "JNI" (page 47).

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 for download at the Apple Developer Connection (ADC) Member Site http://connect.apple.com/. 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/products/.

Get the Current Tools

Apple frequently releases updates to both the Mac OS X Developer Tools and developer documentation. Even if you already have the Xcode Tools installed, you should check the Member Site for the most up-to-date versions of both.

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

With the Xcode Tools and the Java documentation updates, 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/Documentation/`

To remain current with Apple documentation between updates to the Xcode snapshot, visit the Mac Dev Center Reference Library at http://developer.apple.com. The Reference Library includes RSS feeds that announce documentation updates for many technologies, including Java. Subscribe to these feeds at http://developer.apple.com/rss.

Xcode

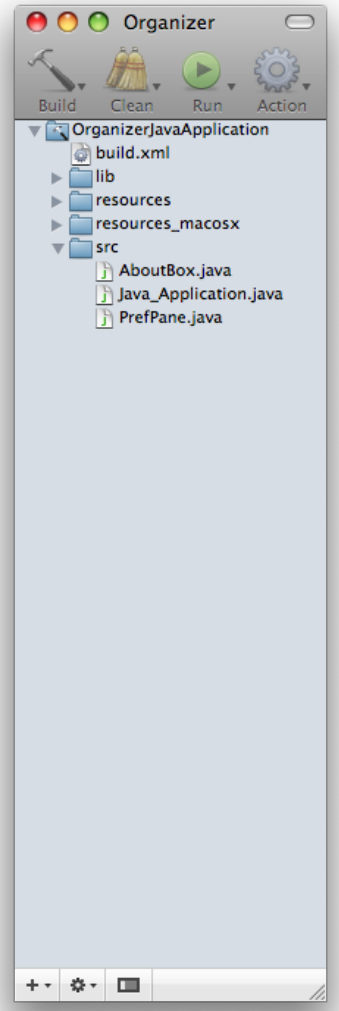
The core component of the Mac OS X development environment is Xcode. Xcode is a complete IDE that allows you to edit, compile, debug, and package Mac OS X applications written in multiple languages. Even if you do not intend to use it for your primary Java development, it is helpful to become familiar 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 the Xcode Help menu.

The Xcode Organizer

Xcode helps you manage Java applications with the Organizer. You can open it by choosing Organizer from the Window menu. Figure 1 shows the Organizer window.

Figure 1 The Xcode Organizer



The Organizer shows your project exactly as it is laid out in the file system. This is in contrast to the main Xcode project windows, which allow you to arrange files arbitrarily without altering their location on disk. The Organizer's direct reflection of the file system better serves Java development and is similar to other Java IDEs.

To create a new Java project in Xcode, choose New From Template from the New menu in the bottom-left corner of the Organizer.

Xcode and Ant

Xcode uses Apache Ant to compile and run Java applications. You can customize your build settings by modifying the `build.xml` file that is automatically generated when you create a new Java project. By default, the `source` and `target` compiler flags in `build.xml` are set to `1.3` and `1.2`, respectively. This is to ensure compatibility with as many Java versions as possible. Raise these default values to take advantage of APIs and features, such as assertions and generics, that are available only with later versions of Java.

Jar Bundler

Jar Bundler is an application that takes Java applications deployed as standalone Jar files and turns them 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/Utilities/`. More information on Jar Bundler is available in *Jar Bundler User Guide*.

Applet Launcher

Applet Launcher (in `/Developer/Applications/Utilities/`) 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 testing your applets in Mac OS X. Performance and behavior settings for applets may be adjusted in the Java Preferences application installed in `/Applications/Utilities/Java/`.

Other Tools

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

- *Java Preferences* for specifying settings for all Java applications, plug-ins, and applets. When you specify a new preference for a default Java runtime in Java Preferences, the `java` tool in `/usr/bin` will dynamically launch that runtime.
- *Java Web Start*, to allow you to launch and modify settings for JNLP-aware Java Web Start applications
- *Input Method HotKey* to set the keyboard combination that invokes the input method dialog in applications with multiple input methods

In addition to containing Xcode and Jar Bundler, `/Developer/Applications/Utilities/` contains some applications that you can use for Java development though they are not Java-specific:

- Package Maker helps you create an installer PKG for your application.
- File Merge provides a graphical interface for comparing and merging source files.
- Icon Composer helps you create an ICNS file for your application bundle.

Additional development tools for Java and other languages can be found in `/usr/share/`. Of particular use for Java development are:

- JUnit, a common Java unit-testing framework.
- Apache Ant, a tool for automating builds. The Ant executable can also be found in `/usr/bin/`.
- Apache Maven, a tool for consolidating multiple elements of development, including dependency management and release management.

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 when you install the Mac OS X Developer Tools. This documentation is easily accessible in Xcode by selecting Documentation from the Help menu. Man pages for the command-line tools are accessible from the command line `man` program and through the Xcode 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 website.

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 on Java documentation in Mac OS X, please use `Java Documentation (developer)` for Component and `X` as Version.

Java Deployment Options for Mac OS X

When deploying Java applications in Mac OS X, you have access to Java Web Start and the Java applet as you do on other platforms. You may also deploy Java applications as native Mac OS X application bundles. This article discusses these deployment technologies. Make sure you know whether you are using a 32-bit or a 64-bit version of Java to ensure that your application is compatible with the architectures you are writing for.

Java Web Start

Mac OS X 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 can run your application with a single click in their web browser. Java Web Start also automatically updates your application by checking your website for a new version before launch. Listing 1 provides a sample JNLP file that you can modify to accommodate your application.

Listing 1 A Sample JNLP file

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec="1.0+"
codebase="http://developer.apple.com/java/javawebstart/apps/welcome"
href="JWS_Demo.jnlp">
  <information>
    <title>Welcome to Web Start!</title>
    <vendor>Apple Computer, Inc.</vendor>
    <homepage href="http://developer.apple.com/java/javawebstart" />
    <offline-allowed />
  </information>
  <resources>
    <j2se version="1.5+" />
    <jar href="WebStartDemo.jar" />
  </resources>
  <application-desc main-class="apple.dts.javawebstart.DemoMain" />
</jnlp>
```

Mac OS X supports desktop integration with Java Web Start, meaning users can create a local application bundle from any Java Web Start application. The Shortcut Creation setting in Java Preferences controls whether the user is prompted to create an application bundle when opening a Java Web Start application. Bundled Java Web Start applications have all of the benefits of native application bundles, which are described in "Mac OS X Application Bundles."

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:

- Java Web Start on the Mac does not support downloading of additional Java Runtime Environments (JREs). New Java versions are provided by Apple via Software Update to all current Mac OS X customers. Valid version keys for Mac OS X application bundles are the same as those of Java Web Start. A list of these keys can be found 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/` directory.

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 can include 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 deployment 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*.

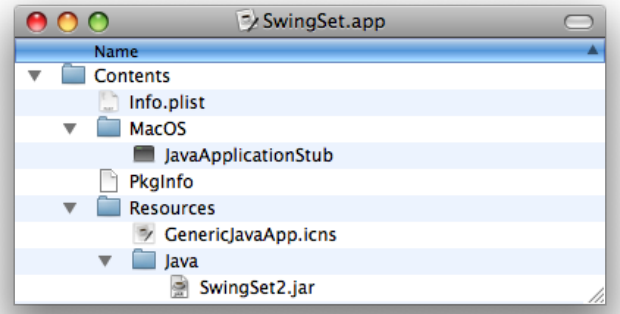
When deploying Java applications in Mac OS X, consider making your Java application into a Mac OS X application bundle. It is easy to do and offers many benefits:

- Users can simply double-click the application to launch it. They can also drag it to the Trash to delete it.
- If you add an appropriate icon, it shows the application icon in the Dock, clearly identifying your application. (Otherwise, a default Java coffee cup icon appears in the Dock.)
- An application bundle lets you easily set Mac OS X–specific system properties that can make your Java application look more like 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 multi-button mouse) an application icon and selecting Show Package Contents. You should see something similar to the directory structure shown in Figure 1.

Figure 1 Contents of a Java application bundle



Applications bundles for Java applications should have the following components:

- An `Info.plist` file in the Contents folder. This contains 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 concatenated with 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 the Resources folder.
- A native executable stub in the MacOS folder that launches the Java VM.
- Optional localized versions of strings may be included in folders designated by the `.lproj` suffix. If your application contains localized strings, you must include corresponding `.lproj` folders in your bundle, even if the strings are in `.properties` files in a Jar. See "Localizing Java Applications" (page 25) 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 Information Property List 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 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. This is most easily done with the Property List Editor application in `/Developer/Applications/Utilities`. Since property lists are simple XML files, you can also modify them with any text editor.

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, a `Java` key designates the beginning of a Java dictionary. This dictionary requires a `MainClass` key and should also include a `JVMVersion` key if your application requires a particular minimum version of Java. 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, you might notice that certain keys are missing from the `Properties` dictionary. This is because 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 Swing application using one of the Xcode Organizer's templates, Xcode automatically generates an application bundle complete with a default `Info.plist` file. You can fine-tune the `Info.plist` file directly in Xcode or with Property List Editor. For more information on using Xcode for Java development, see Xcode Help (available from the Help menu in Xcode).

If you want to turn your existing Java application into a Mac OS X Java application, use the Jar Bundler application available in `/Developer/Applications/Utilities`. 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 from an existing Java application (such as Jar Bundler) 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. Use Icon Composer in `Developer/Applications/Utilities` for help creating your icon file.

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`. For more information on `SetFile`, see the man page.

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

Localizing Java 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 `ResourceBundle`s, 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 `jp.lproj` folder and a `fr.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, and then compress 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.

Making a Mac OS X Java Application Bundle on other Platforms

You can create Java application bundles for Mac OS X on another platform by following the steps outlined for creating a bundle from the command-line, in "Making a Java Application Bundle." Ignore the step involving `JavaApplicationStub` and the step involving setting the bundle bit when bundling on another platform. Bundles created on other platforms are recognized by Mac OS X; however, they lack certain features, such as the resource fork and access control list (ACL) support.

Double-Clickable JAR Files

You can deploy an application as a JAR file, but this method should be used for testing purposes only. This technique requires very few, if any, changes from the JAR files you distribute on other platforms. However, it also has significant drawbacks for your users. Applications distributed as JAR files are given a default Java application icon instead of one specific to the application, and JAR files 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, this deployment method is not recommended.

Double-clickable JAR files launch with the default version of Java. If a JAR file needs to be launched in another version of Java, wrap the JAR file as a Mac OS X application bundle using Jar Bundler and specify the minimum version required in the application bundle's `Info.plist` file.

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 information, the JAR file is not double-clickable and users see an error message like the one shown in Figure 2.

Figure 2 Jar Launcher error

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 the command `jar xvf` *myjar*`.jar`.
2. In the resulting `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, the file looks like this:

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

3. Archive your files again, but this time use the `-m` option with the `jar` command and designate the relative path to the manifest file you just modified, for example, `jar cmf` *`YourModifiedManifestFile`*`.txt` *`YourJARFile`*`.jar *.class`.

This basic example 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.

The Java Plug-in

J2SE 5.0 for Mac OS X includes the Java Plug-in for you to deploy applets in web browsers and other Java embedding applications.

The Applet Launcher application in `/Applications/Utilities/Java/` also launches applets for testing purposes, without using a browser or the applet plug-in. For more information on Applet Launcher see "Applet Launcher" (page 18).

For use in Safari, the `<APPLET>` tag is preferred over the `<OBJECT>` and `<EMBED>` tags.

Mac OS X Integration for Java

The more your application fits in with the native environment, the less users have to learn unique behaviors to use your application. A great application looks and feels like an extension of the platform it runs on. This article 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

Java SE cross-platform design demands a lot of flexibility from 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 in Mac OS X.

This section aims to help you make the right user interface decisions, so that your Java application will look like a Mac OS X native application, and will perform almost as well as one, too. In fact, by following these same suggestions, your application can approach the look and performance of 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 that are often encountered in Java applications. The complete guidelines for the Aqua user interface can be found in*Apple Human Interface Guidelines*. For information on customization options for Swing components in the Aqua Look and Feel, see "*New Control Styles available within J2SE 5.0 on Mac OS X 10.5*".

Working with Menus

The appearance and behavior of menu items varies across platforms. This section offers some techniques for improving how your Java menus are presented, and how they perform, specifically in Mac OS X.

The Menu Bar

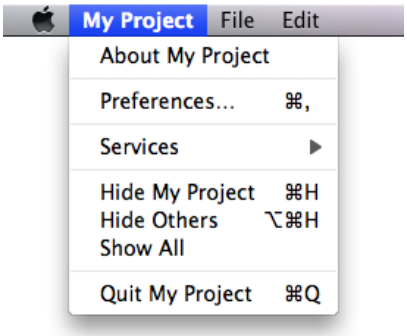
Removing menus from your windows and putting them in the menu bar is highly encouraged, but that approach does not perfectly emulate the native experience of Mac OS X menus. In Mac OS X, a native application's menu bar is always visible when an application is the active application, whether or not any windows are currently open. In Java for Mac OS X, the menus in the menu bar are associated with a top-level frame, and the menus will disappear if the frame closes.

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 Information Property

List File" (page 23). According to the Aqua guidelines, the name you specify for the application menu should be the simplest name of the application (generally no more than 16 characters) and should not include extraneous information like a company name. Figure 1 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 J2SE 5.0 Apple Extensions Reference. Examples of how to use these can also be found in a default Java application project in Xcode. Just open a new project in Xcode by selecting Java Application from the Organizer window. The resulting project uses all of these handlers. For more on the Xcode Organizer, see "The Xcode Organizer" (page 17).

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 should 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 the `java.awt.Tookit.getMenuShortcutKeyMask` method 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. For example, in the case of adding a Copy item to a menu, using `getMenuShortcutKeyMask` means that you can replace the complexity of Listing 1 with the simplicity of Listing 2.

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. When assigning keyboard shortcuts to items for menu items, make sure that you are not overriding any of the keyboard commands that Macintosh users are accustomed to. See *Apple Human Interface Guidelines* for the definitive list of the most common and reserved keyboard shortcuts (keyboard equivalents).

You should make your keyboard shortcuts conditional based on the current platform, because standard shortcuts vary across platforms.

Mnemonics

The `JMenuItem` class inherits the concept of mnemonics from the `JAbstractButton` class. In the context of menus, mnemonics are shortcuts to menus and their contents, which are executed by using a modifier key in conjunction with a single letter. When you set a mnemonic in a menu item, Java underscores the mnemonic letter in the title of the `JMenuItem` or `JMenu` component when the Option key is held down. You are discouraged from using mnemonics in Mac OS X, because they violate the principles of *Apple Human Interface Guidelines*. If you are developing a Java application for multiple platforms and some of those platforms recommend the use of mnemonics, 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 sets an accelerator of Command-Shift-S for a Save As menu.

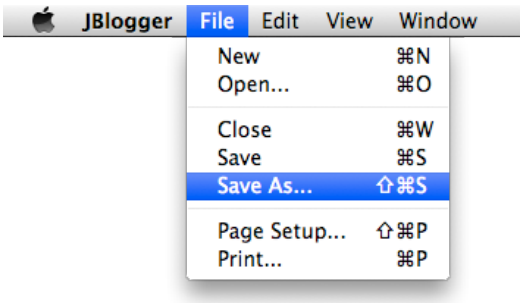
Listing 3 Setting an accelerator

```
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 2 shows the result of this code, along with similar settings for the other items. Note that the symbols representing the Command and Shift keys are automatically included.

Figure 2 A File menu



In addition to accelerators, 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

Menu item icons are available and functional in Mac OS X, via Swing. 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*.

Note: `KeyEvent.getKeyText()` returns the unicode characters of the Command (Meta), Option (Alt), Control, and Shift keys, not the textual descriptions of those keys.

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. Even though both clicks trigger a contextual menu, they are not the same mouse event. In Windows, the right mouse button is the standard trigger for contextual menus.

The different triggers present in Mac OS X 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.

Listing 4 Detecting 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...
       }
    }
});
```

Like the application menu, contextual menus can differ between platforms. You should make the layout of your contextual menus conditional based on the platform.

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.

Designing for Component Layout, Size, and Color

There are several key concepts to keep mind when designing the components in your user interface.

Laying Out and Sizing Components

Do not explicitly set the x and y coordinates of components when placing them; instead make use of 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 preferred 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.

You can create both small and miniature versions of Swing controls in Mac OS X by setting the `sizeVariant` client property. See *New Control Styles available within J2SE 5.0 on Mac OS X 10.5* for more information.

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 approach 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 nonstandard component. To ensure that your custom control matches standard components, query the `UIManager` class for the desired colors. One example 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 call returns the color appropriate for the current look and feel.

Working with Windows and Dialogs

Mac OS X window coordinates and insets are compatible with the JDK. 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` object can provide a confusing user experience in Mac OS X. Therefore when building applications for Mac OS X, try to avoid using this class. Windows minimized in a `JDesktopPane` object move around as the pane changes size. In `JDesktopPane`, windows minimize to the bottom of the pane while independent windows minimize to the Dock. Furthermore, the pane restricts users from moving windows where they want. They are forced to deal with two different scopes of windows, those within the pane and the pane 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.

Windows with Scroll Bars

In Mac OS X, scrollable document windows display a scrollbar whether or not there is enough content in the window to require scrolling (the scroller itself appears only when the content exceeds a window's viewable area). To mimic this behavior in Java applications, place your content inside a scroll pane (`JScrollPane`). You do this because a Swing `JFrame` object by default has no scroll bars, no matter how it is resized. When you use `JScrollPane`, make sure you set the scrollbar policy to always display scrollbars (to mimic Mac OS X), as shown in Listing 5.

Listing 5 Setting `JScrollBar` policies to be more like those of Aqua

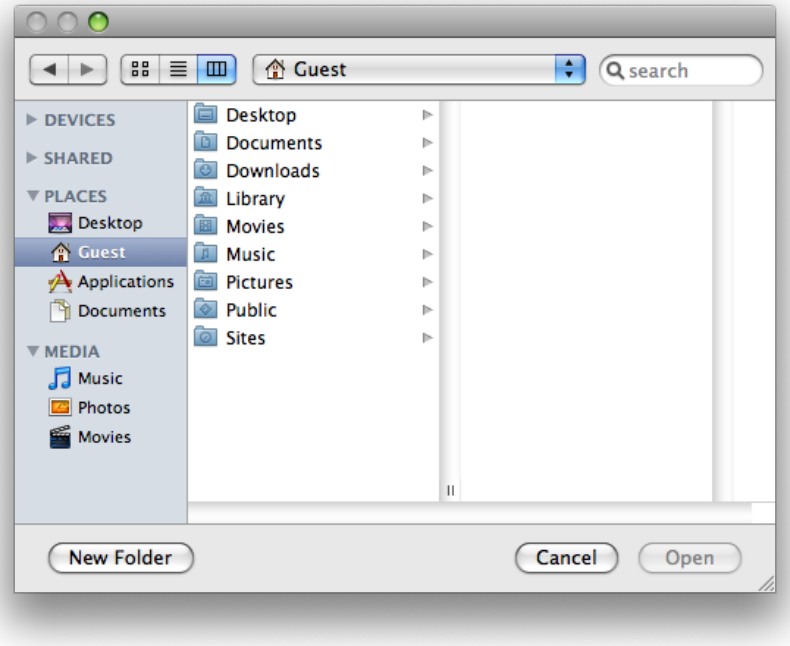
```
JScrollPane jsp = new JScrollPane();
jsp.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
jsp.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
```

When you are using a host platform other than Mac OS X, you may find that the `JScrollBar` default policy, `AS_NEEDED`, more closely resembles its native behavior.

File-Choosing Dialogs

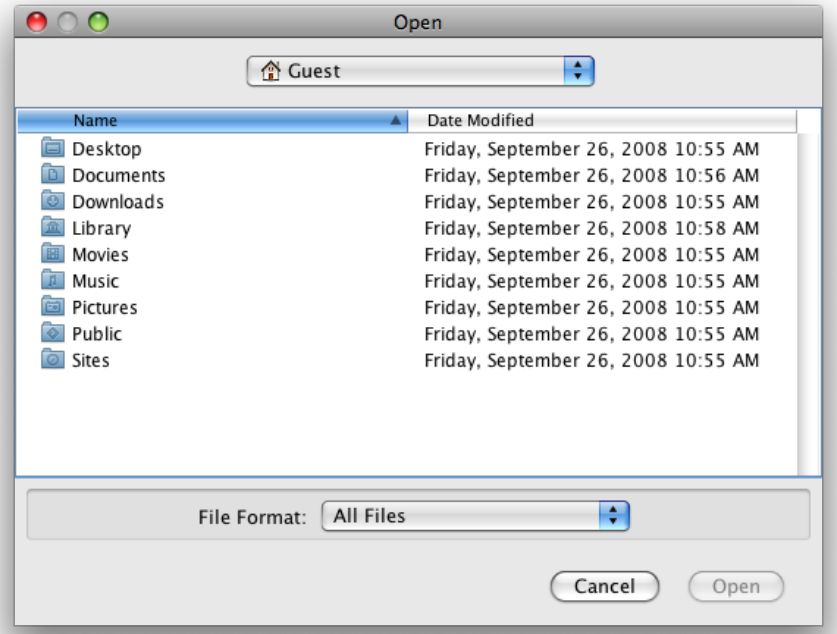
File-choosing dialogs in Java applications are of two main types: dialogs created with `java.awt.FileDialog` and those created with `javax.swing.JFileChooser`. Although each has its advantages, `java.awt.FileDialog` makes your applications behave more like a native Mac OS X application. This dialog, shown in Figure 3, looks much like a Finder window in Mac OS X.

Figure 3 Dialog created with `java.awt.FileDialog`



The Swing dialog, shown in Figure 4, looks much less like a Mac OS X dialog.

Figure 4 Dialog created with `javax.swing.JFileChooser`



Unless you need the functional advantages of `JFileChooser`, use `FileDialog` instead.

When using `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

In Mac OS X, when a document has unsaved changes, the window's close button displays a dot. Adopting this same approach, that is, using a window-modified indicator, in your application makes it look more like a Mac OS X native application and so conform to user expectations.

To display an indicator that a window was modified, you need to use the Swing property `Window.documentModified`. It can be set on any subclass of `JComponent` that implements a top-level window 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 *New Control Styles available within J2SE 5.0 on Mac OS X 10.5*.

Apple Events and 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 can nevertheless let users take some level of control of your application through AppleScript. To do so, 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 such as Print and Open. Information on these two classes is available in J2SE 5.0 Apple Extensions Reference.

Java SE 6 also enables you to invoke AppleScript with the `javax.script` API. Listing 6 (page 37) provides a sample implementation of this functionality. Full documentation for the API can be found at http://java.sun.com/javase/6/docs/api/javax/script/package-summary.html.

Listing 6 Invoking AppleScript with the `javax.script` API

```
public static void main(String[] args) throws Throwable {
    String script = "say \"Hello from Java\"";

    ScriptEngineManager mgr = new ScriptEngineManager();
    ScriptEngine engine = mgr.getEngineByName("AppleScript");
    engine.eval(script);
}
```

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

System Properties

There are many Mac OS X–specific system properties you can set to modify the behavior of your Java application. A complete list of supported Mac OS X system properties, including how to use them, is available in Java System Properties.

User Interface Toolkits for Java

This article discusses how the Mac OS X implementation of the user interface toolkits Swing, AWT, accessibility, and sound differ from the toolkits on 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 article does not discuss user interface design issues that you should consider in Mac OS X. For that information, see "Making User Interface Decisions" (page 29).

Swing

In Mac OS X, Swing uses the Aqua Look and Feel as the default look and feel (LAF). Swing attempts to be platform neutral, but some aspects of it are an impedance mismatch with the Aqua user interface. Apple attempts to bridge the gap with a common ground that provides both developers and users an experience that is not foreign. This section discusses where the Aqua LAF differs from the default implementation on other platforms.

Note: While testing your application, you should test it on the standard Java cross-platform LAF as well as Aqua. To do this, add `UIManager.setLookAndFeel(UIManager.getCrossPlatformLookAndFeelClassname())` to your code.

Menu Bars (JMenuBar)

In Java's default cross-platform LAF, as well as the Windows LAF, menus are applied on a per-frame basis inside the window under the title bar. On a Mac, in contrast, menus appear in one spot no matter what windows users have open—at the top of the screen, in the menu 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
```

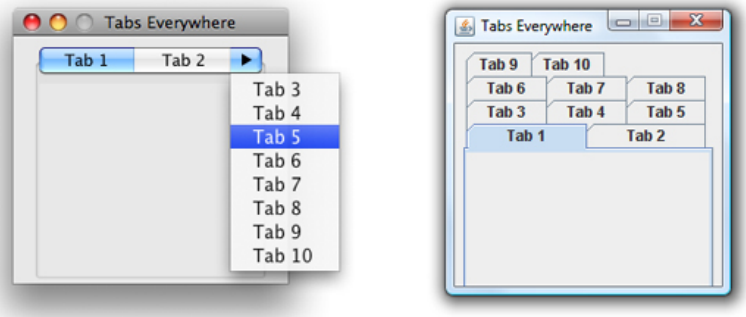
This property can have a value of `true` or `false`. By default, it is `false`, which means menus are in the window instead of the menu bar. When this property is set to `true`, the Java runtime moves the menu bar of any Java frame to the top of the screen, where Macintosh users expect it. Since this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

Note that this setting does not work for Java dialogs having menus. 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 using a `JFrame` object instead of a `JDialog` object.

Tabbed Panes (JTabbedPane)

On other platforms, if you have a tabbed pane (`JTabbedPane`) with too many tabs to fit in the parent window, the tabs are simply stacked on top of each other. 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, allows you to program your application just as you would on any other platform while providing users an experience that is more consistent with Mac OS X guidelines. The difference between a tabbed pane in Mac OS X and a tabbed pane in Windows is shown in Figure 1.

Figure 1 Tabbed panes with multiple tabs in Mac OS X and Windows

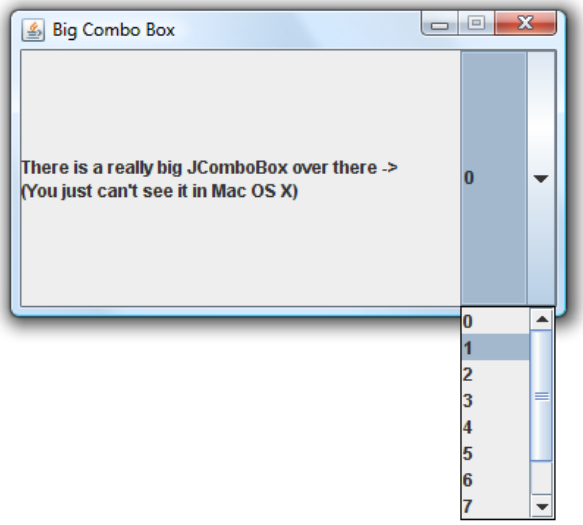


One other thing to keep in mind about `JTabbedPane` objects in Mac OS X is that they have a standard size. If you put an image in a tab, the image is scaled to fit the tab instead of the tab to the image. This standard size applies to several other Swing components as well.

Component Sizing

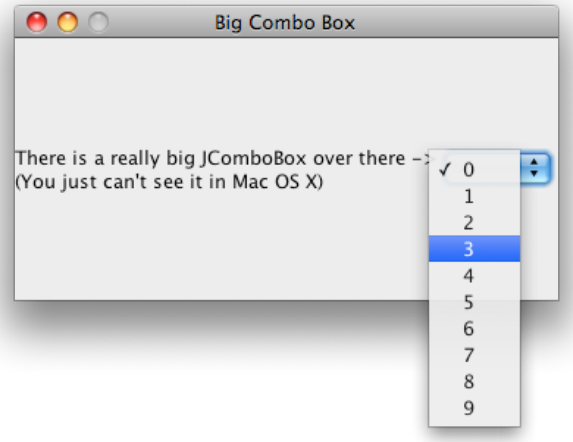
Aqua has very well-defined guidelines for the size of its controls. Swing, on the other hand, does not. The Aqua LAF tries to find a common ground. For example, since any combo box larger than twenty pixels would look out of place in Mac OS X, that is all that is displayed, even if the actual size of the combo box is bigger. Figure 2 shows a very large `JComboBox` component in Windows XP. Note that the drop-down scrolling list appears at the bottom of the button.

Figure 2 An oversize `JComboBox` component in Windows



The same code yields quite a different look in Mac OS X, as can be seen in Figure 3. 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 An oversize `JComboBox` component in the Aqua LAF



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 some nuances of Swing buttons that you should be aware of:

- `JButton` components with images in them are rendered as bevel buttons by default.
- A default `JButton` component 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.)
- `JButton` components 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 three values:

- `toolbar` gives you square bevel button.
- `text` gives you a push button.
- `icon` gives you a rounded bevel button.

Keep Apple's human interface guidelines in mind if you explicitly set a button type.

For a more thorough treatment of the Aqua LAF in Swing, see *New Control Styles available within J2SE 5.0 on Mac OS X 10.5*.

Abstract Window Toolkit (AWT)

By its nature, AWT is very different on every platform. When developing Java applications in Mac OS X, follow these tips for best results:

- The value of the accelerator key can be determined by calling `Toolkit.getDefaultToolkit().getMenuShortcutKeyMask()`. This is further discussed in "Accelerators (Keyboard Shortcuts)" (page 30).
- Mac OS X does not specify a default minimum size for windows. To avoid a 0 by 0 (0x0) pixel window being opened, top-level frames have a minimum size of 128 by 37 (128x37).
- `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.

Character Encoding

The default character encoding in Java for Mac OS X is MacRoman. The default font encoding on some other platforms is ISO-Latin-1 or WinLatin-1; unlike MacRoman, these encodings are subsets of UTF-8. Programs that assume that filenames can be turned into UTF-8 by just turning a byte into a char will 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 a Unicode subset 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, because 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 that certain characters—for example, e-acute (é)—are stored 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.

Accessibility

With some platforms, to use the Java Accessibility API, you must use a native bridge. This is not necessary in Mac OS X because the bridging code is built in. Users can configure the accessibility features of Mac OS X through the Universal Access pane of System Preferences. As a result, 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 called VoiceOver is included with the operating system. Your Java application automatically uses this technology.

Security

In Mac OS X v10.5, Java applications that use the Kerberos computer network authentication protocol automatically 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 v10.5 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");
```

For more usage information, see the reference documentation on `java.security.KeyStore` at http://java.sun.com/j2se/1.5.0/docs/api/java/security/KeyStore.html.

Sound

Java on Mac OS X allows you to sample sound with Apple's Core Audio framework at any frame rate supported by your input device. Input can be signed or unsigned PCM encoding, mono or stereo, 8 or 16 bits per sample.

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

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, use `Toolkit.sync`.

By default, Java on Mac OS X uses the Sun2D renderer, which exactly mimics the behavior of Java 2D on other platforms. If you are developing a graphically intensive application specifically for the Mac OS X platform and the Sun2D renderer's performance is inadequate, you may find better success using Apple's Quartz graphics engine for your Java rendering instead (see http://developer.apple.com/graphicsimaging/quartz/ for more information). Quartz is optimized for a different set of operations from the Sun2D renderer, and as a consequence its behavior is by no means identical to that of Java on other platforms.

By default, Quartz displays text anti-aliased. Therefore, if you use Quartz as your renderer, Java2D turns anti-aliasing on in order to render text in the Aqua look and feel for Swing (it does this by setting `KEY_ANTIALIASING` to `VALUE_ANTIALIAS_ON`). If you want the pixels of your images and text to more closely approximate that same content on other platforms, turn off anti-aliasing. You can do so by using the properties described in Java System Properties or by calling `java.awt.Graphics.setRenderingHint` from within your Java application. In applets, anti-aliasing is turned off by default.

Tip: When you are using anti-aliasing, if you need to replace text or an image, repaint the graphics context. Do not use XOR mode to repaint images.

Resolution Independence

Java is not explicitly designed for resolution independence (also known as HiDPI); therefore, Java for Mac OS X borrows some functionality from the Cocoa framework. To load a resolution-independent `tiff`, `icns`, or `pdf` file from the Resources folder of your application bundle into your Java application, use the `getImage` method of `java.awt.Toolkit`. The string you pass into `getImage` is of the form `"NSImage://MyImage"`. Do not include the file extension of the image. Also be aware that if you are using the Sun2D renderer, Java will automatically switch to the Quartz engine and enable anti-aliasing if you load a resolution-independent image.

You can test resolution independence in your application with the Quartz Debug tool, located in `/Developer/Applications/Performance Tools`. Quartz Debug allows you to launch your application at up to three times the default screen resolution. For a full list of features included in Quartz Debug, consult the Quartz Debug Help.

Mac OS X includes resolution-independent standard images for user interfaces that you can also access with the `getImage` method of `java.awt.Toolkit`. For instance, `Toolkit.getDefaultToolkit().getImage("NSImage://NSColorPanel")` will return an Image reference representing the color wheel icon seen on the Colors button in applications such as Mail. For a comprehensive list of the standard images available, see "Constants" in *NSImage Class Reference*.

Note: The standard image constants defined in `NSImage.h` all include the substring `ImageName`. For instance, the constant for the `NSColorPanel` image has the name `NSImageNameColorPanel`. When passing a string to `getImage`, do not include `ImageName` in the string—it is included in the name of the Objective-C constant, but not the value of the name itself.

Core Java APIs and the Java Runtime on Mac OS X

This article discusses the differences between the Core Java APIs on Mac OS X and other platforms. 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 article. There are a couple of details concerning Preferences that you should be aware of, as discussed in "Other Tools" (page 18).

Networking

Mac OS X v10.3 and later supports IPv6 (Internet Protocol version 6). Because J2SE 5.0 and Java SE 6 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 their home directory (`~/Library/Preferences/`). The system preferences are stored in `/Library/Preferences/` and are only persisted to disk if the user is an administrator.
- 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 J2SE 5.0 Apple Extensions Reference for more information.

JNI

It is recommended that you use the Java JNI Application template in the Xcode Organizer as a starting point for your JNI development. For more on the Xcode Organizer, see "The Xcode Organizer" (page 17).

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`. Java HotSpot also recognizes `.dylib` as a valid JNI library format as of Mac OS X v10.5.

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
```

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
   ```

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

Note: After you have built your JNI libraries, make sure to let Java know where they are. It is recommended that you do this by putting your libraries into your application bundle and passing in the path with the `-Djava.library.path` option. It is also possible to do this by putting your libraries in `/Library/Java/Extensions/`, but this is discouraged, as it breaks the encapsulation of your bundle.

A complete example of JNI development 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.

The Java Runtime

The Java implementation for Mac OS X includes the Java HotSpot VM runtime and the Java HotSpot client VM, both 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.

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`	1.5.0_13	Mac OS X v10.4 and earlier ships with earlier versions of Java. Use this property to test for the minimal version your application requires.
`java.vm.version`	1.5.0_13	
`file.separator`	'/'	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`	10.5.4	Java 1.5 runs only in Mac OS X v10.4 or later.

Note : The `mrj.version` system property is still exposed by the VM in Java 1.5. 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 Mac OS X. The reason is that this property may go away in future attempts to further synchronize the Apple source with the source from Sun.

Document Revision History

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

Date	Notes
2008-10-15	Updated to focus on J2SE 5.0 and Java SE6 and reflect recent updates to Java for Mac OS X.
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.