MAC OS RUNTIME
FOR JAVA

Programming With JManager

For MRJ 2.0

12/9/97
Technical Publications

Contents

Figures, Tables, and Listings

About This Document

Programming With JManager describes how to create a Java™ runtime environment and execute Java applets and applications on the Mac OS platform. You should read this document if you want to create Mac OS applications that can support the execution of Java code. For example, if you are writing a Web browser, you can use JManager to display Java applets within the browser. You can also use JManager to create Mac OS–compatible Java applications.

The Java™ Applet/Application Manager (JManager) is a set of C-based functions that you use for instantiating an environment to run Java code on the Mac OS platform and to interact with the code within it. Many JManager functions are also available as Java methods, so you can access them from Java code as well; a listing of the corresponding methods is included.

This document does not describe the Java language, low-level details of the Java virtual machine, or the Java Native Interface (JNI). For that information, you should consult JavaSoft documentation, which you can access through the Java home page:

<http://java.sun.com/>

How to Use This Document

To understand how to use JManager functions to prepare and execute Java applets and applications on the Mac OS platform, you should first read Chapter 1, "Using JManager," which gives tutorial information and code samples. Chapter 2, "JManager Reference," contains descriptions of all the JManager functions and the required application-defined callback functions. You can reference this chapter while reading Chapter 1 or while writing your code.

If you would like to access JManager functions from Java code, read the first two chapters and then see Chapter 3, "JManager Java Class Reference," for a listing of the Java equivalents to the C functions.

IMPORTANT

This document does not describe any older JManager 1.0 functionality. Although many functions are unchanged from the older version, you should consult JManager 1.0 documentation if you need the older functions. However, if you want to update code that uses JManager 1.0 calls, you can check Appendix A for a listing of changes to individual functions. ▲

If you are new to the Mac OS platform, check Appendix B for a brief list of Mac OS–related issues that might affect your Java code.

Additional Resources

Some JManager functions require that you already know how to manipulate windows or handle user events on the Mac OS platform. If you are not familiar with these concepts, please consult *Inside Macintosh: Macintosh Toolbox Essentials* and *Inside Macintosh: More Macintosh Toolbox* before using JManager functions. You can find more information about drawing inside windows in *Inside Macintosh: Imaging With QuickDraw.*

For more information about Apple's use of Java technology, see the following Web page:

<http://www.applejava.apple.com/>

If you simply want to package a Java application so that you can launch it like a Mac OS application, you can use the JBindery tool to do so. JBindery does not require any Mac OS programming knowledge. For information on JBindery, see the document *Using JBindery.*

Conventions

This book uses special conventions to present certain types of information. Words that indicate special meanings appear in specific fonts or font styles.

Special Fonts

All code listings, reserved words, command options, resource types, and the names of actual libraries are shown in Letter Gothic `(this is Letter Gothic)`.

Words that appear in **boldface** are key terms or concepts and are defined in the glossary.

Command Syntax

This book uses the following syntax conventions:

`literal`	Letter Gothic text indicates a word that must appear exactly as shown.
italics	Italics indicate a parameter that you must replace with a term that matches the parameter's definition.

Notes

Note
A note like this contains information that is useful but that you do not have to read to understand the main text. ◆

IMPORTANT
A note like this contains information that is crucial to understanding the main text. ▲

CHAPTER 1

Using JManager

Contents

Using JManager

This chapter describes how JManager interacts with the Java runtime environment and how you use JManager to prepare and execute Java applets and applications.

Changes in JManager 2.0

JManager 2.0, as included with Mac OS Runtime for Java 2.0, extends or improves the functionality of previous versions. The JManager 2.0 library still supports the older functions; you are not required to update older code for MRJ 2.0 compatibility. However, if you build your application using JManager 2.0 headers, you can no longer use the older 1.0 functions.

The major changes for JManager 2.0 are as follows:

- The new `JMTextRef` object used to encapsulate strings. This object allows you to pass Unicode strings as well as Mac OS strings. All strings passed by JManager functions are now passed as text objects.
- New flags used with the `JMOpenSession` function, to allow use of the Just In Time Compiler (JITC), InternetConfig, and the application heap.
- Abstract Window Toolkit (AWT) contexts no longer have a suspended state.
- Security options are now bound to individual applets, not to a session.
- Graphics ports (`grafPort`) are now bound directly to frames, rather than accessed through a callback.
- The applet viewer callback `MyShowDocument` now specifies the name of the frame in which the document is to be shown.
- New callbacks for handling calls to `java.lang.System.exit`,URL password authentication, and low-memory conditions.
- Java bindings for most JManager functions. These allow your Java application to call JManager methods. See Chapter 3, "JManager Java Class Reference," for listings of JManager Java classes and interfaces.
- Use of the Java Native Interface (JNI), rather than the Java Runtime Interface (JRI) to access Java methods. If desired, however, you can still access methods using the Java Runtime Interface.

For a listing of changes to individual functions, see Appendix A. You can use this listing to determine quickly which function calls or callbacks you may need to update.

JManager and the Java Runtime Environment

Java is an object-oriented programming language you can use to construct programs that will run without modification on multiple platforms. To support this capability, Java requires that each platform provide a virtual machine (VM) that can interpret and execute compiled Java code. A **virtual machine** is software that simulates an abstract microprocessor, complete with its own registers and instruction set. This virtual machine executes "system software" that

- loads and executes Java programs
- creates windows and graphics to interact with the user
- provides a secure environment for executing untrusted code
- provides networking capabilities and security measures
- enables garbage collection (automatic memory deallocation and cleanup)

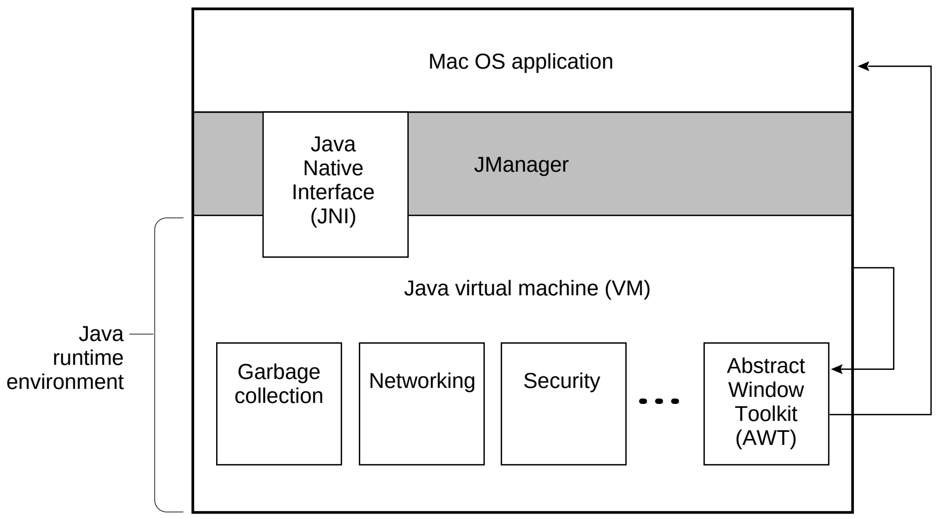
Each platform that supports Java must contain software that emulates this virtual machine. The combination of the virtual machine and its associated system software is called the **Java runtime environment.**

You can think of the Java runtime environment as a black box platform running within the Mac OS. This platform can handle multiple programs, and each program can contain virtual windows, buttons, and text. In order for the virtual machine to interact with the outer world, the actions that occur within the Java VM must be mapped to similar actions on the Mac OS. To do so requires an **embedding application** that accesses the Java runtime environment. This application can be a full-featured program (such as a Web browser) or a simple "wrapper" whose sole purpose is to run Java programs (such as JBindery).

For example, if a Java applet creates a window, a Mac OS application must map that window to an actual one that the user can see. Similarly, a mouse click by the user must be passed by the application to the Java VM so the applet can take proper action. JManager is the interface that handles these transactions.

Figure 1-1 shows the relationship between the Java runtime environment, JManager, and a Mac OS application.

Figure 1-1 The Java runtime environment, JManager, and a Mac OS application



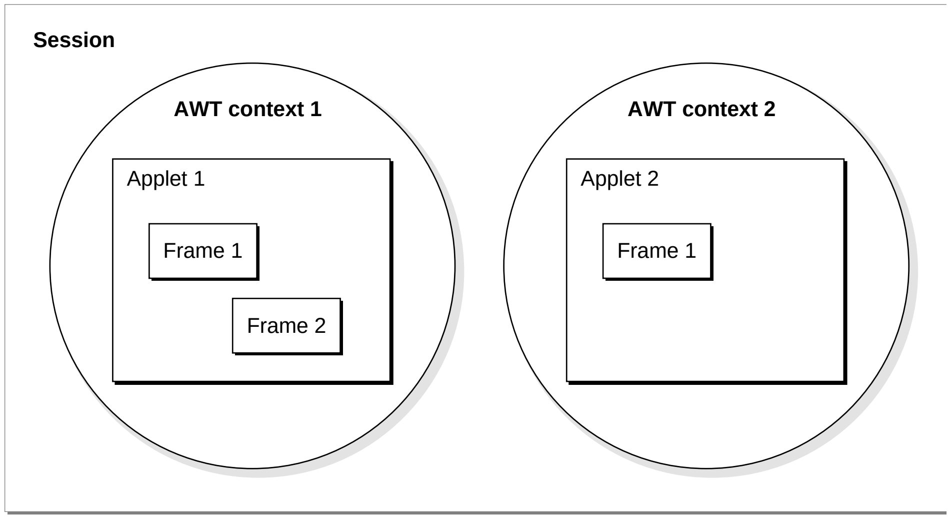
You can use JManager to accomplish the following on the Mac OS platform:

- create an instantiation of the Java runtime environment (the Java virtual machine and its associated system software).
- create execution environments within the Java runtime environment that can be mapped to the Mac OS user interface.
- find and instantiate Java applets or applications.
- set security options and specify proxy servers when accessing remote Java code.
- pass user events and window manipulations between the Mac OS user interface and the abstract interface provided by the Java runtime environment.
- call Java methods from Mac OS code.
- gain access to the Java Native Interface and the Java Runtime Interface.

Java Sessions, AWT Contexts, and Frames

To run Java applets on the Mac OS platform, you must first create a Java runtime **session**, which is an instantiation of the Java runtime environment. This environment can then load and execute Java code. Figure 1-2 shows an instantiated Java session and elements associated with it.

Figure 1-2 An instantiated Java session

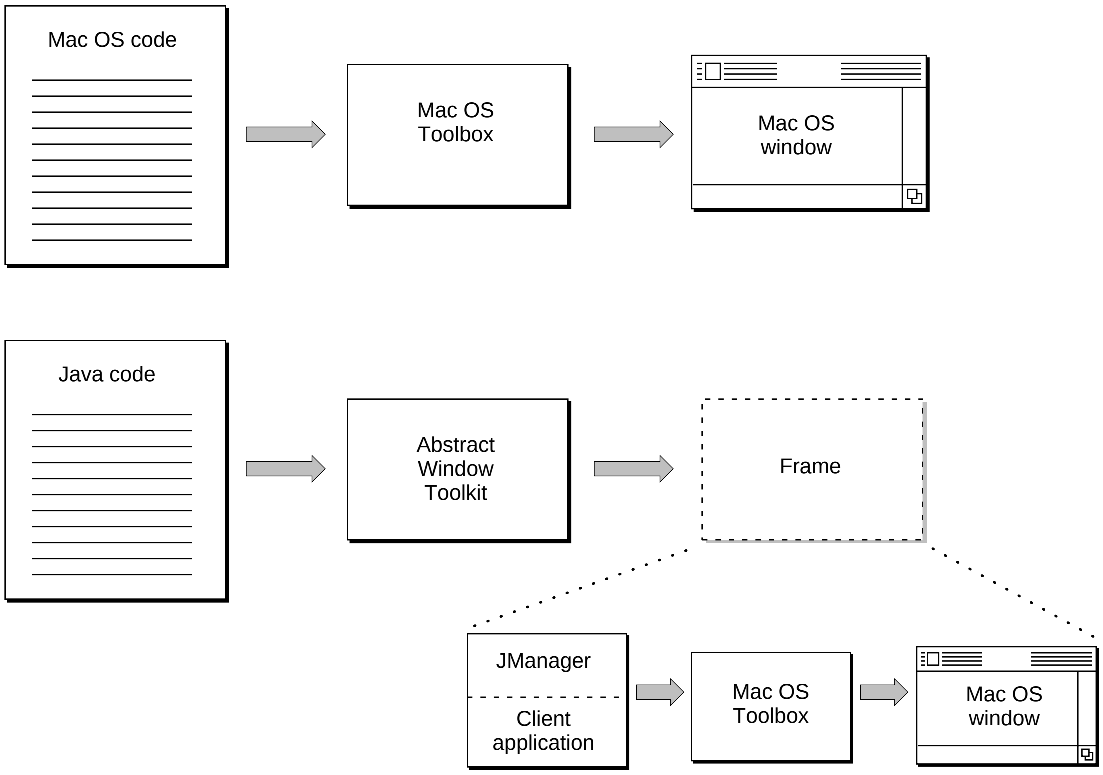


Within a session, a Java applet must run within an Abstract Window Toolkit (AWT) context. The **AWT context** provides an execution environment and a thread group for the Java program. Each Java applet must have its own AWT context. However, a given session can contain multiple contexts, each of which runs independently.

In addition to providing an execution environment, the AWT context allows access to the Abstract Window Toolkit. Java programs that display any graphical information (such as a text window, a button, or an image) must do so by calling the AWT. A call to the AWT manipulates images in a virtual window called a **frame.** This action is analogous to a Mac OS program calling the Mac OS Toolbox to manipulate images in a user-visible screen window.

To make the contents of a frame visible to the user, JManager cooperates with the AWT to pass information between the Java program and the Mac OS. For example, if the Java program decides to open a window, it tells the AWT, which can then call the appropriate Mac OS Toolbox functions to display a new window. Similarly, if the user selects a button, the selection is passed back to the Java program through the AWT. Figure 1-3 compares the elements needed by Mac OS code and Java code to display graphical information in a Mac OS window.

Figure 1-3 Frames versus windows



A Java applet can have any number of frames. A frame does not necessarily have to correspond to a visible Mac OS window. For example, you could assign a frame to a graphics port that displays data on a pen plotter instead of a window.

Since the Java runtime environment is object-oriented, each element associated with it is defined by an object. For example, an AWT context is defined by the `JMAWTContextRef` object, and the Java session itself is defined by an object of type `JMSessionRef`. Creating any element (for example, a frame) involves instantiating an instance of that object type.

To run a Java applet on the Mac OS, you must do the following:

1. Create the Java runtime environment.
2. Find the applet.
3. Create an AWT context.
4. Instantiate the applet.

Text Objects

Java programs typically handle text as Unicode strings, while the Mac OS platform uses special Mac OS–specific encodings (for example, MacRoman). To ensure compatibility, JManager functions pass all character strings as text objects. A **text object** is an object of type `JMTextRef`, and it encapsulates the string to pass along with its length and text encoding information. You can use other JManager functions to retrieve the encapsulated text as either a Unicode or a Mac OS encoding.

For example, if you wanted to encapsulate the string “Happy days are here again” in a text object, you would call the `JMNewTextRef` function (page 88):

```
OSStatus JMNewTextRef ( myJavaSession, &myHappyRef,
                        kTextEncodingMacRoman,
                        "Happy days are here again", 25);
```

The `JMNewTextRef` function requires you to specify the Java session that is to contain the text object (`myJavaSession` in this example), the reference to use for the created object (`myHappyRef`), the text encoding to use for the object (`kTextEncodingMacRoman`), the string, and the length of the string (25 characters).

You can specify any text encoding defined by the Text Encoding Converter; see the document *Programming with the Text Encoding Converter Manager* for a listing of possible encodings. After creating the text object, you would pass the reference `myHappyRef` anywhere you wanted to pass the string "Happy days are here again."

After use, it is your responsibility to remove any text objects you created by calling the `JMDisposeTextRef` function (page 89). Any text objects passed by the session, however (for example, during a callback) are automatically removed when no longer required. For more information about `JMNewTextRef` and other text object handling functions, see "Text Handling Functions" (page 88).

Creating a Java Runtime Session

If you want to run Java applets on the Mac OS platform, your embedding application must first create a Java runtime session. This session can then load and execute Java code.

Beginning a Java Runtime Session

On the Mac OS platform, the Java runtime session is defined by the `JMSessionRef` object. To instantiate this object, you must call the function `JMOpenSession` (page 80). Listing 1-1 gives an example of creating a session.

Listing 1-1 Creating a session

```
static JMSessionRef theSession;

static Boolean initializeMRJ()
{

    JMSessionCallbacks sessionCallbacks = {
        kJMVersion,         /* the current version */
        MyStandardOutput,   /* designated standard output */
        MyStandardError,    /* designated standard error */
        MyStandardIn        /* designated standard input */
        MyExit              /* System.exit handler */
        MyAuthenticate      /* URL Authentication handler */
        MyLowMem            /* Low memory condition handler */
    };

    return JMOpenSession(&theSession, eJManager2Defaults,
                         eCheckRemoteCode, &sessionCallbacks,
                         kTextEncodingMacRoman, 0) == noErr;
}
```

The instantiated `JMSessionRef` object is referenced by the value of `theSession`. Other JManager functions require you to pass this value to identify the session. (You can create more than one instantiation of the Java runtime environment if you wish.) Note that the `JMSessionCallbacks` structure you pass contains a field indicating the version of JManager you are using in your program. You should always set this value to `kJMVersion`. Setting this value prevents your program from accessing older (possibly incompatible) JManager functions.

The text encoding you specify when calling `JMOpenSession` (`kTextEncodingMacRoman` in this example) indicates the encoding used for any data sent to the designated standard output or standard error.

Session and Security Options

When calling `JMOpenSession`, you pass two parameters that indicate the desired session options, and whether you want to use the code verifier.

- The session options parameter is actually a mask that allows you to select various options, such as the following:
  - Whether the Java session can use temporary memory in addition to application heap memory. The default uses application heap memory only.
  - Whether to use the Just In Time (JIT) compiler. The default enables the compiler.
  - Whether to allow debugging. The default disables the debugger.
  - Whether to use preferences determined by InternetConfig. The default uses InternetConfig settings.
  - Whether to inhibit class unloading (that is, to prevent garbage collection of classes that are not being used). The default allows class unloading.

  The example in Listing 1-1 passes `eJManager2Defaults`, which selects all the default settings. See “Runtime Session Options” (page 63) for a list of the available options.

- The code verifier parameter (set to `eCheckRemoteCode` in Listing 1-1) specifies whether you want the **code verifier** to check the Java code before attempting to execute it. The code verifier analyzes the code to make sure that it is valid Java code and that it does not attempt any illegal or questionable actions (such as pointer arithmetic) that could give the code access to the Mac OS runtime environment. Typically you should use the code verifier if you plan to receive Java code from an untrustworthy source (such as over a network). See "Security Level Indicators" (page 61) for the available options.

  After calling `JMOpenSession`, you can read or modify the code verifier setting by calling the functions `JMGetVerifyMode` (page 84) or `JMSetVerifyMode` (page 84) respectively.

Callbacks

The data structure you must pass to the `JMOpenSession` function is a set of callback functions to handle console input and output, calls to exit from a Java application, low memory conditions, and URL authentication.

- The standard output, standard error, and standard input callbacks all deal with communications with the command line console. For example, you could specify a function that would receive and parse text sent to the standard output. Since the Mac OS runtime environment does not have a command line, these callbacks are often unused and set to `nil`. (By default, any text sent to standard output or standard error is redirected to a file.) For information about the form of these functions, see `MyStandardOutput` (page 135), `MyStandardError` (page 135), and `MyStandardIn` (page 136).
- The exit handler handles the case where the Java application quits (by calling `java.lang.System.exit`). Your callback can dispose of the applet or session as it sees fit, or it can simply allow the `System.exit` call to execute normally. For more information about the form of the exit handler, see `MyExit` (page 137).
- The URL authentication handler is called in cases where the user must enter a name and password to gain access to a URL. The handler should prompt the user for the proper information and pass it back to the Java program, which then decides whether the information is valid. For more information about the form of the authentication handler, see `MyAuthenticate` (page 137).
- The low memory handler is called when the Java runtime session runs low on memory. For more information about the form of the low memory handler, see `MyLowMem` (page 138).

For more information about the session callback structure, see "Session Callbacks Structure" (page 67).

Specifying Proxy Servers

If you want to define proxy servers for a session, you can do so using the `JMSetProxyInfo` function (page 83). A proxy server essentially acts as a gateway when you access data over a network. For example, if your company has a security firewall, all requests for code or data external to the company network must pass through the firewall before reaching the desired server. You can designate proxy servers for HTTP access, FTP access, and firewall access.

Note
If you allowed the use of the InternetConfig settings when creating the session, any proxy information defined there is used for the default settings. ◆

You pass a proxy server options structure to the `JMSetProxyInfo` function for each type of server. For example, Listing 1-2 sets a firewall proxy server.

Listing 1-2 Specifying a firewall proxy server

```
JMProxyInfo myFirewallProxyInfo {
    true,           /* allow a proxy for this type of server access */
    "TheWall.myCompany.com",    /* the name of the server */
    80};            /* the port number of the server */

JMSetProxyInfo(theSession, eFirewallProxy, &myFirewallProxyInfo);
```

The `myFirewallProxyInfo` structure specifies the firewall server by name and by port number. (If you wanted to specify HTTP or FTP proxy servers, you would create a structure for each of them as well.) You then set these values by calling the `JMSetProxyInfo` function and specifying the firewall proxy.

To read proxy information for a given session, you must call the `JMGetProxyInfo` function (page 82). See "Proxy Server Options" (page 69) and "Session Security Indicators" (page 61) for more information about the values you pass to these functions.

Checking JManager Versions

Many JManager data structures require that you specify the version of JManager (`kJMVersion`) you are compiling against. Before beginning a session, you should compare this value to the version of JManager available on the host computer to make sure that the two are compatible. The function `JMGetVersion` (page 128) returns the JManager version available on the host computer.

IMPORTANT

If you do not specify JManager as a weak library when compiling, your application will automatically fail to launch if the JManager library is not present. If you weak-link to the JManager library, your code should check that the `JMGetVersion` symbol is valid (that is, its value is not `nil`) before calling it. ▲

Properties and Client-Specific Session Data

Since the session is a `JMSessionRef` object, you can assign or change **properties** associated with it. You can do this using the functions `JMGetSessionProperty` (page 86) and `JMPutSessionProperty` (page 87). These functions correspond respectively to the Java methods `java.lang.System.getProperty` and `java.lang.System.setProperty`. If the property name you specify does not exist, then JManager creates a new property with that name.

You can also read or set optional client data for a given session using the functions `JMGetSessionData` (page 85) and `JMSetSessionData` (page 86). For example, if you have multiple sessions running at the same time, you might want to store specific data with each one.

Servicing Other Threads

When you are running a Macintosh embedding application, you must explicitly tell JManager to give up time to the Java virtual machine. You do so by using the `JMIdle` function (page 82) in your main event loop. Listing 1-3 shows an example of using `JMIdle`.

Listing 1-3 Using the `JMIdle` function

```
Boolean MainEventLoopContinues = true;

while (MainEventLoopContinues) {
    EventRecord eve;

    if (! WaitNextEvent(everyEvent, &eve, 30, nil) ||
        eve.what == nullEvent)
            JMIdle(theSession, 100);
        else
            handleEvent(&eve);
    }
```

The value specified in `JMIdle` indicates how many milliseconds to allot to other threads; you can specify a default wait period by using the value `kDefaultJMTime`. `JMIdle` returns immediately if no threads need servicing. `JMIdle` also returns if a user event occurs in the current session.

Ending a Java Runtime Session

After you have finished executing your Java programs, you should end the Java runtime session by calling the function `JMCloseSession` (page 81). This function disposes of the `JMSessionRef` object and removes any resources that JManager may have allocated for it. However, if you created any resources (such as AWT contexts and applets) within the session, you should explicitly remove them before calling `JMCloseSession`.

Finding Applets

Before you can instantiate and execute your applet, you must find the applet code by creating a `JMAppletLocatorRef` object. You can create such an object either synchronously or asynchronously. A synchronous search assumes that the applet's location can be immediately verified (if, for example, it is contained in a local file). You use the `JMNewAppletLocatorFromInfo` function (page 102) for a synchronous search, and you must provide information about the location of the applet in a `JMLocatorInfoBlock` data structure. The location information is the same as you would find in an **applet tag** in an HTML document. Listing 1-4 shows an example of using `JMNewAppletLocatorFromInfo`.

Listing 1-4 Using the `JMNewAppletLocatorFromInfo` function

```
OSStatus err = noErr;
JMAppletLocatorRef locatorRef;
JMTextRef URLTextRef, appTextRef;

/* Build the text objects for the strings to pass */
JMNewTextRef (theSession, &URLTextRef, kTextEncodingMacRoman,
                "file:///$APPLICATION/applets/DrawTest/example1.html",
                51);

JMNewTextRef (theSession, &appTextRef,      /* from the applet tag */
                kTextEncodingMacRoman,"DrawTest.class",14);

JMLocatorInfoBlock infoBlock = {
    kJMVersion,                             /* should be kJMVersion */
    URLTextRef,
    appTextRef,
    400, 400,                               /* width, height */
    0, nil              /* no optional parameters in this example */
};

/* create the locator */
/* If noErr is returned, the infoBlock was valid */
err =  JMNewAppletLocatorFromInfo(&locatorRef,
        theSession, &infoBlock, 0);

if (err == noErr) {
    /* instantiate and execute applet */
    }

/* Dispose text objects after use */
JMDisposeTextRef(URLTextRef);
JMDisposeTextRef(appTextRef);
```

Note that the two strings passed in the information block (the URL and the name of the class containing the applet code) are passed as text objects.

The /$APPLICATION/ indicator in the URL is an Apple-specific designation that indicates the current application directory.

For more information about the JMLocatorInfoBlock structure, see “The Applet Locator Information Block” (page 72).

If the applet is located on a remote server, you should search for it asynchronously using the JMNewAppletLocator function (page 102). Listing 1-5 shows an example of using JMNewAppletLocator.

Listing 1-5 Using the JMNewAppletLocator function

```
JMAppletLocatorRef locatorRef;
JMTextRef sampleURLTextRef;

struct JMAppletLocatorCallbacks locatorCallbacks = {
    kJMVersion,         /* should be kJMVersion */
    MyFetchCompleted    /* called on completion */
    };

JMNewTextRef (theSession, &sampleURLTextRef, kTextEncodingMacRoman,
               "http://www.hypno.com/javabeta/bongo/bongo.html", 46);

/* ignore the result--no pointer is passed to the */
/* html text, since it might not exist locally.*/
(void) JMNewAppletLocator(&locatorRef, theSession,
            &locatorCallbacks, sampleURLTextRef, nil, 0);

JMDisposeTextRef(sampleURLTextRef); /* dispose text object after use */
JMIdle(theSession, kDefaultJMTime);

/* this is the callback function specified in locatorCallbacks */
static void MyFetchCompleted(JMAppletLocatorRef locatorRef,
        JMLocatorErrors status){
    if (status != eLocatorNoErr) {
        /* handle the error here--perhaps put up a dialog box */
        }
    else {
        /* instantiate and execute applet */
        }
}
```

In the asynchronous search, you pass HTML text (as a text object) indicating the location of the applet and specify a callback function to execute when the search is completed. The callback function can take various actions, depending on the status value returned. For more information about the callback function, see `MyFetchCompleted` (page 146)

IMPORTANT
It is possible that the `MyFetchCompleted` function will be called before `JMNewAppletLocator` returns. ▲

One you have found your applet's location, you should call the `JMCountApplets` function (page 105) to determine the number of applets associated with the HTML page. `JMCountApplets` counts the number of applets and assigns an index value to each. Then you can use the functions `JMGetAppletDimensions` (page 106), `JMGetAppletTag` (page 107), and `JMGetAppletName` (page 107) to determine which applet to instantiate or to get more information about a particular applet. Listing 1-6 shows an example that counts the number of applets and returns information about each.

Listing 1-6 Retrieving information from an applet's HTML page

```
UInt32 appletCount;
UInt32 appWidth, appHeight;
UInt32 i;
JMTextRef appNameTextRef;
Handle appName;

/* iterate over the applets */
err = JMCountApplets(locatorRef, &appletCount);
printf("Number of Applets: "appletCount);

for (i = 0;  i < appletCount && err == noErr;  i++) {
    err = JMGetAppletName( locatorRef, i, &appNameTextRef);

    appName = JMTextToMacOSCStringHandle(appNameTextRef);
    Hlock(appName);

    if (!err) {
        err = JMGetAppletDimensions(locatorRef, i, &appWidth,
                &appHeight);
        if (!err) {
            printf("\nApplet #"%d" is "%s,i+1,*appName);
            printf("Dimensions:"%d" by "%d" pixels",appWidth,appHeight);
            }
        }
    HUnlock(appName);
    }
DisposeHandle(appName);
```

In some cases you might want to associate some client-specific data with an applet locator. To do so you can use the functions `JMGetAppletLocatorData` (page 104) and `JMSetAppletLocatorData` (page 105).

Both the `JMNewAppletLocatorFromInfo` and `JMNewAppletLocator` functions provide a valid `JMLocatorRef` object, which you can then use to instantiate and execute the applet. After instantiating the applet, however, you no longer need the `JMLocatorRef` object, so you can remove it by calling the `JMDisposeAppletLocator` function (page 104).

Creating an AWT Context

On the Mac OS, an AWT context is defined by a `JMAWTContextRef` object. Every Java program running within a session has its own AWT context. You can create an AWT context before or after instantiating a locator for the applet you want to run, but you must instantiate the AWT context before instantiating the applet.

To instantiate a `JMAWTContextRef` object, you call the `JMNewAWTContext` function (page 93) as shown in Listing 1-7. You must have instantiated a session already before creating an AWT context.

Listing 1-7 Creating an AWT context

```
/* define callbacks for the AWT context */
JMAWTContextCallbacks sessionCallbacks = {
    kJMVersion,         /* should be kJMVersion */
    MyRequestFrame,     /* callback to create a frame */
    MyReleaseFrame,     /* callback to release a frame */
    MyUniqueMenuID,     /* callback to give the AWT a valid MenuID */
    MyExceptionOccurred,/* notify that an exception occurred */
    };

/* create an AWT context for this applet */
JMAWTContextRef context;
err = JMNewAWTContext(&context, theSession, &sessionCallbacks, 0);
```

The value `context` references the `JMAWTContext` object, and you should pass this value in other JManager functions to specify this particular context.

You must specify a number of callbacks when calling `JMNewAWTContext`. JManager uses these callbacks to handle requests from the Java program for new frames (that is, windows). The `MyRequestFrame` callback (page 139) creates a new window, `MyReleaseFrame` releases a window (page 140), and `MyUniqueMenuID` creates a new menu ID (page 140). For example, if the Java program requests that a frame be made available, the application-defined callback function `MyRequestFrame` should request a new Mac OS window. Listing 1-8 shows an example of such a function.

Listing 1-8 Application-defined new frame function

```
OSStatus MyRequestFrame(JMAWTContextRef context, JMFrameRef newFrame,
    JMFrameKind kind, Rect bounds, Boolean resizeable,
    JMFrameCallbacks* callbacks)
{
    WindowPtr win;
    Point zeroPt = { 0, 0 };

    /* callbacks with pointers to your implementation-- */
    /* note that you also fill in the version number that you */
    /* compiled against (based on what you passed to JMOpenSession) */

    callbacks->fVersion = kJMVersion;
    callbacks->fSetFrameSize = MyResizeRequest;
    callbacks->fInvalRect = MyInvalRect;
    callbacks->fShowHide = MyShowHide;
    callbacks->fSetTitle = MySetTitle;
    callbacks->fCheckUpdate = MyCheckUpdate;
    callbacks->fReorderFrame = MyFrameReorder
    callbacks->fSetResizeable = MySetResizeable

    win = NewCWindow(nil, &bounds, "\p", false, documentProc,
        (WindowPtr) -1, true, (long) newFrame);
    if (win == nil)
        return memFullErr;

    JMSetFrameVisibility(newFrame, win, zeroPt, nil);

    return JMSetFrameData(newFrame, (JMClientData) win);
}
```

The `MyRequestFrame` function in this example calls the Mac OS Toolbox function `NewCWindow` to request a new window that corresponds to the frame. If you prefer, you can use an existing window instead. This example always creates a window of type `documentProc` (a simple document window without size or zoom boxes), but you can select different window types depending on the `kind` parameter passed into the callback function. See "Frame Types" (page 64) for a listing of possible requests.

After creating the window, the `JMSetFrameVisibility` function (page 117) registers the window characteristics (the graphics port, its position, and its clipping region) with the frame. Whenever the visibility of the window changes (for example, due to scrolling), you must call `JMSetFrameVisibility` again to update the visibility information.

The function `MyRequestFrame` also requires a number of callback functions that allow the Java program to manipulate the new window (for example, to show, hide, or update the window). For more information on these functions, see "Displaying Frames" (page 33) and "Application-Defined Functions" (page 134).

As part of the `NewCWindow` call, the reference to the frame (as held in `newFrame`) is stored in the `refCon` field of the window record (a `WindowRecord` structure). Doing so allows you to determine the frame associated with a window by simply calling the Mac OS Toolbox function `GetWRefCon`.

In a similar fashion, the `JMSetFrameData` function is used to store a pointer to the new window record in the frame's client data. You can then easily determine the window associated with a given frame by using a function such as that in Listing 1-9.

Listing 1-9 Determining the window associated with a frame

```
WindowPtr getFrameWindow(JMFrameRef frame)
{
    if (frame) {
        WindowPtr win = nil;
        if (JMGetFrameData(frame, (JMClientData*) &win) == noErr)
            return win;
    }
    return nil;
}
```

Displaying Frames

As explained earlier, a Java program displays graphical output in virtual windows called frames. Since frames correspond to Mac OS windows, you can manipulate them in a similar manner (for example, create or destroy frames, resize them, and so on).

In order to communicate between the abstract frames and the actual Mac OS windows, you must designate a number of application-defined callback functions. Many of these functions correspond to similar Mac OS Toolbox functions. The application-defined functions and their corresponding Mac OS Toolbox functions are shown in Table 1-1. For details of the structure of these functions, see “Application-Defined Functions” (page 134).

Table 1-1 Application-defined frame functions

Frame function	Description	Corresponding Mac OS Toolbox function
`MyRequestFrame`	Creates a new window	`GetNewCWindow`, `NewCWindow`, `GetNewWindow`, or `NewWindow`
`MyReleaseFrame`	Disposes of a window	`DisposeWindow`
`MyResizeRequest`	Requests that a window be resized	`SizeWindow`
`MyInvalRect`	Invalidates a portion of a window	`InvalRect`

Table 1-1 Application-defined frame functions

Frame function	Description	Corresponding Mac OS Toolbox function
`MyShowHide`	Shows or hides a window	`ShowHide` or `ShowWindow` and `HideWindow`
`MySetTitle`	Sets the window title bar	`SetWTitle`
`MyCheckUpdate`	Checks to see if a window update is necessary	`CheckUpdate` or `BeginUpdate` and `EndUpdate`
`MyFrameReorder`	Changes the ordering of the frame (bring to front, send to back, etc)	`BringToFront` or `SendBehind`
`MySetResizeable`	Sets whether a frame is resizeable or not	No corresponding function, although the state set by this function affects whether your `DoGrowWindow` callback calls the `SizeWindow` function.

Typically the bulk of an application-defined frame function prepares a call to the corresponding Mac OS Toolbox function. For example, assuming that the application uses the functions in Listing 1-8 (page 31) and Listing 1-9 (page 33), you can use the callback function in Listing 1-10 to set the window title.

Listing 1-10 A callback function to change the title of a window

```
void MySetTitle(JMFrameRef frame, JMTextRef titleObj)
{
    Handle title;
    Str255 ptitle;

    title = JMTextToMacOSCStringHandle(titleObj);
    Hlock(title);

    convertToPascalString(*title, &ptitle); /* this is a dummy utility */
    WindowPtr win = getFrameWindow(frame);
    if (win)
        SetWTitle(win, ptitle);

    HUnlock(title);
    DisposeHandle(title);
}
```

Since the Mac OS Toolbox function `SetWTitle` requires a Pascal string, you must convert the handle `title` before calling `SetWTitle`.

Getting Information About AWT Contexts and Frames

JManager provides a number of functions that return information about an AWT context or a frame. For example, since multiple applets can appear onscreen, if a user clicks in a window that corresponds to a frame, you may need to find out what applet or AWT context the frame belongs to.

The `JMCountAWTContextFrames` function (page 95) counts the number of frames associated with an AWT context.

The `JMGetAWTContextFrame` function (page 96) lets you find a particular frame (as indexed by the `JMCountAWTContextFrames` function) associated with an AWT context.

The `JMGetFrameContext` function (page 127) finds the AWT context associated with a frame.

The `JMGetFrameViewer` function (page 114) lets you determine the frame associated with an applet.

The `JMGetViewerFrame` function (page 115) finds an applet's **parent frame,** which is the frame created when the applet is created.

If you want to set or read client-specific data associated with an AWT context, you can do so using the functions `JMSetAWTContextData` (page 94) and `JMGetAWTContextData` (page 94).

If you want to read or set client-specific data associated with a particular frame, you can do so using the functions `JMGetFrameData` (page 117) and `JMSetFrameData` (page 118). You can store the window record as client data to make it easy to find a window corresponding to a frame. See Listing 1-8 (page 31) and Listing 1-9 (page 33) for an example.

Removing an AWT Context

When you have finished executing an applet and no longer need the AWT context, you should dispose of it using the `JMDisposeAWTContext` function (page 93). However, you should have already removed any applets from the context before calling `JMDisposeAWTContext`. You can reuse an AWT context by disposing of an instantiated applet and then instantiating a new one in the same context.

Instantiating Applets

After you have created a session, located the applet, and created an AWT context for the applet, you can instantiate it. An instantiated applet is defined by an `JMAppletViewerRef` object, and you use the `JMNewAppletViewer` function (page 108) to create one. Listing 1-11 shows an example of instantiating an applet.

Listing 1-11 Instantiating an applet

```
JMAppletSecurity securitySettings = {
    kJMVersion,         /* should be kJMVersion */
    eAppletHostAccess,  /* applet network access option */
    eLocalAppletAccess, /* applet access to local file system */
    true,               /* restrict system.access packages */
    true                /* restrict system.define packages */
    true,               /* restrict application.access packages */
    true                /* restrict application.define packages */
    };

JMAppletViewerCallbacks viewerCallbacks = {
    kJMVersion,         /* should be kJMVersion */
    MyShowDocument,     /* showDocument callback */
    MySetStatusMsg      /* setMessage callback */
    };

JMAppletViewerRef viewer;
err = JMNewAppletViewer(&viewer, context, locatorRef,
        appletIndex, &securitySettings, &viewerCallbacks, 0);

/* reload the applet to get it going */
if (err == noErr) {
    err = JMReloadApplet(viewer);
    }
```

The `JMNewAppletViewer` function requires you to pass a security options data structure indicating the following security settings:

- The applet network access field designates the applet's level of network access. Typically you should set this to `eAppletHostAccess` to indicate that an applet can access only its host server. See "Applet Security Indicators" (page 62) for a list of available options.
- The applet local file system access field designates whether an applet can access files stored on the host computer. Typically you should set this to `eLocalAppletAccess` to indicate that only an applet stored locally can access the local file system. See "Applet Security Indicators" (page 62) for a list of available options.
- The next four fields are flags that let you specify that whether to allow class access or class definitions outside the `java.*` classes. A flag that is set to true restricts access to the packages defined in the corresponding property:
  - `mrj.security.system.access`
  - `mrj.security.system.define`
  - `mrj.security.application.access`
  - `mrj.security.application.define`

  See "Applet Security Structure" (page 76) for more information about using these flags.

If you want to determine the current security levels for a particular applet, you can use the `JMGetAppletViewerSecurity` function (page 111). To change existing security levels, you can use the `JMSetAppletViewerSecurity` function (page 111).

In addition, you must define two callbacks, `MyShowDocument` (page 146) and `MySetStatusMsg` (page 147), when calling the `JMNewAppletViewer` function. The `MyShowDocument` function displays the contents of a URL passed back from the applet, and `MySetStatusMsg` displays any status messages the applet may pass to the application. You must also specify the index of the applet you want to instantiate (as returned by the `JMCountApplets` function (page 105)).

After you have instantiated the applet viewer object, you must load and execute the applet code by calling the function `JMReloadApplet` (page 112).

You can set or retrieve client data associated with the applet by using the functions `JMSetAppletViewerData` (page 110) and `JMGetAppletViewerData` (page 110).

At any time you can stop execution of the applet by calling the `JMSuspendApplet` function (page 113) and resume it by calling the `JMResumeApplet` function (page 113). If you want to restart the applet without reloading it from the server, use the `JMRestartApplet` function (page 112).

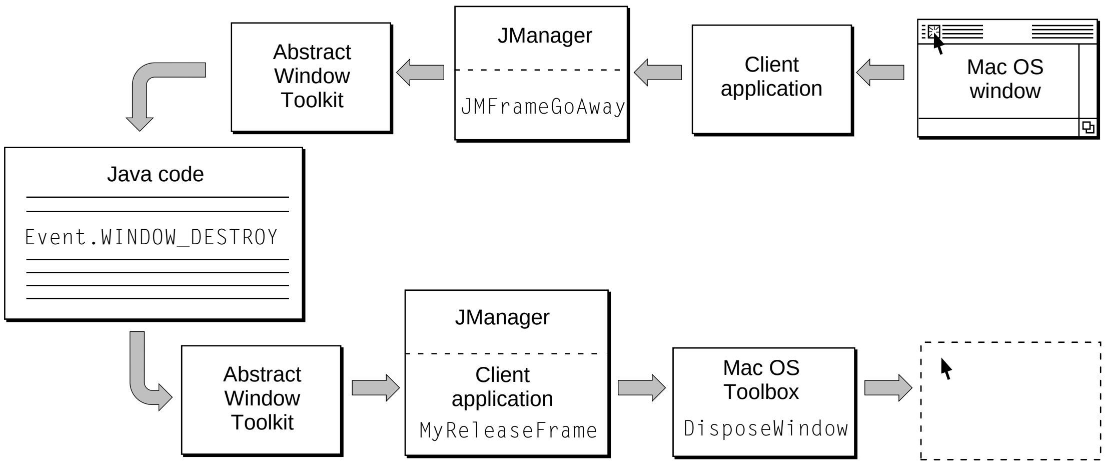
When you are finished with an applet, you can dispose of it by calling the `JMDisposeAppletViewer` function (page 109). This function automatically halts applet execution and removes any frames associated with the applet.

Handling Events

Although the Java applet runs within its own runtime environment, it must often interact with events that occur in the Mac OS. For example, user events (such as mouse clicks) or system events may require some response from the Java program. This section describes JManager functions for handling such events. For more information about how the Mac OS handles events, see "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials.*

Actions performed on a frame may need to be reflected on the user-visible window as well, but in most cases, such actions are handled by JManager through a callback. For example, if the user clicks on an applet window's close box, your application must notify the frame that this has occurred. The Java program can then take action based upon this event, such as removing the frame or displaying a message in a dialog box. In either case, the AWT uses the callback functions you defined in your application to display the results. Figure 1-4 shows the steps for removing an applet window.

Figure 1-4 Removing an applet window



See “Displaying Frames” (page 33) for more information about using the callbacks.

When an event occurs, you typically call an event-handling function from your main loop. Listing 1-12 shows an example of an event-handling function.

Listing 1-12 Handling events

```
static void handleEvent(const EventRecord* eve)
{
    switch (eve->what) {

        case updateEvt:
            handleUpdate((WindowPtr) eve->message);
            break;

        case activateEvt:
            handleActivate((eve->modifiers & activeFlag) != 0,
                (WindowPtr) eve->message);
            break;
        case osEvt:
            /* everyone should care about this */
            handleResume((eve->message & resumeFlag) != 0);
            break;

        case kHighLevelEvent:
            AEProcessAppleEvent(eve);
            break;

        case mouseDown:
            handleMouse(eve);
            break;

        /* assume no one cares about these */
        case mouseUp:
            break;

        case keyDown:
        case autoKey:
        case keyUp:
            handleKey(eve);
            break;
    }

}
```

High-level events (generally Apple events) are handled by calling the Mac OS Toolbox function `AEProcessAppleEvent`. In other cases, the handling functions should check to see if the event needs to be passed to the embedded Java program. The sections that follow describe the JManager functions needed to pass events and give sample implementations of the `handleUpdate`, `handleActivate`, `handleResume`, `handleMouse`, and `handleKey` functions.

Update, Activate, and Resume Events

If the application receives an update event, it must update the currently active window. If the window corresponds to a frame, you must pass the update event to the frame using the `JMFrameUpdate` function (page 122). The AWT context can then update the actual window using a callback. Listing 1-13 shows an example of an update function.

Listing 1-13 Handling a frame update event

```
static void handleUpdate(WindowPtr win)
{
    JMFrameRef frame;
    BeginUpdate(win);
    SetPort(win);
    frame = (JMFrameRef) GetWRefCon(win);
    if (frame)
        JMFrameUpdate(frame, win->visRgn);
    else
        EraseRgn(win->visRgn);
    EndUpdate(win);
}
```

If an activate event occurs, then a window was made active (that is, brought to the front), and if that window is associated with a frame, you must activate the frame using the `JMFrameActivate` function (page 123). This action gives the frame the opportunity to highlight title bars, scroll bars, and so on. Activating a frame also installs the menu bar associated with the frame. Listing 1-14 gives an example of activating a frame.

Listing 1-14 Sending an activate event to a frame

```
static void handleActivate(Boolean active, WindowPtr window)
{
    JMFrameRef frame = (JMFrameRef) GetWRefCon(window);
    if (frame)
        JMFrameActivate(frame, active);
}
```

Note
The `JMFrameActivate` function can either activate or deactivate a frame, depending on the Boolean value passed to it (the value of `active` in this example). ◆

Suspend and resume events can also occur when a window is activated or deactivated. When the user switches from one application to another, the newly selected application is sent a resume event, and the previously active one is sent a suspend event. This event affects all the applets embedded within an application, so you must call the `JMFrameResume` function (page 124) to suspend or resume all the existing frames.

Listing 1-15 shows how to send a resume event to all the frames associated with a client application.

Listing 1-15 Sending a resume event to frames

```
static void handleResume(Boolean resume)
{
    WindowPtr win = FrontWindow();
    while (win) {
        JMFrameRef frame = (JMFrameRef) GetWRefCon(win);
        if (frame)
            JMFrameResume(frame, resume);
        win = (WindowPtr) ((WindowPeek) win)->nextWindow;
    }
}
```

Note
The `JMFrameResume` function can either suspend or resume a frame, depending on the Boolean value passed to it (the value of `resume` in this example). ◆

This example cycles through all the visible windows used by the application and sends the event to those associated with frames. However, this example only works if every frame is associated with a window. If this is not the case, you must use some other method to send the resume event.

Mouse Events

A mouse event occurs when you click somewhere in the visible screen area. A function that handles mouse events must check the location of the mouse click and take action as appropriate. If the mouse event took place in a window that corresponds to a frame, you must pass the event to the frame so the Java applet can take proper action.

Listing 1-16 shows a function, `handleMouse`, that handles a mouse event.

Listing 1-16 Handling a mouse event

```
void handleMouse(const EventRecord* eve)
{
    WindowPtr win;
    short part = FindWindow(eve->where, &win);

    switch (part) {
        case inMenuBar: {
            long mResult = MenuSelect(eve->where);
            if (mResult != 0)
                menuHit(mResult >> 16, mResult & 0xffff);
        }   break;

        case inDesk:
            break;

        case inSysWindow:
            SystemClick(eve, win);
            break;

        case inContent:
            if (win != FrontWindow())
                SelectWindow(win);
            else {
                JMFrameRef frame = (JMFrameRef) GetWRefCon(win);
                if (frame) {
                    /* convert the mouse position to window local */
                    /* coordinates and pass it into the Java */
                    /* environment */
                    Point localPos = eve->where;
                    SetPort(win);
                    GlobalToLocal(&localPos);
                    JMFrameClick(frame, localPos, eve->modifiers);
                }
            }
            break;

        case inDrag: {
            Rect r = (**GetGrayRgn()).rgnBBox;
            DragWindow(win, eve->where, &r);
        }   break;

        case inGoAway: {
            /* request that the frame go away--it will call to the */
            /* frame through a callback if it actually does */
            JMFrameRef frame = (JMFrameRef) GetWRefCon(win);
            if (frame)
                JMFrameGoAway(frame);
             }  break;

        case inGrow: {
            union GrowResults {
                Point asPt;
                long asLong;
            } results;

            JMFrameRef frame = (JMFrameRef) GetWRefCon(win);
            Rect rGrow = { 30, 30, 5000, 5000 };
            results.asLong = GrowWindow(win, eve->where, &rGrow);
            if (frame != nil && results.asLong != 0) {
                /* request that the frame resize itself--it will call */
                /* to the frame through a callback if it actually does */
                Rect r;
                r.left = 0;
                r.top = 0;
                r.right = results.asPt.h;
                r.bottom = results.asPt.v;
                r.bottom -= 15;
                JMSetFrameSize(frame, &r);
                }
            } break;

        default:
            break;
    }
}
```

This example uses the Mac OS Toolbox function `FindWindow` to determine the location of the mouse click and then takes action depending on the location.

The `inMenuBar`, `inDesk`, and `inSysWindow` cases are handled as in any Mac OS application.

If the mouse click occurs in a window's content area (`inContent`), `handleMouse` checks to see if the window is active. If not, the window is selected (it receives an activate event and possibly a resume event). If the window is currently active, the local position of the mouse within the window is calculated and the coordinates sent to the corresponding frame using the `JMFrameClick` function (page 120). In a similar manner, you can also send a mouse-over event by calling the `JMFrameMouseOver` function (page 124) before the `JMIdle` call in the main event loop.

If the mouse click is in the drag region (`inDrag`), the Mac OS Toolbox function `DragWindow` is called to move the window. You do not have to pass any information to the corresponding frame, since the Java runtime environment does not worry about the relative position of frames.

If the mouse click is on the close box (`inGoAway`), the code notifies the corresponding frame using the `JMFrameGoAway` function (page 125). Any user-visible response to this action (such as removing the window) is handled by the AWT using the callbacks you specified when instantiating the AWT context.

If the mouse click is in a window's size box (`inGrow`), the code calls the Mac OS Toolbox function `GrowWindow` to track the new size of the window. The new dimensions are passed to the frame using the `JMSetFrameSize` function (page 119). The dimensions of the window are updated using an AWT context callback.

Note
If the new window dimensions are too large or too small (because of screen constraints or some arbitrary limit), the window should be adjusted to a preferred size. ◆

Keyboard Events

Keyboard events occur whenever the user presses a key. These keypresses may correspond to text entered into a window, a keyboard-equivalent menu selection, or a similar action (for example, selecting the default button in a dialog box by pressing the return key). If the keyboard event occurs in a window that corresponds to a frame, you must pass the event to the frame using either the `JMFrameKey` function (page 121) for key-down events or the `JMFrameKeyRelease` function (page 122) for key-up events.

Listing 1-17 shows a simple example that handles a keyboard event.

Listing 1-17 Handling a keyboard event

```
static void handleKey(const EventRecord* eve)
{
    WindowPtr win;
    JMFrameRef frame;

    /* see if a menu item was selected */
    if (eve->what == keyDown && (eve->modifiers & cmdKey) == cmdKey) {
        long menuResult = MenuKey(eve->message & charCodeMask);
        if (menuResult != 0) {
            menuHit(menuResult >> 16, menuResult & 0xffff);
            return;
        }
    }
    /* otherwise, just let JManager deal with it */
    win = FrontWindow();
    if (win) {
        frame = (JMFrameRef) GetWRefCon(win);
        if (frame)
            JMFrameKey(frame, eve->message & charCodeMask,
                (eve->message & keyCodeMask) >> 8, eve->modifiers);
    }
}
```

This code first checks to see if the keyboard input was a keyboard equivalent for a menu item (for example, Command-Q for Quit). If so, control passes to the menu-event routine (see Listing 1-18 (page 47) for an example of handling a menu selection). In all other cases, the keyboard input is passed to the frame corresponding to the window, and the Java program can then determine the appropriate response. The content of the keyboard input is determined from the event record (the `EventRecord` structure) returned by the Event Manager.

Menu Selections

Both mouse events and keyboard events can select menu items. In either case, the event should be handled by a menu selection function. If the selection corresponds to a Java applet's menu item, you must pass the selection to the applet's AWT context using the `JMMenuSelected` function (page 96). Listing 1-18 shows a simple menu selection function.

Listing 1-18 Handling a menu item selection

```
/* enumerators to define the Mac OS menus*/
enum Menus {
    eAppleMenu = 1000,
    eFileMenu,
    eEditMenu,
    eLastMenu
    };

/* enumerators to define the menu items */
enum AppleMenuItems {
    eAboutItem = 1
    };

enum FileMenuItems {
    eMyAction1 = 1,     /* nonstandard menu item defined by */
                        /* the application */
    eQuitItem = eMyAction1 + 2
    };

enum EditMenuItems {
    eUndoItem = 1,
    eCutItem = eUndoItem + 2,
    eCopyItem,
    ePasteItem,
    eClearItem,
    eSelectAllItem = eClearItem + 2
    };

static void menuHit(short menuID, short menuItem)
{
```

```
switch (menuID) {
    case eAppleMenu:
        switch (menuItem) {
            case eAboutItem:    /* show About box */
                MyAboutBox();
                break;

            default: {      /* open the appropriate desk accessory */
                Str255 s;
                SetPort(LMGetWMgrPort());
                GetItem(GetMHandle(eAppleMenu), menuItem, s);
                if (s[0] > 0)
                    OpenDeskAcc(s);
            }   break;
        }
        break;

    case eFileMenu:
        switch (menuItem) {
            case eMyAction1:
                DoMyAction1();
                break;

            case eQuitItem:
                MainEventLoopContinues = false;
                break;
        }
        break;

    case eEditMenu:
        break;

    default: {
        /* pass the menu hit to the AWTContext for processing */
        WindowPtr win = FrontWindow();
        if (win != nil) {
            JMFrameRef frame = (JMFrameRef) GetWRefCon(win);
            if (frame != nil) {
                JMAWTContextRef context = JMGetFrameContext(frame);
                if (context != nil)
                    JMMenuSelected(context, GetMHandle(menuID),
                             menuItem);
                    }
                }
            }   break;
        }

        HiliteMenu(0);
}
```

If the user did not select a standard menu item, the `menuHit` function passes the menu selection to the applet's AWT context. On the Mac OS, the menu bar is always associated with the active window (and, consequently, with the active frame). After determining the frame associated with the window (using the Mac OS Toolbox function `GetWRefCon`), `JMGetFrameContext` returns the AWT context associated with the frame. The menu handle of the selection is then passed to the AWT context using the `JMMenuSelected` function (page 96).

Drag-And-Drop Support

If your embedding application supports the Drag Manager, you can pass drag-and-drop information to a frame using JManager functions. The `JMFrameDragTracking` function (page 126) allows your frame to respond to an item dragged over it (for example, creating a highlight to signal that the drag item is valid for the frame). If the user drops the item within the frame boundaries, the `JMFrameDragReceive` function (page 127) lets you pass information about the dropped item to the frame.

These functions correspond respectively to the application-defined Drag Manager functions `DragTrackingHandler` and `DragReceiveHandler`, and are called from within these handlers. For more information about the Drag Manager, see the *Drag Manager Programmer's Guide*.

Note
The Java JDK standards 1.1.x and earlier do not support drag and drop. ◆

Executing Java Applications

Most of the information in the previous sections describes how to load and execute Java applets, which are designed to be run within an embedding application. However, you can also execute Java applications, which can be launched just like any other application. To do so on the Mac OS, you must create a wrapper program around the Java application.

Note
You can also use the utility application JBindery to create a wrapper for Java applications. JBindery allows you to create standalone Java applications that you can launch just like any Mac OS application. ◆

To launch a Java application using JManager, you must take the following steps:

1. Instantiate a Java runtime session.
2. Create an AWT context for the application.
3. Find the Java application's code.
4. Call the application's `main` method.

The first two steps are the same steps you used to instantiate and execute a Java applet. However, finding and executing the Java application requires some interaction with the Java Native Interface (JNI). Listing 1-19 shows an example of finding and launching a Java application.

Listing 1-19 Launching a Java application

```
static Boolean initializeSampleApp()
{
    JNIEnv* env;
    JNIMethodID method;

    static const char* kSampleAppZipFile = "file:///$APPLICATION/
                AppSample.zip";
    JMTextRef theURLRef;
    FSSpec appSpec;
    /* make the file URL into a text object */
    JMNewTextRef(theSession, &theURLRef, kTextEncodingMacRoman,
                    kSampleAppZipFile, len(kSampleAppZipFile);

    /* now locate the application's code and add it to the class path */
    if (JMURLToFSS(theSession, theURLRef, &appSpec) != noErr)
        return false;

    if (JMAddToClassPath(theSession, &appSpec) != noErr)
        return false;

    /* next, use the JNI to locate the class */
    /* begin by getting a JNI_Env object */
    env = JMGetCurrentEnv(theSession);
    if (env == nil)
        return false;

    /* find the class--if it’s in a package, separate with */
    /* slashes (/), for example, sun/applet/AppletViewer */
    theAppClass = env->FindClass(env, "AppSample");
    if (theAppClass == nil)
        return false;

    /* now find the method by name & signature */
    method = env->GetStaticMethodID(env, theAppClass, "main",
                    "([Ljava/lang/String;)V");
    if (method == nil)
        return false;

    /* request that the method be executed within the AWT context */
    /* note that there are no arguments to pass to this method */
    return noErr == JMExecJNIStaticMethodInContext(theContext,
                        theAppClass, method, 0, nil);

    /* remove the text object now that it’s no longer needed */
    JMDisposeTextRef(theURLRef);
}
```

In this example, the application’s code is stored in a zip file. The location of the file is specified as a URL, and this is then converted to a file specification record using the JMURLToFSS function (page 130). The JMAddToClassPath function (page 130) adds this record to the class path, so the Java runtime environment knows where to search for additional Java classes.

To find the class and main method of the Java application, you must use the Java Native Interface. First call the JManager function JMGetCurrentEnv (page 131) to get information about the JNI environment associated with this session. You can then call JNI functions to find the class and method. In Listing 1-19, the call to FindClass returns the class associated with the application appSample. The call GetStaticMethodID returns the ID of the main method in appSample (that is, the main routine). The GetStaticMethodID function requires that you pass the method's signature, which is a string that describes the method's parameters and return values. For a full description of the signature format, see the Java Native Interface documentation available at the Java home page:

<http://java.sun.com/>

Once you know the class and method ID, you can then call the JManager function JMExecJNIStaticMethodInContext (page 98) to call the method and execute the application within the created AWT context. If the method requires any arguments, you pass them when you call JMExecJNIStaticMethodInContext.

Note
Execution of the Java application is asynchronous. That is, execution of the application begins when the AWT context can devote time to doing so. ◆

Although the launch process differs, Java applications rely on JManager to interact with the Mac OS in the same manner as applets do. Therefore, when writing your wrapper application, you must include frame callbacks and user-event handling routines just as you would for applets.

Since the Java program is an application, you cannot call a JManager function to exit. However, you can trap the call to the Java method java.lang.System.exit (which quits the Java application) by implementing a MyExit callback that disposes of the Java runtime and quits the wrapper application. See MyExit (page 137) for more information about this application-defined function.

Alternatively, since JManager automatically generates a Quit Application Apple event when java.lang.System.exit executes, you can install an Apple event handler to quit the wrapper application. Listing 1-20 gives an example of using an event handler.

Listing 1-20 Using an Apple event handler to quit a Java application

```
static pascal OSErr _handleQUIT(AppleEvent* event, AppleEvent* reply,
    long refcon)
{
    theLoopContinues = false;
    return noErr;
}

void main(void)
{
...
    AEInstallEventHandler(kCoreEventClass, kAEQuitApplication,
        NewAEEventHandlerProc(_handleQUIT), 0, false);

/* main event loop */
    while (theLoopContinues) {
        ...
        }
    JMCloseSession(theSession);
}
```

The Apple event handler `_handleQUIT` halts the main event loop; the wrapper application then ends the Java session and exits.

For more information about how to use Apple events, see *Inside Macintosh: Interapplication Communication.*

Obtaining Java References

In some cases, you may need to access the actual Java objects used in a session, rather than the encapsulated objects passed by the embedding application. For example, if you wanted to call a method contained in an instantiated frame, you would need a reference to the actual Java frame, not the `JMFrameRef` object.

Table 1-2 lists the functions you can use to return Java references. These functions all return a pointer of type `jref`.

Table 1-2 Functions that return pointers to Java objects

Function	Returns pointer to object
`JMGetSessionObject` (page 87)	`com.apple.mrj.JManager.JMSession`
`JMGetAwtContextObject` (page 101)	`com.apple.mrj.JManager.AWTContext`
`JMGetAppletViewerObject` (page 115)	`com.apple.mrj.JManager.JMAppletViewer`
`JMGetJMFrameObject` (page 128)	`com.apple.mrj.JManager.JMFrame`
`JMGetAppletObject` (page 116)	`java.applet.Applet`
`JMGetAWTFrameObject` (page 128)	`java.awt.Frame`

The first four functions return the `jref` equivalents of the corresponding JManager objects. For example, `JMGetSessionObject` returns the equivalent of the `JMSessionRef` reference. `JMGetAppletObject` and `JMGetAWTFrameObject`, however, return references to their actual Java objects.

You can use these references directly if you are accessing the Java Runtime Interface (JRI), but if you need to access the Java Native Interface (JNI), you must convert references of type `jref` to ones of type `jobject` using the `JMJRIRefToJNIObject` function (page 133). To convert references of type `jobject` to type `jref`, use the `JMJNIObjectToJRIRef` function (page 134).

CHAPTER 2

JManager Reference

Contents

This chapter describes all the JManager constants, data types, and functions.

JManager Constants and Data Types

Security Level Indicators

In JManager 2.0, some security options are bound to the applet rather than to the session; that is, the same session can instantiate applets with different security options.

Session Security Indicators

When calling the `JMSetVerifyMode` function, you use the `CodeVerifierOptions` type to specify which Java code should be run through the verifier before execution.

```
enum CodeVerifierOptions {
    eDontCheckCode = 0,
    eCheckRemoteCode,
    eCheckAllCode
    };
```

Constant descriptions

`eDontCheckCode`	Don't verify any code.
`eCheckRemoteCode`	Verify any code that is read from a network.
`eCheckAllCode`	Verify all code.

When calling the `JMSetProxyInfo` function you must use the `JMProxyType` type to specify the type of proxy to use.

```
enum JMProxyType {
    eHTTPProxy = 0,
    eFirewallProxy,
    eFTPProxy
};
```

Constant descriptions

`eHTTPProxy`	An HTTP proxy
`eFirewallProxy`	A firewall proxy
`eFTPProxy`	An FTP proxy

Applet Security Indicators

When you set up an applet security data structure, you must use the `JMNetworkSecurityOptions` type to specify the security level for the applet when accessing a network.

```
enum JMNetworkSecurityOptions {
    eNoNetworkAccess = 0,
    eAppletHostAccess,
    eUnrestrictedAccess
    };
```

Constant descriptions

`eNoNetworkAccess`	The applet cannot access any networks.
`eAppletHostAccess`	The applet may access only its host server.
`eUnrestrictedAccess`	The applet has unrestricted access to all networks.

In addition, you must use the `JMFileSystemOptions` type to specify the security level allowed for applets accessing the local file system.

```
enum JMFileSystemOptions {
    eNoFSAccess = 0,
    eLocalAppletAccess,
    eAllFSAccess
};
```

Constant descriptions

`eNoFSAccess`	Applets have no access to the local file system.
`eLocalAppletAccess`	Only applets that are stored locally may access the local file system.
`eAllFSAccess`	All applets have access to the local file system.

See “The AWT Context” (page 70) for more information about using these security indicator types.

Runtime Session Options

When instantiating a Java runtime session, you can specify certain attributes using the `JMRuntimeOptions` mask.

```
enum JMRuntimeOptions {
    eJManager2Defaults      = 0,

    eUseAppHeapOnly         = (1 << 0),
    eDisableJITC            = (1 << 1),
    eEnableDebugger         = (1 << 2),
    eDisableInternetConfig  = (1 << 3),
    eInhibitClassUnloading  = (1 << 4),

    eJManager1Compatible = (eDisableJITC | eDisableInternetConfig)
};
```

Constant descriptions

`eJManager2Defaults`	The default group of settings. No mask attributes are set (application and temporary memory allowed, JITC enabled, debugger disabled, and so on).
`eUseAppHeapOnly`	When this bit is set, the Java runtime session only uses application heap memory (as opposed to temporary memory).
`eDisableJITC`	When this bit is set, the MRJ Just In Time Compiler is disabled.
`eEnableDebugger`	You should set this bit if you want to use any Java runtime debuggers.
`eDisableInternetConfig`	When this bit is set, InternetConfig is disabled (no default proxy server information can be obtained).
`eInhibitClassLoading`	When this bit is set, unreferenced classes are kept in memory (that is, they are not garbage-collected).

`eJManager1Compatible`
: This mask disables the JITC and InternetConfig, which makes the session compatible with JManager 1.0 functions.

You can choose one of the two preset masks or specify your own, depending on your needs.

The Text Object

Many JManager functions require you to pass strings in their parameter lists. These strings must be passed as a text object to allow use of different text encodings. Such a text object has the following type definition:

```
typedef struct JMText* JMTextRef;
```

Text Encoding Specifications

Some JManager functions require you to pass a text encoding specification in their parameter lists. Such text encoding specifications have the following type definition:

```
typedef TextEncoding JMTextEncoding;
```

JManager supports all the text encoding specifications used by the Text Encoding Converter. See *Programming with the Text Encoding Converter Manager* for a list of possible text encoding specifications.

Frame Types

When an AWT context requests a new frame through the `MyRequestFrame` callback, it specifies the type of frame desired. The type of frame is specified by the `JMFrameKind` type.

```
enum JMFrameKind {
    eBorderlessModelessWindowFrame = 0,
    eModelessWindowFrame,
    eModalWindowFrame
    };
```

Constant descriptions

`eBorderlessModelessWindowFrame`	A modeless borderless frame. This frame type is analogous to a Mac OS borderless window (for example, a window of type `plainDBox`). Other frames can appear on top of this one.
`eModelessWindowFrame`	A modeless frame. This frame type is analogous to a standard Mac OS window (with title bar, size box, and so on) such as those of type `zoomDocProc`.
`eModalWindowFrame`	A modal frame. This frame type is analogous to a Mac OS modal dialog window (for example a window of type `dBoxProc`). You should not create other frames on top of a modal frame, but you can create frames underneath it. Note that your application is responsible for enforcing the modal behavior of these windows.

See the description of the function `MyRequestFrame` (page 139) for more information about using these values.

For more information about Mac OS window types, see "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials.*

Frame Ordering Indicators

An AWT context may request that a frame be reordered using the `MyFrameReorderRequest` callback. The context specifies the new placement of the frame using the `ReorderRequest` type.

```
enum ReorderRequest {
    eBringToFront = 0,
    eSendToBack,
    eSendBehindFront
};
```

Constant descriptions

`eBringToFront`	Bring the frame to the front.
`eSendToBack`	Send the frame to the back.
`eSendBehindFront`	Send the frame behind the front frame.

See the description of the function `MyRequestFrame` (page 139) for more information about using these values.

Applet Locator Status Values

When the `JMNewAppletLocator` function (page 102) has retrieved an HTML document, it passes a status value to the `MyFetchCompleted` callback function. These status values are specified by the `JMLocatorErrors` type.

```
enum JMLocatorErrors {
    eLocatorNoErr = 0,
    eHostNotFound,
    eFileNotFound,
    eLocatorTimeout,
    eLocatorKilled
    };
```

Constant descriptions

`eLocatorNoErr`	The HTML text was retrieved successfully.
`eHostNotFound`	The host specified by the URL was not found.
`eFileNotFound`	The HTML file was not found on the host.
`eLocatorTimeout`	A timeout occurred while waiting for the HTML text.
`eLocatorKilled`	The `JMDisposeAppletLocator` function was called before the text could be retrieved.

See the description of the application-defined function `MyFetchCompleted` (page 146) for more information about using these values.

Miscellaneous Constants

A Java session may require you to pass one of these two constants when calling JManager functions.

```
enum {
    kJMVersion      = 0x11300003,
    kDefaultJMTime  = 0x00000400,
    };
```

Constant descriptions

`kJMVersion`	The version of JManager being used. The value for JManager 2.0 reflects adherence with Sun's 1.1.3 Java specifications.
`kDefaultJMTime`	The default idle time (1024 milliseconds) for `JMIdle` calls. For more information, see the `JMIdle` function (page 82).

The Java Runtime Session

Session Reference

A Java runtime session is defined by the `JMSessionRef` object, which has the following type definition:

```
typedef struct JMSession* JMSessionRef;
```

Session Callbacks Structure

When you create an instantiation of the Java runtime environment using the `JMOpenSession` function, you must pass a data structure that supplies callback information for that instantiation. The session callbacks data structure is defined by the `JMSessionCallbacks` data type.

```
struct JMSessionCallbacks {
    UInt32                  fVersion;           /* set to kJMVersion */
    JMConsoleProcPtr        fStandardOutput;    /* standard output */
    JMConsoleProcPtr        fStandardError;     /* standard error */
    JMConsoleReadProcPtr    fStandardIn;        /* standard input */
    JMExitProcPtr           fExitProc;          /* handle System.exit */
    JMAuthenticateURLProcPtrfAuthenticateProc;  /* for authentication */
    JMLowMemoryProcPtr      fLowMemProc;        /* low memory warning */
};
```

Field descriptions

`fVersion`
: The version of JManager. You should set this field to `kJMVersion`.

`fStandardOutput`
: A pointer to a function that handles text sent to the standard output. JManager sends all console output to this function. This callback has the following type definition:

```
typedef void (*JMConsoleProcPtr) (
    JMSessionRef session, const char* message,
    UInt32 messageLen);
```

For more information, see the description of the application-defined function `MyStandardOutput` (page 135).

Note that all text sent to this function will be encoded using the text encoding you specified when calling `JMOpenSession`.

`fStandardError`
: A pointer to a function that handles standard error output. JManager sends any error messages to this function. This callback has the following type definition:

```
typedef void (*JMConsoleProcPtr) (
    JMSessionRef session, const char* message,
    UInt32 messageLen);
```

For more information, see the description of the application-defined function `MyStandardError` (page 135).

Note that all text sent to this function will be encoded using the text encoding you specified when calling `JMOpenSession`.

`fStandardIn`
: A pointer to a function that handles console input. JManager accepts input from this routine. This value can be a null pointer, which indicates default behavior (no console input). This callback has the following type definition:

```
typedef SInt32 (*JMConsoleReadProcPtr) (
    JMSessionRef session, char* buffer,
    SInt32 maxBufferLength);
```

For more information, see the description of the application-defined function `MyStandardIn` (page 136).

`fExitProc`
A pointer to a function that handles calls to `java.lang.System.exit` (that is, requests to quit). This callback has the following type definition:

```
typedef Boolean (*JMExitProcPtr) (
    JMSessionRef session, int value);
```

For more information, see the description of the application-defined function `MyExit` (page 137).

`fAuthenticateProc`
A pointer to a function that handles user authentication requests (such as a request for a password) for a URL. This callback has the following type definition:

```
typedef Boolean (*JMAuthenticateURLProcPtr) (
    JMSessionRef session, const char* url,
    const char* realm, char userName[255],
    char password[255]);
```

For more information, see the description of the application-defined function `MyAuthenticate` (page 137).

`fLowMemProc`
A pointer to a function that handles low-memory conditions. This callback has the following type definition:

```
typedef Boolean (*JMLowMemoryProcPtr) (
    JMSessionRef session);
```

For more information, see the description of the application-defined function `MyLowMem` (page 138).

Proxy Server Options

When calling the `JMSetProxyInfo` function, you must pass a data structure containing information about the proxy server. The `JMGetProxyInfo` function returns information in this structure. The proxy server data structure is defined by the `JMProxyInfo` data type.

```
struct JMProxyInfo {
    Boolean useProxy;
    char proxyHost[255];
    UInt16 proxyPort;
};
```

Field descriptions

`useProxy`	If set to true, the specified proxy is to be used.
`proxyHost`	The name of the proxy server.
`proxyPort`	The port number of the proxy server.

For more information about using this structure, see the `JMGetProxyInfo` function (page 82) and `JMSetProxyInfo` function (page 83).

The AWT Context

AWT Reference

The Abstract Window Toolkit (AWT) context is defined by the `JMAWTContextRef` object, which has the following type definition:

```
typedef struct JMAWTContext* JMAWTContextRef;
```

AWT Context Callbacks Structure

When you create an AWT context associated with a session, you must pass a data structure that supplies callback information. The AWT context callback data structure is defined by the `JMAWTContextCallbacks` data type.

```
struct JMAWTContextCallbacks {
    UInt32                      fVersion;
    JMRequestFrameProcPtr       fRequestFrame;
    JMReleaseFrameProcPtr       fReleaseFrame;
    JMUniqueMenuIDProcPtr       fUniqueMenuID;
    JMExceptionOccurredProcPtr  fExceptionOccurred;
};
```

Field descriptions

`fVersion`	The version of JManager. You should set this field to `kJMVersion`.

`fRequestFrame` A pointer to a function that creates a new frame. The client application must set up the new frame and supply callbacks for the new frame. This callback function has the following type definition:

```
typedef OSStatus (*JMRequestFrameProcPtr) (
    JMAWTContextRef context, JMFrameRef newFrame,
    JMFrameKind kind, const Rect* InitialBounds,
    Boolean resizeable, JMFrameCallbacks* callbacks);
```

For more information, see the description of the application-defined function `MyRequestFrame` (page 139).

`fReleaseFrame` A pointer to a function that removes a frame. This callback function has the following type definition:

```
typedef OSStatus (*JMReleaseFrameProcPtr) (
    JMAWTContextRef context, JMFrameRef oldFrame);
```

For more information, see the application-defined function `MyReleaseFrame` (page 140).

`fUniqueMenuID` A pointer to a function that allocates a unique menu ID for later use in creating a menu. This callback function has the following type definition:

```
typedef SInt16 (*JMUniqueMenuIDProcPtr) (
    JMAWTContextRef context, Boolean isSubmenu);
```

For more information, see the description of the application-defined function `MyUniqueMenuID` (page 140).

`fExceptionOccurred` A pointer to an exception notification function. This function can indicate only that an error occurred; you cannot recover from the exception by using this function. This callback function has the following type definition:

```
typedef void (*JMExceptionOccurredProcPtr) (
    JMAWTContextRef context,
    const JMTextRef exceptionName,
    const JMTextRef exceptionMsg,
    const JMTextRef stackTrace);
```

For more information, see the description of the application-defined function `MyExceptionOccurred` (page 141).

The Applet Locator

Applet Locator Reference

An applet locator is defined by the `JMAppletLocatorRef` object, which has the following type definition:

```
typedef struct JMAppletLocator* JMAppletLocatorRef;
```

The Applet Locator Information Block

If you are synchronously creating an applet locator using the `JMNewAppletLocatorFromInfo` function, you must pass an information block that describes the location of the applet. This applet locator structure is defined by the `JMLocatorInfoBlock` data type.

```
struct JMLocatorInfoBlock {
    UInt32                  fVersion;
    JMTextRef               fBaseURL;
    JMTextRef               fAppletCode;
    short                   fWidth;
    short                   fHeight;
    int                     fOptionalParameterCount;
    JMLIBOptionalParams*    fParams;
};
```

Field descriptions

fVersion	The version of JManager. You should set this field to kJMVersion.
fBaseURL	A text object containing the URL of this applet's host page..
fAppletCode	A text object containing the location of the applet's code (that is, the name of the class file that contains the code). This information is the same as the code location described in an applet tag.
fWidth	The width of the applet, in pixels.
fHeight	The height of the applet, in pixels.
fOptionalParameterCount	The number of optional parameters.
fParams	A pointer to the first element in the array of optional parameters.

The fields fVersion, fBaseURL, fAppletCode, fWidth, and fHeight must be present and cannot be null values. The other fields are optional and can contain any parameters that need to be passed to the applet for execution. If there are no optional parameters, fOptionalParameterCount should be 0 and fParams should be null.

See the description of the JMNewAppletLocatorFromInfo function (page 102) for information on using this strucure.

For more information about the format of an applet tag, check the JavaSoft documentation available at the Web site

<http://java.sun.com/>

Applet Locator Optional Parameters

When passing the applet locator data structure to the JMNewAppletLocatorFromInfo function, you can provide optional parameters to be passed to the applet for execution. Such parameters are defined by the JMLibOptionalParams data type.

```
struct JMLIBOptionalParams {
    char* fParamName;
    char* fParamValue;
};
```

Field descriptions

`fParamName`	The name of the optional parameter (as found in the `NAME` field of a `<PARAM>` applet tag).
`fParamValue`	The value of the optional parameter (as found in the `VALUE` field of a `<PARAM>` applet tag).

Applet Locator Callback Structure

If you are asynchronously retrieving HTML information when creating an applet locator, you must pass a data structure that supplies a callback function. This data structure is defined by the `JMAppletLocatorCallbacks` data type.

```
struct JMAppletLocatorCallbacks {
    UInt32              fVersion;       /* set to kJMVersion */
    JMFetchCompleted    fCompleted;     /* text has been retrieved */
};
```

Field descriptions

`fVersion` The version of JManager. You should set this field to `kJMVersion`.

`fCompleted` A pointer to a function that should execute after the HTML data has been successfully retrieved and parsed. The callback function has the following type definition:

```
typedef void (*JMFetchCompleted) (
    JMAppletLocatorRef ref, JMLocatorErrors status);
```

For more information, see the description of the application-defined function `MyFetchCompleted` (page 146).

The Applet Object

Applet Reference

An instantiated applet is defined by the `JMAppletViewerRef` object, which has the following type definition:

```
typedef struct JMAppletViewer* JMAppletViewerRef;
```

Note that the `JMAppletViewer` object is not the same as a `java.applet.Applet` object. The `JMAppletViewer` object encapsulates the Java applet object so that it can be referenced outside the Java environment.

Applet Callbacks Structure

When you instantiate an applet using the `JMNewAppletViewer` function (page 108), you must pass a data structure that supplies callback information for the applet. This data structure is defined by the `JMAppletViewerCallbacks` data type.

```
struct JMAppletViewerCallbacks {
    UInt32                  fVersion;
    JMShowDocumentProcPtr   fShowDocument;
    JMSetStatusMsgProcPtr   fSetStatusMsg;
};
```

Field descriptions

`fVersion`	The version of JManager. You should set this field to `kJMVersion`.
`fShowDocument`	A pointer to a function that displays the contents of a URL passed to it, possibly in a new window. This callback function has the following type definition: `typedef void (*JMShowDocumentProcPtr) ( JMAppletViewerRef viewer, const JMTextRef urlString, const JMTextRef windowName);` For more information, see the description of the application-defined function `MyShowDocument` (page 146).
`fSetStatusMsg`	A pointer to a function that handles messages from the applet. The client application can display the message (in a status bar, for example) or ignore it. This callback function has the following type definition:

```
typedef void (*JMShowDocumentProcPtr) (
    JMAppletViewerRef viewer,
    const JMTextRef urlString,
    const JMTextRef windowName);
```

```
typedef void (*JMSetStatusMsgProcPtr) (
    JMAppletViewerRef viewer,
    const JMTextRef statusMsg);
```

For more information, see the description of the application-defined function `MySetStatusMsg` (page 147).

Applet Security Structure

When you instantiate an applet using the `JMNewAppletViewer` function (page 108), you must pass a data structure that supplies security information for the applet. This data structure is defined by the `JMAppletSecurity` data type.

```
struct JMAppletSecurity {
    UInt32 fVersion;
    JMNetworkSecurityOptions fNetworkSecurity;
    JMFileSystemOptions fFileSystemSecurity;
    Boolean fRestrictSystemAccess;
    Boolean fRestrictSystemDefine;
    Boolean fRestrictApplicationAccess;
    Boolean fRestrictApplicationDefine;
};
```

Field descriptions

`fVersion`	The version of JManager. You should set this field to `kJMVersion`.
`JMNetworkSecurityOptions`	A flag indicating access privileges for applets connecting to networks. See "Applet Security Indicators" (page 62) for a list of possible values for this field.
`JMFileSystemOptions`	A flag indicating applet access privileges to the local file system. See "Applet Security Indicators" (page 62) for a list of possible values for this field.
`fRestrictSystemAccess`	If set to true, the applet cannot access system packages found in the `mrj.security.system.access` property. (The default packages in the property are `com.apple.*` and `sun.*`.)

`fRestrictSystemDefine`	If set to true, the applet cannot load system packages found in the `mrj.security.system.define` property. (The default packages in the property are `com.apple.*` and `sun.*`.)
`fRestrictApplicationAccess`	If set to true, the applet cannot access application packages found in the `mrj.security.application.access` property.
`fRestrictApplicationDefine`	If set to true, the applet cannot load application packages found in the `mrj.security.application.define` property.

The Frame Object

Frame Reference

A frame is defined by the `JMFrameRef` object, which has the following type definition:

```
typedef struct JMFrame* JMFrameRef;
```

Note that the `JMFrameRef` object is not the same as a `java.awt.Frame` object. The `JMFrameRef` object encapsulates the Java frame object so that it can be referenced outside the Java environment.

Frame Callbacks Structure

When you create an AWT context associated with a session, you must pass a data structure that provides frame callback information. This data structure is defined by the `JMFrameCallbacks` data type.

```
struct JMFrameCallbacks {
    UInt32 fVersion;
    JMSetFrameSizeProcPtr   fSetFrameSize;
    JMFrameInvalRectProcPtr fInvalRect;
    JMFrameShowHideProcPtr  fShowHide;
    JMSetTitleProcPtr       fSetTitle;
    JMCheckUpdateProcPtr    fCheckUpdate;
    JMReorderFrame          fReorderFrame;
    JMSetResizeable         fSetResizeable;
};
```

Field descriptions

`fVersion`	The version of JManager. You should set this field to `kJMVersion`.
`fSetFrameSize`	A pointer to a function that handles a frame sizing request. This callback function has the following type definition:

```
typedef void (*JMSetFrameSizeProcPtr) (
    JMFrameRef frame, const Rect* newBounds);
```

For more information, see the description of the application-defined function `MyResizeRequest` (page 142).

`fInvalRect`	A pointer to a function that handles a frame invalidation request. This callback function has the following type definition:

```
typedef void (*JMFrameInvalRectProcPtr) (
    JMFrameRef frame, const Rect* r);
```

For more information, see the description of the application-defined function `MyInvalRect` (page 142)

`fShowHide`	A pointer to a window show/hide function. This callback function has the following type definition:

```
typedef void (*JMFrameShowHideProcPtr) (
    JMFrameRef frame, Boolean showFrameRequested);
```

For more information, see the description of the application-defined function `MyShowHide` (page 143).

`fSetTitle`	A pointer to a function that sets the title bar text for a frame. This callback function has the following type definition:

```
typedef void (*JMSetTitleProcPtr) (
    JMFrameRef frame, const JMTextRef title);
```

For more information, see the description of the application-defined function `MySetTitle` (page 143).

`fCheckUpdate`
: A pointer to a function that allows the frame to be updated during an interaction (such as a mouse drag). This callback function has the following type definition:

```
typedef void (*JMCheckUpdateProcPtr) (
    JMFrameRef frame);
```

For more information, see the description of the application-defined function `MyCheckUpdate` (page 144).

`fReorderFrame`
: A pointer to a function that changes the ordering of the frame. For example, you can bring a frame to the front or send it to the back. This callback function has the following type definition:

```
typedef void (*JMReorderFrame) (
    JMFrameRef frame,
    enum ReorderRequest theRequest);
```

For more information, see the description of the application-defined function `MyFrameReorder` (page 144).

`fSetResizeable`
: A pointer to a function that sets a frame as resizeable or not. This callback has the following type definition:

```
typedef void (*JMSetResizeable) (
    JMFrameRef frame, Boolean resizeable);
```

For more information, see the description of the application-defined function `MySetResizeable` (page 145).

Client-Specific Data

When working with a session, AWT context, applet locator, applet, or frame, you must often set or retrieve arbitrary client-specific data. Such data has the following type definition:

```
typedef void* JMClientData;
```

JManager Functions

Runtime Invocation Functions

All JManager API calls reference an instantiation of the Java runtime environment called a session or a `JMSessionRef` object. This section describes routines that create and dispose of a Java runtime session.

JMOpenSession

Instantiates a Java runtime session and returns a session pointer.

```
OSStatus JMOpenSession (
                    JMSessionRef* session,
                    JMRuntimeOptions runtimeOptions,
                    JMVerifierOptions verifyMode,
                    const JMSessionCallbacks* callbacks,
                    JMTextEncoding desiredEncoding,
                    JMClientData data);
```

`session`	A pointer to a Java runtime session. On return, this parameter is the allocated session.
`runtimeOptions`	A pointer to a runtime options structure. See “Runtime Session Options” (page 63) for more information.
`verifyMode`	A flag indicating whether to use the code verifier. See “Session Security Indicators” (page 61) for a list of possible values.
`callbacks`	A pointer to a session callbacks structure. See “Session Callbacks Structure” (page 67) for more information.
`desiredEncoding`	The text encoding to use for any text sent to the designated standard output or standard error.

`data`	Any optional client-specific data.
function result	A result code; see “JManager Result Codes” (page 148).

DISCUSSION

A valid session pointer has a value other than `null`. If not enough system memory exists to instantiate a session, `JMOpenSession` returns `memFullErr`.

SEE ALSO

“Beginning a Java Runtime Session” (page 21).

The `JMCloseSession` function (page 81).

JMCloseSession

Ends a Java runtime session and removes the allocated `JMSessionRef` object.

```
OSStatus JMCloseSession (JMSessionRef session);
```

`session`	The session to be removed.

DISCUSSION

A successful call to `JMCloseSession` also frees any resources allocated by JManager. However, any resources that you explicitly allocated on behalf of JManager (such as AWT contexts or applets) must be explicitly removed before calling `JMCloseSession`.

SEE ALSO

The `JMOpenSession` function (page 80).

JMIdle

Allows JManager time to service other threads.

```
OSStatus JMIdle     (JMSessionRef session,
                     UInt32 JMTimeMillis);
```

`session`	The current session.
`JMTimeMillis`	The amount of time, in milliseconds, allowed to service other threads. For the default wait period, you can set this parameter to `kDefaultJMTime`.
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

If no threads need to be serviced, `JMIdle` returns immediately. `JMIdle` also returns, suspending other threads, if a user event occurs in the current session. You should call the `JMIdle` function once each time through the event loop.

SEE ALSO

"Servicing Other Threads" (page 25).

The `JMFrameMouseOver` function (page 124).

JMGetProxyInfo

Gets the proxy information for a given session.

```
OSStatus JMGetProxyInfo (
                    JMSessionRef session,
                    JMProxyType type,
                    JMProxyInfo* proxyInfo);
```

`session`	The session whose proxy information you wish to determine.
`type`	The type of proxy you want to query. See "Session Security Indicators" (page 61) for a list of possible values to pass.

`proxyInfo`	A pointer to a proxy information structure. on return, this structure contains the proxy information for the specified proxy type. For more information, see "Proxy Server Options" (page 69).
function result	A result code; see "JManager Result Codes" (page 148).

SEE ALSO

The `JMSetProxyInfo` function (page 83).

"Specifying Proxy Servers" (page 24).

JMSetProxyInfo

Sets the proxy information for an existing session.

```
OSStatus JMSetProxyInfo (
                  JMSessionRef session,
                  JMProxyType type,
                  const JMProxyInfo* proxyInfo);
```

`session`	The session whose proxy information you wish to set.
`type`	The type of proxy you want to query. See "Session Security Indicators" (page 61) for a list of possible values to pass.
`proxyInfo`	A pointer to a proxy information structure containing the proxy information to set. For more information, see "Proxy Server Options" (page 69).
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

You may specify different proxy servers for HTTP, firewall, and FTP access.

SEE ALSO

The `JMGetProxyInfo` function (page 82).

"Specifying Proxy Servers" (page 24).

JMGetVerifyMode

Gets the state of the code verifier for an existing session.

```
OSStatus JMGetVerifyMode (
                        JMSessionRef session,
                        JMVerifierOptions* verifierOptions);
```

`session`	The session whose code verifier mode you wish to set.
`verifierOptions`	A pointer to a code verifier option. on return, this parameter is the state of the code verifier. See "Session Security Indicators" (page 61) for a list of possible values.
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

The code verifier checks to see that the Java code is valid and that it does not attempt any illegal actions that could affect the host platform. You set the code verifier either when instantiating the session (using the `JMOpenSession` function (page 80)) or by calling the `JMSetVerifyMode` function (page 84).

JMSetVerifyMode

Sets the code verifier mode for an existing session.

```
OSStatus JMSetVerifyMode (JMSessionRef session,
                        JMVerifierOptions verifierOptions);
```

`session`	The session whose code verifier mode you wish to set.
`verifierOptions`	The code verifier option you want to set. See "Session Security Indicators" (page 61) for a list of possible values.

function result A result code; see "JManager Result Codes" (page 148).

DISCUSSION

The verifier checks to see that the Java code is valid and that it does not attempt any illegal actions that could affect the host platform. You can also set the verifier mode when calling the `JMOpenSession` function (page 80).

SEE ALSO

The `JMGetVerifyMode` function (page 84).

JMGetSessionData

Obtains client data for an existing session.

```
OSStatus JMGetSessionData (
                        JMSessionRef session,
                        JMClientData* data);
```

`session`	The session whose client data you wish to obtain.
`data`	A pointer to client data. on return, this parameter contains the received client data.
function result	A result code; see "JManager Result Codes" (page 148).

SEE ALSO

The `JMSetSessionData` function (page 86).

JMSetSessionData

Sets client data for an existing session.

```
OSStatus JMSetSessionData (
                    JMSessionRef session,
                    JMClientData data);
```

`session`	The session whose client data you wish to change.
`data`	A pointer to the new client data.
function result	A result code; see "JManager Result Codes" (page 148).

SEE ALSO

The `JMGetSessionData` function (page 85).

JMGetSessionProperty

Obtains a property value for an existing session.

```
OSStatus JMGetSessionProperty (
                    JMSessionRef session,
                    const JMTextRef propertyName,
                    JMTextRef* propertyValue;
```

`session`	The session whose property value you wish to obtain.
`propertyName`	A text object holding the name of the property whose value you want to obtain.
`propertyValue`	A pointer to the text object holding the property value. on return, this location contains the value of `propertyName`.
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

This function corresponds to the Java method `java.lang.System.getProperty`.

SEE ALSO

The `JMPutSessionProperty` function (page 87).

JMPutSessionProperty

Adds or modifies a session property.

```
OSStatus JMPutSessionProperty (
                    JMSessionRef session,
                    const JMTextRef propertyName,
                    const JMTextRef propertyValue);
```

`session`	The session whose property you wish to set.
`propertyName`	A text object holding the name of the property whose value you want to add or modify.
`propertyValue`	The value to set for `propertyName`, as a `JMTextRef` object.
function result	A result code; see “JManager Result Codes” (page 148).

DISCUSSION

If the property does not exist, JManager creates a new one with the name in `propertyName` and the value in `propertyValue`. This function corresponds to the Java method `java.lang.System.setProperty`.

SEE ALSO

The `JMGetSessionProperty` function (page 86).

JMGetSessionObject

Obtains a JRI reference for an existing session.

```
jref JMGetSessionObject (JMSessionRef session);
```

`session`	The session whose object reference you wish to obtain.
function result	A pointer to the `com.apple.mrj.JManager.JMSession` object.

DISCUSSION

The `JMGetSessionObject` returns the JRI reference equivalent (of type `jref`) of the `JMSessionRef` reference.

Text Handling Functions

All text handled by JManager functions are stored as `JMTextRef` objects, which , in addition to the actual text, also specify the text encoding and the length of the string. The following functions let you create or manipulate a `JMTextRef` object.

JMNewTextRef

Creates a new text object.

```
OSStatus JMNewTextRef (
                    JMSessionRef session,
                    JMTextRef* textRef,
                    JMTextEncoding encoding,
                    const void* charBuffer,
                    UInt32 bufferLengthInBytes
                    );
```

`session`	The session in which you want to create the text object.
`textRef`	A pointer to a text object. on return, this parameter is the new text object.
`encoding`	The text encoding of the string to encapsulate in the text object. You can pass any text encoding specification defined by the Text Encoding Converter. See "Text Encoding Specifications" (page 64) for more information.
`charBuffer`	The string to encapsulate in the text object.

`bufferLengthInBytes`
The length of the string.

function result A result code; see "JManager Result Codes" (page 148).

DISCUSSION

If you create a text object, it is your responsibility to dispose of it after use.

SEE ALSO

The `JMDisposeTextRef` function (page 89).

The `JMGetTextBytes` function (page 90).

JMDisposeTextRef

Removes a text object.

```
OSStatus JMDisposeTextRef(JMTextRef textRef);
```

`textRef`	The text object you want to remove.
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

If you create a text object, it is your responsibility to dispose of it after use.

SEE ALSO

The `JMNewTextRef` function (page 88).

JMCopyTextRef

Duplicates a text object.

```
OSStatus JMCopyTextRef (
              const JMTextRef textRefSrc,
              JMTextRef* textRefDst);
```

`textRefSrc`	The text object you want to duplicate.
`textRefDst`	A pointer to a text object. On return, this parameter contains a copy of the `textRefSrc` text object.
function result	A result code; see "JManager Result Codes" (page 148).

JMGetTextLength

Returns the length of the string in a text object.

```
OSStatus JMGetTextLength (
              const JMTextRef textRef,
              UInt32* textLengthInCharacters);
```

`textRef`	The text object containing the string.
`textLengthInCharacters`	A pointer to an integer. On return, this parameter contains the length of the string, in characters.
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

This function returns the string length in characters rather than bytes, since the Unicode standard uses 2 bytes per character, while standard Mac OS encodings and UTF-8 use only 1.

JMGetTextBytes

Retrieves characters in a text object in the appropriate text encoding.

```
OSStatus JMGetTextBytes (
              const JMTextRef textRef,
              JMTextEncoding dstEncoding,
              void* textBuffer,
              UInt32 textBufferLength,
              UInt32* numCharsCopied);
```

`textRef`	The text object containing the string to retrieve.
`dstEncoding`	The text encoding you want to use for the string.
`textBuffer`	A pointer to a buffer. on return, this parameter is the retrieved string.
`textBufferLength`	The length of the buffer, in bytes.
`numCharsCopied`	A pointer to an integer. On return, this parameter is the actual length of the string retrieved, in characters.
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

The number of characters retrieved is dependent on the size you specify for the buffer as well as on the text encoding. For example, Unicode characters can take 2 bytes per character.

SEE ALSO

The `JMNewTextRef` function (page 88).

JMTextToJavaString

Returns the text object as a Java string.

```
jref JMTextToJavaString (const JMTextRef textRef);
```

`textRef`	The text object to convert.
function result	A reference to a Java string

DISCUSSION

This reference is the only reference to the string; if you do not use it, the session will garbage-collect it.

SEE ALSO

The `JMTextToMacOSCStringHandle` function (page 92).

JMTextToMacOSCStringHandle

Returns the text object as a null-terminated Mac OS C string in the current system text encoding.

```
Handle JMTextToMacOSCStringHandle(const JMTextRef textRef);
```

`textRef`	The text object to convert.
function result	A handle to a Mac OS C string.

DISCUSSION

You must dispose of the handle (by calling the Mac OS Toolbox function `DisposeHandle`) after you are finished using it.

SEE ALSO

The `JMTextToJavaString` function (page 91).

Abstract Window Toolkit Control Functions

Every Java program that needs access to the Abstract Window Toolkit (AWT) must have an AWT context associated with it. This AWT context supplies an execution environment for the Java program and an associated thread group. The following functions let you create or remove an AWT context or manipulate elements associated with the context.

JMNewAWTContext

Creates an AWT context.

```
OSStatus JMNewAWTContext (
                    JMAWTContextRef* context,
                    JMSessionRef session,
                    const JMAWTContextCallbacks* callbacks,
                    JMClientData data);
```

`context`	A pointer to the AWT context. on return, this parameter is the new AWT context.
`session`	The session in which you want to create the AWT context.
`callbacks`	A pointer to the context's callbacks. See "AWT Context Callbacks Structure" (page 70) for more information.
`data`	Any optional client-specific data.
function result	A result code; see "JManager Result Codes" (page 148).

SEE ALSO

"Creating an AWT Context" (page 30).

The `JMDisposeAWTContext` function (page 93).

JMDisposeAWTContext

Removes an AWT context.

```
OSStatus JMDisposeAWTContext (JMAWTContextRef context);
```

`context`	The AWT context you want to remove.
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

A successful call to the `JMDisposeAWTContext` function also removes any frames associated with the AWT context.

SEE ALSO

"Creating an AWT Context" (page 30).

The `JMNewAWTContext` function (page 93).

JMGetAWTContextData

Receives client-specific data associated with an AWT context.

```
OSStatus JMGetAWTContextData (
                    JMAWTContextRef context,
                    JMClientData* data);
```

`context`	The AWT context whose data you want to receive.
`data`	A pointer to the client-specific data. On return, this parameter points to the client-specific data.
function result	A result code; see "JManager Result Codes" (page 148).

SEE ALSO

The `JMSetAWTContextData` function (page 94).

JMSetAWTContextData

Assigns client-specific data to an AWT context.

```
OSStatus JMSetAWTContextData (
                    JMAWTContextRef context,
                    JMClientData data);
```

`context`	The AWT context whose client-specific data you want to change.
`data`	The new value of the client-specific data.
function result	A result code; see “JManager Result Codes” (page 148).

SEE ALSO

The `JMGetAWTContextData` function (page 94).

JMCountAWTContextFrames

Counts the number of frames associated with an AWT context.

```
OSStatus JMCountAWTContextFrames (
                        JMAWTContextRef context,
                        UInt32* frameCount);
```

`context`	The AWT context whose frames are being counted.
`frameCount`	A pointer to the frame count. On return, this parameter holds the number of allocated frames associated with the context. (This value may be 0.)
function result	A result code; see “JManager Result Codes” (page 148).

DISCUSSION

After determining the number of frames, you can then access an individual frame by calling the `JMGetAWTContextFrame` function (page 96).

JMGetAWTContextFrame

Gets a particular frame associated with an AWT context.

```
OSStatus JMGetAWTContextFrame (
                    JMAWTContextRef context,
                    UInt32 frameIndex,
                    JMFrameRef* frame);
```

`context`	The AWT context that contains the frame.
`frameIndex`	The index number of the frame.
`frame`	A pointer to the frame. On return, this parameter holds the frame with index `frameIndex`.
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

The index numbers for frames range from 0 to `frameCount` –1 (as determined by the `JMCountAWTContextFrames` function (page 95)), with the most recently added frame having the highest number. The index number of a particular frame is not necessarily constant; removing or adding frames can cause the index number to change.

JMMenuSelected

Dispatch a menu event to an AWT context.

```
OSStatus JMMenuSelected (
                    JMAWTContextRef context,
                    MenuHandle hMenu,
                    short menuItem);
```

`context`	The AWT context that owns the menu.
`hMenu`	The menu handle that was selected.
`menuItem`	The item that was selected.

function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

You can use the Mac OS Toolbox function `GetMenuHandle` to get the appropriate value of the `hMenu` parameter to pass.

JMExecJNIMethodInContext

Executes a nonstatic Java method in a given AWT context thread using the Java Native Interface (JNI).

```
OSStatus JMExecJNIMethodInContext (
                    JMAWTContextRef context,
                    JNIEnv* env,
                    jobject objref,
                    jmethodID methodID,
                    UInt32 argCount,
                    jvalue args[]);
```

`context`	The AWT context in whose thread you want the method to execute.
`env`	A pointer to the current `JNIEnv` data structure.
`objref`	A pointer to the Java object that contains the method you want to call.
`methodID`	The ID of the method.
`argCount`	The number of arguments in the method.
`args[]`	The argument list.
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

Before calling this function, you must call the `JMGetCurrentEnv` function (page 131) to get the `JNIEnv` pointer.

If you want to execute a static Java method (that is, one that is not local to an object) using the JNI, you must call the `JMExecJNIStaticMethodInContext` function (page 100) instead.

You can find documentation on the Java Native Interface (JNI) at the Web page

<http://java.sun.com/>

JMExecJNIStaticMethodInContext

Executes a static Java method in a given AWT context thread using the Java Native Interface (JNI).

```
OSStatus JMExecJNIStaticMethodInContext (
                    JMAWTContextRef context,
                    JNIEnv* env,
                    jclass classID,
                    jmethodID methodID,
                    UInt32 argCount,
                    jvalue args[]);
```

`context`	The AWT context in whose thread you want the method to execute.
`env`	A pointer to the current `JNIEnv` data structure.
`classID`	The class ID of the class that contains the method.
`methodID`	The ID of the method.
`argCount`	The number of arguments in the method
`args[]`	The argument list.
function result	A result code; see “JManager Result Codes” (page 148).

DISCUSSION

Before calling this function, you must call the `JMGetCurrentEnv` function (page 131) to get the `JNIEnv` pointer.

For information about using this function to launch a Java application, see “Executing Java Applications” (page 50).

If you want to execute a nonstatic Java method, you should call the `JMExecJNIMethodInContext` function (page 97) instead.

You can find documentation on the Java Native Interface (JNI) at the Web page

<http://java.sun.com/>

JMExecMethodInContext

Executes a nonstatic Java method in a given AWT context thread using the Java Runtime Interface (JRI).

```
OSStatus JMExecMethodInContext (
                     JMAWTContextRef context,
                     jref objref,
                     JRIMethodID methodID,
                     UInt32 argCount,
                     JRIValue args[]);
```

`context`	The AWT context in whose thread you want the method to execute.
`objref`	A pointer to the Java object that contains the method you want to call.
`methodID`	The ID of the method.
`argCount`	The number of arguments in the method.
`args[]`	The argument list.
function result	A result code; see “JManager Result Codes” (page 148).

DISCUSSION

Unless you have a particular reason to access the Java Runtime Interface, you should instead make calls to the Java Native Interface, which provides similar functionality.

If you want to execute a static Java method (that is, one that is not local to an object) using the JRI, you must call the `JMExecStaticMethodInContext` function (page 100) instead.

You can find documentation on the Java Runtime Interface (JRI) at the Web page

<http://developer.netscape.com/>

SEE ALSO

The `JMExecJNIMethodInContext` function (page 97).

The `JMExecJNIStaticMethodInContext` function (page 100).

JMExecStaticMethodInContext

Executes a static Java method in a given AWT context thread using the Java Runtime Interface (JRI).

```
OSStatus JMExecStaticMethodInContext (
                    JMAWTContextRef context,
                    JRIClassID classID,
                    JRIMethodID methodID,
                    UInt32 argCount,
                    JRIValue args[]);
```

`context`	The AWT context in whose thread you want the method to execute.
`classID`	The class ID of the class that contains the method.
`methodID`	The ID of the method.
`argCount`	The number of arguments in the method
`args[]`	The argument list.
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

Unless you have a particular reason to access the Java Runtime Interface, you should instead make calls to the Java Native Interface, which provides similar functionality.

If you want to execute a nonstatic Java method using the JRI, you should call the `JMExecMethodInContext` function (page 99) instead.

You can find documentation on the Java Runtime Interface (JRI) at the Web page

<http://developer.netscape.com/>

SEE ALSO

The `JMExecJNIMethodInContext` function (page 97).

The `JMExecJNIStaticMethodInContext` function (page 100).

JMGetAwtContextObject

Returns a reference to a context's Java object.

```
jref JMGetAwtContextObject (JMAWTContextRef context);
```

`context`	The context whose Java object you want to find.
function result	A pointer to the `com.apple.mrj.JManager.AWTContext` object associated with the frame.

DISCUSSION

This function returns the Java equivalent of the `JMAWTContextRef` reference.

Applet Control Functions

With the following functions you can find an applet through an HTML page, instantiate it within an AWT context, and set or receive information associated with it.

JMNewAppletLocatorFromInfo

Locates an applet synchronously using information in the applet locator information block structure.

```
OSStatus JMNewAppletLocatorFromInfo (
                    JMAppletLocatorRef* locatorRef,
                    JMSessionRef session,
                    const JMLocatorInfoBlock* info,
                    JMClientData data);
```

`locatorRef`	A pointer to the locator. On return, this parameter is the new applet locator.
`session`	The session in which you want to instantiate the applet.
`info`	A pointer to an applet locator information block structure. For more information, see "The Applet Locator Information Block" (page 72).
`data`	Optional client-specific data.
function result	A result code; see "JManager Result Codes" (page 148).

SEE ALSO

"Finding Applets" (page 26).

The `JMNewAppletLocator` function (page 102).

The `JMDisposeAppletLocator` function (page 104).

JMNewAppletLocator

Locates an applet asynchronously by fetching an HTML document from a specified URL.

```
OSStatus JMNewAppletLocator (
                    JMAppletLocatorRef* locatorRef,
                    JMSessionRef session,
                    const JMAppletLocatorCallbacks* callbacks,
                    const JMTextRef url,
                    const JMTextRef htmlText,
                    JMClientData data);
```

`locatorRef`	A pointer to the locator. On return, this parameter is the new applet locator.
`session`	The session in which you want to instantiate the applet.
`callbacks`	A pointer to an applet locator callbacks structure. For more information, see "Applet Locator Callback Structure" (page 74).
`url`	A text object containing the text of the URL.
`htmlText`	Optional text (stored as a text object) containing one or more applet tags.
`data`	Optional client-specific data.
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

If you have already retrieved the HTML document, you can pass it to `JMNewAppletLocator` in the `htmlText` argument. Otherwise, `JMNewAppletLocator` starts a separate thread to retrieve the document. The client application must call `JIdle` to allow the new thread time to retrieve the document. The callback that executes upon fetching the HTML document can occur either while this function is executing or some time after it has returned.

SEE ALSO

"Finding Applets" (page 26).

The `JMNewAppletLocatorFromInfo` function (page 102).

The `JMDisposeAppletLocator` function (page 104).

JMDisposeAppletLocator

Removes an applet locator.

```
OSStatus JMDisposeAppletLocator (JMAppletLocatorRef locatorRef);
```

`locatorRef`	The applet locator to remove.
function result	A result code; see “JManager Result Codes” (page 148).

DISCUSSION

Since the applet locator merely locates an applet, you can dispose of it after instantiating the applet. You can call `JMDisposeAppletLocator` while the thread created by `JMNewAppletLocator` function (page 102) is still searching for the applet.

SEE ALSO

The `JMNewAppletLocatorFromInfo` function (page 102).

JMGetAppletLocatorData

Retrieves client data associated with an applet locator.

```
OSStatus JMGetAppletLocatorData (
                        JMAppletLocatorRef locatorRef,
                        JMClientData* data);
```

`locatorRef`	The applet locator whose client data you want to obtain.
`data`	A pointer to the client data. on return, this parameter points to the client data.
function result	A result code; see “JManager Result Codes” (page 148).

SEE ALSO

The `JMSetAppletLocatorData` function (page 105).

JMSetAppletLocatorData

Assigns client data to an applet locator.

```
OSStatus JMSetAppletLocatorData (
                    JMAppletLocatorRef locatorRef,
                    JMClientData data);
```

`locatorRef`	The applet locator whose client data you want to set.
`data`	The client data.
function result	A result code; see "JManager Result Codes" (page 148).

SEE ALSO

The `JMGetAppletLocatorData` function (page 104).

JMCountApplets

Counts the number of applets associated with an HTML page.

```
OSStatus JMCountApplets (
                    JMAppletLocatorRef locatorRef,
                    UInt32* appletCount);
```

`locatorRef`	The applet locator that contains the parsed HTML text.
`appletCount`	A pointer to the applet count. On return, this parameter points to the number of applets.
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

If there are no applets associated with the text, `JMCountApplets` returns 0. You can call this function only after successfully retrieving HTML text using the `JMNewAppletLocator` function (page 102).

SEE ALSO

"Finding Applets" (page 26).

JMGetAppletDimensions

Returns the dimensions of an applet.

```
OSStatus JMGetAppletDimensions (
                    JMAppletLocatorRef locatorRef,
                    UInt32 appletIndex,
                    UInt32* width,
                    UInt32* height);
```

`locatorRef`	The applet locator that contains the retrieved HTML text.
`appletIndex`	The index number of the applet you want to query.
`width`	A pointer that, on return, contains the width, in pixels, of the applet.
`height`	A pointer that, on return, contains the height, in pixels, of the applet.
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

The `appletIndex` value is an index number from 0 to `appletCount` –1, where `appletCount` is determined by the `JMCountApplets` function (page 105). You can call this function only after successfully retrieving HTML text using the `JMNewAppletLocator` function (page 102).

SEE ALSO

"Finding Applets" (page 26).

JMGetAppletTag

Returns the tag associated with an applet.

```
OSStatus JMGetAppletTag (
                    JMAppletLocatorRef locatorRef,
                    UInt32 appletIndex,
                    JMTextRef* tagRef);
```

`locatorRef`	The applet locator that contains the retrieved HTML text.
`appletIndex`	The index number of the applet you want to query.
`tagRef`	A pointer to a text object. on return, the `tagRef` object contains the applet tag.
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

The `JMGetAppletTag` function returns the text bounded by the `<APPLET>` and `</APPLET>` delimeters in an HTML document. The `appletIndex` value is an index number from 0 to `appletCount` –1, where `appletCount` is determined by the `JMCountApplets` function (page 105). You can call this function only after successfully retrieving HTML text using the `JMNewAppletLocator` function (page 102).

SEE ALSO

"Finding Applets" (page 26).

JMGetAppletName

Returns the name of an applet.

```
OSStatus JMGetAppletName (
                    JMAppletLocatorRef locatorRef,
                    UInt32 appletIndex,
                    JMTextRef* nameRef);
```

`locatorRef`	The applet locator that contains the retrieved HTML text.
`appletIndex`	The index number of the applet you want to query.
`nameRef`	A pointer to a text object. On return, the `nameRef` object contains the name of the applet.
function result	A result code; see “JManager Result Codes” (page 148).

DISCUSSION

The `appletIndex` value is an index number from 0 to `appletCount` –1, where `appletCount` is determined by the `JMCountApplets` function (page 105). You must reserve a buffer for the text object containing the applet name and pass a pointer to the buffer when you call `JMGetAppletName`. You can call this function only after successfully retrieving HTML text using the `JMNewAppletLocator` function (page 102).

SEE ALSO

“Finding Applets” (page 26).

JMNewAppletViewer

Instantiates an applet.

```
OSStatus JMNewAppletViewer (
                    JMAppletViewerRef* viewer,
                    JMAWTContextRef context,
                    JMAppletLocatorRef locatorRef,
                    UInt32 appletIndex,
                    const JMAppletSecurity* security
                    const JMAppletViewerCallbacks* callbacks,
                    JMClientData data);
```

`viewer`	A pointer to an applet. On return, this parameter is the newly instantiated applet.
`context`	The AWT context associated with the applet.
`locatorRef`	The applet locator containing the applet.

`appletIndex`	The index of the applet to instantiate.
`security`	A pointer to the applet security structure. See "Applet Security Structure" (page 76) for more information.
`callbacks`	A pointer to the applet viewer callbacks structure. See "Applet Callbacks Structure" (page 75) for more information.
`data`	Optional client-specific data.
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

To instantiate an applet, you must first create an AWT context (and start the thread associated with it) and create an applet locator. The `appletIndex` value is an index number from 0 to `appletCount` –1, where `appletCount` is determined by the `JMCountApplets` function.

SEE ALSO

"Instantiating Applets" (page 36).

The `JMCountApplets` function (page 105).

The `JMDisposeAppletViewer` function (page 109).

JMDisposeAppletViewer

Removes an applet.

```
OSStatus JMDisposeAppletViewer (JMAppletViewerRef viewer);
```

`viewer`	The instantiated applet to remove.
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

Calling the `JMDisposeAppletLocator` function first halts execution of the applet if necessary. This function also disposes of the applet's frame (if visible) and any other frames created by the applet.

SEE ALSO

The `JMNewAppletViewer` function (page 108).

JMGetAppletViewerData

Retrieves client data associated with an applet.

```
OSStatus JMGetAppletViewerData (
                    JMAppletViewerRef viewer,
                    JMClientData* data);
```

`viewer`	The applet to query.
`data`	A pointer to the client data. On return, this parameter holds the client data.
function result	A result code; see "JManager Result Codes" (page 148).

SEE ALSO

The `JMSetAppletViewerData` function (page 110).

JMSetAppletViewerData

Assigns client data to an applet.

```
OSStatus JMSetAppletViewerData (
                    JMAppletViewerRef viewer,
                    JMClientData data);
```

`viewer`	The applet whose client data you want to set.

`data`	The client data.
function result	A result code; see "JManager Result Codes" (page 148).

SEE ALSO

The `JMGetAppletViewerData` function (page 110).

JMGetAppletViewerSecurity

Gets the security options for an applet.

```
extern OSStatus JMGetAppletViewerSecurity (
                         JMAppletViewerRef viewer,
                         JMAppletSecurity* data);
```

`viewer`	The applet whose security options you wish to determine.
`data`	A pointer to an applet security options structure. on return, this structure contains the security options for the specified applet. For more information, see "Applet Security Structure" (page 76).
function result	A result code; see "JManager Result Codes" (page 148).

SEE ALSO

The `JMSetAppletViewerSecurity` function (page 111).

The `JMNewAppletViewer` function (page 108).

JMSetAppletViewerSecurity

Sets the security options for an applet.

```
OSStatus JMSetAppletViewerSecurity (
                         JMAppletViewerRef viewer,
                         const JMAppletSecurity* data);
```

`viewer`	The applet whose security options you wish to change.
`data`	A pointer to an applet security options structure containing the new values to set. For more information, see “Applet Security Structure” (page 76).
function result	A result code; see “JManager Result Codes” (page 148).

SEE ALSO

The `JMGetAppletViewerSecurity` function (page 111).

JMReloadApplet

Loads (or reloads) an applet from its source server and executes it.

```
OSStatus JMReloadApplet (JMAppletViewerRef viewer);
```

`viewer`	The applet you want to execute.
function result	A result code; see “JManager Result Codes” (page 148).

DISCUSSION

You can also use this function to reload and restart the applet at any time (if the code has changed, for example). If you want to restart the applet without reloading the applet code, you should use the `JMRestartApplet` function (page 112).

JMRestartApplet

Restarts an applet without reloading it from the source server.

```
OSStatus JMRestartApplet (JMAppletViewerRef viewer);
```

`viewer`	The applet you want to execute.
function result	A result code; see “JManager Result Codes” (page 148).

DISCUSSION

This function corresponds to the Java method `java.applet.Applet.start`. If you want to reload the applet code before execution, you should use the `JMReloadApplet` function (page 112).

JMSuspendApplet

Suspends execution of an applet and any associated threads.

```
OSStatus JMSuspendApplet (JMAppletViewerRef viewer);
```

`viewer`	The applet you want to suspend.
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

This function corresponds to the Java method `java.applet.Applet.stop`.

SEE ALSO

The `JMResumeApplet` function (page 113).

JMResumeApplet

Resumes execution of a suspended applet.

```
OSStatus JMResumeApplet (JMAppletViewerRef viewer);
```

`viewer`	The applet you want to execute.
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

You can use the `JMResumeApplet` function to resume execution of an applet halted by the `JMSuspendApplet` function (page 113). If you want to restart the applet from the beginning, you should use the `JMRestartApplet` function (page 112) or the `JMReloadApplet` function (page 112).

JMGetFrameViewer

Finds the applet that owns a particular frame.

```
OSStatus JMGetFrameViewer (
                    JMFrameRef frame,
                    JMAppletViewerRef* viewer,
                    JMFrameRef* parentFrame);
```

`frame`	The frame whose applet you want to determine.
`viewer`	A pointer to the applet. On return, this parameter is the applet associated with the `frame` parameter.
`parentFrame`	A pointer to the parent frame. On return, this parameter is the applet's parent frame (that is, the one created for the applet when it was instantiated).
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

In the AWT context associated with the applet, the frame index of the parent frame is 0.

SEE ALSO

The `JMCountAWTContextFrames`function (page 95).

JMGetViewerFrame

Finds the parent frame for a given applet.

```
OSStatus JMGetViewerFrame (
                    JMAppletViewerRef viewer,
                    JMFrameRef* frame);
```

`viewer`	The applet whose parent frame you want to find.
`frame`	A pointer to the parent frame. On return, this parameter is the parent frame associated with the applet.
function result	A result code; see “JManager Result Codes” (page 148).

DISCUSSION

In the AWT context associated with the applet, the frame index of the parent frame is 0.

SEE ALSO

The `JMCountAWTContextFrames`function (page 95).

JMGetAppletViewerObject

Returns a JRI reference to a `JMAppletViewerRef` object.

```
jref JMGetAppletViewerObject (JMAppletViewerRef viewer);
```

`viewer`	The applet whose Java object you want to find.
function result	A pointer to the `com.apple.mrj.JManager.JMAppletViewer` Java object.

DISCUSSION

The `JMAppletViewerRef` object is not the same as an applet object (that is, a `java. applet.Applet` object). A `JMAppletViewer` object encapsulates the Java applet object so it may be handled outside the Java environment.

JMGetAppletObject

Returns a reference to the Java applet object.

```
jref JMGetAppletObject (JMAppletViewerRef viewer);
```

`viewer`	The applet whose Java object you want to find.
function result	A pointer to the `java.applet.Applet` Java object.

DISCUSSION

This function returns a reference to the actual applet object rather than to the `JMAppletViewerRef` object handled by the embedding application.

Frame Manipulation Functions

Each applet has one or more frames associated with it. A frame is analogous to a Mac OS window record and usually represents a user window. The following functions let you pass events between a visible user window and the abstract applet frame.

JMSetFrameVisibility

Assigns Mac OS window visibility properties to a frame.

```
OSStatus JMSetFrameVisibility (
                     JMFrameRef frame,
                     GrafPtr framePort,
                     Point frameOrigin,
                     const RgnHandle frameClip);
```

`frame`	The frame whose visibility you want to set.
`framePort`	The graphics port to assign to the frame.
`frameOrigin`	The location of the frame's origin, in global coordinates.
`frameClip`	The clipping region for the frame.
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

You use the `JMSetFrameVisibility` function to pass Mac OS window visibility elements to the abstract frame. For example, if the user moves the window associated with the frame (by dragging or scrolling), you must update the frame's visibility by passing the new position of the frame in `JMSetFrameVisibility`.

SEE ALSO

"Displaying Frames" (page 33).

JMGetFrameData

Reads client data associated with a frame.

```
OSStatus JMGetFrameData (
                     JMFrameRef frame,
                     JMClientData* data);
```

`frame`	The frame whose client data you want to read.
`data`	A pointer to the client data. Upon return, this parameter holds the client data for the frame.
function result	A result code; see "JManager Result Codes" (page 148).

SEE ALSO

"Displaying Frames" (page 33).

The `JMSetFrameData` function (page 118).

JMSetFrameData

Sets or changes the client data for a frame.

```
OSStatus JMSetFrameData (
                    JMFrameRef frame,
                    JMClientData data);
```

`frame`	The frame whose client data you want to set or change.
`data`	The new client data.
function result	A result code; see "JManager Result Codes" (page 148).

SEE ALSO

"Displaying Frames" (page 33).

The `JMGetFrameData` function (page 117).

JMGetFrameSize

Gets the coordinates of the frame.

```
OSStatus JMGetFrameSize (
                    JMFrameRef frame,
                    Rect* result);
```

`frame`	The frame whose dimensions you want to determine.
`result`	A pointer to the frame coordinates. Upon return, this parameter holds the coordinates of the frame in pixels.
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

Frames are described using a zero-based coordinate system with the top-left corner of the frame having the coordinates (0,0).

SEE ALSO

The `JMSetFrameSize` function (page 119).

JMSetFrameSize

Sets the size of a frame.

```
OSStatus JMSetFrameSize (
                    JMFrameRef frame,
                    const Rect* newSize);
```

`frame`	The frame whose size you want to set.
`newSize`	The coordinates of the new frame size in pixels.
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

The function automatically calculates the width and height of the frame, given the coordinates you pass in the `newSize` parameter. The top left corner indicates the global position of this frame, so you can use this to update the position of the frame if it gets moved.

IMPORTANT
On the Mac OS platform, the global frame coordinates (0,0) does not map to the actual top left corner of the screen, but rather is offset to accomodate the title and side bars of the corresponding window as well as the menu bar. ▲

SEE ALSO

The `JMGetFrameSize` function (page 119).

JMFrameClick

Dispatches a mouse event to a frame.

```
OSStatus JMFrameClick (
                        JMFrameRef frame,
                        Point localPos,
                        short modifiers);
```

`frame`	The frame where the mouse event occurred.
`localPos`	The position of the mouse in frame coordinates.
`modifiers`	Any keyboard modifiers from the `EventRecord` data structure. For more information about `EventRecord`, see *Inside Macintosh: Macintosh Toolbox Essentials.*
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

The client application must keep track of the currently active frame.

SEE ALSO

"Mouse Events" (page 42).

JMFrameKey

Dispatches a key-down event to a frame.

```
OSStatus JMFrameKey (JMFrameRef frame,
                     char asciiChar,
                     char keyCode,
                     short modifiers);
```

`frame`	The frame in which the key-down event occurred.
`asciiChar`	The ASCII character typed.
`keyCode`	The machine key code typed.
`modifiers`	Any keyboard modifiers from the `EventRecord` data structure. You can determine the ASCII character (and corresponding machine key code) from `EventRecord.message`. For more information about `EventRecord`, see *Inside Macintosh: Macintosh Toolbox Essentials.*
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

The client application must keep track of the currently active frame.

SEE ALSO

"Keyboard Events" (page 45).

The `JMFrameKeyRelease` function (page 122).

JMFrameKeyRelease

Dispatches a key-up event to a frame.

```
OSStatus JMFrameKeyRelease (JMFrameRef frame,
                         char asciiChar,
                         char keyCode,
                         short modifiers);
```

`frame`	The frame in which the key up event occurred.
`asciiChar`	The ASCII character typed.
`keyCode`	The machine key code typed.
`modifiers`	Any keyboard modifiers from the `EventRecord` data structure. You can determine the ASCII character (and corresponding machine key code) from `EventRecord.message`. For more information about `EventRecord`, see *Inside Macintosh: Macintosh Toolbox Essentials.*
function result	A result code; see “JManager Result Codes” (page 148).

DISCUSSION

The client application must keep track of the currently active frame.

SEE ALSO

“Keyboard Events” (page 45).

The `JMFrameKey` function (page 121).

JMFrameUpdate

Updates a frame.

```
OSStatus JMFrameUpdate (
                         JMFrameRef frame,
                         const RgnHandle updateRgn);
```

`frame`	The frame to be updated.
`updateRgn`	The region to be updated, in frame coordinates.
function result	A result code; see “JManager Result Codes” (page 148).

DISCUSSION

The region to update must be specified in frame coordinates, where the top-left corner of the frame has the coordinates (0,0).

SEE ALSO

“Update, Activate, and Resume Events” (page 40).

JMFrameActivate

Activates or deactivates a frame.

```
OSStatus JMFrameActivate (
                    JMFrameRef frame,
                    Boolean activate);
```

`frame`	The frame to be activated or deactivated.
`activate`	A Boolean value. If true, the frame is to be made active; if false, the frame is to be deactivated.
function result	A result code; see “JManager Result Codes” (page 148).

DISCUSSION

Only one frame should be active at one time, so if you make a frame active, deactivate the other frames in that AWT context thread. The client application must keep track of the currently active frame.

SEE ALSO

“Update, Activate, and Resume Events” (page 40).

JMFrameResume

Passes a resume event to a frame.

```
OSStatus JMFrameResume (
                    JMFrameRef frame,
                    Boolean resume);
```

`frame`	The frame to receive the resume event.
`resume`	A Boolean value. If true, the process associated with the frame is brought to the foreground; if false, the process is sent to the background.
function result	A result code; see “JManager Result Codes” (page 148).

DISCUSSION

The resume event means that the frame becomes part of the running foreground process. When a client application receives a resume event, it must notify all its associated frames.

SEE ALSO

“Update, Activate, and Resume Events” (page 40).

JMFrameMouseOver

Passes a mouse-over event to a frame.

```
OSStatus JMFrameMouseOver (
                    JMFrameRef frame,
                    Point localPos,
                    short modifiers);
```

`frame`	The frame containing the mouse. This should be the active frame.
`localPos`	The position of the mouse in frame coordinates.

`modifiers`	Any keyboard modifiers from the `EventRecord` structure. See *Inside Macintosh: Macintosh Toolbox Essentials*, for more information about the `EventRecord` data structure.
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

To monitor the position of the mouse, you should call the `JMFrameMouseOver` (page 124) and `JMIdle` (page 82) functions each time through the event loop. Note that the client application must keep track of the currently active frame.

JMFrameShowHide

Shows or hides a frame.

```
OSStatus JMFrameShowHide (
                        JMFrameRef frame,
                        Boolean showFrame);
```

`frame`	The frame to show or hide.
`showFrame`	A Boolean value. If true, then the frame should be shown; if false, the frame should be hidden.
function result	A result code; see "JManager Result Codes" (page 148).

JMFrameGoAway

Passes a go-away event to the frame.

```
OSStatus JMFrameGoAway (JMFrameRef frame);
```

`frame`	The frame to receive the go-away event.
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

Calling the `JMFrameGoAway` function disposes of the frame, although in some instances the Java program may want to display a dialog box message (asking if the user wants to save the file before closing, for example).

SEE ALSO

"Mouse Events" (page 42).

JMFrameDragTracking

Passes mouse dragging information to a frame.

```
OSStatus JMFrameDragTracking (
                JMFrameRef frame,
                DragTrackingMessage message,
                DragReference theDragRef);
```

`frame`	The frame to receive the mouse dragging information.
`message`	The drag tracking message to pass to the frame. These messages should correspond to the drag tracking messages passed by the Drag Manager to a `DragTrackingHandler` callback.
`theDragRef`	The drag reference of the drag.
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

The `JMFrameDragTracking` function passes drag information to the frame when the user drags an item into the corresponding window. Essentially your application calls this function as though the Drag Manager were calling a `DragTrackingHandler` callback.

Note that JDK versions 1.1.x and earlier do not support drag and drop.

SEE ALSO

"Drag-And-Drop Support" (page 49).

Drag Manager Programmer's Guide.

JMFrameDragReceive

Passes a drag-and-drop item to a frame.

```
OSStatus JMFrameDragReceive (
             JMFrameRef frame,
             DragReference theDragRef);
```

`frame`	The frame to receive the drag-and-drop information.
`theDragRef`	The drag reference of the drag.
function result	A result code; see “JManager Result Codes” (page 148).

DISCUSSION

If the user releases a drag in a window corresponding to a frame, you should pass the drag information to the frame using `JMFrameDragReceive`. The frame can then take action on the drag, depending on the contents of the drag. Essentially your application calls this function as though the Drag Manager were calling a `DragReceiveHandler` callback.

Note that JDK versions 1.1.x and earlier do not support drag and drop.

SEE ALSO

“Drag-And-Drop Support” (page 49).

Drag Manager Programmer’s Guide.

JMGetFrameContext

Returns the AWT context associated with a frame.

```
JMAWTContextRef JMGetFrameContext (JMFrameRef frame);
```

`frame`	The frame whose AWT context you want to determine.
function result	A pointer to the AWT context that owns the frame.

JMGetAWTFrameObject

Returns a reference to a frame's Java object.

```
jref JMGetAWTFrameObject(JMFrameRef frame);
```

`frame`	The frame whose Java object you want to find.
function result	A pointer to the `java.awt.Frame` object associated with the frame.

JMGetJMFrameObject

Returns a JRI reference to a frame's `JMFrame` object.

```
jref JMGetJMFrameObject(JMFrameRef frame);
```

`frame`	The frame whose Java object you want to find.
function result	A pointer to the `com.apple.mrj.JManager.JMFrame` object associated with the frame.

DISCUSSION

The `JMGetJMFrameObject` function returns the Java equivalent of a `JMFrameRef` reference. A `JMFrameRef` object encapsulates a Java frame object (that is, a `java.awt.Frame` object) so it can be handled outside the Java environment.

Utility Functions

JMGetVersion

Returns the version of JManager available on the host computer.

```
UInt32 JMGetVersion (void);
```

function result	A version code. This value is similar to the value of the `kJMVersion` constant.

DISCUSSION

A version of JManager older than the one your embedding application compiled against may not have the same functionality; you can use `JMGetVersion` and compare the result against `kJMVersion` to avoid calling nonexisting functions.

JMFSSToURL

Converts a Mac OS file system specification record (`FSSpec`) into a Uniform Resource Locator (URL) string.

```
Handle JMFSSToURL   (JMSessionRef session,
                     const FSSpec* spec);
```

`session`	The current session.
`spec`	A pointer to a file system specification record.
function result	A handle containing a URL string in the form `file://xxxx`. If the file cannot be found, the function returns `null`.

DISCUSSION

The `JMFSSToURL` function resolves the path of the file represented by the file system specification record and returns this information as a URL string. The handle returned points to a null-terminated string. Your application is responsible for calling the Mac OS Toolbox function `DisposeHandle` to release the allocated handle.

SEE ALSO

The `JMURLToFSS` function (page 130).

JMURLToFSS

Converts a Uniform Resource Locator (URL) into a Mac OS file system specification record (`FSSpec`).

```
OSStatus JMURLToFSS (JMSessionRef session,
                     const JMTextRef urlString,
                     FSSpec* spec);
```

`session`	The current session.
`urlString`	The URL string to be converted, as a text object.
`spec`	A pointer to a file system specification record. Upon return, `spec` points to the converted URL.
function result	A result code; see "JManager Result Codes" (page 148).

DISCUSSION

The URL string must follow the format `file:///disk/dir1/.../dirN/file`. Other formats cause the function to return the result code `paramErr`.

SEE ALSO

The `JMFSSToURL` function (page 129).

"Executing Java Applications" (page 50).

JMAddToClassPath

Adds a directory, zip file, or Java archive (JAR) file to the class path.

```
OSStatus JMAddToClassPath (
                     JMSessionRef session,
                     const FSSpec* spec);
```

`session`	The current session.
`spec`	A pointer to a file system specification record.

function result A result code; see “JManager Result Codes” (page 148).

DISCUSSION

The `JMAddToClassPath` function adds an additional file to the Java class definition search path by prepending a Mac OS file system specification record (`FSSpec`) to the class path of the current session. You can add any number of files to the class path. The `FSSpec` value can indicate an uncompressed zip file or a directory. A directory is considered to be the root of a class hierarchy.

SEE ALSO

“Executing Java Applications” (page 50).

JMGetCurrentEnv

Returns the current `JNIEnv` data structure.

```
struct JNIEnv* JMGetCurrentEnv (JMSessionRef session);
```

`session`	The current session.
function result	A pointer to the `JNIEnv` data structure.

DISCUSSION

The `JMGetCurrentEnv` function allows you to access the current Java Native Interface (JNI) structure. Once you have the `JNIEnv` structure, you can call JNI functions. The data structure `JNIEnv` is defined in the header file `JNI.h`.

You can find documentation for the Java Native Interface at the Web site

<http://java.sun.com/>

SEE ALSO

“Executing Java Applications” (page 50).

JMGetJRIRuntimeInstance

Returns the current `JRIRuntimeInstance` data structure.

```
struct JRIRuntimeInstance* JMGetJRIRuntimeInstance (
                    JMSessionRef session);
```

`session`	The current session.
function result	A pointer to the `JRIRuntimeInstance` data structure.

DISCUSSION

The `JMGetJRIRuntimeInstance` function allows you to get information about the current Java Runtime Interface (JRI) structure. The data structure `JRIRuntimeInstance` is defined in the header file `JRI.h`.

Note
Unless you have a particular reason to access the Java Runtime Interface, you should access the Java Native Interface, which provides similar functionality, instead. ◆

You can find documentation for the Java Runtime Interface at the Web site

<http://developer.netscape.com/>

JMGetCurrentJRIEnv

Returns the current `JRIEnv` data structure.

```
struct JRIEnv* JMGetCurrentJRIEnv (JMSessionRef session);
```

`session`	The current session.
function result	A pointer to the `JRIEnv` data structure.

DISCUSSION

The `JMGetCurrentJRIEnv` function allows you to obtain the current Java Runtime Interface (JRI) structure. Once you have the `JRIEnv` structure, you can call JRI functions. The data structure `JRIEnv` is defined in the header file `JRI.h`.

Note
Unless you have a particular reason to access the Java Runtime Interface, you should access the Java Native Interface, which provides similar functionality, instead. ◆

You can find documentation for the Java Runtime Interface at the Web site

<http://developer.netscape.com/>

JMJRIRefToJNIObject

Converts a JRI-based `jref` to a JNI-based `jobject`.

```
jobject JMJRIRefToJNIObject (
                    JMSessionRef session,
                    JNIEnv* env,
                    jref jriRef);
```

`session`	The current session.
`env`	The current `JNIEnv` data structure.
`jriRef`	The `jref` to convert.
function result	A pointer to the `jobject` object.

DISCUSSION

If you want to use `jref` object references with the Java Native Interface (JNI), you must convert them to `jobject` references using `JMJRIRefToJNIObject`. Note that you must have a pointer to the current `JNIEnv` data structure before calling this function.

SEE ALSO

The `JMGetCurrentEnv` function (page 131).

JMJNIObjectToJRIRef

Converts a JNI-based `jobject` to a JRI-based `jref`.

```
jref JMJNIObjectToJRIRef (
                        JMSessionRef session,
                        JNIEnv* env,
                        jobject jniObject);
```

`session`	The current session.
`env`	The current `JNIEnv` data structure.
`jniObject`	The `jObject` to convert.
function result	A pointer to the `jref` object.

DISCUSSION

If you want to use `jobject` object references with the Java Runtime Interface (JRI), you must convert them to `jref` references using `JMJNIObjectToJRIRef`. Note that you must have a pointer to the current `JNIEnv` data structure before calling this function.

SEE ALSO

The `JMGetCurrentEnv` function (page 131).

Application-Defined Functions

This section describes the format of JManager callback functions you must implement in your application.

MyStandardOutput

Sends text to output. This is how you would define your output function if you were to name it `MyStandardOutput`:

```
void MyStandardOutput (
                     JMSessionRef session,
                     const char* message,
                     UInt32 messageLen);
```

`session`	The session sending the text.
`message`	The text to display.
`messageLen`	The length of the text.

DISCUSSION

When invoking the Java runtime environment using `JMOpenSession`, you must designate a callback function to display any console output received from the session.

MyStandardError

Directs error output. This is how you would define your error function if you were to name it `MyStandardError`:

```
void MyStandardError (
                     JMSessionRef session,
                     const char* message,
                     UInt32 messageLen);
```

`session`	The session sending the error text.
`message`	The error message to display.
`messageLen`	The length of the text.

DISCUSSION

When invoking the Java runtime environment using `JMOpenSession`, you must designate a callback function to direct any error output received from the session.The `MyStandardError` function has the same form as the `MyStandardOutput` callback function.

SEE ALSO

The `MyStandardOutput` function(page 135).

MyStandardIn

Reads text input. This is how you would define your input function if you were to name it `MyStandardIn`:

```
SInt32 MyStandardIn (JMSessionRef session,
                     char* buffer,
                     SInt32 maxBufferLength);
```

`session`	The session to receive the text input.
`buffer`	The buffer to hold the input.
`maxBufferLength`	The maximum length allowed by the buffer.
function result	The number of characters actually read (up to `maxBufferLength`) or –1 if an error occurred.

DISCUSSION

When invoking the Java runtime environment using `JMOpenSession`, you must designate a callback function to direct any console input to the session.

MyExit

Handles calls to `java.lang.System.exit`. This is how you would define your input function if you were to name it `MyExit`:

```
Boolean MyExit (JMSessionRef session,
                int value);
```

`session`	The session to receive the text input.
`value`	The buffer to hold the input.
function result	A Boolean value. If true, then the current thread is killed. If false, a QUIT Apple event is sent to the current process.

DISCUSSION

When invoking the Java runtime environment using `JMOpenSession`, you must designate a callback function (called `MyExit` here) to handle requests to quit. When a Java applet or application calls `java.lang.System.exit`, the session calls `MyExit`. Note that instead of passing false back to the session, you can simply dispose of the session and exit from within the `MyExit` function.

MyAuthenticate

Handles an authentication request for a URL. This is how you would define your authentication function if you were to name it `MyAuthenticate`:

```
Boolean MyAuthenticate (
                JMSessionRef session,
                const char* url,
                const char* realm,
                char userName[255],
                char password[255]);
```

`session`	The session to receive the text input.
`url`	The URL making the authentication request.
`realm`	The realm associated with the URL.

`userName`	The name entered by the user.
`password`	The password entered by the user.
function result	A Boolean value. You should return false if the user selects to cancel the authentication, true otherwise.

DISCUSSION

When invoking the Java runtime environment using `JMOpenSession`, you can designate a callback function to handle any authentication requests from a URL. This callback should prompt the user for a name and password, and pass them back to the session. If you do not indicate an authentication callback, the Java session will prompt the user with its own authentication dialog box.

MyLowMem

Handles a low-memory condition. This is how you would define your low-memory function if you were to name it `MyLowMem`:

```
void MyLowMem (JMSessionRef session);
```

`session`	The session indicating the low-memory condition.

DISCUSSION

When invoking the Java runtime environment using `JMOpenSession`, you can designate a callback function to be called if the Java session runs low on memory. This callback typically notifies the user of the low-memory condition and suggests possible actions to take.

MyRequestFrame

Creates a new frame. This is how you would define your frame request function if you were to name it `MyRequestFrame`:

```
OSStatus MyRequestFrame (
                  JMAWTContextRef context,
                  JMFrameRef newFrame,
                  JMFrameKind kind,
                  const Rect* initialBounds,
                  Boolean resizeable,
                  JMFrameCallbacks* callbacks);
```

`context`	The AWT context making the frame request.
`newFrame`	A pointer to the new frame. on return, this parameter is the new frame.
`kind`	The type of frame desired. See "Frame Types" (page 64) for a list of possible values for this field.
`initialBounds`	The initial dimensions of the frame.
`resizeable`	A Boolean value. If false, this frame is not resizeable; if true, you can resize the frame.
`callbacks`	A pointer to the frame callbacks data structure. on return, this parameter should specify the frame's callback functions. The AWT can then use these callbacks when it needs to modify a frame. See "Frame Callbacks Structure" (page 77) for more information about this data structure.
function result	A result code. The function should return a standard result code.

DISCUSSION

When instantiating an AWT context, you must designate a callback function to handle requests for new frames.

MyReleaseFrame

Releases the frame. This is how you would define your frame release function if you were to name it `MyReleaseFrame`:

```
OSStatus MyReleaseFrame (
                         JMAWTContextRef context,
                         JMFrameRef oldFrame);
```

`context`	The AWT context containing the frame.
`oldFrame`	The frame to be released.
function result	A result code. The function should return a standard result code.

DISCUSSION

When instantiating an AWT context, you must designate a callback function to release existing frames.

MyUniqueMenuID

Allocates a new menu ID. This is how you would define your menu ID allocation function if you were to name it `MyUniqueMenuID`:

```
typedef SInt16 MyUniqueMenuID (
                         JMAWTContextRef context,
                         Boolean isSubmenu);
```

`context`	The AWT context containing the frame.
`isSubMenu`	A Boolean value. If false, the menu to add is a standard menu. If true, the menu is a submenu. The menu ID value of a submenu must be in the range 1 to 255.
function result	If successful, the function should return the ID of the new menu. Otherwise the function returns 0.

DISCUSSION

When instantiating an AWT context, you must designate a callback function to create new menu IDs if necessary.

MyExceptionOccurred

Indicates that an exception occurred. This is how you would define your exception notification function if you were to name it `MyExceptionOccurred`:

```
void MyExceptionOccurred (
                    JMAWTContextRef context,
                    const JMTextRef exceptionName,
                    const JMTextRef exceptionMsg,
                    const JMTextRef stackTrace);
```

`context`	The AWT context in which the exception occurred.
`exceptionName`	A text object containing the name of the exception.
`exceptionMsg`	A text object containing the exception message to display. This value is not guaranteed to be present and may be `null`.
`stackTrace`	A text object containing information about the call chain where the exception occurred. This value is not guaranteed to be present and may be `null`.

DISCUSSION

When instantiating an AWT context, you must designate a callback function to notify that an exception has occurred. You cannot use this function to recover from the exception; this function indicates only that an exception has occurred.

MyResizeRequest

Handles a request to change the size of a frame. This is how you would define your frame resize function if you were to name it `MyResizeRequest`:

```
Boolean MyResizeRequest (
                         JMFrameRef frame,
                         Rect* newBounds);
```

`frame`	The frame to be resized.
`newBounds`	A pointer to the new desired dimensions of the frame.
function result	True if the frame can be resized to the requested dimensions.

DISCUSSION

When creating a frame, you must designate a callback function to resize the frame if necessary. The function can refuse the resize request or adjust the frame size to something other than the requested dimensions. If your function sets a new frame size, you can modify the value of the `newBounds` parameter to reflect the new dimensions.

MyInvalRect

Handles a frame invalidation request. This is how you would define your invalidation request function if you were to name it `MyInvalRect`:

```
void MyInvalRect    (JMFrameRef frame,
                     const Rect* dimens);
```

`frame`	The frame that contains the area to be invalidated.
`dimens`	A pointer to the dimensions of the frame.

DISCUSSION

When creating a frame you must designate a callback function to invalidate a portion of the frame if necessary (in a manner similar to the MacOS Toolbox function call `InvalRect`). The invalid portion can be updated later using the `JMFrameUpdate` function.

SEE ALSO

The `JMFrameUpdate` function (page 122).

MyShowHide

Shows or hides a window associated with a frame. This is how you would define your show/hide function if you were to name it `MyShowHide`:

```
void MyShowHide     (JMFrameRef frame,
                      Boolean showFrameRequested);
```

`frame`	The frame to be shown or hidden.
`showFrameRequested`	A Boolean value. If true, the window should be displayed. If false, the window should be hidden.

DISCUSSION

When creating a frame, you must designate a callback function to show or hide the window associated with it.

MySetTitle

Sets the title bar text for the frame. This is how you would define your title bar function if you were to name it `MySetTitle`:

```
void MySetTitle     (JMFrameRef frame,
                      const JMTextRef title);
```

`frame`	The frame that contains the title bar to be set or changed.
`title`	The title to display, as a text object.

DISCUSSION

When creating a frame you must designate a callback function to set or modify the title bar associated with it.

MyCheckUpdate

Checks to see if a frame update is necessary. This is how you would define your update check function if you were to name it `MyCheckUpdate`:

```
void MyCheckUpdate  (JMFrameRef frame);
```

`frame`	The frame to be checked.

DISCUSSION

When creating a frame you must designate a callback function to check if a frame update is needed. This function may be called to enable updates for interactions such as live scrolling or other mouse-tracking maneuvers. If the function determines that an update is necessary, it should call the `JMFrameUpdate` function to perform the update.

SEE ALSO

The `JMFrameUpdate` function (page 122).

MyFrameReorder

Reorders the frame. This is how you would define your frame reorder function if you were to name it `MyFrameReorder`:

```
void MyFrameReorder (
                    JMFrameRef frame,
                    enum ReorderRequest theRequest);
```

`frame`	The frame to be reordered.

`theRequest`	The desired reordering. See "Frame Ordering Indicators" (page 65) for a list of possible values.

DISCUSSION

When creating a frame you must designate a callback function to reorder the frame if necessary (for example, to bring it to the front or send it to the back). Note that you should not reorder frames such that a modal frame appears on top of a nonmodal one.

MySetResizeable

Designates whether a frame is resizeable. This is how you would define your set resizeable function if you were to name it `MySetResizeable`:

```
void MySetResizeable (
                        JMFrameRef frame,
                        Boolean resizeable);
```

`frame`	The frame to designate as resizeable or not.
`resizeable`	If true, the frame should be designated as resizeable.

DISCUSSION

When creating a frame you must designate a callback function to set the frame as resizeable or not. The callback can allow or disallow the use of the grow control depending on the value of `resizeable`.

MyFetchCompleted

Executes after an attempt to retrieve HTML data using the `JMNewAppletLocator` function. This is how you would define your output function if you were to name it `MyFetchCompleted`:

```
void MyFetchCompleted (
                    JMAppletLocatorRef ref,
                    JMLocatorErrors status);
```

`ref`	The newly created applet locator.
`status`	The status of the HTML data retrieval. See “Applet Locator Status Values” (page 66) for a listing of values that `JMNewAppletLocator` may pass in this parameter.

DISCUSSION

When calling `JMNewAppletLocator`, you must designate this function in the applet locator callbacks structure. The actions taken by the completion function depend on the status value it receives from `JMNewAppletLocator`. For example, if the HTML text is retrieved successfully, the function can then proceed to instantiate an applet associated with the HTML page.

SEE ALSO

The `JMNewAppletLocator` function (page 102).

MyShowDocument

Displays the contents of a URL passed back by an instantiated applet. This is how you would define your output function if you were to name it `MyShowDocument`:

```
void MyShowDocument (JMAppletViewerRef viewer,
                    const JMTextRef urlString,
                    const JMTextRef windowName);
```

`viewer`	The current applet.
`urlString`	A text object containing the URL passed by the applet.
`windowName`	A text object describing the type of window to display the URL contents.

DISCUSSION

When calling `JMNewAppletViewer`, you must designate this function in the applet callbacks structure.

The session passes one of the strings in Table 2-1 in the `WindowName` parameter, and your application should display the URL contents accordingly.

Table 2-1 Window strings passed to `MyShowDocument`

String	Action
`_self`	Show contents in the current frame
`_parent`	Show contents in the parent frame
`_top`	Show contents in the top-most (that is, front-most) frame
`_blank`	Show contents in a new unnamed window
frameName	Show contents in a new window named *frameName*

SEE ALSO

The `JMNewAppletViewer` function (page 108).

MySetStatusMsg

Handles any status messages passed back by an instantiated applet. This is how you would define your output function if you were to name it `MySetStatusMsg`:

```
void MySetStatusMsg (JMAppletViewerRef viewer,
                     const char* statusMsg);
```

`viewer`	The current applet.
`statusMsg`	The status text passed by the applet.

DISCUSSION

When calling `JMNewAppletViewer`, you must designate this function in the applet callbacks data structure. The function can display the status message or ignore it, whichever is appropriate.

SEE ALSO

The `JMNewAppletViewer` function (page 108).

JManager Result Codes

Many JManager functions return result codes. The various result codes specific to JManager are shown in Table 2-2. In addition, JManager functions may also return File Manager, Code Fragment Manager, and Process Manager result codes, which are described in *Inside Macintosh.*

Table 2-2 JManager result codes

`noErr`	No error
`paramErr`	Invalid parameter in function call
`memFullErr`	Out of memory
`kJMBadClassPathError`	An invalid path was found in the ClassPath list.
`kJMExceptionOccurred`	An exception occurred
`kJMVersionError`	Incompatible JManager version

C H A P T E R 3

JManager Java Class Reference

Contents

JManager Java Class Reference

This chapter presents Java versions of many of the JManager C constants, structures, and functions, grouped by class or interface. For detailed information about using these items, however, you should refer to the corresponding C version in Chapter 2, "JManager Reference."

The JManagerException Class

The `JManagerException` class implements JManager errors as JManager exceptions.

```
package com.apple.mrj.JManager;

public class JManagerException extends Exception {
    JManagerException(int i) {
        super("JManagerError(" + i + ")");
        }

    static void checkError(int i) throws JManagerException {
        if (i != 0)
            throw new JManagerException(i);
        }
}
```

The JMConstants Interface

The `JMConstants` interface contains all the JManager numerical constants accessible from Java code.

```
package com.apple.mrj.JManager;

/* JMConstants provides access to numerical constants from JManager. */

public interface JMConstants {
    /* using Sun's 1.0.2 APIs, our current APIs. */
    public static final int kJMVersion = 0x11300003,
    /* how much time to give the JM library on "empty" events */
    kDefaultJMTime= 0x00000400;

    public static final int /* JMVerifierOptions */
        eDontCheckCode = 0,
        eCheckRemoteCode = 1,
        eCheckAllCode = 2;

    public static final int /* JMProxyType */
        eHTTPProxy = 0,
        eFirewallProxy = 1,
        eFTPProxy = 2;

    public static final int /* ReorderRequest */
        eBringToFront = 0, // bring the window to front
        eSendToBack = 1, // send the window to back
        eSendBehindFront = 2; // send the window behind the front window

    public static final int /* JMFrameKind */
        eBorderlessModelessWindowFrame = 0,
        eModelessWindowFrame = 1,
        eModalWindowFrame = 2;

    public static final int /* JMLocatorErrors */
        eLocatorNoErr = 0, // the html was retrieved successfully
        eHostNotFound = 1, // the host could not be found
        eFileNotFound = 2, // the file could not be found on the host
        eLocatorTimeout = 3, // timeout while retrieving the html text
        eLocatorKilled = 4; // in response to a JMDisposeAppletLocator
                            // before it has completed

    public static final int /* JMNetworkSecurityOptions */
        eNoNetworkAccess = 0,
        eAppletHostAccess = 1,
        eUnrestrictedAccess = 2;

    public static final int /* JMFileSystemOptions */
        eNoFSAccess = 0,
        eLocalAppletAccess = 1,
        eAllFSAccess = 2;
};
```

The JMProxyInfo Class

The `JMProxyInfo` class contains constants and methods related to setting or reading proxy server information.

```
package com.apple.mrj.JManager;

public class JMProxyInfo {

    boolean itsSet;
    String itsName;
    int itsPort;

    public JMProxyInfo() {
        itsSet = false;
        itsName = "";
        itsPort = 0;
    }

    public JMProxyInfo(boolean set, String name, int port) {
        itsSet = set;
        itsName = name;
        itsPort = port;
    }

    public boolean getSet() {
        return itsSet;
    }

    public String getName() {
        return itsName;
    }

    public int getPort() {
        return itsPort;
    }
};
```

The JMSession Interface

The `JMSession` interface contains structures and methods related to instantiating a Java session. It is analogous to the `java.lang.System` or `java.lang.Runtime` classes.

```
package com.apple.mrj.JManager;

import java.io.*;

/* JMSession represents the encapsulation of the MRJ runtime. */

public interface JMSession {

    /* @return the "C" version of the JMSessionRef pointer */
    public int getSessionRef();

    /* @return the client specific data associated with this session */
    public int getClientData() throws JManagerException;

    /* Sets the client specific data */
    /* @param data the new data to be set */
    public void setClientData(int data) throws JManagerException;

    /* Adds the specified .zip file or .class folder to */
    /* the class path. */
    /* @param path the path to the entity to add */
    /* @exception throws FileNotFoundException if the entity */
    /* isn't found */
    public void addToClassPath(File path)
        throws FileNotFoundException, JManagerException;

    /* Turn a Java File object into a URL string */
    /* @param filePath the path to the file - it may not exist */
    /* @return a URL representing a local file */
    public String fileToURL(File filePath) throws JManagerException;

    /* Turn a URL string to a File object. (The object may not exist) */
    /* @param urlString the name of the url */
    /* @return a File object */
    public File urlToFile(String urlString) throws JManagerException;

    /* @return the current proxy information for the given proxy. */
    public JMProxyInfo getProxyProps(int proxyKind)
        throws JManagerException;

    /* Set the specified proxy configuration */
    public void setProxyProps(int proxyKind, JMProxyInfo info)
        throws JManagerException;
}
```

The JMText Interface

The `JMText` interface contains constants and methods used to manipulate `JMText` objects.

```
package com.apple.mrj.JManager;

public interface JMText {

    public static final int
        eMacEncoding = 0,
        eUTF8Encoding = 1,
        eUnicodeEncoding = 2;

    /* @Return the "C" version of the JMTextRef pointer */
    public int getTextRef();

    /* @Return the number of characters (not bytes) in */
    /* the text segment */
    public int getTextLength() throws JManagerException
    /* Copies the characters to a destination byte array, using */
    /* the specified destination encoding. */
    /* @param encoding the destination encoding */
    /* @buffer the destination buffer */
    /* @bufferOffset the offset in the buffer where data is written */
    /* @bufferLength the number of characters to write into the buffer*/
    /* @return the number of characters actually written to the buffer */
    public int getTextBytes(int encoding, byte buffer[],
        int bufferOffset, int bufferLength) throws JManagerException;

    /* Return a proper java string for this object */
    public String toString();
}
```

The JMFrame Interface

The `JMFrame` interface contains structures and methods related to manipulating frames.

```
package com.apple.mrj.JManager;

/* A JMFrame is part of the peer implementation of a native window. */
/* It is created through a JMAWTContext callback. */

public interface JMFrame {

    /* @Return the "C" version of the JMFrameRef object */
    public int getFrameRef();/* returns the JMFrameRef value */

    /* @Return the client specified data associated with this frame */
    public int getClientData() throws JManagerException;

    /* Set the client specified data associated with this frame */
    /* @param data the new data to set. */
    public void setClientData(int data) throws JManagerException;

    /* Update the Frame’s visibility characteristics. This is */
    /* usually only called from C code. */
    /* @param framePort the port to bind this frame to */
    /* @frameOrigin offset in the port for 0,0 top left of the frame */
    /* @frameClip clipping region overlayed accross the framePort */
    public void setFrameVisibility(com.apple.MacOS.GrafPtr framePort,
        com.apple.MacOS.Point frameOrigin,
        int /*com.apple.MacOS.RgnHandle*/ frameClip)
        throws JManagerException;

    /* @returns the size and position of the frame */
    public com.apple.MacOS.Rect getFrameSize() throws JManagerException;

    /* Requests that the frame be resized to this size and location */
    /* @param newSize the new window size, including position */
    public void setFrameSize(com.apple.MacOS.Rect newSize)
        throws JManagerException;

    /* Dispatch a mouse click to the frame. */
    /* @param localPos the mouse click in local coordinates */
    /* @param modifiers the modifiers from the event record */
    public void frameClick(com.apple.MacOS.Point localPos,
        short modifiers) throws JManagerException;

    /* Dispatch a key press to the frame. */
    /* @param asciiChar the ascii value from the event record */
    /* @param keyCode the machine keycode value from the event record */
    /* @param modifiers the modifiers from the event record */
    public void frameKey(byte asciiChar, byte keyCode,
        short modifiers) throws JManagerException;

    /* Dispatch a key release to the frame. */
    /* @param asciiChar the ascii value from the event record */
    /* @param keyCode the machine keycode value from the event record */
    /* @param modifiers the modifiers from the event record */
    public void frameKeyRelease(byte asciiChar, byte keyCode,
        short modifiers) throws JManagerException;

    /* Dispatch an update event to the frame. */
    /* @param updateRgn the region to be updated in local coordinates */
    public void frameUpdate(int/*com.apple.MacOS.RgnHandle*/ updateRgn)
        throws JManagerException;
```

```
/* Dispatch an activate event to the frame */
/* @param activate set activate (true) or deactivate (false) event */
public void frameActivate(boolean activate) throws JManagerException;

/* Dispatch an application resume event to the frame */
/* @param resume set a resume (true) or suspend (false) event */
public void frameResume(boolean resume) throws JManagerException;

/* Dispatch a mouse over event to the frame. */
/* Used for cursor shaping */
/* @param localPos the mouse location in local coordinates */
/* @param modifiers the modifiers from the event record */
public void frameMouseOver(com.apple.MacOS.Point localPos,
    short modifiers) throws JManagerException;

/* Show or hide the frame. */
/* @param showFrame true to make the frame visible */
public void frameShowHide(boolean showFrame)
    throws JManagerException;

/* Request that a frame be disposed. */
public void frameGoAway() throws JManagerException;

/* Return the AWTContext for this frame. */
/* @see JMAWTContext */
/* @return a JMAWTContext */
public JMAWTContext getFrameContext() throws JManagerException;

/* Post drag & drop events to the frame */
/* @param message the message from the drag handler */
/* @param theDragRef the drag reference from the drag handler */
public void frameDragTracking(
    com.apple.MacOS.DragTrackingMessage message,
    com.apple.MacOS.DragReference theDragRef)
    throws JManagerException;

/* A drag was received for the frame. */
/* @param theDragRef the drag reference from the drag handler */
public void frameDragRecieve(
    com.apple.MacOS.DragReference theDragRef)
    throws JManagerException;
    /* Return the java.awt.Frame for this frame */
    public java.awt.Frame getAwtFrame() throws JManagerException;

    /* If we're representing an applet viewer, which one? */
    /* @returns a JMAppletViewer representing the viewer */
    public JMAppletViewer getFrameViewer() throws JManagerException;
}
```

The JMAppletViewer Interface

The `JMAppletViewer` interface contains methods related to manipulating applets.

```
package com.apple.mrj.JManager;

/* A JMAppletViewerObject represents an object that displays applets. */
/* It has an independent securty object, per applet */
/* @see JMAWTContext */
/* @see JMAppletViewerSecurity */

public interface JMAppletViewer {

    /* @Return the "C" version of the JMAppletViewerRef */
    public int getViewerRef();

    /* @Return the client specified data associated with this applet */
    public int getClientData() throws JManagerException;

    /* Set the client-specific data associated with this applet */
    /* @param data the new data to set. */
    public void setClientData(int data) throws JManagerException;

    /* Return the context associated with this applet */
    public JMAWTContext getContext() throws JManagerException;

    /* Return the java.applet.Applet object instantiated for this applet
*/
    /* @see java.applet.Applet */
    public java.applet.Applet getApplet() throws JManagerException;

    /* Reload the applet’s bytecodes and reinitialize it. */
    public void reloadApplet() throws JManagerException;

    /* Restart the applet without reloading its byte codes */
    public void restartApplet() throws JManagerException;

    /* Call the applets stop() method. */
    public void suspendApplet() throws JManagerException;

    /* Call the applets start() method. */
    public void resumeApplet() throws JManagerException;

    /* Return the frame that this viewer is visible in (if any) */
    public JMFrame getViewerFrame() throws JManagerException;
}
```

The JMAWTContext Interface

The `JMAWTContext` interface contains methods related to AWT contexts, including those to create frames and dispatch menu events to frames.

```
package com.apple.mrj.JManager;

public interface JMAWTContext {

    /* @Return the "C" version of the JMAWTContextRef */
    public int getContextRef();

    /* @Return the client specified data associated with this context */
    public int getClientData() throws JManagerException;

    /* Set the client specified data associated with this context */
    /* @param data the new data to set. */
    public void setClientData(int data) throws JManagerException;

    /* Return the number of frames created for this context */
    public int countAWTContextFrames() throws JManagerException;

    /* Return a JMFrame for the specified frame */
    /* @param frameIndex the 0 based index of the requested frame */
    public JMFrame getContextFrame(int frameIndex)
        throws JManagerException;

    /* Dispatch a menu selected event to the owning Frame */
    /* @param hMenu a MacOS MenuHandle */
    /* @param menuItem the one based menu item index */
    public void menuSelected(com.apple.MacOS.MenuHandle hMenu,
        short menuItem) throws JManagerException;
}
```

The JMAppletSecurity Class

The `JMAppletSecurity` class contains constants methods related to setting or reading security levels for applets.

```
package com.apple.mrj.JManager;

/* JMAppletSecurity is the java representation of a JMAppletSecurity */
/* structure.*/
/* The corresponding C structure is similar, but NOT compatible. */
/* @see JMAppletViewer */

public class JMAppletSecurity {

    public static final int         /* NetworkSecurityOptions */
        eNoNetworkAccess = 0,
        eAppletHostAccess = 1,
        eUnrestrictedAccess = 2;

    public static final int         /* FileSystemOptions */
        eNoFSAccess = 0,
        eLocalAppletAccess = 1,
        eAllFSAccess = 2;
    int itsNetworkAccess;
    int itsFileSystemAccess;

    boolean itsRestrictSystemAccess;
    boolean itsRestrictSystemDefine;

    boolean itsRestrictApplAccess;
    boolean itsRestrictApplDefine;

    /* Create an applet security structure with "good" defaults. */
    public JMAppletSecurity() {

        itsNetworkAccess = eAppletHostAccess;
        itsFileSystemAccess = eLocalAppletAccess;

        itsRestrictSystemAccess = true;
        itsRestrictSystemDefine = true;

        itsRestrictApplAccess = false;
        itsRestrictApplDefine = false;
    }

    /* Create an applet security with specified defaults */
    public JMAppletSecurity(int networkAccess, int fileSystemAccess,
        boolean systemAccess, boolean systemDefine, boolean applAccess,
        boolean applDefine) {

        itsNetworkAccess = networkAccess;
        itsFileSystemAccess = fileSystemAccess;

        itsRestrictSystemAccess = systemAccess;
        itsRestrictSystemDefine = systemDefine;

        itsRestrictApplAccess = applAccess;
        itsRestrictApplDefine = applDefine;
    }

    public final int getNetworkAccess() {
        return itsNetworkAccess;
    }
    public final int getFilesystemAccess() {
        return itsFileSystemAccess;
    }

    public final boolean getRestrictSystemAccess() {
        return itsRestrictSystemAccess;
    }

    public final boolean getRestrictSystemDefine() {
        return itsRestrictSystemDefine;
    }

    public final boolean getRestrictApplAccess() {
        return itsRestrictApplAccess;
    }

    public final boolean getRestrictApplDefine() {
        return itsRestrictApplDefine;
    }

};
```

Appendixes

Changes from JManager 1.0

Table A-1 alphabetically lists all the JManager 2.0 functions and how they differ (if at all) from the older 1.0 functions. Changes to data structures and constants are indicated under the appropriate function.

Table A-1 Changes from JManager 1.0 functions

Function Name	Changes from 1.0
`JMAddToClassPath`	None.
`JMCloseSession`	None.
`JMCopyTextRef`	New with 2.0.
`JMCountApplets`	None.
`JMCountAWTContextFrames`	None.
`JMDisposeAppletLocator`	None.
`JMDisposeAppletViewer`	None.
`JMDisposeAWTContext`	None.
`JMDisposeTextRef`	New with 2.0.
`JMExecJNIMethodInContext`	New with 2.0.
`JMExecJNIStaticMethodInContext`	New with 2.0.
`JMExecMethodInContext`	None.
`JMExecStaticMethodInContext`	None.
`JMFrameActivate`	None.
`JMFrameClick`	None.
`JMFrameDragReceive`	New with 2.0.
`JMFrameDragTracking`	New with 2.0.
`JMFrameGoAway`	None.

Table A-1 Changes from JManager 1.0 functions

Function Name	Changes from 1.0
`JMFrameKey`	None.
`JMFrameKeyRelease`	None.
`JMFrameMouseOver`	None.
`JMFrameResume`	None.
`JMFrameShowHide`	None.
`JMFrameUpdate`	None.
`JMFSSToURL`	None.
`JMGetAppletDimensions`	None.
`JMGetAppletLocatorData`	None.
`JMGetAppletName`	Strings now passed as text objects.
`JMGetAppletObject`	New with 2.0.
`JMGetAppletViewerData`	None.
`JMGetAppletViewerObject`	New with 2.0.
`JMGetAppletViewerSecurity`	New with 2.0.
`JMGetAppletTag`	Strings now passed as text objects.
`JMGetAWTContextData`	None.
`JMGetAWTContextFrame`	None.
`JMGetAwtContextObject`	New with 2.0.
`JMGetAWTFrameObject`	New with 2.0.
`JMGetCurrentEnv`	Now returns a pointer to the Java Native Interface (JNI) structure rather than the Java Runtime Interface (JRI) structure.
`JMGetCurrentJRIEnv`	New with 2.0. Has the same functionality as the JManager 1.0 `JMGetCurrentEnv` function.
`JMGetFrameContext`	None.
`JMGetFrameData`	None.

Table A-1 Changes from JManager 1.0 functions

Function Name	Changes from 1.0
`JMGetFrameSize`	None.
`JMGetFrameViewer`	None.
`JMGetJMFrameObject`	New with 2.0.
`JMGetJRIRuntimeInstance`	New with 2.0.
`JMGetProxyInfo`	New with 2.0.
`JMGetSessionData`	None.
`JMGetSessionObject`	New with 2.0.
`JMGetSessionProperty`	Strings now passed as text objects.
`JMGetTextBytes`	New with 2.0.
`JMGetTextLength`	New with 2.0.
`JMGetVersion`	None.
`JMGetViewerFrame`	None.
`JMGetVerifyMode`	New with 2.0.
`JMIdle`	None.
`JMJNIObjectToJRIRef`	New with 2.0.
`JMJRIRefToJNIObject`	New with 2.0.
`JMMenuSelected`	None.
`JMNewAppletLocator`	Strings now passed as text objects.
`JMNewAppletLocatorFromInfo`	Strings passed in the `JMLocatorInfoBlock` data structure are now text objects.
`JMNewAppletViewer`	New applet security data structure required (`JMAppletSecurity`).
`JMNewAWTContext`	None.
`JMNewTextRef`	New with 2.0.

Table A-1 Changes from JManager 1.0 functions

Function Name	**Changes from 1.0**
`JMOpenSession`	New code verifier options.
	New `JMRuntimeOptions` mask.
	New callbacks in `JMSessionCallbacks` data structure.
	Many security options now bound to individual applets.
`JMPutSessionProperty`	Strings now passed as text objects.
`JMReloadApplet`	None.
`JMRestartApplet`	None.
`JMResumeApplet`	None.
`JMResumeAWTContext`	No longer required. AWT contexts no longer have a suspended state.
`JMSetAppletLocatorData`	None.
`JMSetAppletViewerData`	None.
`JMSetAppletViewerSecurity`	New with 2.0.
`JMSetAWTContextData`	None.
`JMSetFrameData`	None.
`JMSetFrameSize`	None.
`JMSetFrameVisibility`	New with 2.0.
`JMSetProxyInfo`	New with 2.0.
`JMSetSessionData`	None.
`JMSetVerifyMode`	New with 2.0.
`JMShowPropsDialog`	No longer used. Calls from JManager 1.0 still supported.
`JMSuspendApplet`	None.
`JMSuspendAWTContext`	No longer required. AWT contexts no longer have a suspended state.

Table A-1 Changes from JManager 1.0 functions

Function Name	Changes from 1.0
`JMTextToJavaString`	New with 2.0.
`JMTextToMacOSCStringHandle`	New with 2.0.
`JMURLToFSS`	Strings now passed as text objects.
`MyAuthenticate`	New with 2.0.
`MyCheckUpdate`	None.
`MyExceptionOccurred`	Strings now passed as text objects.
`MyExit`	New with 2.0.
`MyFetchCompleted`	None.
`MyFrameReorder`	New with 2.0.
`MyInvalRect`	None.
`MyLowMem`	New with 2.0.
`MyReleaseFrame`	None.
`MyRequestFrame`	New callbacks in the `JMFrameCallbacks` data structure. Dimensions of the frame now passed as a `Rect` structure.
`MySetResizeable`	New with 2.0.
`MySetStatusMsg`	Strings now passed as text objects.
`MySetTitle`	Strings now passed as text objects.
`MyResizeRequest`	None.
`MyShowDocument`	Strings now passed as text objects.
`MyShowHide`	None.
`MyStandardError`	None.
`MyStandardIn`	None.
`MyStandardOutput`	None.
`MyUniqueMenuID`	None.

A P P E N D I X   B

Mac OS–Related Issues

The Mac OS runtime environment has some restrictions and peculiarities that you should keep in mind when writing either Java code or Mac OS–based code that interacts with Java code. Table B-1 describes these issues and possible solutions.

Table B-1 Mac OS–related issues

Description	Possible limitations	Solutions
31-character limit on filenames.	Java class filenames can easily exceed 31 characters.	Class files stored in zip files can have names longer than 31 characters.
Only the active frame's menu bar is visible.	Some Java programs may specify more than one menu bar.	N/A
Cooperative rather than preemptive multitasking.	Some odd timing effects may occur.	N/A
File and directory delimiter character is a colon (`:`), not a slash (`/`). (The slash is a valid character in a filename.)	May cause confusion or odd behavior when working with file systems that return a URL or similar path (such as not finding a file located in a folder named `Reports May/June`).	Embedded slash characters should be encoded as a percent sign (`%`) plus the hex ASCII value (that is, `%2F`).
A space is a valid character in a Mac OS filename.	May cause unexpected interpretations of strings obtained from other platforms.	Embedded space characters should be encoded as a percent sign (`%`) plus the hex ASCII value (that is `%20`).

The mechanism for handling dynamically linked libraries (DLLs) on the Mac OS platform is the Code Fragment Manager (CFM). For information about the Code Fragment Manager, see *Mac OS Runtime Architectures* and *Inside Macintosh: PowerPC System Software.*

In addition, the standards for the Mac OS user interface are often different from those on other platforms. Here are a few things to keep in mind:

- The Preferences menu item is typically stored under the Edit menu (not the File menu).
- Dialog boxes typically have the OK button located near the lower right corner, with the Cancel button to its immediate left.
- The default grayscale appearance of windows and menus in System 7.6 and later is lighter than the corresponding Windows 95 appearance.

Glossary

Abstract Window Toolkit (AWT) In the Java runtime environment, a collection of functions that allows Java programs to manipulate virtual graphics (windows, images, buttons, and so on). These abstract graphics can be translated into user-visible windows and controls on the client platform. See also **AWT Context.**

applet In the Java runtime environment, an executable program that must run within a larger host application. In JManager, an instantiated applet is called a `JMAppletViewerRef` object.

applet tag Text in an HTML document that describes an embedded applet. This text is bounded by the `<APPLET>` and `</APPLET>` delimiters. See also **Hypertext Markup Language (HTML).**

AWT context An instantiation of an execution environment in the Java runtime environment. An AWT context is a separate thread and may represent a thread group. An AWT context typically contains an applet and one or more frames. In JManager, an AWT context is called a `JMAWTContextRef` object. See also **Abstract Window Toolkit (AWT).**

code verifier A bytecode verifier that is part of the Java runtime environment. The code verifier acts as a security measure to make sure the Java code to be executed cannot crash the Java virtual machine or otherwise attempt illegal actions that might allow the code access to the host platform.

embedding application The application on a host platform (for example, a Web browser) that instantiates a Java session and executes Java applets or applications.

file system specification record On Mac OS–based platforms, a method of describing the name and location of a file or directory. File system specification records are defined by the `FSSpec` data type.

frame A user interface window in the Java virtual machine. Frames usually contain a title bar and often correspond to a user-visible window. Frames are analogous to a window record on the Mac OS. See also **parent frame.**

HTML See **Hypertext Markup Language.**

Hypertext Markup Language (HTML) A standard for describing the layout and contents of a hypertext document. An HTML document can contain an applet tag that specifies the name and location of an applet. See also **applet tag.**

Java runtime environment The Java virtual machine and the associated software required to load and execute Java code. See also **virtual machine.**

Java runtime session An instantiation of the Java runtime environment (that is, an instantiation of the Java virtual machine and associated software). In JManager a Java runtime session is called a `JMSessionRef` object. See also **virtual machine.**

parent frame The main user interface window associated with an applet. The parent frame is created when the applet is instantiated. In an AWT context, the parent frame has the index value 0. See also **frame.**

property A data item associated with an object.

session See **Java runtime session.**

text object An object of type `JMTextRef` used to encapsulate strings passed by JManager functions. In addition to the actual text, a text object also contains text encoding information and the length of the string.

thread An independent event loop in the Java virtual machine. Multiple threads can run concurrently in a Java virtual machine. A thread is also called a *lightweight process.*

Uniform Resource Locator (URL) A text string that describes the location of an HTML document. A URL may point to a file or to a server that contains the file.

URL See **Uniform Resource Locator.**

virtual machine (VM) A software package that simulates the actions of a microprocessor. A virtual machine can mimic an existing processor (such as the 68K emulator on PowerPC-based, Mac OS–compatible computers) or parse special VM-specific code. Java code requires a virtual machine environment to execute. See also **Java runtime environment, Java runtime session.**

Index

T H E   A P P L E   P U B L I S H I N G   S Y S T E M

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Line art was created using Adobe™ Illustrator and Adobe Photoshop.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Adobe Letter Gothic.

WRITER
Jun Suzuki

ILLUSTRATORS
Deborah Dennis and Ruth Anderson

DEVELOPMENTAL EDITORS
Wendy Krafft, Laurel Rezeau, Donna S. Lee, and Robin Joly

PRODUCTION EDITOR
Glen Frank

Special thanks to Patrick Beard and Steve Zellers.

Acknowledgments to Peri Frantz, Gary Little, Barry Langdon-Lassagne, Tom O'Brien, Pete Steinauer, and the rest of the MRJ reviewers.