MAC OS RUNTIME FOR JAVA

Programming Note

Using JDirect to Access Mac OS Code From Java

For MRJ 2.1

10/8/98
Technical Publications

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

Simultaneously published in the United States and Canada.

CHAPTER 1

Using JDirect to Access Mac OS Code From Java

Contents

JDirect is a programming interface that allows you to access native PowerPC C code from a Java™ application. You can use JDirect if you have either of the following needs:

- You want to call Mac OS system functions from your Java application
- You want your Java application to call an existing C library

You can use JDirect to access any Mac OS functions that are stored in Code Fragment Manager–based shared libraries (or *fragments*). For more detailed information about the Code Fragment Manager and shared libraries, you should read the document *Mac OS Runtime Architectures*.

Note
Standard Java classes, such as `java.io.File` already call through to the Mac OS Toolbox. In addition, you can access many basic Mac OS–specific functions using MRJToolkit, which uses JDirect to handle calls. Before using JDirect, you should make sure that the functionality you require is not already handled by MRJToolkit or MRJ itself. ◆

You can use JDirect with any Java application (but not applet) that can run in the Mac OS Runtime for Java (MRJ) Java environment.

Changes From JDirect 1.0

This document describes the version of JDirect available with MRJ 2.1 and later.

▲ **WARNING**
The JDirect mechanism as well as the Apple-defined wrapper classes described in JDirect 1.0 documentation have changed and may not work with versions of MRJ later than 2.0 ▲

JDirect 2.0 no longer uses `com.apple.NativeObject` to denote native functions that use JDirect. You must now create interfaces to represent the shared library fragments containing the native functions and implement them in your Java code. See “Calling Native Code Using JDirect” (page 7) for more details.

JDirect 2.0 changes names and packages for the existing wrapper classes, and adds new classes to handle opaque structures and callback functions. Table 1-1 lists changes to the wrapper classes.

Table 1-1 Changes to wrapper classes

Old	New
`com.apple.memory.MemoryObject`	`com.apple.mrj.jdirect.Struct`
`com.apple.memory.ByteObject`	`com.apple.mrj.jdirect.ByteArrayStruct`
`com.apple.memory.PointerObject`	`com.apple.mrj.jdirect.PointerStruct`
`com.apple.memory.HandleObject`	`com.apple.mrj.jdirect.HandleStruct`

In addition, Apple now supplies generated interfaces for many Mac OS system software functions. In most cases you can access Mac OS functions by simply copying the appropriate classes to your Java program and making the calls. If you only want to access Mac OS system software functions, read “Using the Mac OS Binding Sample Code” (page 24).

JDirect Versus JNI

If you want to call Mac OS C code from your Java code, there are currently two mechanisms for doing so: JDirect and the Java Native Interface (JNI). Each has advantages and disadvantages. Generally you would want to use JNI if

- You are writing your Java code and C code concurrently
- You want your code to be cross-platform
- You need to access full Java functionality from your C code (for example, if you need to manipulate Java objects in C)

JDirect is useful if

- You want to call older C functions from Java, or code whose source you cannot change (for example, legacy code or system software functions).
- You do not want the overhead of writing C glue code to establish a JNI environment and translating parameters for native calls

Overall, JDirect provides a simpler interface to native C code, but it does not allow the full flexibility of JNI.

Calling Native Code Using JDirect

JDirect lets you access Mac OS code with minimal overhead and no extra C glue code. For example, say you want to call a Mac OS function that has the header declaration shown in Listing 1-1.

Listing 1-1 Declaration for Mac OS C code

```
int someFunc (short firstParam, long secondParam, infoStruct* moreInfo);

struct infoStruct {
    long data1;
    float data2;
    }
```

To use JDirect to access Mac OS code, you must take the following steps:

1. Create an interface that extends `com.apple.jdirect.SharedLibrary`.
2. Define a static value named `libraryInstance` of type `Object` that is initialized by the method `JDirectLinker.loadLibrary`. The parameter you pass to `loadLibrary` should be the name of the shared library fragment that contains the methods you want to call.
3. Create a class that implements the interface you created in step 1.
4. If your native code manipulates pointers or data structures, you must create "wrapper" classes to represent them in the Java environment. See "Using Wrapper Objects" (page 14) for more information.
5. Declare your native methods within the class. These methods must be static, and they must have the same case-sensitive name as the corresponding symbol exported from the Mac OS shared library fragment.

IMPORTANT

The shared library fragment name is not necessarily the same as the name of the shared library file (for example, several different shared library fragments may be packaged in one shared library file). For more information about fragments, see *Mac OS Runtime Architectures.* ▲

Listing 1-2 shows the Java interface that you could use to call the `someFunc` function from Java code.

Listing 1-2 An interface to call `someFunc` from Java

```
import com.apple.jdirect.SharedLibrary;
import com.apple.mrj.jdirect.ByteArrayStruct;

interface myCodeLib extends SharedLibrary {
    static Object libraryInstance = JDirectLinker.loadLibrary ("myCodeLibFrag");
    }

public class myCodeLibFuncs implements myCodeLib {
    public static int someFunc (short firstParam, int secondParam, infoStructWrapper
                                moreInfo) {
        return someFunc (short firstParam, int secondParam, moreInfo.getByteArray());
        }

    private native static int someFunc (short firstParam, int secondParam, byte[]
                                        moreInfo);
    }

…

public class infoStructWrapper extends ByteArrayStruct {
    static final int sizeOfInfoStruct = 8;

    infoStructWrapper () {
        super (sizeOfInfoStruct);
        }

    int setData1 { return setIntAt (0);}
    float setData2 { return setFloatAt (4);}
```

```
int getData1 { return getIntAt (0);}
float getData2 { return getFloatAt (4);}

}
```

The interface `myCodeLib` extends the interface `com.apple.jdirect.SharedLibrary`.

When a class is loaded, MRJ checks to see if the class implements an interface derived from `com.apple.JDirect.SharedLibrary`. If it does, any native methods are linked from the shared library specified in the `libraryInstance` object. In this example, JDirect will look for the `someFunc` function in the shared library `myCodeLibFrag`.

Note
You can subclass any class that implements (or extends any class that implements) `com.apple.JDirect.SharedLibrary`. ◆

The class `myCodeLibFuncs` declares the native methods you want to call (in this case only `someFunc`). The types of the parameters may differ from the C declarations depending on the mappings between C and Java data types. See “Parameter Passing between Java and C” (page 11) for more information.

Note that the Java code uses a special class to represent the `infoStruct` data structure. Since Java does not use data structures, in order to manipulate them in the Java environment you must encapsulate them as objects. Apple provides several predefined classes (such as `ByteArrayStruct`) that you can extend to represent your data structures (as well as pointers and other references foreign to Java). Rather than manipulate the structure directly, you must use accessor methods to set or retrieve any elements within it. In Listing 1-2, the `infoStructWrapper` class contains a constructor as well as get/set methods for each field.

For more information about using wrapper objects for data structures and pointers, see “Using Wrapper Objects” (page 14).

IMPORTANT

You must be very careful if you are calling native code that retains pointers to objects after the call. During the call, JDirect holds the Java heap in a fixed state, but afterwards, the garbage collectors are free to compact the heap and move the contents of objects around in memory; any references retained by native code will likely become invalid. For example, if during a call you pass a reference to an array you have allocated in the Java heap, you should not expect the native code to retain access to the array after the call has completed. However, this restriction does not apply to object that have been allocated in the Mac OS heap. For example, if you had requested a new window record (by calling Mac OS Toolbox function `NewCWindow` using JDirect), references to it will always remain valid. ▲

Searching Multiple Shared Libraries

If you have a class whose native methods are implemented in several different shared libraries, you can implement each of the necessary shared library interfaces, such as in Listing 1-3.

Listing 1-3 Implementing multiple shared library interfaces

```
public class someFuncs implements InterfaceLib, someLib, someOtherLib {

    public native static void SysBeep(short duration);
    public native static int OtherBeep (short duration);
    }
```

However, problems may occur if multiple libraries implement the same methods. For example, if both `someLib` and `someOtherLib` implement `OtherBeep`, which one will get called? To minimize confusion, you must understand how JDirect prepares and searches shared libraries.

JDirect searches the shared library list from left to right. That is, in Listing 1-3 `InterfaceLib` is searched first, followed by `someLib` and `someOtherLib`. Note,

however, that JDirect will search "up" the interface hierarchy before moving to the next library. For example, say you have the following three shared libraries:

- Interface `A`, which extends `SharedLibrary`
- Interface `B`, which extends `SharedLibrary`
- Interface `C`, which extends interface `B`

Given the class definition

```
public class someFuncs implements C, A{
…
}
```

JDirect searches the libraries in the order `C`, `B`, `A`.

Note
JDirect uses a "lazy" binding method. That is, it will not search the libraries for a function until the native method is actually called for the first time. ◆

Parameter Passing between Java and C

When the native method is called, the actual signature of the method (the types of its arguments) is used to build a parameter list for the Mac OS function. All of the basic Java types – `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, and `double` – are passed as is, by value, in the order specified. Table 1-2 shows the correspondence between Java and Mac OS C data types.

Table 1-2 Java versus C data types in parameters

Java Type	C Type
`void`	`void`
`boolean`	`unsigned char` (only 0x00 or 0x01 allowed)
`byte`	`signed char`
`char`	`unsigned short`

Table 1-2 Java versus C data types in parameters

Java Type	C Type
`short`	`signed short`
`int`	`signed long`
`long`	`signed long long`
`float`	`float`
`double`	`double`
`byte[]`	`signed char*`
`char[]`	`unsigned short*`
`short[]`	`signed short*`
`int[]`	`signed long*`
`long[]`	`signed long long*`
`float[]`	`float*`
`double[]`	`double*`

Note
Data type `long long` is a 64-bit scalar type provided by the PowerPC ABI. ◆

Java arrays are passed as a pointer to the first element of the array. `Boolean` arrays are not supported.

IMPORTANT
Unlike in Java, native methods in C have no way to determine the length of an array unless it is explicitly stated (for example, if the length is passed as an additional parameter). ▲

Return values are mapped in the same fashion as the function parameters, but only built-in types are allowed. That is, JDirect does not support return types

that are arrays or Java objects. Table 1-2 shows the correspondence between Java and Mac OS C return types.

Table 1-3 Java versus C data types in return values

Java Type	C Type
`void`	`void`
`boolean`	`unsigned char` or `bool`
`byte`	`signed char`
`char`	`unsigned short`
`short`	`signed short`
`int`	`signed long`
`long`	`signed long long`
`float`	`float`
`double`	`double`

Note that JDirect compares the low byte of the C return value against zero to determine the value of a `boolean`.

Table 1-4 shows some examples of C function declarations and their corresponding Java versions.

Table 1-4 Some C function declarations and their Java versions.

C Version	Java Version
`extern short MyGetResCount( ResType theType);`	`static native short MyGetResCount(int theType);`
`extern Handle MyGetResource( ResType theType, short theID);`	`static native int MyGetResource(int theType, short theID);`

Table 1-4 Some C function declarations and their Java versions.

C Version	Java Version
`extern long MyHashFunction ( const char* cString);`	`static native int MyHashFunction(byte [] cString);`
`extern Boolean MyCompareString( ConstStr255Param s1, ConstStr255Param s2);`	`static native boolean MyCompareString(byte [] s1, byte [] s2);`
`extern void MyBlockCopy ( const void* src, void* dst, unsigned long length);`	`static native void MyBlockCopy(byte [] src, byte [] dst, int length);`

Since method overloading is allowed in Java, C functions that pass arrays or strings may be represented in more than one way in Java. For example, the Java version of `MyHashFunction` could take an integer parameter rather than a byte array:

```
static native int MyHashFunction(int cString);
```

Similarly, `MyCompareString` could also take type `int` parameters. The `MyBlockCopy` method can be overloaded in any number of ways (such as taking type `int[]`), since the C version takes `void*` parameters (which can represent any data type).

Note that if you pass an array value as `null` in Java, the value `NULL` is passed to the C function. Doing so is useful in cases where the C code has an optional parameter (pass `NULL` to ignore).

Using Wrapper Objects

To access Mac OS functions, the Java code must often be able to do things it was not meant to do, such as manipulate pointers and structures. The way to accomplish this is through wrapper classes,which hide the representation of the native data structure. If you want to call your own Mac OS code, you will have to create wrapper classes that encapsulate your data structures.

IMPORTANT

Apple has already created wrapper objects for many of the structures used by Mac OS system software functions. If you are calling system software, you should check to see if appropriate wrappers exist before writing your own. See “Using the Mac OS Binding Sample Code” (page 24) for more information. ▲

If you choose not to use the Apple-defined wrapper classes described in this section to encapsulate your objects, you must create your own. That is, you must define any objects as `byte[]` arrays and enforce a field alignment within it. Then, to access any fields (that is, elements in the array), you must define additional methods to read them (for example, `getByteAt(offset)`).

MRJ 2.1 provides primitive classes in the package `com.apple.mrj.jdirect`, which lets you manipulate memory from your Java code.

Accessing Memory

As mentioned earlier, the basic approach to representing toolbox objects in Java is to provide wrapper classes that hide the representation of the Mac OS data structure. Accessing the data inside such objects is accomplished by writing “get” and “set” methods. Since toolbox objects are either pointer-based or handle-based, we provide base-class wrapper objects, `PointerStruct`, and `HandleStruct`, which provide get/set methods for primitive values at a known offset. Both are subclasses of the abstract class `Struct`, which is shown in Listing 1-4.

Listing 1-4 The `Struct` Class

```
package com.apple.mrj.jdirect;

public abstract class Struct {

    protected abstract boolean getBooleanAt(int offset);
    protected abstract byte getByteAt(int offset);
    protected abstract byte[] getBytesAt(int offset, int numBytes);
    protected abstract char getCharAt(int offset);
    protected abstract short getShortAt(int offset);
    protected abstract int getIntAt(int offset);
```

```
    protected abstract long getLongAt(int offset);
    protected abstract float getFloatAt(int offset);
    protected abstract double getDoubleAt(int offset);

    protected abstract void setBooleanAt(int offset, boolean value);
    protected abstract void setByteAt(int offset, byte value);
    protected abstract void setBytesAt(int offset, byte[] byteVals);
    protected abstract void setCharAt(int offset, char value);
    protected abstract void setShortAt(int offset, short value);
    protected abstract void setIntAt(int offset, int value);
    protected abstract void setLongAt(int offset, long value);
    protected abstract void setFloatAt(int offset, float value);
    protected abstract void setDoubleAt(int offset, double value);
    protected abstract void setStructAt(int offset, Struct value);

    protected abstract byte[] getBytes();
    protected abstract int getSize();
    }
```

The classes `ByteArrayStruct`, `PointerStruct`, and `HandleStruct` implement the abstract `Struct` method using a byte array in the Java heap, a Mac OS pointer, and a Mac OS handle respectively. Listing 1-5 shows the `ByteStruct` class, Listing 1-6 shows the `PointerStruct` class, and Listing 1-7 shows the `HandleStruct` class.

Listing 1-5 The `ByteArrayStruct` class

```
package com.apple.mrj.jdirect;

public class ByteArrayStruct extends Struct {

/**
* Constructor that sets the size of the byte array.
* @param size the number of bytes for memory allocation.
* @return this.
*/
    protected ByteArrayStruct(int size) {...}

    /**
     * Returns the byte array containing the toolbox data structure.
```

```
     * @return the actual byte array NOT a copy
     */
    public final byte[] getByteArray() {...}

    public final int getSize() { return bytes.length; }

    protected byte[] bytes;

    ...

}
```

Listing 1-6 The `PointerStruct` class

```
package com.apple.mrj.jdirect;

public class PointerStruct extends Struct {

    /**
     * Creates a PointerStruct from an existing pointer
     * @param address the address of an existing pointer
     */
    public PointerStruct(int address) { ... }

    public final int getPointer() { return this.pointer; }

    protected int pointer;

    ...

}
```

Listing 1-7 The `HandleObject` class

```
package com.apple.mrj.jdirect;

public class HandleStruct extends Struct {
```

```
/**
* Creates a HandleStruct from the specified memory handle.
* @param handle a handle to memory.
*/
    public HandleStruct(int handle) { ... }

    public final int getHandle() { return this.handle; }

    protected int handle;

    ...

}
```

Representing Pointers and Handles

You can manipulate pointers and handles in the Java environment by creating subclasses of `PointerStruct` or `HandleStruct` respectively. For example, you can create a subclass of `PointerStruct` to represent a Mac OS window record. In addition to being able to pass the window record in the Java environment, you can include get/set methods to present a familiar interface to the fields of the Mac OS data structure. Listing 1-8 shows an example wrapper class for a Mac OS window record.

Listing 1-8 A wrapper class for a Window Record

```
import  com.apple.mrj.jdirect.PointerStruct;

class WindowPtr extends PointerStruct {

    public boolean isVisible() {
        return (getByteAt(110) != 0);       // offsetof(WindowRecord,
                                            // visible)
        }
    }
```

This simple example shows how Mac OS `Boolean` values are mapped to Java `boolean` values. Since any nonzero value can be a true value in C , we can represent Mac OS `Boolean` values as `byte` values in Java, which we convert to a

type `boolean` by comparing with zero. Note that you could also simply return `getBooleanAt(110)` to accomplish the same result.

The value 110 is the offset of the `visible` field in the `WindowRecord` data structure.

In a similar manner, you can manipulate handles in Java by wrapping them as objects derived from `HandleStruct`.

Representing Strings

Since JDirect allows you to pass a Java array directly to Mac OS functions as a pointer to its first array element, you can represent C and Pascal strings as `byte[]` array objects. C strings are merely byte arrays with an extra zero byte to signal the end of string. Pascal strings are simply byte arrays with the first element indicating the length of the string. Note that some Mac OS functions accept C strings (such as those in the standard C library) and others require Pascal string. JDirect does not perform string conversion.

Representing C Data Structures

Not all Mac OS objects are easily represented in Java. For example, C structures can contain embedded arrays, as well as nested structures. In Java, arrays are always separate objects, and fields that refer to objects are always references. You can solve this representation problem by using an opaque representation of the object defined by the `ByteArrayStruct` class.

Consider the file system specification record (`FSSpec`) in Listing 1-9.

Listing 1-9 A file system specification record

```
struct FSSpec {
    short           vRefNum;
    long            parID;
    Str63           name ;
    };
```

You can think of the structure as being one long array of byte values. You can then access a field using offsets. For example, the `parID` field would begin at an offset of 2 from the beginning of the structure (since `vRefNum` takes up 2 bytes).

The file system specification record field `name` is a Pascal string consisting of a length byte (`name[0]`) followed by 63 bytes to represent the filename.

The `ByteArrayStruct` class provides a way to represent a structure as a `byte[]` array. To access the fields in your structure, you must write get/set methods in your class. Listing 1-10 shows how you could implement the `FSSpec` structure by representing it as a `byte[]` array.

Listing 1-10 Accessing a file system specification record using the `ByteArrayStruct` class

```
import  com.apple.mrj.jdirect.ByteArrayStruct;

class FSSpec extends ByteArrayStruct {

    static final int sizeOfFSSpec = 70; // 70 bytes in size

    FSSpec() {
        super(sizeOfFSSpec);          // allocate 70 bytes please.
        }

    short getVRefNum() { return getShortAt(0); }
    int getParID() { return getIntAt(2); }

    byte[] getName() {
        int length = getByteAt(6); // filename is a Pascal string
        byte result[] = new byte[length];
        int offset = 7;
        for (int i = 0; i < length; i++)
            result[i] = getByteAt(offset++);
        return result;
        }
    }
```

The `getName` method in thie example simply copies the bytes from the name and then returns that value.

Representing Opaque Structures

JDirect uses the class `OpaqueStruct` to wrap opaque Mac OS data structures. For example, structures of type `DragReference` used by the Drag Manager are opaque and therefore inaccessible. If you wanted to handle a drag reference in Java, you must encapsulate it as an object. Listing 1-11 shows a class that you could use to wrap the drag reference.

Listing 1-11 A wrapper class for a drag reference

```
import  com.apple.mrj.jdirect.OpaqueStruct;

public class DragReferenceOpaque extends OpaqueStruct {

    public DragReferenceOpaque(int opaquePointer) {
        super(opaquePointer);
    }
}
```

Handling Callbacks

If you are calling a Mac OS function that calls back to your code, you normally provide a pointer to your application-defined function. If you are calling from Java code, however, you must encapsulate the pointer as a Java object. JDirect provides the `MethodClosure` and `MethodClosureUPP` wrapper classes to encapsulate pointers and allow C code to call back to Java code.

To create a `MethodClosure` object to pass to C code, you use the `MethodClosure` constructor, which has the following declaration:

```
protected MethodClosure (
                    Object targetObject,
                    String methodName,
                    String methodSignature);
```

For example, say you have a Mac OS function `InstallTwizzler` that takes a function pointer parameter as shown in Listing 1-12.

Listing 1-12 C Declaration of a function that takes a function pointer parameter

```
typedef short (*TwizzlerProcPtr)(long index);
extern void InstallTwizzler(TwizzlerProcPtr proc);
```

If you have a Java method named `Twizzler` that you would like to pass to `InstallTwizzler`, you must create a method closure for `Twizzler` as shown in Listing 1-13.

Listing 1-13 Creating a method closure for the Twizzle function

```
// declare class that handles Twizzler callbacks
public class MyTwizzler {

    public short Twizzle(int index) { // this is our callback method
        ...
    }
}

public class TwizzlerClosure extends MethodClosure {

    public TwizzlerClosure(Object target) {
        super(target, "Twizzle", "(I)S");
    }
}
```

Note
The method closure constructor allocates an additional block of memory in the Mac OS heap via `NewPtr`. This block is initialized with the PowerPC code necessary to reenter the Java VM and call the specified method on the specified object. When you are finished with the closure object, you should call the `dispose()` method to free this block. ◆

You must then create the standard JDirect interface to call the native function `InstallTwizzler`, as shown in Listing 1-14. This example assumes that `InstallTwizzler` is implemented in the shared library `MyLib`.

Listing 1-14 Setting up the JDirect interface for InstallTwizzler

```
// class with native method for InstallTwizzler
public class MyFuncs implements MyLib {

    public static void InstallTwizzler(TwizzlerClosure closure) {
        InstallTwizzler(closure.getProc();
    }

    public static native void InstallTwizzler(int proc);
}
```

You can then invoke the Mac OS function and pass a function pointer using the code in Listing 1-15. The Mac OS code can then call back to your Java method `Twizzle`.

Listing 1-15 Passing a method object to C code from Java

```
MyTwizzler callbackTarget = new MyTwizzler();
TwizzlerClosure thunk = new TwizzlerClosure(callbackTarget);
MyFuncs.InstallTwizzler(thunk);
…
thunk.dispose(); // remove closure object when finished
```

Note that you must maintain a reference to the method closure object in the Java environment after you pass it to the Mac OS function. If not, the object may be garbage-collected and the system could crash when the Mac OS code attempts to call back to an object that no longer exists. You can remove the reference when you no longer require the callback.

In a similar fashion, if the Mac OS function expects to receive a universal procedure pointer (UPP) to the application-defined function, you can use the `MethodClosureUPP` class to create the appropriate Java object.

Using the Mac OS Binding Sample Code

To simplify development with JDirect, Apple supplies generated Java classes for many of the Mac OS system software programming interfaces. You can easily access Mac OS system calls by copying the appropriate classes into your code base. For example, you can call the `SysBeep` function (contained in `OSUtils.h)` by copying some or all of `OSUtilFunctions` and calling `OSUtilFunctions.SysBeep`.

Apple's interface generator creates the following classes for each header file:

- One class containing all the functions
- One class containing all the constants
- One class per data structure
- One interface and one method closure class for every application-defined function (that is, every callback).

The name of each class is derived from the header file. Generally this means the class prefix is the name of the header without the `.h` suffix. However, if the name ends in `s`, the `s` is dropped to make the name more readable. Also, any header files that begin with `Mac` lose their prefix in the generated interfaces (that is, any classes generated from `MacWindows.h` would simply have the prefix `Window`).

For example, the file `Quickdraw.h` would have the following classes and interfaces associated with it:

- The class `QuickDrawConstants`, containing all the constants
- The class `QuickDrawFunctions`, containing all the functions
- A class ending with `Struct` for every data structure (for example, the class for the structure `QDGlobals` would be `QDGlobalsStruct`).
- An interface ending with `Interface` and a method closure class ending with `ClosureUPP` for every application-defined function. For example, the class and interface to support the application-defined function specified by `ColorSearchProcPtr` are `ColorSearchInterface` and `ColorSearchClosureUPP` respectively.

For more information about obtaining generated Java equivalents for Mac OS interfaces, see the following Web page and follow the links to download prerelease software:

<http://developer.apple.com/java/>

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

Special thanks to Nick Kledzik and Steve Zellers.