Summary of QuickTime for Java

Since its introduction in 1991, QuickTime has evolved into the equivalent of five large multimedia toolkits, with an extensive API in the C language. It is designed to produce full-length movies, music, animated sprites, virtual reality, and 3D modeling. Besides its widespread use for delivering high-quality digital content over the Internet, QuickTime has become the core multimedia technology used in over 11,500 CD-ROM titles and hundreds of new DVD titles.

QuickTime for Java came about to meet developers' need for a way to get at QuickTime besides using C calls. The idea was to integrate the write-once, run-everywhere capabilities of Java with QuickTime's robust media architecture.

An early version of the QuickTime for Java technology was first demonstrated at the JavaOne conference in 1998. The current version is now available free to the Java community and other QuickTime developers at `www.apple.com/quicktime/qtjava`.

This document summarizes the highlights of QuickTime for Java under the following headings:

- "The QuickTime for Java API" (page 2) summarizes the principal programming elements of the Java API for QuickTime
- "Using QuickTime in Multimedia Production" (page 3) presents several scenarios for using QuickTime for Java in the multimedia production space.
- "An Overview of QuickTime for Java" (page 5) gives you basic facts about this software.

- "Integrating QuickTime with Java" (page 9) discusses how the QuickTime for Java Application Framework works.
- "The QuickTime for Java Package Structure" (page 14) describes the packages that make up QuickTime for Java.
- "Comparing QuickTime C and Java Code" (page 25) discusses how you relate Java code to QuickTime's native C code.
- "QuickTime for Java Classes and Interfaces" (page 33) describes the `QTCanvas` and `QTFactory` classes and the `QTDrawable` interface.
- "Displaying and Streaming Movies" (page 44) summarizes the ways you can stream multimedia content using QuickTime for Java.
- "Media and Presenters" (page 57) discusses how your Java code can present QuickTime media.
- "Imaging and Effects" (page 65) summarizes the ways you can use Java to image QuickTime media and apply QuickTime effects.
- "Animation and Compositing" (page 78) introduces some of the techniques you use to animate and composite images in QuickTime for Java programs.

For full technical details see the book *QuickTime for Java,* being introduced at JavaOne in June, 1999.

THE QUICKTIME FOR JAVA API

If you're a Java or QuickTime programmer and want to harness the power of QuickTime's multimedia engine, you'll find a number of important advantages to using the QuickTime for Java API. For one thing, the API lets you access QuickTime's native runtime libraries and, additionally, provides you with a feature-rich Application Framework that enables you to integrate QuickTime capabilities into Java software.

Aside from representation in Java, QuickTime for Java also provides a set of packages that forms the basis for the Application Framework found in the `quicktime.app` group. The focus of these packages is on using QuickTime to present different kinds of media. The framework uses the interfaces in the `quicktime.app` packages to abstract and express common functionality that exists between different QuickTime objects.

As such, the services that the QuickTime for Java Application Framework renders to the developer can be viewed as belonging to the following categories:

- creation of objects that present different forms of media, using `QTFactory.makeDrawable()` methods
- various utilities (classes and methods) for dealing with single images as well as groups of related images
- spaces and controllers architecture, which enables you to deal with complex data generation or presentation requirements
- composition services that allow the complex layering and blending of different image sources
- timing services that enable you to schedule and control time-related activities
- video and audio media capture from external sources
- exposure of the QuickTime visual effects architecture

All of these are built on top of the services that QuickTime provides. The provided interfaces and classes in the `quicktime.app` packages can be used as a basis for developers to build on and extend in new ways, not just as a set of utility classes you can use.

USING QUICKTIME IN MULTIMEDIA PRODUCTION

QuickTime provides some powerful constructions that can simplify the production of applications or applets that contain or require multimedia content. Although we do not address directly the often complex production processes that can be involved, a combination of some of the techniques that are illustrated in the example code presented in this document can provide a basis to both simplify and extend the work that is involved in such productions.

For example, a multimedia presentation often includes many different facets: some animation, some video, and a desire for complex presentation and interactive delivery of this content. The spaces and controllers architecture, and the services that it provides to build animations, provides the ability to deliver presentations that can be both complex in their construction and powerful in the interactive capabilities that they provide to the user.

The media requirements for such presentations can also be complex. They may include “standard” digital video, animated characters, and customized musical instruments. QuickTime’s ability to reference movies that exist on local and remote servers provides a great deal of flexibility in the delivery of digital content.

A movie can also be used to contain the media for animated characters and / or customized musical instruments. For example, a cell-based sprite animation can be built where the images that make up the character are retrieved from a movie that is built specifically for that purpose. In another scenario, a movie can be constructed that contains both custom instruments and a description of instruments to be used from QuickTime’s built-in Software Synthesizer to play a tune.

In both cases we see a QuickTime movie used to contain media and transport this media around. Your application then uses this media to recreate its presentation. The movie in these cases is not meant to be played but is used solely as a media container. This movie can be stored locally or remotely and retrieved by the application when it is actually viewed. Of course, the same technique can be applied to any of the media types that QuickTime supports. The sprite images and custom instruments are only two possible applications of this technique.

A further interesting use of QuickTime in this production space is the ability of a QuickTime movie to contain the media data that it presents as well as to hold a reference to external media data. For example, this enables both an artist to be working on the images for an animated character and a programmer to be building the animation using these same images. This can save time, as the production house does not need to keep importing the character images, building intermediate data containers, and so on. As the artist enhances the characters, the programmer can immediately see these in his or her animation, because the animation references the same images.

AN OVERVIEW OF QUICKTIME FOR JAVA

QuickTime for Java enables Java and QuickTime programmers alike to take full advantage of QuickTime's multimedia capabilities. At its simplest level, QuickTime for Java lets you write a Java applet and run that applet on a variety of platforms. Java can be used in an applet, for example, to make Web pages more interactive so that users will be able to interact with those pages in ways they haven't before. At an advanced level, you can write applications that composite images, capture music and audio, create special effects, and use sprites for animation.

A SET OF JAVA CLASSES WITH TWO LAYERS

The QuickTime API is implemented as a set of Java classes in QuickTime for Java. These classes offer equivalent APIs for using QuickTime functionality on both Mac OS and Windows platforms.

The QuickTime for Java API consists of two layers. There is a core layer that provides the ability to access the QuickTime native runtime libraries (its API) and an Application Framework layer that makes it easy for Java programmers to integrate QuickTime capabilities into their Java software. The Application Framework layer includes

- the integration of QuickTime with the Java runtime, which includes sharing display space between Java and QuickTime and passing events from Java into QuickTime
- a set of classes that provides a number of services that simplify the authoring of QuickTime content and operation

The QuickTime for Java classes are grouped into a set of packages on the basis of common functionality, common usage, and their organization in the standard QuickTime header files. The packages provide both an object model for the QuickTime API and a logical translation or *binding* of the native function calls into Java method calls. In "Integrating QuickTime with Java" (page 9) we discuss in detail the grouping and usage of these packages, as well as how this logical translation works.

SUPPORT FOR DIFFERENT JAVA VIRTUAL MACHINES

The QuickTime for Java API supports all fully compliant Virtual Machines (VMs). On the Macintosh, it supports Apple's Macintosh Run-time for Java (MRJ) 2.1 or later; under Windows, JDK 1.1 or later. The Java VM used must be fully compliant with at least the JDK 1.1 specification and implementation from Sun Microsystems.

QuickTime for Java can also run in an applet, provided that the browser or applet viewer has one of the supported Java VM's chosen. Currently this requires the use of the Java Plug-in on Windows for Netscape Navigator and Internet Explorer browsers because those browsers do not provide a fully compliant Java 1.1 VM. On the Macintosh, you can use the Internet Explorer version 4 browser with an applet tag if MRJ 2.1 is chosen as your default VM. For Netscape Navigator version 4 on the Macintosh, QuickTime for Java applets must be viewed using the MRJ Plug-in.

This requires the HTML page to have different tags (OBJECT, EMBED or APPLET, for instance), depending on the browser. In the Software Development Kit (SDK), available for download from Apple's QuickTime for Java website (`http://www.apple.com/quicktime/qtjava`), is a JavaScript script (`AppletTag.JS`) is provided that inserts the appropriate tag when the page is viewed.

VERSION NUMBERING

The minimal version required of QuickTime is expressed in the version number of QuickTime for Java. Thus, the current release is known as QuickTime for Java version 3, which means that it is basically compatible with QuickTime 3. Subsequent releases of QuickTime for Java that have QuickTime 4 APIs in abundance will be renumbered QuickTime for Java version 4.

THE QTSIMPLEAPPLET: AN EXAMPLE

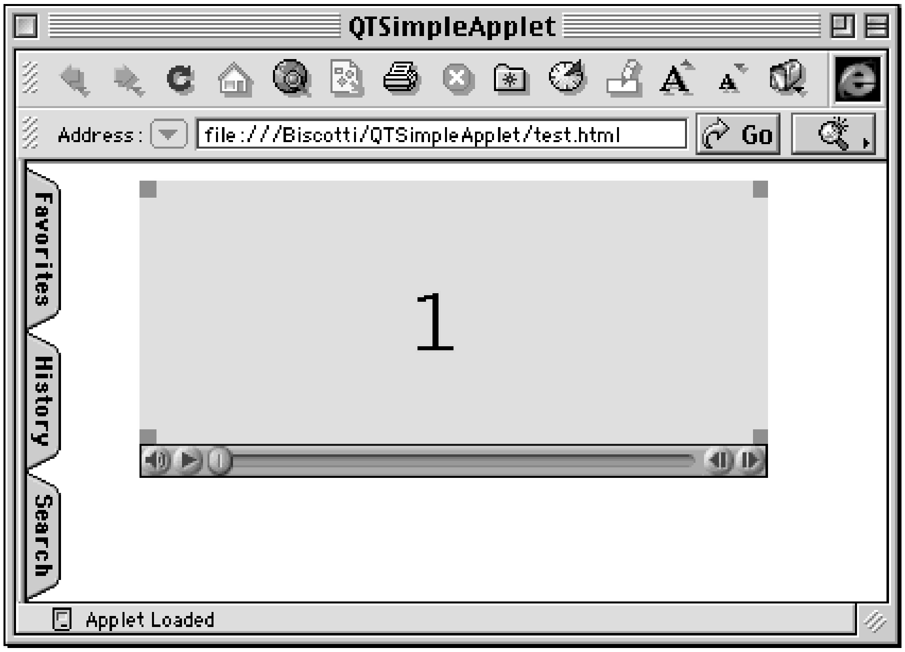
A good starting point for understanding QuickTime for Java is the `QTSimpleApplet` program, a code snippet of which is shown in Example 1.1. This program is useful for taking advantage of QuickTime presentation on the Internet to display multimedia content—and this is accomplished in less than a dozen lines of Java code. Figure 1.1 shows the output of the `QTSimpleApplet`, running locally on the user's computer in an Internet Explorer browser window. The resulting QuickTime movie, which you can control with the standard QuickTime movie controller buttons, uses the media as specified in the applet tag—in this case, `sample.mov`.

The `QTSimpleApplet` example uses `QTFactory()` methods that belong to the `quicktime.app` package to manufacture QuickTime objects that are able to present any of the wide range of media that QuickTime supports. That media may include video, audio, text, timecode, music/MIDI, sprite animation, tween, MPEG, or even movies that let you use Apple's QuickTime VR and QuickDraw 3D capabilities. The code sample is discussed in detail in "The QTSimpleApplet Code" (page 22).

EXAMPLE 1.1 *QTSimpleApplet.java*

```
public class QTSimpleApplet extends Applet {
    private Drawable myQTContent;
    private QTCanvas myQTCanvas;

    public void init () {
        try {
            QTSession.open();
            setLayout (new BorderLayout());
            myQTCanvas = new QTCanvas (QTCanvas.kInitialSize, 0.5F, 0.5F);
            add (myQTCanvas, "Center");

            QTFile file = new QTFile (getCodeBase().getFile() + getParameter("file"));
            myQTContent = QTFactory.makeDrawable (file);
        } catch (Exception qtE) {
                ...
                // handle exception
        }
    }

    public void start () {
        try {
            if (myQTCanvas != null)
                myQTCanvas.setClient (myQTContent, true);
        } catch (QTException e) {
                    }
    }

    public void stop () {
        if (myQTCanvas != null)
            myQTCanvas.removeClient();
    }

    public void destroy () {
        QTSession.close();
    }
}
```

FIGURE 1.1 *The QTSimpleApplet running in a browser window*



INTEGRATING QUICKTIME WITH JAVA

As we saw in the previous section, QuickTime for Java represents a library of classes that are designed to bring the power and functionality of QuickTime to Java. The core library of these classes provides you with the ability to access many features and capabilities in the QuickTime API. The second set of classes—an Application Framework—lets you integrate those capabilities into your Java programs.

This section discusses how that integration works. The topics discussed include the following:

- a brief introduction to some Java terminology
- grouping QuickTime for Java classes into a set of packages based on common functionality and usage

- classes in the Application Framework built on top of QuickTime for Java native binding classes
- the `QTSimpleApplet` and `PlayMovie` programs explained method by method

In this and the following sections, many code listings are given that use the various QuickTime components and structures. For detailed technical information about QuickTime, see `http://www.apple.com/quicktime`, where the complete QuickTime reference documentation is available.

SOME JAVA TERMINOLOGY

If you're a C or C++ programmer, you'll need to understand some of the key terms in the Java programming language before proceeding with the QuickTime for Java API.

In Java, you can think of an *object* as a collection of data values, or *fields*. In addition, there are *methods* that operate on that data. The data type of an object is called a *class*; and an object is referred to as an *instance* of its class. In object-oriented programming, the class defines the type of each field in the object. The class also provides the methods that operate on data that is contained in an instance of the class. You create an object using the *new* keyword. This invokes a constructor method of the class to initialize the new object. You access the fields and methods of an object by using the dot (.) operator.

In Java, methods that operate on the object itself are known as *instance methods*. These are different from the *class* methods. Class methods are declared `static`, and they operate on the class itself rather than on an individual instance of the class. The fields of a class may also be declared `static`, which makes them *class fields* instead of *instance fields*. Each object that you instantiate in Java has its own copy of each instance field, but there is only one copy of a class field, which is shared by all instances of the class.

Fields and methods of a class may have different *visibility levels*, namely, `public`, `protected`, `package`, and `private`. These different levels allow fields and methods to be used in different ways.

Every class has a *superclass*. And from that superclass it inherits fields and methods. When a class inherits from another class, it is called a *subclass* of that class. This inheritance relationship forms what is known as a *class hierarchy*. The `java.lang.Object` class is the root class of all Java classes; `Object` is the ultimate superclass of all other classes in Java.

An *interface* is a Java construct that defines methods, like a class. However, it does *not* provide implementations for those methods. A class can implement an interface by defining an appropriate implementation for each of the methods in the interface. An interface expresses the methods an object can perform—what a class can do—while making no assumptions about how the object implements these methods.

When compiled, Java classes generate a class file that is a byte-coded representation of the class. When a Java program is run, these byte codes are interpreted and often compiled (with a Just-in-Time Compiler) into the native or machine code of the runtime environment and then executed. This is the part of the work done by the Java Virtual Machine (VM). These byte codes are platform-independent and can be executed on any platform that has a Java VM.

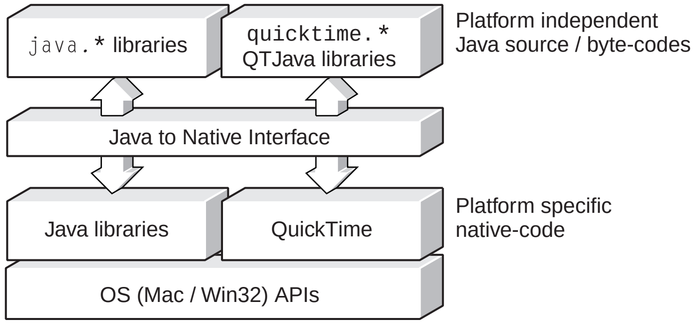
A method in a Java class can be declared to be a *native* method. A native method has no Java code; it assumes that the method is actually defined in a native library, typically in C. Native methods are used for a number of reasons: performance, access to native services provided by the operating system, and so on. In fact, many of the classes in the `java.*` packages contain native methods in order for the Java classes to intergrate with an existing operating system.

Part of the distribution of QuickTime for Java is a framework of classes. A package name (package and import) is a qualification that precedes the name of a class, i.e., it defines a name-space. The `java.*` packages are the standard set available on any distribution. QuickTime for Java uses `quicktime.*` to delineate the QuickTime for Java name-space.

QUICKTIME TO JAVA INTEGRATION STRUCTURE

Figure 1.2 illustrates a top-level view of the QuickTime to Java integration.

FIGURE 1.2 *QuickTime and Java integration*


java.* libraries
quicktime.* QTJava libraries
Platform independent Java source / byte-codes
Java to Native Interface
Java libraries
QuickTime
Platform specific native-code
OS (Mac / Win32) APIs

BINDING QUICKTIME FUNCTIONS TO JAVA METHODS

Java classes are created from structures and data types found in the standard QuickTime C language header files. These data types provide the basic class structure of the QuickTime for Java API. For example, the `Movie` data type in `Movies.h` becomes the `Movie` class. In general, the C function calls list the main data structure they operate on as the first parameter of the function. These calls become methods in this class. In line with Java conventions, all class names are capitalized, while method names are not.

The methods of a class are created from C functions. There is generally a one-to-one relationship between a native function call and a Java method. The Java method's name is derived by the following procedure:

The QuickTime native function

```
SetMovieGWorld
```

logically translates (or is bound by) the Java method `setGWorld` on the `Movie` class.

The QuickTime native function

`MCSetControllerPort`

logically translates (or is bound by) the Java method

`setPort` on the `MovieController` class.

A complete list of the QuickTime functions that QuickTime for Java binds is provided on the QuickTime for Java SDK, which is included with the book *QuickTime for Java.* The javadoc-generated documentation in HTML, also on the SDK, lists for each method the related QuickTime function call in **bold**. For example:

QuickTime::EnterMovies()

The supplied HTML documentation for these binding calls provides only brief descriptions. You need to refer to the QuickTime documentation in this website for specific details of a particular API, as well as for general discussions on the usage of particular services.

GARBAGE COLLECTION

As Java has a built-in garbage collection mechanism, the QuickTime for Java classes perform their own memory management. There are no explicit dispose calls in the QuickTime for Java API. These calls are called by the objects themselves when they perform garbage collection. The `quicktime.util.QTUtils.reclaimMemory()` method requests that the garbage collector run and can be used to help ensure disposed of memory that is no longer referenced.

The QuickTime for Java API contains no direct access to pointers or other features that are common in a C-based API. The Java method calls provide very little overhead to the native call; they do parameter marshalling and check the result of the native call for any error conditions. If an error is returned by the native call, then an exception is thrown.

THREADS

Although Java is a multi-threaded environment, the method calls that map a QuickTime function to a Java method do not provide any implicit synchronization support. If you share any QuickTime object between threads, you are responsible for dealing with any synchronization issues that may arise. The Java language provides easy services to let you do this by means of the following syntax as well as synchronized method calls:

```
synchronize (aJavaObject) { /*synchronized block of code*/ }
```

THE QUICKTIME FOR JAVA PACKAGE STRUCTURE

The QuickTime for Java classes are grouped into a set of packages. The grouping is based on common functionality and usage and on their organization in the standard QuickTime header files. The packages provide both an object model for the QuickTime API and a logical translation or binding of the native function calls into Java method calls. A number of packages also have subpackages that group together smaller sets of functionality.

The major packages generally have a constants interface that presents all of the constants that relate to this general grouping and an exception class that all errors that derive from a call in this package group will throw. The packages, with descriptions of their principal classes and interfaces, are shown in Table 1.1.

TABLE 1.1 *QuickTime for Java packages*

Package	Principal classes and interfaces	Description
`quicktime`	`QTSession`, `QTException`	The `QTSession` class has calls that set up and intialize the QuickTime engine, such as `initialize`, `gestalt`, and `enterMovies`.
`quicktime.io`	`OpenFile`, `QTFile`, `OpenMovieFile`, `QTIOException`	Contains calls that deal with file I/O. These calls are derived from the `Movies.h` file.
`quicktime.qd`	`QDGraphics`, `PixMap`, `Region`, `QDRect`, `QDColor`, `QDConstants`, `QDException`	Contains classes that represent the QuickDraw data structures that are required for the rest of the QuickTime API. These calls are derived from the `QuickDraw.h` and `QDOffscreen.h` files. The QuickTime API expects data structures that belong to QuickDraw, such as graphics ports, GWorlds, rectangles, and points.
`quicktime.qd3d`	`CameraData`, `Q3Point`, `Q3Vector`	Contains classes that represent the QuickDraw 3D data structures that are required for the rest of the QuickTime API, predominantly the tweener and 3D media services.
`quicktime.sound`	`SndChannel`, `Sound`, `SPBDevice`, `SoundConstants`, `SoundException`	Contains classes that represent the Sound Manager API. These calls are derived from the `Sound.h` file. While some basic sound recording services are provided, for more demanding sound input and output the sequence grabber components and movie playback services should be used.

Package	Principal classes and interfaces	Description
`quicktime.std`	`StdQTConstants`, `StdQTException`	The original QuickTime interfaces on the Mac OS are contained in a collection of eight header files that describe the standard QuickTime API. As such, nearly all of the functions defined in these files are to be found in classes in the `quicktime.std` group of packages.
`quicktime.std.anim`	`Sprite`, `SpriteWorld`	Classes that provide support for animation. QuickTime can be used as a real time rendering system for animation, distinct from a data format—that is, the movie. Thus, you can create a graphics space (`SpriteWorld`) within which characters (`Sprite` objects) can be manipulated.
`quicktime.std.clocks`	`Clock`, `TimeBase`, `QTCallback` and subclasses	Contains classes that provide timing services, including support for the creation of hierarchical dependencies between time bases, the usage of callbacks for user scheduling of events or notification, and the capability of instantiating the system clocks that provide the timing services.
`quicktime.std.com`	`Component`, `Component-Description`	QuickTime is a component-based architecture, with much of its funtionality being provided through the creation and implementation of a particular component's API. This package contains classes that provide basic support for this component architecture; a full implementation is forthcoming.

Package	Principal classes and interfaces	Description
`quicktime.std.image`	`CodecComponent`, `QTImage`, `CSequence`, `Matrix`	Contains classes that present the Image Compression Manager. These classes provide control for the compression and decompression of both single images and sequences of images. It also contains the `Matrix` class, which (like the `Region` class in the `qd` package) is used generally throughout QuickTime to alter and control the rendering of 2D images.
`quicktime.std.movies`	`AtomContainer`, `Movie`, `MovieController`, `Track`	Contains the principal data structures of QuickTime, including classes that represent QuickTime atom containers, movies, movie controllers, and tracks—all essential for creating and manipulating QuickTime movies. A movie containing one or more tracks is the primary way that data is organized and managed in QuickTime. A `Movie` object can be created from a file or from memory and can be saved to a file. The `MovieController` class provides the standard way that QuickTime data (movies) are presented and controlled. `AtomContainer` objects are the standard data structures used to store and retrieve data in QuickTime.
`quicktime.std.movies.media`	`DataRef`, `Media` and subclasses, `MediaHandler` and subclasses, `Sample Description` and subclasses	A `Track` object is a media neutral structure, but it contains a single `Media` type that defines the kind of data that a `Track` is representing. The `Media`, `MediaHandler` and `SampleDescription` subclasses describe the various media types that QuickTime can present. `Media` classes control references to data that comprise the raw media data.

Package	Principal classes and interfaces	Description
`quicktime.std.music`	`AtomicInstrument`, `NoteChannel`, `NoteAllocator`	Contains classes that deal with the general music architecture provided by QuickTime. This architecture can be used to capture and generate music (MIDI) events in real time, customize and create instruments, and eventually provide your own algorithmic synthesis engines.
`quicktime.std.qtcomponets`	`MovieExporter`, `MovieImporter`, `TimeCoder`	Contains classes that interface with some of the components that are provided to supply different services. The import and export components are supported, as are tween and timecode media components.
`quicktime.std.sg`	`SequenceGrabber`, `SGVideoChannel`, `SGSoundChannel`	Contains classes that implement the sequence grabber component for capturing video and audio media data.
`quicktime.util`	`QTHandle`, `QTByteObject`, `QTPointer`, `UtilException`	Contains classes that represent utility functionality required by the general QuickTime API. The most commonly used feature of this package is a set of classes for memory management from `Memory.h`. These classes typically form the base class for actual QuickTime objects.
`quicktime.vr`	`QTVRConstants`, `QTVRInstance`, `QTVRException`	Contains classes that represent the QuickTime Virtual Reality API. The package contains all the QuickTime VR interface constants, the `QTVRInstance` class and some QTVR callbacks for presentation of QTVR content.

QUICKTIME HEADERS AND JAVA CLASSES

As we’ve seen, Java classes are created from structures and data types found in the standard QuickTime C language header files. These provide the basic class structure of the QuickTime for Java API. The original QuickTime interfaces on the Mac OS are contained in a collection of eight header files that describe the standard QuickTime API. As such, nearly all of the functions defined in these files are to be found in classes in the `quicktime.std` group of packages.

The standard QuickTime C header files with their corresponding packages in the QuickTime for Java API are shown in Table 1.2.

TABLE 1.2 *C header files and corresponding QuickTime for Java packages*

QuickTime C header files	**Description**
`Components.h`	Calls from this file are in the `quicktime.std.comp` package.
`ImageCompression.h` and `ImageCodec.h`	Calls from this file are in the `quicktime.std.image` package.
`MediaHandlers.h`	Not required in QuickTime for Java.
`Movies.h`	This file has been separated into a number of packages to present a finer degree of definition and functional grouping. Sprite animation calls are in the `quicktime.std.anim` package. Callback and time-base calls are in the `quicktime.std.clocks` package. File I/O calls are in the `quicktime.io` package. All media-related calls are in the `quicktime.std.movies.media` package. Movies, movie controllers, tracks, and atom containers are in the `quicktime.std.movies` package.

QuickTime C header files	Description
`MoviesFormat.h`	Not required in QuickTime for Java.
`QuickTimeComponents.h`	This file has been separated into a number of packages to present a finer degree of definition and functional grouping. The clocks component is found in the `quicktime.std.clocks` package. Sequence grabber components calls are found in the `quicktime.std.sg` package. The remaining components are found in the `quicktime.std.qtcomponents` package.
`QuickTimeMusic.h`	All calls from this file are in the `quicktime.std.music` package.

THE APPLICATION FRAMEWORK

The classes in the QuickTime for Java Application Framework are built entirely on top of QuickTime for Java native binding classes.

The Application Framework classes are designed to simplify the usage of the QuickTime for Java API and to provide a close integration with Java's display and event distribution system. They offer a set of services that are commonly used by QuickTime programs. In addition, they provide useful abstractions and capabilities that make the use of these services simpler and easier for the developer.

The Framework itself is also designed with reusability and extensibility of classes in mind. It uses Java interfaces to express some of the functionality that can be shared or is common among different classes. You can also implement your own versions of these interfaces, or extend existing implementations, to more specifically meet a particular requirement, and in so doing, use these custom classes with other classes of the Framework itself. Table 1.3 describes the various Framework packages and their principal classes.

TABLE 1.3 *QuickTime for Java Application Framework packages*

Package	Description
`quicktime.app`	Provides a set of "factory" methods for creating classes that you can use to present media that QuickTime can import. In addition, it provides some utility methods for finding directories and files in the local file system.
`quicktime.app.action`	Contains a large number of useful controller classes for mouse drags and for handling mouse events. It also contains action classes that can be used to apply actions to target objects.
`quicktime.app.anim`	Contains classes that present all of the functionality of the `Sprite` and `SpriteWorld`.
`quicktime.app.audio`	Contains a number of interfaces and classes that deal specifically with the audio capabilities of QuickTime.
`quicktime.app.display`	Contains a number of classes that are important for using the QuickTime for Java API. `QTCanvas` and `QTDrawable` negotiate with `java.awt` classes to allow the presentation of QuickTime content within a Java window or display space.
`quicktime.app.image`	Handles the presentation and manipulation of images. Included are utility classes for setting transparent colors in images, applying visual effects, creating objects for handling sequences of images, and `QTDrawable` objects that read image data from a file or load the data into memory.
`quicktime.app.players`	`QTPlayer` and `MoviePlayer` define a set of useful methods that enables you to present QuickTime movies, using `QTCanvas` objects and the `QTDrawable` interface.
`quicktime.app.sg`	Contains a single class, `SGDrawer`.
`quicktime.app.spaces`	Interfaces in this package provide a uniform means of dealing with a collection of objects in QuickTime for Java.
`quicktime.app.time`	Provides a set of useful classes to handle timing services used to schedule regular tasks that need to be performed on an ongoing basis.

THE QTSIMPLEAPPLET CODE

The sample code `QTSimpleApplet` (available in the SDK that comes with the book *QuickTime for Java*) is a QuickTime for Java applet you can create that presents any of the media file formats that QuickTime supports. QuickTime includes support for a vast array of common file formats: QuickTime movies (including QuickTime VR), pictures, sounds, MIDI, and QuickDraw 3D.

The applet tag for this applet is

```
<applet code="QTSimpleApplet.class" width=200 height=100>
    <param name="file" value="media/crossfad.gif">
</applet>
```

The `QTSimpleApplet` code takes any of the media file types supported by QuickTime as a parameter in the HTML applet tag and creates the appropriate object for that media type:

```
param name=file value=”MyMediaFile.xxx”
```

The `QTCanvas`, `QTDrawable`, and `QTFactory` classes, which are part of the `QTSimpleApplet` code, are discussed in greater detail in the next section.

As with all Java applets, we begin in the `QTSimpleApplet` code by declaring a list of Java packages and QuickTime for Java packages that contain the required classes you need to import:

```
import java.applet.Applet;
import java.awt.*;
import quicktime.QTSession;
import quicktime.io.QTFile;

import quicktime.app.QTFactory;
import quicktime.app.display.QTCanvas;
import quicktime.app.display.Drawable;
import quicktime.QTException;
```

To get the resources for the simple applet and set up the environment, including the creation of the `QTCanvas` object, you use the `init()` method, as shown in the snippet below. `QTCanvas` is the object responsible for handling the integration from the `java.awt` side between Java and QuickTime. The `QTCanvas` also has parameters that let you control the resizing of the client that it presents. In this case, we tell the canvas to center the client within the space given by the applet's layout manager. This ensures that the client is only as big as its initial size (or smaller if you make the canvas smaller).

The `QTSession.open()` call performs a gestalt check to make sure that QuickTime is present and is initialized. Note that this is a *required* call before any QuickTime for Java classes can be used:

```
public void init () {
    try {
        QTSession.open()
```

To set up a `QTCanvas` object that displays its content at its original size or smaller and is centered in the space given to the `QTCanvas` when the applet is laid out, we do the following:

```
        setLayout (new BorderLayout());
        myQTCanvas = new QTCanvas (QTCanvas.kInitialSize, 0.5F,
                0.5F);
        add (myQTCanvas, "Center");

        QTFile file = new QTFile (getCodeBase().getFile() +
                                    getParameter("file"));
        myQTContent = QTFactory.makeDrawable (file);
    } catch (Exception e) {
        e.printStackTrace();
        ...
    }
}
```

The `QTFactory.makeDrawable()` method is used to create an appropriate QuickTime object for the media that is specified in the <PARAM> tag.

If a `QTException` is thrown in the `init()` method, an appropriate action should be taken by the applet, depending on the error reported.

In the `start()` method shown in the next code snippet, you set the client of the `QTCanvas`. This `QTCanvas.client` is the QuickTime object (i.e., an object that implements the QTDrawable interface) that draws to the area of the screen that the QTCanvas occupies. This is the QuickTime side of the integration between Java and QuickTime:

```
public void start () {
    try { myQTCanvas.setClient (myQTContent, true);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

You use the stop() method to remove the client from the QTCanvas. It will be reset in the start() method if the applet is restarted. destroy() is used to close the QTSession. This protocol enables the applet to be reloaded, suspended, and resumed—for example, if the user is leaving and returning to the page with the applet. The init()/destroy(), and start()/stop() methods are reciprocal in their activities.

```
public void stop () {
    myQTCanvas.removeClient();
}

public void destroy () {
    QTSession.close();
}
```

You need to call QTSession.close() if you have previously called QTSession.open() in order to shut down QuickTime properly.

The init() method may throw exceptions because the required file was not found or the applet does not have permission from Java's security manager to read that file. Alternatively, the required version of QuickTime may not be installed. The applet should deal with these issues appropriately.

COMPARING QUICKTIME C AND JAVA CODE

Much of the sample code available for QuickTime is presented in the C programming language. Comparing Example 1.2 and Example 1.3 in C with the Java version shown in Example 1.4 can aid in understanding how to translate C to Java code.

GETTING A MOVIE FROM A FILE

Before your application can work with a movie, you must load the movie from its file. You must open the movie file and create a new movie from the movie stored in the file. You can then work with the movie. You use the `OpenMovieFile` function to open a movie file and the `NewMovieFromFile` function to load a movie from a movie file. The code in Example 1.2 shows how you can use these functions.

EXAMPLE 1.2 *Getting a movie from a file using C code*

```
Movie GetMovie (void)
{
    OSErr               err;
    SFTypeList          typeList = {MovieFileType,0,0,0};
    StandardFileReply   reply;
    Movie               aMovie = nil;
    short               movieResFile;

    StandardGetFilePreview (nil, 1, typeList, &reply);
    if (reply.sfGood)
    {
        err = OpenMovieFile (&reply.sfFile, &movieResFile,
                             fsRdPerm);

        if (err == noErr)
        {
            short movieResID = 0; /* want first movie */
            Str255 movieName;
            Boolean wasChanged;
            err = NewMovieFromFile (&aMovie, movieResFile,
                                    &movieResID,
                                    movieName,
                                    newMovieActive, /* flags */
                                    &wasChanged);
            CloseMovieFile (movieResFile);
        }
    }
    return aMovie;
}
```

The QuickTime Movie Toolbox uses Alias Manager and File Manager functions to manage a movie's references to its data. A movie file does not necessarily contain the movie's data. The movie's data may reside in other files, which are referred to by the movie file. When your application instructs the Movie Toolbox to play a movie, the Toolbox attempts to collect the movie's data. If the movie has become separated from its data, the Movie Toolbox uses the Alias Manager to locate the data files. During this search, the Movie Toolbox automatically displays a dialog box. The user can cancel the search by clicking the Stop button.

The code in Example 1.3 shows the steps your application must follow in order to play a movie. This program retrieves a movie, sizes the window properly, plays the movie forward, and exits. This program uses the `GetMovie` function shown in Example 1.2 to retrieve a movie from a movie file.

EXAMPLE 1.3 *Playing a movie*

```
#include <Types.h>
#include <Traps.h>
#include <Menus.h>
#include <Fonts.h>
#include <Packages.h>
#include <GestaltEqu.h>
#include "Movies.h"
#include "ImageCompression.h"

/* #include "QuickTimeComponents.h" */
```

```
#define doTheRightThing 5000

void main (void)
{
    WindowPtr aWindow;
    Rect windowRect;
    Rect movieBox;
    Movie aMovie;
    Boolean done = false;
    OSErr err;
    EventRecord theEvent;
    WindowPtr whichWindow;
    short part;

    InitGraf (&qd.thePort);
    InitFonts ();
    InitWindows ();
    InitMenus ();
    TEInit ();
    InitDialogs (nil);
    err = EnterMovies ();
    if (err) return;

    SetRect (&windowRect, 100, 100, 200, 200);
    aWindow = NewCWindow (nil, &windowRect, "\pMovie",
                         false, noGrowDocProc, (WindowPtr)-1,
                         true, 0);

    SetPort (aWindow);
    aMovie = GetMovie ();
    if (aMovie == nil) return;

    GetMovieBox (aMovie, &movieBox);
    OffsetRect (&movieBox, -movieBox.left, -movieBox.top);
    SetMovieBox (aMovie, &movieBox);

    SizeWindow (aWindow, movieBox.right, movieBox.bottom, true);
    ShowWindow (aWindow);

    SetMovieGWorld (aMovie, (CGrafPtr)aWindow, nil);
    StartMovie (aMovie);
while ( !IsMovieDone(aMovie) && !done )
{
    if (WaitNextEvent (everyEvent, &theEvent, 0, nil))
    {
        switch ( theEvent.what )
    {
            case updateEvt:
                whichWindow = (WindowPtr)theEvent.message;
                if (whichWindow == aWindow)
                {
                    BeginUpdate (whichWindow);
                    UpdateMovie(aMovie);
                    SetPort (whichWindow);
                    EraseRect (&whichWindow->portRect);
                    EndUpdate (whichWindow);
                }
                break;

            case mouseDown:
                part = FindWindow (theEvent.where,
                                        &whichWindow);
                if (whichWindow == aWindow)
                {
                    switch (part)
                    {
                        case inGoAway:
                            done = TrackGoAway (whichWindow,
                                                    theEvent.where);
                            break;
                        case inDrag:
                            DragWindow (whichWindow,
                                            theEvent.where,
                                            &qd.screenBits.bounds);
                            break;
                    }
                }
                break;
        }
}
            MoviesTask (aMovie, DoTheRightThing);
        }
        DisposeMovie (aMovie);
        DisposeWindow (aWindow);
}
```

PLAYING A QUICKTIME MOVIE

Example 1.4 shows how to display any QuickTime content within a `java.awt` display space using the `QTCanvas`. It also demonstrates the use of the different resize options of the `QTCanvas` (with the alignment set to center it in the display space). You use the movie controller to select and then play a QuickTime movie, which can be a local file or a URL specified by the user.

You call `QTSession.open()` to perform a gestalt check to ensure that QuickTime is present and is initialized. This is a required call before any QuickTime Java classes can be used.

The window is the size of the movie and resizing the window will resize the movie. The `QTCanvas` is set to allow any size and is the central component in a `java.awt.BorderLayout` of its parent `Frame`.

You use the following methods to lay out and resize the `Frame` to the size of the `Movie`:

```
pm.pack();
pm.show();
pm.toFront();
```

You now prompt the user to select a movie file:

```
QTFile qtf = QTFile.standardGetFilePreview(QTFile.kStandardQTFileTypes);
```

You open the selected file and make a movie from it, using these calls:

```
OpenMovieFile movieFile = OpenMovieFile.asRead(qtf);
Movie m = Movie.fromFile (movieFile);
```

You construct a movie controller from the resultant movie, enabling the keys so the user can interact with the movie by using the keyboard:

```
MovieController mc = new MovieController (m);
mc.setKeysEnabled (true);
```

You create a `QTCanvas` so that the `MovieController` has somewhere to draw and add it to the `Frame`:

```
QTCanvas myQTCanvas = new QTCanvas();
add (myQTCanvas);
```

You construct the `QTDrawable` object to present a movie controller:

```
QTPlayer myQTPlayer = new QTPlayer (mc);
```

Now you set it as the drawing client of the `QTCanvas` for a `QTPlayer`. This also registers interests for both mouse and key events that originate in the `QTCanvas`:

```
myQTCanvas.setClient (myQTPlayer, true);
```

You add a `WindowListener` to this frame that will close down the `QTSession`. Finally, you dispose of the `Frame` that closes down the window and you exit:

```
addWindowListener(new WindowAdapter () {
    public void windowClosing (WindowEvent e) {
        QTSession.close();
        dispose();
    }

    public void windowClosed (WindowEvent e) {
        System.exit(0);
    }
});
```

When the user closes the window, the program quits, first calling `QTSession.close` to terminate QuickTime. You need to call `QTSession.close()` if you have previously called `QTSession.open()` in order to shut down QuickTime properly. `QTSession.close()` is called before the canvas that the QuickTime object is attached to is disposed of. This enables QuickTime to clean up its graphics objects, which it attaches to the native implementation of the `QTCanvas`.

EXAMPLE 1.4 *PlayMove.java*

```
import java.awt.*;
import java.awt.event.*;

import quicktime.*;
import quicktime.io.*;
import quicktime.std.movies.*;
import quicktime.app.display.QTCanvas;
import quicktime.app.players.QTPlayer;

public class PlayMovie extends Frame {

    public static void main (String args[]) {
        try {
            QTSession.open ();
            PlayMovie pm = new PlayMovie("QT in Java");
            pm.pack();
            pm.show();
            pm.toFront();
        } catch (QTException e) {
                // handle errors
                . . .
        }
    }

    PlayMovie (String title) throws QTException {
        super (title);

        QTFile qtf = QTFile.standardGetFilePreview(QTFile.kStandardQTFileTypes);

        OpenMovieFile movieFile = OpenMovieFile.asRead(qtf);
        Movie m = Movie.fromFile (movieFile);

        MovieController mc = new MovieController (m);
        mc.setKeysEnabled (true);

        QTCanvas myQTCanvas = new QTCanvas();
        add (myQTCanvas);

        QTPlayer myQTPlayer = new QTPlayer (mc);
        myQTCanvas.setClient (myQTPlayer, true);

        addWindowListener(new WindowAdapter () {
            public void windowClosing (WindowEvent e) {
                QTSession.close();
                dispose();
            }

            public void windowClosed (WindowEvent e) {
                System.exit(0);
            }
        });
    }
}
```

SUMMARY COMPARISION

We've seen two bodies of code illustrated in several examples, one in C and the other in Java. In summary, we could note the following points:

- Both pieces of code can open and play a vast number of media files.
- The C code is specific to the Macintosh; the Windows version (not shown) is different—though only slightly. Of course, as an application in C is developed around QuickTime, more and more platform-specific code needs to be written, whereas with Java, a framework is provided that is a cross-platform API as well as a cross-platform execution model.
- The Java code benefits from the Java class framework with which a developer may already be familiar.
- Java runs anywhere, unchanged, so long as QuickTime is available. As other client operating systems gain QuickTime support, the QuickTime for Java code will run there, too.
- The Java code is arguably simpler.

QUICKTIME FOR JAVA CLASSES AND INTERFACES

This section discusses the usage of two principal classes and one principal interface in the QuickTime for Java API. These are the `QTCanvas`, `QTDrawable`, and `QTFactory`. As we saw in the preceding section, the `QTCanvas` is a subclass of the `java.awt.Canvas` and represents primarily a way to gain access to the underlying native graphics structure of the platform. QuickTime requires this in order to draw to the screen. `QTDrawable` is an interface that expresses this requirement of QuickTime objects to draw to this native graphics structure. `QTFactory` is a class that uses the importing capabilities of QuickTime to "manufacture" `QTDrawable` objects to present that media.

In this section, we discuss how to

- use the `QTCanvas` class to interact with the Java display and event system
- use the `QTDrawable` interface to encapsulate common operations that can be applied to a QuickTime drawing object
- present media that QuickTime can import by using "factory" methods provided by the `QTFactory` class

The section also introduces briefly an abstraction known as a *space*, which is part of the QuickTime for Java spaces and controllers architecture. A space defines and organizes the behavior of objects and allows for the complex representation of disparate media types.

THE QTCANVAS CLASS

To present QuickTime content within Java, you need a mechanism for interacting with the Java display and event system. This is provided through the services of the `QTCanvas` class and its client, an object that implements the `Drawable` interface. `QTCanvas` is a specialized canvas that supplies access to the native graphics environment and offers expanded display functionality.

`QTCanvas` encapsulates much of the display and presentation functionality of the QuickTime for Java API. It is responsible for “punching a hole” within the Java display surface and telling its client that it can draw to this display surface and receive events that occur therein. Its clients are generally some kind of QuickTime object, and the events it receives may be mouse or key events. An instance of a `QTCanvas` object can display any object that implements the `Drawable` interface.

The client of a `QTCanvas` object is called `Drawable` because QuickTime generally uses the word “draw”—for example, `MCDraw`. Java uses the word “paint,” so this enables us to make a distinction between Java objects that “paint” and QuickTime objects that “draw.”

Your code can interact with the `QTCanvas` methods as with those of any `java.awt.Component`. The `QTCanvas` delegates calls as appropriate to its drawing client. A client essentially draws itself in the display area of a `QTCanvas`. If the client is a `QTDrawable`, the `QTCanvas` also gets the graphics structure (`QDGraphics`) of the native implementation of the canvas’s peer and sets this `QDGraphics` as the destination `QDGraphics` of such a client. All `QTDrawable` objects require a destination `QDGraphics` in which to draw.

Using the `setClient()` method, you can associate a new client, a `Drawable` object, with this `QTCanvas`. The flag determines if `awt` performs a layout and how the client is integrated with the canvas. If the flag is `false`, the new client takes on the current size and position of the canvas. If the flag is `true`, then `awt` lays out the canvas again, using the initial size of the client and the resize flags to resize the canvas and its client. The `getClient()` method returns the `Drawable` object currently associated with this `QTCanvas`.

Interacting With Java Layout Managers

QuickTime media content is often sensitive to the size and proportions of screen space that it uses—for instance, a movie that is made to display at 320 x 240 pixels can look bad or have serious performance repercussions if drawn at 600 x 60 pixels on the screen.

The QTCanvas provides flags for controlling how the Java layout managers allocate space to it (resizeFlags) and where, within that space, the QTCanvas client actually draws (alignmentFlags). You can also set the minimum, preferred, and maximum sizes of the QTCanvas, as with any other java.awt.Component. The preferred size is automatically set for you as the initial size of the QTCanvas client if your application does not specifically set the preferred size itself.

THE QTDRAWABLE INTERFACE

The QTDrawable interface encapsulates the common operations that can be applied to a QuickTime drawing object. It also presents an interface that expresses the required methods that the QTCanvas needs to call upon its drawing client.

Objects that implement the QTDrawable interface draw into the QDGraphics object presented to the client by its QTCanvas. As a consequence, this is not an appropriate interface for objects that are not QuickTime-based drawing objects. QTExceptions can be thrown by any of the methods in this interface and would indicate that either the graphics environment has changed in some unexpected way or that the media object itself is in some unexpected state.

Figure 1.3 represents the QTDrawable interfaces and the classes in QuickTime for Java that implement this interface.

Working With the QTDrawable Interface

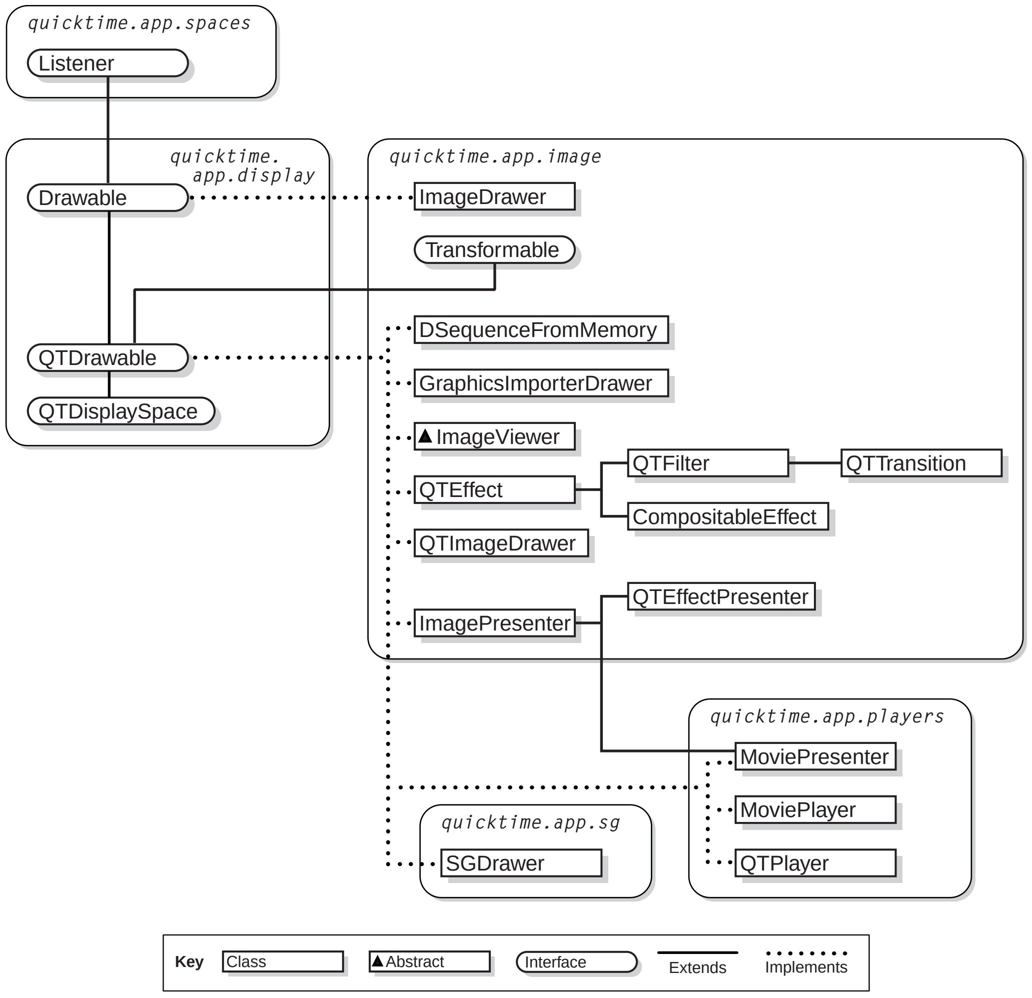
As we've seen, the QTDrawable interface is used to handle the negotiation between the QTCanvas and one of several QuickTime objects. The QuickTime objects that implement the QTDrawable interface are extensive and include the following:

- the QTPlayer class, which presents the movie controller
- the MoviePlayer or MoviePresenter classes, which present the Movie data type
- the GraphicsImporterDrawer class, which wraps the graphics importer to present images from a data source, typically image files

- the `ImagePresenter` class, which presents the `DSequence` but for a single image only, allowing image data to be loaded and kept in memory, providing a faster redraw than the `GraphicsImporter` object
- the `SWCompositor` class, which presents the sprite world
- the `SGDrawer` class, which presents the sequence grabber and sequence grabber channel (video) components
- the `GroupDrawable` class, which groups `QTDrawable` objects into the same canvas.
- the `QTEffect` class, which wraps the visual effects architecture of QuickTime
- the `QTImageDrawer` class, which allows the results of Java painting into a `java.awt.Image` to be captured and drawn by QuickTime

The `QTDrawable` interface is designed to work hand-in-hand with a `QTCanvas` object. The class that implements this interface draws into the supplied `QDGraphics` object. The `QTCanvas` will call the methods of its client (setting its destination `QDGraphics`, setting display bounds, and redrawing it) as required.

FIGURE 1.3 *The QTDrawable implementation*



QTDrawable Methods

The `QTDrawable` interface expresses the capabilities that all QuickTime drawing objects possess. The `QTCanvas` uses the following methods of the `QTDrawable` interface:

- `addedTo()`, `removedFrom()`

These methods are used by the `QTCanvas` to notify the client object that it has been added to or removed from the `QTCanvas`. Various clients require this notification:

- ❑ `QTPlayer`, so it can declare its interest in mouse and key events
- ❑ `QTImageDrawer`, so that it can create an `java.awt.Image` object to paint into
- ❑ `QTDisplaySpaces`, to allow the attachment of controllers that are interested in receiving events whose source is the `QTCanvas`

- `setDisplayBounds()`, `getDisplayBounds()`

  These methods are used to get the current size of a `QTCanvas` client and set its size. The `QTCanvas` determines the size of itself and its client based on a complex interaction between the size the `QTCanvas` parent container allocates to the `QTCanvas`, the initial (or best) size of its client, and the setting of the resize and alignment flags of the `QTCanvas` itself.

  Once the `QTCanvas` has determined the correct size of itself and its client, it uses the `setDisplayBounds()` method to resize and locate its client. The size and location of a `QTCanvas` and its client is always the same.

- `redraw()`

  This method is used to tell the client to redraw itself.

  Both the `setDisplayBounds()` and `redraw()` methods of a client are only called by the `QTCanvas paint()` method, with the `setDisplayBounds()` call, if required, being always called before a `redraw()` call.

- `getGWorld()`, `setGWorld()`

  These methods are used to set the `QTDrawable` client's destination `QDGraphics` if the client of a `QTCanvas` is a `QTDrawable`, as is normally the case. All `QTDrawable` objects must have a destination `QDGraphics` at all times. As such, if a `QTDrawable` client is removed from a `QTCanvas` or the `QTCanvas` has been hidden

```
myQTCanvas.setVisible (false);
```

then a special `QDGraphics`, `QDGraphics.scratch()`, is used to indicate to the `QTDrawable` that it is invisible and basically disabled.

While the following methods are not used specifically in the relationship between `QTCanvas` and its client, they are presented here for the sake of completeness and also express capabilities that all `QTDrawable` objects possess:

- `getClip()`, `setClip()`

  All `QTDrawable` objects have the ability to clip their drawn pixels to a specified region. The `getClip()` and `setClip()` methods are used to get and set a clipping region. `null` can be used and returned, and it indicates that the `QTDrawable` currently has no clipping region set.

  The `DirectGroup` display space enables one or more `QTDrawable` objects to draw into the same `QTCanvas` destination `QDGraphics`. It provides the capability to layer the objects in such a group, so that the objects do not draw over each other. This layering is achieved through the use of clipping regions. The `DirectGroup` clips its members so that members that are behind others cannot draw into the area where those in front are positioned. As this group by definition draws directly to the screen, this is the only means that can be used to avoid the flickering that would occur if the `QTDrawable` objects could not be clipped.

- `getMatrix()`, `setMatrix()`, `getInitialSize()`

  The `QTDrawable` interface also extends the `Transformable` interface. The `Transformable` interface expresses the ability of QuickTime drawers to have a matrix applied that will map the source pixels of a drawing object to some transformed destination appearance. A matrix allows the following transformations to be applied in the rendering process:

  - *translation*. Drawing pixels from an x or y location
  - *scaling*. Scaling the image by an x or y scaling factor
  - *rotation*. Rotating the image a specified number of degrees
  - *skew*. Skewing the image by a specified amount

- ❑ *perspective.* Applying an appearance of perspective to the image

Your application can freely mix and match these different transformations at its own discretion.

All `QTDrawable` objects can have a matrix applied to them that will transform their visual appearance. A `Sprite` in a `SpriteWorld` can also have matrix transformations applied to it, as can a 3D model. The `TwoDSprite` also implements the `Transformable` interface.

The `Transformable` interface allows for applications to define behaviors that can be applied to any of these objects. For example, the `quicktime.app.actions.Dragger`, which will position objects in response to a `mouseDragged` event, is defined totally in terms of the `Transformable` interface. Thus, the `Dragger` can reposition any `TwoDSprite` in a `SWCompositor` or any `QTDrawable` in a `GroupDrawable`.

Using interfaces to express common functionality is a powerful concept and is relied upon extensively in the QuickTime for Java Application Framework.

THE QTFACTORY CLASS

The `QTFactory` class provides “factory” methods for creating classes used to present media that QuickTime can import. The `makeDrawable()` methods of `QTFactory` use the QuickTime importers to return an appropriate QuickTime object that can present any of a wide range of media types (images, movies, sounds, MIDI, and so on) that QuickTime can import.

Given a file, a Java `InputStream` object, or a URL, the `makeDrawable()` methods that belong to the `QTFactory` class examine the contained media and return an object that best presents that kind of media. For example:

- movies—a `QTPlayer` object
- images—a `GraphicsImporterDrawer` object
- sound media—a `QTPlayer` object
- MIDI media—a `QTPlayer` object

Once you have the `QTDrawable` object, you merely add it to the canvas and the visual component of the media is presented in the canvas' display space.

There are three versions of the `makeDrawable()` method. The first two methods deal with a file, either local or remote:

- `QTFactory.makeDrawable( QTFile, . . .)`
- `QTFactory.makeDrawable( String URL, . . .)`

  The `QTFile` version is a local file, whereas the URL version can use any of the protocols known to QuickTime:

  - file: for a local file
  - HTTP: for a remote file
  - FTP: for a remote file
  - RTSP: for a movie with streaming content

- `QTFactory.makeDrawable (InputStream, . . . )`

  In this method, the media data can be derived from any source—for example, from a `ZipEntry`, from a local or remote file, or from memory. The `readBytes()` method of the input stream is used to read all of the source object's bytes into memory. Then this memory is used to create the appropriate `QTDrawable`.

In the first two cases, QuickTime can use details about the file to determine the type of media that the file contains. Many media formats are not self-describing. That is, they don't have information in the data that describes what they are. Usually, the file type describes to QuickTime this important detail.

In the case of the `InputStream`, however, because the data can come from an unknown or arbitrary source, your application must describe to QuickTime the format of the source data so that it can import it successfully. As such, these methods require the provision of a `hintString` and `hintType`. You can specify that the hint is either the file extension, Mac OS file type, or MIMEType of the source data.

Once the `makeDrawable()` methods determine the media, they create either a `Movie` or a `GraphicsImporter` as the QuickTime object to present that media. They then pass off this `Movie` or `GraphicsImporter` to a `QTDrawableMaker` to return a `QTDrawable` object to present that `Movie` or `GraphicsImporter`.

Your application uses the default `QTDrawableMaker` if you do not specify one. This returns a `QTPlayer` that creates a `MovieController` for the `Movie` or a `GraphicsImporterDrawer` for the `GraphicsImporter`. An application can also specify a custom `QTDrawableMaker` that will create a required object for one or both of these two cases.

The `QTFactory` also provides methods to locate files in the known directories of the Java application when it executes. In order for QuickTime to open a local file, it must have the absolute path and name of the file. Your application may want to open files where it can know the relative location of the file from where it is executing. For instance, many of the code samples in the QuickTime for Java SDK will look for files in the media directory that is in the SDK directory. This media directory is added to the class path when the application is launched. The `QTFactory.findAbsolutePath()` method is used by the application to find the absolute path of the file, and to find those files, at runtime.

- `addDirectory()`

  Allows your application to specify a directory that is added to the known directories that are used in searching for files

- `removeDirectory()`

  Allows your application to remove directories where it doesn't want files to be found. This can shorten the search process, as the application can remove the specified directory from the search paths that the `find()...` methods will search for specified files in.

- `findAbsolutePath()`

  Given a relative file (or directory), this method will search for this specified path in

  - ❑ application-specified directories
  - ❑ `user.dir`, which is a directory that is one of the system properties of the Java runtime

- ❑ any directory known in the class path directories. `class.path` is also a system property of the Java runtime.

  This method searches for the specified path as this path is appended to these known directories. It returns the first occurrence of the file (or directory) that exists in these locations. If the specified file is not found, a `FileNotFoundException` is thrown.

- ■ `findInSystemPaths()`

  Whereas the `findAbsolutePath()` will look for the specified file by appending that file to the known directories, this method will do a recursive search for the specified file or subdirectory of all of the known or registered directories and their parents. This can be a time-consuming search, and typically your application will only use this method to find a file that the user has misplaced or is not found in the `findAbsolutePath()` method. If the specified file is not found, a `FileNotFoundException` is thrown.

SPACES AND CONTROLLERS ARCHITECTURE

The QuickTime for Java API introduces an abstraction known as a *space*. A space defines and organizes the behavior of objects and allows for the complex representation of disparate media types. Spaces provide a powerful and useful means to assemble and control a complex presentation.

You use a space to create a "world" that can be controlled dynamically from Java at runtime. This world could have certain characteristics that define how the objects exist and interact with others. Your application assembles the space with its objects or *members* and uses Java's event model to allow for user interaction with those objects. Because it is a dynamic environment, decisions about behavior and even which members belong to the space can be deferred until runtime. The `Space` interface, which is part of the `quicktime.app.spaces` package, provides the standard API that all spaces support.

QuickTime for Java controllers manipulate Java objects in spaces. These controllers can define the standard behavior for a group of objects by defining the behavior of objects over time, by monitoring objects, and by responding to user events. Controllers provide a uniform way of enforcing the same behavior on a group of objects. The behavior of a controller depends upon the support protocols defined by a space and by a controller itself.

The QuickTime for Java API provides a `Controller` interface and defines a protocol for the interaction between spaces and controllers. While the focus in this release of QuickTime for Java is in presentation, the architecture is general enough to be applied in the model space for generating data.

DISPLAYING AND STREAMING MOVIES

QuickTime provides a set of APIs that enable you to stream multimedia content over a network in real time, as opposed to downloading that content and storing it locally prior to presentation. With streaming, the timing and speed of transmission as well as the display of data are determined by the nature of the content rather than the speed of the network, server, or the client. Thus, a one-minute-long QuickTime movie is streamed over a network so that it can be displayed or presented in one minute of real time.

In the QuickTime streaming architecture, a *stream* is, simply, a track in a movie. QuickTime lets you stream a broad variety of content—audio, video, text, and MIDI; the output of any audio or video codec supported in QuickTime can be streamed, in fact. If your application is QuickTime-savvy, you can automatically take advantage of this multimedia streaming capability.

In addition to demonstrating some useful techniques for displaying QuickTime movies, this section shows you how to play a streaming movie from a URL. It builds on the concepts and examples discussed in “Integrating QuickTime with Java” (page 9) and “QuickTime for Java Classes and Interfaces” (page 33) and shows you how to

- select and then play a QuickTime movie with its controller detached

- convert a screen to full-screen mode and back to normal mode
- display a QuickTime movie within a window and add callbacks that are triggered at some specific time during movie playback

PLAY A STREAMING MOVIE

Example 1.5 builds on the `QTSimpleApplet` code discussed in "The QTSimpleApplet Code" (page 22). This applet enables you to play a steaming movie from a URL.

You define the instance variables for the applet:

```
private Drawable myQTContent;
private QTCanvas myQTCanvas;
```

Just as with the `QTSimpleApplet` code sample, you can use the standard `init()`, `start()`, `stop()`, and `destroy()` methods to initialize, execute, and terminate the applet. Likewise, you call `QTSession.open()` in order to make sure that QuickTime is present and initialized. Again, this is a *required* call before any QuickTime for Java classes can be used. It is called first in the `init()` method. In order to shut down QuickTime properly, you also need to call `QTSession.close()` if you have previously called `QTSession.open()`. This is called in the `destroy()` method.

`QTCanvas`, as we've seen, is a display space into which QuickTime can draw and receive events. `QTCanvas` provides the output destination for QuickTime drawing. You set up a `QTCanvas` to display its content at its original size or smaller and centered in the space given to the `QTCanvas` when the applet is laid out. The `QTCanvas` is initialized to display its client up to as large as that client's initial size. And 0.5F flags are used to position the canvas at the center of the space allocated to it by its parent container's layout manager:

```
setLayout (new BorderLayout());
myQTCanvas = new QTCanvas (QTCanvas.kInitialSize, 0.5F, 0.5F);
add (myQTCanvas, "Center");
```

You need to set the client as a `Drawable` object that can display into the canvas. The QuickTime logo is displayed when there is no movie to display. Thus, `ImageDrawer` is set up as the initial client of `QTCanvas`.

```
myQTContent = ImageDrawer.getQTLogo();
```

You enter the URL to a QuickTime movie to be displayed in a text field:

```
final TextField urlTextField = new TextField (
                            "file:///... Enter an URL to a movie",
                            30);
```

You set the font and font size in the text field for the URL. The initial string is displayed in the text field. You add an `ActionListener` so that the events taking place on the text field are captured and executed. `tf.getText()` returns the URL that the user has entered:

```
urlTextField.setFont (new Font ("Dialog", Font.PLAIN, 10));
urlTextField.setEditable (true);
urlTextField.addActionListener (new ActionListener () {
    TextField tf = urlTextField;

    public void actionPerformed (ActionEvent ae) {
        myQTContent = QTFactory.makeDrawable (tf.getText());
        myQTCanvas.setClient (myQTContent, true);
    }
});
```

The URL can support the following protocols:

- file—a local file on the user's computer
- HTTP—HyperText Transfer Protocol
- FTP—File Transfer Protocol
- RTSP—Real Time Streaming Protocol

The URL can also point to any media file (movies, images, and so on) that QuickTime can present. The single `makeDrawable()` method will return the appropriate QuickTime object to present the specified media. Once created, this QuickTime object is set as the client of the `QTCanvas` and can then be viewed and / or played by the user.

Note that no error handling is done in the code in Example 1.5.

EXAMPLE 1.5 *QTStreamingApplet.java*

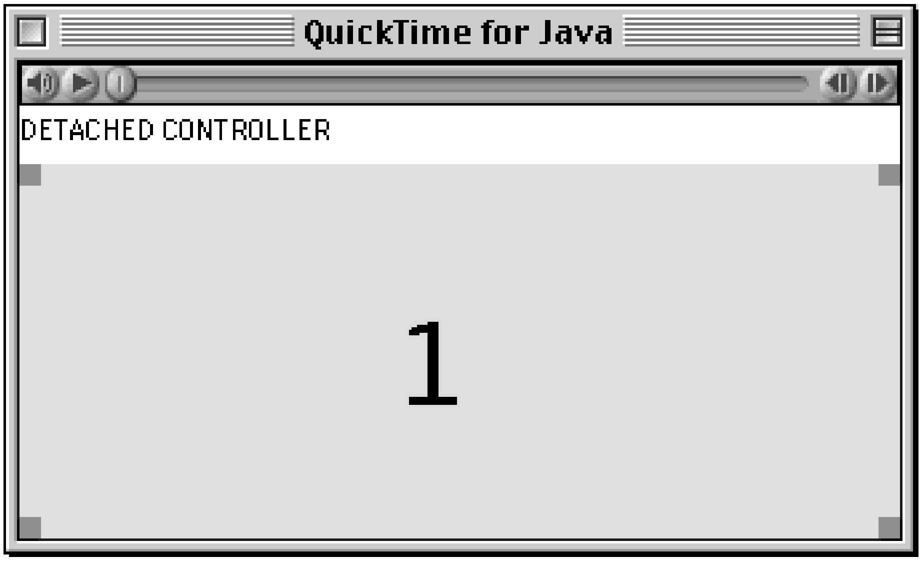
```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

import quicktime.*;
import quicktime.io.QTFile;

import quicktime.app.QTFactory;
import quicktime.app.display.*;
import quicktime.app.image.ImageDrawer;

public class QTStreamingApplet extends Applet {
    private Drawable myQTContent;
    private QTCanvas myQTCanvas;

    public void init () {
        try {
            QTSession.open();
            setLayout (new BorderLayout());
            myQTCanvas = new QTCanvas (QTCanvas.kInitialSize, 0.5F, 0.5F);
            add (myQTCanvas, "Center");

            myQTContent = ImageDrawer.getQTLogo();

            final TextField urlTextField = new TextField ("Enter URL to movie here",
                                                                30);
            urlTextField.setFont (new Font ("Dialog", Font.PLAIN, 10));
            urlTextField.setEditable (true);
            urlTextField.addActionListener (new ActionListener () {
                TextField tf = urlTextField;

                public void actionPerformed (ActionEvent ae) {
                    if (myQTCanvas != null) {
                        try {
                            myQTContent = QTFactory.makeDrawable (tf.getText());
                            myQTCanvas.setClient (myQTContent, true);
                        } catch (QTException e) {
                            e.printStackTrace();
                        }
                    }
                }
            });
            add (urlTextField, "South");
        } catch (QTException qtE) {
            throw new RuntimeException (qtE.getMessage());
        }
    }

    public void start () {
        try {
            if (myQTCanvas != null)
                myQTCanvas.setClient (myQTContent, true);
        } catch (QTException e) {
            e.printStackTrace();
        }
    }

    public void stop () {
        if (myQTCanvas != null)
            myQTCanvas.removeClient();
    }

    public void destroy () {
        QTSession.close();
    }
}
```

USING THE DETACHED CONTROLLER

The code sample in this section shows how to select and then play a QuickTime movie with its controller detached. The media required is a QuickTime movie of the user's choice.

You are prompted to select a movie file. If you make the selection, a window is constructed and the movie and its controller are presented, as shown in Figure 1.4.

FIGURE 1.4 *The detached controller*



The movie and its controller are displayed separately in the same window, with the controller at the top and the actual movie at the bottom. They are separated by a label. Though the movie and its controller are shown here in the same window, they could also be displayed in different windows, even on different monitors.

You use a `QTPlayer` object to play the movie in its canvas, and you can use a `MoviePlayer` object to present the movie.

```
void setControllerCanvas(Movie mMovie) throws QTException {
    QTCanvas controllerCanvas = new QTCanvas();

    MovieController mController = new MovieController(mMovie,
                                              mcScaleMovieToFit);

    mController.setAttached(false);

    QTPlayer qtPlayer = new QTPlayer(mController);
```

```
        add(controllerCanvas, "North");
        controllerCanvas.setClient(qtPlayer, true);
    }
```

You can attach or detach the movie from its controller using this method:

```
    mController.setAttached(false);
```

Once the controller is a client of its canvas, if the movie is reattached to the controller, you need to notify the canvas of this change in the client's display characteristics.

Now you create the canvas for the detached movie:

```
    void setMovieCanvas(Movie mMovie) throws QTException{
        QTCanvas movieCanvas = new QTCanvas();

        MoviePlayer mPlayer = new MoviePlayer(mMovie);

        add(movieCanvas, "South");
        movieCanvas.setClient(mPlayer, true);
    }
```

The following snippet shows the code that assembles the `DetachedController` window. Resizing the window resizes the movie and the width of the controller but not its height.

```
    DetachedController(Movie mMovie) throws QTException {
        super ("QT in Java");

        setControllerCanvas(mMovie);
        setMovieCanvas(mMovie);

        add (new Label("DETACHED CONTROLLER"), "Center");
        .
        .
        .
    }
```

CONVERTING TO FULL SCREEN

The `FullScreen` class provides the capability for converting a screen to full-screen mode and back to normal mode. The QuickTime for Java API allows you to put the specified screen into full screen mode and then use a Java window to fill the screen.

To do this, you use the `FullScreenWindow` class, which is a subclass of the `java.awt.Window` object. The `FullScreenWindow` class internally manages a `FullScreen` object, and when the `show()` method is called, it puts the screen into full-screen mode and fills up the screen with an `awt.Window`. This is very useful because you can get the complete functionality of using a Java AWT Window but in full-screen mode.

The movie is created in a similar fashion as the `QTStreamingApplet`. The program also creates a menu that allows the user to select a movie and once opened provides a **Present Movie** menu item to present the movie in full screen mode.

The full code listing is shown in Example 1.6. You present the movie in full screen mode and use the current screen resolution and current movie. The `QTCanvas` is created using the performance resize flag. This ensures that the movie is displayed at its original size or at a multiple of 2:

```
FullScreenWindow w = new FullScreenWindow(new FullScreen(),
                                        myPlayMovie);
MoviePlayer mp = new MoviePlayer (myPlayMovie.getMovie());
QTCanvas c = new QTCanvas (QTCanvas.kPerformanceResize, 0.5F, 0.5F);
w.add (c);
w.setBackground (Color.black);
```

You remove the movie from its current `QTCanvas` and put the movie into the new canvas of the `FullScreenWindow`. You do this because a `QT-Drawable` can only draw to a single destination `QDGraphics`:

```
myPlayMovie.getCanvas().removeClient();
c.setClient (mp, false);
```

HideFSWindow is a MouseListener. A MouseListener is installed on both the QTCanvas and the Window. The window is then shown, which will put the window into full-screen mode:

```
w.show();
HideFSWindow hw = new HideFSWindow (w, myPlayMovie, c);
w.addMouseListener (hw);
c.addMouseListener (hw);
```

As MoviePlayer object is used to present the movie, this shows just the movie and not a controller. So finally, you start the movie playing:

```
mp.setRate (1);
```

When the user presses the mouse, the movie is restored to its previous QTCanvas and the full-screen window is hidden:

```
public void mousePressed (MouseEvent me) {
    try {
        c.removeClient();
        pm.getCanvas().setClient (pm.getPlayer(), false);
    } catch (QTException e) {
        e.printStackTrace();
    } finally {
         w.hide();
    }
}
```

The user must explicitly press the mouse to hide the FullScreenWindow. However, your application could also define an ExtremeCallBack that would automatically hide the FullScreenWindow when the movie is finished playing. Callbacks are discussed in the next section.

Example 1.6 *FileMenu.java*

```
presentMovieMenuItem.addActionListener (new ActionListener () {
    public void actionPerformed(ActionEvent event) {
        try {
            if (myPlayMovie.getPlayer() == null) return;

        FullScreenWindow w = new FullScreenWindow(new FullScreen(), myPlayMovie);
        MoviePlayer mp = new MoviePlayer (myPlayMovie.getMovie());
        QTCanvas c = new QTCanvas (QTCanvas.kPerformanceResize, 0.5F, 0.5F);
        w.add (c);
        w.setBackground (Color.black);

        myPlayMovie.getCanvas().removeClient();
        c.setClient (mp, false);

        w.show();
        HideFSWindow hw = new HideFSWindow (w, myPlayMovie, c);
        w.addMouseListener (hw);
        c.addMouseListener (hw);

        mp.setRate (1);
    } catch (QTException err) {
        err.printStackTrace();
    }
}

static class HideFSWindow extends MouseAdapter {
    HideFSWindow (FullScreenWindow w, PlayMovie pm, QTCanvas c) {
        this.w = w;
        this.pm = pm;
        this.c = c;
    }

    private FullScreenWindow w;
    private PlayMovie pm;
    private QTCanvas c;


    public void mousePressed (MouseEvent me) {
        try {
            c.removeClient();
            pm.getCanvas().setClient (pm.getPlayer(), false);
        } catch (QTException e) {
            e.printStackTrace();
        } finally {
            w.hide();
        }
    }
}
```

USING MOVIE CALLBACKS

This section explains how to display a QuickTime movie within a window and add callbacks. The callbacks are QuickTime calling back into Java through the movie controller, movie, and QuickTime VR APIs.

Callbacks can be used by an application to perform its own tasks when certain conditions occur within QuickTime itself. The callbacks used in the `MovieCallbacks` program are invoked when some condition having to do with the presentation of a movie is changed:

- In the movie, the `DrawingComplete` procedure is used to notify the Java program whenever QuickTime draws to the screen.
- `ActionFilter` procedures are used. This subclass overrides those actions that pass on no parameters and a float parameter.
- The movie contains QuickTime VR content, and a number of QTVR callbacks are installed for panning and tilting, for hot spots, and for entering and leaving nodes.

The `QTCallBack`, `ActionFilter`, or `DrawingComplete` callbacks are invoked through a direct or indirect call of the QuickTime `MoviesTask` function.

Many of the callback methods in QuickTime in Java are required to execute in place, in that QuickTime requires a result code in order to proceed. These callbacks provide meaningful feedback when their `execute()` method returns. The subclasses of `QTCallBack`, however, can execute asynchronously, in which case QuickTime does not require a result code in order to proceed. This is also true of any of the `execute()` methods with no return value.

The program just prints out details about the callback when it is invoked. The QuickTime API documentation provides examples and discussions on the usage of these.

To set up a movie drawing callback:

```
static class MovieDrawing implements MovieDrawingComplete {
    public short execute (Movie m) {
        System.out.println ("drawing:" + m);
```

```
                        return 0;
                    }
                }
```

To set up an action filter:

```
static class PMFilter extends ActionFilter {
    public boolean execute (MovieController mc, int action) {
        System.out.println (mc + "," + "action:" + action);
        return false;
    }

    public boolean execute (MovieController mc, int action, float value) {
        System.out.println (mc + "," + "action:" + action + ",value=" + value);
        return false;
    }
}
```

The following code sets up callbacks for QuickTime VR content:

```
    Track t = m.getQTVRTrack (1);
    if (t != null) {
        QTVRInstance vr = new QTVRInstance (t, mc);
        vr.setEnteringNodeProc (new EnteringNode(), 0);
        vr.setLeavingNodeProc (new LeavingNode(), 0);
        vr.setMouseOverHotSpotProc (new HotSpot(), 0);
        Interceptor ip = new Interceptor();
        vr.installInterceptProc (QTVRConstants.kQTVRSetPanAngleSelector, ip, 0);
        vr.installInterceptProc (QTVRConstants.kQTVRSetTiltAngleSelector, ip, 0);
        vr.installInterceptProc (QTVRConstants.kQTVRSetFieldOfViewSelector, ip, 0);
        vr.installInterceptProc (QTVRConstants.kQTVRSetViewCenterSelector, ip, 0);
        vr.installInterceptProc (QTVRConstants.kQTVRTriggerHotSpotSelector, ip, 0);
        vr.installInterceptProc (QTVRConstants.kQTVRGetHotSpotTypeSelector, ip, 0);
    }

static class EnteringNode implements QTVREnteringNode {
    public short execute (QTVRInstance vr, int nodeID) {
        System.out.println (vr + ",entering:" + nodeID);
        return 0;
    }
}
```

```
static class LeavingNode implements QTVRLeavingNode {
    public short execute (QTVRInstance vr, int fromNodeID, int toNodeID, boolean[]
                        cancel) {
        System.out.println (vr + ",leaving:" + fromNodeID + ",entering:" +
                            toNodeID);
            // cancel[0] = true;
        return 0;
    }
}

static class HotSpot implements QTVRMouseOverHotSpot {
    public short execute (QTVRInstance vr, int hotSpotID, int flags) {
        System.out.println (vr + ",hotSpot:" + hotSpotID + ",flags=" + flags);
        return 0;
    }
}

static class Interceptor implements QTVRInterceptor {
    public boolean execute (QTVRInstance vr, QTVRInterceptRecord qtvrMsg) {
        System.out.println (vr + "," + qtvrMsg);
        return false;
    }
}
```

The following code shows how to install a `QTRuntimeException` handler:

```
QTRuntimeException.registerHandler (new Handler());
```

Runtime exceptions can be thrown by many methods and are thrown where the method does not explicitly declare that it throws an exception. Using the `QTRuntimeHandler` allows your application to examine the exception and determine if it can be ignored or recovered from.

An example of this is the `paint()` method of a Java component. Its signature is

```
public void paint (Graphics g);
```

It is declared not to throw any exceptions. However, the `QTDrawable redraw()` or `setDisplayBounds()` calls (which are both invoked on a `QTCanvas` client) are defined to throw QTExceptions.

If the `QTCanvas` client does throw an exception in this case, it passes it off to a `QTRuntimeHandler` if the application has registered one, with details about the cause of the exception. The application can then either throw the exception or rectify the situation. If no runtime exception handler is registered, the exception is thrown and caught by the Java VM thread itself.

MEDIA AND PRESENTERS

QuickTime understands media through the pairing of media data and a metadata object that describes the format and characteristics of that media data. This media data can exist either on some external storage device or in memory. The media data can further be stored in the movie itself or can exist in other files, either local or remote. The meta-data object that describes the format of the media data is known as a `SampleDescription`. There are various extensions to the base-level `SampleDescription` that are customized to contain information about specific media types (e.g., `ImageDescription` for image data).

This section discusses the following topics:

- retrieving media data and QuickTime's use of this complex and powerful capability, particularly with the introduction of multimedia streaming
- using presenters to express an object that renders media data loaded into or residing in memory
- the `ImagePresenter` class, which is the principal class in the QuickTime for Java API that presents such image data

MEDIA DATA AND MOVIES

With the introduction of QuickTime streaming, QuickTime has extended its ability to retrieve data. Traditionally, media data had to exist as local data. With streaming, media data can now exist on remote servers and retrieved using the HTTP or FTP Internet protocols, as well as the more traditional file protocol of previous releases. The location of media is contained in a structure called a `DataRef`. This tells QuickTime where media data is and how to retrieve it.

QuickTime streaming also adds support for broadcast media data, using the RTSP protocol. Media data of this format is delivered through network protocols, typically sourced through the broadcast of either live or stored video. It requires the creation of streaming data handlers to deal with the mechanics of retrieving media data using this protocol.

The tracks in a single movie can have data in different locations. For example, one track's media data might be contained in the movie itself; a second track's media data might exist in another local file; and a third track's media data might exist in a remote file and retrieved through the FTP protocol. In these two cases, the movie references the media data that is stored elsewhere. A fourth track might retrieve broadcasted media data through a network using the RTSP protocol.

Your application is free to mix and match these data references for tracks' media data as appropriate. QuickTime presents a complex and powerful capability to deal with media data; a complexity that provides the user of QuickTime with a powerful tool in both the development and delivery of media content.

Dealing With Media

Despite the complexity of the data retrieval semantics of QuickTime, the process of dealing with media in QuickTime is quite simple. In the `CreateMovie` sample code, the movie's data is constructed by inserting the media sample data and its description into the media object and then inserting the media into the track.

The following is a printout of the `ImageDescription` for the last frame of the image data that is added to the movie. As you can see, the

`ImageDescription` describes the format, the size, and the dimensions of the image itself:

```
quicktime.std.image.ImageDescription[
        cType=rle ,
        temporalQuality=512,
        spatialQulity=512,
        width=330,
        height=140,
        dataSize=0,
        frameCount=1,
        name=Animation,
        depth=32]
```

The following is the `SoundDescription` of the sound track added in this sample code. It describes the sample, sample rate, size, and its format:

```
quicktime.std.movies.media.SoundDescription[
        format=twos,
        numChannels=1,
        sampleSize=8,
        sampleRate=22050.0]
```

If the data is not local to the movie itself, one inserts a sample description (as previously to describe the data) and a `DataRef` that describes to QuickTime both the location of the data and the means it should use to retrieve the data when it is required. Once assembled, QuickTime handles all of the mechanics of retrieving and displaying the media at runtime.

When media is displayed, the media handlers use the various rendering services of QuickTime. These rendering services are based on the same data model, in that the sample description is used to both describe the media and to instantiate the appropriate component responsible for rendering data of that specific format. These rendering components are also available to the application itself outside of movie playback.

For example, the `DSequence` object is used to render image data to a destination `QDGraphics` and used throughout QuickTime to render visual media. Your application can also use the `DSequence` object directly by providing both the image data and an `ImageDescription` that describes the format and other characteristics of this data and, of course, a destination `QDGraphics` where the data should be drawn. Like QuickTime itself, your application can apply matrix transformations, graphics modes, and clipping regions that should be applied in this rendering process.

Similar processes are used and available for all of the other media types (sound, music, and so on) that QuickTime supports, with each of these media types providing their own extended sample descriptions, media handlers, and rendering services.

The ImageSpec Interface

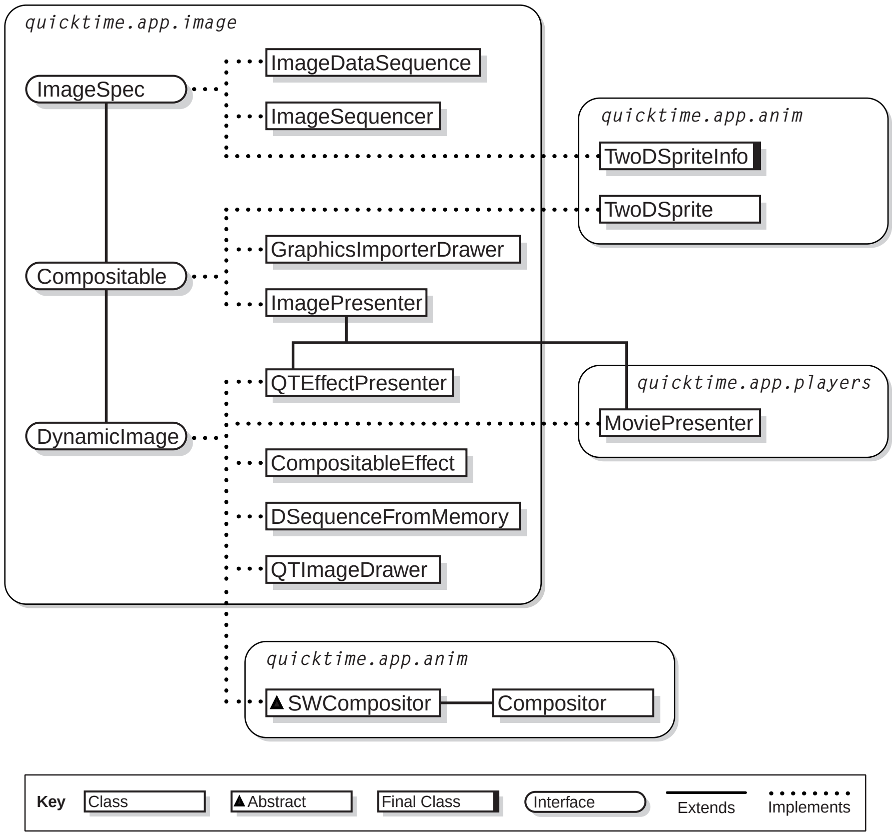
The `ImageSpec` interface expresses the close relationship between image data and an `ImageDescription` that describes it. Figure 1.5 illustrates all of the `ImageSpec`-derived classes in the QuickTime for Java API.

The image data itself is represented by an object that implements the `EncodedImage` interface. This interface allows image data to be stored in either raw memory, accessed using pointers, or in a Java `int` or `byte` array. Any object that implements the `ImageSpec` interface can have its image rendered by either the `ImagePresenter` to a destination `QDGraphics` or by a `TwoDSprite` to its container `SWCompositor`.

The `ImageSpec` interface expresses the commonality of QuickTime's media model, specifically with regard to image data, and unifies the many possible constructions and imaging services that QuickTime provides.

If your application requires a particular format for generating image data, then it can implement the `ImageSpec` interface and thus have QuickTime use this custom class wherever the QuickTime for Java API uses existing `ImageSpec` objects.

FIGURE 1.5 *Image implementations*



The Compositable and DynamicImage Interfaces

The `Compositable` and `DynamicImage` interfaces extend the `ImageSpec` interface. The `Compositable` interface captures the ability of image data to have a graphics mode applied to it when it is rendered. Graphics modes include rendering effects such as transparency, where any pixels of a particular color in the image data won't be drawn, and blending, where all of the drawn colors of an image are blended with a blend color to alter the rendered image.

The `DynamicImage` interface extends `Compositable` and expresses the fact that some pixel data may change. This interface is used by the `Com-`

positor, as a TwoDSprite must invalidate its Sprite if the pixel data changes.

QUICKTIME FOR JAVA PRESENTERS

When QuickTime plays back a movie, it does not generally read all of its media data into memory but rather reads chunks of data as required. A movie can be constructed that requires the media data to be loaded into memory or other parameters that will require more memory to be used but will generally improve the quality of the rendered movie. The QuickTime API documentation covers these customizations that a movie's author can make.

Due to the usefulness of, and in some cases requirement for, loading media data into memory, QuickTime for Java provides *presenters*. A presenter is an object that renders media data that is loaded or resides in memory. A presenter also uses a QuickTime service to render this media data.

QuickTime for Java ships with an ImagePresenter for rendering image data. The ImagePresenter class implements the QTDrawable interface. The ImagePresenter uses a DSequence to perform the rendering of the image data. It is the primary object that is used in QuickTime for Java to render image data. The TwoDSprite is also a presenter that presents image data loaded into memory. However, its role is specific to the membership of its Sprite in a SpriteWorld (which is represented in QuickTime for Java by the SWCompositor).

Though only an ImagePresenter is provided in this release, a similar design strategy could be employed with other media types. For example, let's consider the music media type. MusicMedia is rendered in QuickTime by the TunePlayer class. A MusicPresenter class could be created that used the TunePlayer to render the MusicData. A MusicSpec interface could be described that returns a MusicDescription and the raw MusicData of the tune events.

The ImagePresenter Class

The `ImagePresenter` manages the varying state and conditions of use of QuickTime's `DSequence` renderer. It is a useful abstraction because it hides such details from the user, creating a robust and reusable class.

The `ImagePresenter` is able to render its data faster than its corollary `GraphicsImporterDrawer`, which also implements the `QTDrawable` interface, for two reasons. First, the data is kept in memory and so it is quicker to read. (The `GraphicsImporterDrawer` reads its image data from its `DataRef`, typically a file.) Second, the image data of an `ImagePresenter` can be (and often is) kept in a format that is optimized for rendering or decompression. However, a `GraphicsImporterDrawer` will use less memory and is often more than sufficient for presenting an image where no demanding rendering tasks are required, such as constant, time-sensitive redrawing.

An `ImagePresenter` object can be created from many sources. For example, from a file:

```
ImagePresenter myImage = ImagePresenter.fromFile(imageFile);
```

In this case, the `ImagePresenter` will create a `GraphicsImporter` to read and load into memory the image data from the file. It will then decompress the image if necessary to a format optimized for rendering.

You can also create an `ImagePresenter` from generated image data and an `ImageDescription` that describes it:

```
int width = 100;
int height = 100;

IntEncodedImage myImageData = new IntEncodedImage (width * height);
//...fill in pixel values using standard java ARGB ordering

ImageDescription myDescription =
        ImageDescription.getJavaDefaultPixelDescription (width,
                                                         height);
myDescription.setDataSize (myImageData.getSize());

ImagePresenter myPresenter = ImagePresenter.fromQTImage(myImageData,
                                                myDescription);
```

You can also create an ImagePresenter object from an onscreen or offscreen QDGraphics that you have previously drawn into:

```
QDGraphics gWorld = ....
Rect rect = ....
int colorDepth = ....
int quality = ....
int codecType = ....
CodecComponent codec = ....

ImagePresenter myIP = ImagePresenter.fromGWorld (myGWorld,
                                myGWorld.getBounds(),
                                myGWorld.getPixMap().getPixelSize(),
                                myDerivedCompressionQuality,
                                myDerivedCompressionType,
                                myDerivedCodecComponent);
```

Subclasses of ImagePresenter

There are two basic subclasses of ImagePresenter, which inherit from and build on the features of ImagePresenter: MoviePresenter and QTEffectPresenter. Both of these subclasses work in a similar fashion. They render their target objects (a movie or an effect) into an offscreen QDGraphics object. The pixel data from this QDGraphics is then set as the EncodedImage data of the superclass. An ImageDescription is created that describes this raw pixel data and this description is given to the superclass. Thus, retrieving the image data (getImage()) returns the raw pixel data that the movie has drawn into.

The ImagePresenter superclass then blits this raw pixel data to its destination QDGraphics. If this presenter is added as a Sprite, the raw pixel data becomes that sprite's image data.

MoviePresenter takes a Movie object and implements the same interfaces as MoviePlayer, but it has the extra capability of being a presenter and having a graphics mode set for its overall appearance.

QTEffectPresenter is similar to MoviePresenter but takes a QTEffect object. You could use this presenter to capture the results of applying a filter to a source image. You could then discard the filter object and just present the resultant image. You can also use this presenter to present a character in a sprite animation that could transition on and off the stage.

IMAGING AND EFFECTS

The QuickTime for Java Application Framework provides a number of classes that handle the presentation of images within a Java display space. These classes provide utility methods that you can use to set transparent colors in images, apply visual effects, or create objects for handling sequences of images (such as slide shows). Other methods enable you to create `QTDrawable` objects that read image data from a file or load the data into memory.

This section shows you how to

- draw an image file using the `GraphicsImporterDrawer`
- create a `java.awt.Image` out of an image in QuickTime's format
- use the `QTImageDrawer` class to render Java-painted content in a QuickTime graphics space
- take advantage of QuickTime's visual effects architecture to apply transitions between two images

DRAWING AN IMAGE FILE

Example 1.7 shows how to import and draw an image from a file. This program works with the `GraphicsImporterDrawer` object to import and display a variety of image file formats. The media required for this sample code is any image file that can be imported using the `GraphicsImporterDrawer`.

The `quicktime.app.image` package has two primary image display classes: `GraphicsImporterDrawer` and `ImagePresenter`. These classes implement both `ImageSpec` and `QTDrawable` interfaces. `GraphicsImporterDrawer` uses `GraphicsImporter` to read, decompress, and display image files.

EXAMPLE 1.7 *ImageFileDemo.java*

```
import quicktime.*;
import quicktime.std.StdQTConstants;
import quicktime.std.image.GraphicsImporter;
import quicktime.io.QTFile;

import quicktime.app.display.QTCanvas;
import quicktime.app.image.*;

public class ImageFileDemo extends Frame implements StdQTConstants {

    public static void main (String args[]) {
        try {
            QTSession.open();

            int[] fileTypes = { kQTFileTypeGIF, kQTFileTypeJPEG, kQTFileTypePicture };
            QTFile qtf = QTFile.standardGetFilePreview (fileTypes);

            ImageFileDemo ifd = new ImageFileDemo (qtf);
            ifd.pack();
            ifd.show();
            ifd.toFront();
        } catch (QTException e) {
            if (e.errorCode() != Errors.userCanceledErr)
                e.printStackTrace();
            QTSession.close();
        }
    }

    ImageFileDemo (QTFile qtf) throws QTException {
        super (qtf.getName());

        QTCanvas canv = new QTCanvas();
        add (canv, "Center");

        GraphicsImporterDrawer myImageFile = new GraphicsImporterDrawer (qtf);
        canv.setClient (myImageFile, true);

        addWindowListener(new WindowAdapter () {
            public void windowClosing (WindowEvent e) {
                QTSession.close();
                dispose();
            }

            public void windowClosed (WindowEvent e) {
                System.exit(0);
            }
        });
    }
}
```

QUICKTIME TO JAVA IMAGING

The code in this section shows how to create a `java.awt.Image` from a QuickTime source. The `QTImageProducer` is used to produce this image's pixel data from the original QuickTime source.

The QuickTime image could come from any one of the following sources:

- an image file in a format that Java doesn't directly support but QuickTime does
- recording the drawing actions of a `QDGraphics` into a `Pict`. This can be written out to a file or presented by an `ImagePresenter` class to the `QTImageProducer` directly.
- using the services of QuickTime's sequence grabber component. A sequence grabber can be used to capture just one individual frame from a video source.

In the sample code, the user is prompted to open an image file from one of 20 or more formats that QuickTime's `GraphicsImporter` can import.

The program then uses the `QTImageProducer` to create a `java.awt.Image` that is then drawn in the `paint()` method of the `Frame`.

You prompt the user to select an image file and import that image into QuickTime. You then create a `GraphicsImporterDrawer` that uses the `GraphicsImporter` to draw. This object produces pixels for the `QTImageProducer`:

```
QTFile imageFile =
        QTFile.standardGetFilePreview(QTFile.kStandardQTFileTypes);
GraphicsImporter myGraphicsImporter = new GraphicsImporter
                                                  (imageFile);
GraphicsImporterDrawer myDrawer = new GraphicsImporterDrawer
                                              (myGraphicsImporter);
```

You create a `java.awt.Image` from the pixels supplied to it by the `QTImageProducer`:

```
QDRect r = myDrawer.getDisplayBounds();
imageSize = new Dimension (r.getWidth(), r.getHeight());
QTImageProducer qtProducer = new QTImageProducer (myDrawer,
                                                      imageSize);
javaImage = Toolkit.getDefaultToolkit().createImage(qtProducer);
```

Note that to do Java drawing, your application uses the `java.awt.Graphics.drawImage(...)` calls, which must be defined in a `paint()` method. When using a `QTCanvas` client, this detail is taken care of by the `QTCanvas` itself by establishing the `QTCanvas` `QTDrawable` client relationship, as the above code demonstrates.

In the `paint()` method of the frame, the image that we produced in the above code is drawn, using this `drawImage` call. This method will correctly resize the image to the size of the Frame.

```
public void paint (Graphics g) {
    Insets i = getInsets();
    Dimension d = getSize();
    int width = d.width - i.left - i.right;
    int height = d.height - i.top - i.bottom;
    g.drawImage (javaImage, i.left, i.top, width, height, this);
}
```

This returns the size of the source image, so the `pack()` will correctly resize the frame:

```
public Dimension getPreferredSize () {
    return imageSize;
}
```

IMAGE PRODUCING

The code in this section shows how to display any QuickTime drawing object using Java's `ImageProducer` model.

The program works with the `QTImageProducer` and with Swing components. The `QTImageProducer` in this case is responsible for getting the QuickTime movie and producing the pixels for a `java.awt.Image`. It draws the movie into its own `QDGraphics` world and then feeds the pixels to any `java.image.ImageConsumer` objects that are registered with it in a format that they are able to deal with. The Swing buttons control the movie playback.

The Swing `JComponent` is the `ImageConsumer` of the `QTImageProducer`. It will automatically repaint itself if the `QTImageProducer` source is multiframed media, such as a movie. This is a feature of the `ImageProducer` model of the Java API.

Placing a QuickTime movie within a Swing `JComponent` requires the usage of Java's image producing API. Swing is a framework that uses lightweight components. As such, a heavyweight component, such as a `QTCanvas`, is generally not added to the lightweight components. To put QuickTime content into a lightweight component a `java.awt.Image` is used to capture the pixel data generated by QuickTime. As the previous example showed, the Java image producing API is used, with the `QTImageProducer` implementing the `ImageProducer` interface used for this purpose. More information about Swing and the image producing API can be found in Java documentation and at the Java web site: `http://java.sun.com/`.

We open the movie and set looping of its time base. We make a `MoviePlayer` out of the Movie to pass to the `QTImageProducer`, which takes any `QTDrawable` object as a source of pixel data. We pass in the original size of the movie to the `QTImageProducer`, which will notify the producer how big a `QDGraphics` it should create.

Once the `QTImageProducer` is made, we need to redraw it when each frame of the movie is drawn. To optimize this process, we install a `MovieDrawingComplete` callback. This callback notifies us when the movie draws a complete frame, and in the `execute()` method we redraw the

QTImageProducer. The QTImageProducer redraw() method will pass on the changed pixel data to the registered ImageConsumer objects. The ImageConsumer is registered with the QTImageProducer when we create the IPJComponent, as shown in the code below.

```
OpenMovieFile openMovieFile = OpenMovieFile.asRead(movFile);
Movie m = Movie.fromFile (openMovieFile);
m.getTimeBase().setFlags (loopTimeBase);
MoviePlayer moviePlayer = new MoviePlayer (m);
QDRect r = moviePlayer.getDisplayBounds();
Dimension d = new Dimension (r.getWidth(), r.getHeight());
ip = new QTImageProducer (moviePlayer, d);

    //this tells us that the movie has redrawn and
    //we use this to redraw the QTImageProducer - which will
    //supply more pixel data to its registered consumers
m.setDrawingCompleteProc (movieDrawingCallWhenChanged, this);

IPJComponent canv = new IPJComponent (d, ip);
pan.add("Center", canv);
```

The following is the execute() method of the MovieDrawingComplete interface. This execute() method is invoked whenever the movie draws. We use the updateConsumers() method of the QTImageProducer as the movie has already drawn when this callback is executed, so we only need to notify the image consumers and give them the new pixel data. If the movie hadn't drawn, then the QTImageProducer redraw() method would be called. This both redraws the QTDrawable source and updates the image consumers' pixel data:

```
public int execute (Movie m) {
    try {
        ip.updateConsumers (null);
    } catch (QTException e) {
        return e.errorCode();
    }
    return 0;
}
```

The following shows the construction of the IPJComponent. This JComponent is a Swing component and its paint() method will draw the image that is the consumer of the QTImageProducer. The constructor creates a java.awt.Image, the createImage() method establishing the QTImageProducer as the producer of pixel data for this Image. The prepareImage() call establishes the IPJComponent as the ImageObserver of this process. The paint() method uses the supplied java.awt.Graphic drawImage() call to draw the Image, also passing the IPJComponent as the ImageObserver. Each time the QTImageProducer is redrawn, it notifies its image consumers that it has more pixel data. This will, in turn, notify the ImageObserver, which is the IPJComponent, that it should repaint itself. The paint() method is consequently called and drawImage() will draw the new pixel data to the screen.

```
static class IPJComponent extends JComponent {
    IPJComponent (Dimension prefSize, QTImageProducer ip) {
        pSize = prefSize;
        im = createImage (ip);
        prepareImage (im, this);
    }

    private Dimension pSize;
    private Image im;

    public Dimension getPreferredSize() {
        return pSize;
    }

    public void paint (Graphics g) {
        g.drawImage (im, 0, 0, pSize.width, pSize.height, this);
    }
    // stops flicker as we have no background color to erase
    public void update (Graphics g) {
        paint (g);
    }
}
```

USING THE QTIMAGEDRAWER CLASS

The QTImageDrawer class enables standard Java drawing commands and graphics objects to have their content rendered by QuickTime within a QuickTime graphics space (QDGraphics). Standard AWT paint() calls can be made on the `Graphics` object supplied to the `paint()` call of the `Paintable` interface attached to the `QTImageDrawer`. Because the `QTImage-Drawer` implements the `ImageSpec` interface, you can add it to a `quick-time.app.display.Compositor` object and thus draw into the same display space as QuickTime-generated content. It also implements the `QTDrawable` interface and can thus also be a client of a `QTCanvas`.

A `QTImageDrawer` object is used to convert the results of painting into a `java.awt.Image` object using a `java.awt.Graphics` object to a format that QuickTime can render.

When the `java.awt.Image` object is created and painted into (using the `QTImageDrawer Paintable` object) the `QTImageDrawer` uses Java's `Pix-elGrabber` to grab the raw pixel values that resulted from this painting. It then constructs an `ImageDescription` object that describes to Quick-Time the format, width, and height of this pixel data, and an `ImagePre-senter` is used internally to render the image using QuickTime imaging services.

If the `QTImageDrawer` is used to grab the results of a single paint, then it can make optimizations on the amount of memory that it uses. A single frame is indicated by the `kSingleFrame` flag when constructing it. In this case, your application can grab the resultant image data (`get-Image()`) and its description (`getDescription()`) and then discard the `QTImageDrawer` object completely. This is preferable in many situations where both performance and memory usage are a consideration; performance is enhanced because once the image is retrieved in this way and can be drawn very efficiently by QuickTime, it is already in a format that is suitable. The extra memory and overhead of Java's imaging model and the translation to a format QuickTime can render can thus be discarded.

The `paint()` method of the `QTImageDrawer Paintable` object returns an array of rectangles that tells the drawer which areas of its drawing area were drawn. This optimizes the amount of copying and can greatly increase performance when composited.

VISUAL EFFECTS

QuickTime provides a wide range of sophisticated visual effects. QuickTime's visual effects architecture can be applied to visual media in a movie or can be used in real-time—that is, applied to image sources that are not contained in a movie. The code listed here and the QuickTime for Java classes are designed with this second usage in mind. Example code to apply effects to a movie can be found in the QuickTime SDK that is included with the book *QuickTime for Java.*

The `QTEffect` class forms the base class for visual effects. Depending on the effect itself, you can apply visual effects over a length of time or once only. Effects can be applied to no source images (i.e., the effect acts on a background `QDGraphics` object), such as the ripple effect. An effect can also be applied to a single source image (typically, color correction or embossing) using the `QTFilter` class. Finally, an effect can be applied to two sources (such as wipes and fades), using the `QTTransition` class that transitions from the source to the destination image.

The code snippets in this section show how to use QuickTime's visual effects architecture. The effects in the code are applied, in real time, to two images. The rendering of the transitions is controlled by the user settings in the window's control panel.

Changes to any one of these three fields of the transition can affect the values of the other. Thus, changing the duration or frames per second (fps) will alter how many frames are rendered. The code updates the values in the other fields to reflect these dependencies.

You set the duration of the transition if it is running using the time mode of the `QTTransition` class:

```
timeField.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent event) {
        int output = ...//get value from field
        transition.setTime (output);
    }
});
```

You set the number of frames the transition will take to render:

```
frameField.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent event) {
        int output = ...//get value from field
        transition.setFrames (output);
    }
});
```

You set the number of frames per second that the transition should be rendered in:

```
fpsField.addActionListener (new ActionListener () {
     public void actionPerformed (ActionEvent event) {
        int output = ...//get value from field
        transition.setFramesPerSecond (output);
    }
});
```

You use this code snippet to make the transition run based on the settings. If the transition is profiling, some statistics about the transition are printed.

```
runEffectButton.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent event) {
        try {
            transition.doTransition();
            if (transition.isProfiled()) {
                String profileString = "Transition Profile:"
                    + "requestedDuration:" + transition.getTime()
                    + ",actualDuration:" + transition.profileDuration()
                    + ",requestedFrames:" + transition.getFrames()
                    + ",framesRendered=" + transition.profileFramesRendered()
                    + ",averageRenderTimePerFrame="
                    + (transition.profileDuration()
                            / transition.profileFramesRendered());
                System.out.println (profileString);
            }

        } catch (QTException e) {
            if (e.errorCode() != userCanceledErr)
                e.printStackTrace();
            }
        }
    });
```

You use this code snippet to show QuickTime's Choose Effects dialog box:

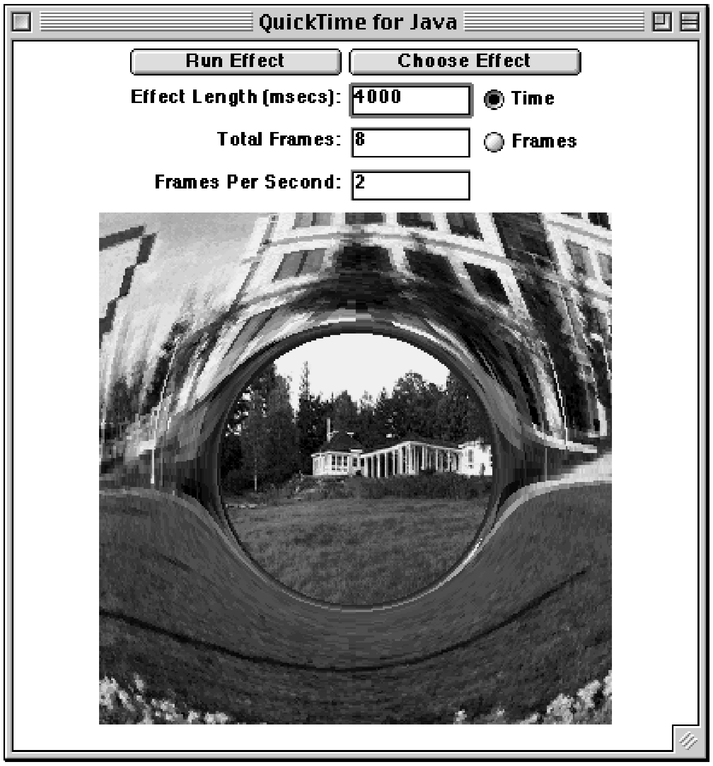
```
chooseEffectButton.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent event) {
        PlayQTEffectApp.showDialog (transition);
    }
});
```

The two buttons in the following code snippet can change the mode of the transition. If the transition renders using time mode, it will potentially drop frames in order to render itself as close to the specified duration as possible. If the transition is set to `doTime(false)`, it will render the currently specified number of frames as quickly as possible. This mode varies considerably from computer to computer based on the processing and video capabilities of the runtime environment.

```
frameButton.addItemListener (new ItemListener () {
    public void itemStateChanged (ItemEvent event) {
        transition.doTime (false);
    }
});
timeButton.addItemListener (new ItemListener () {
    public void itemStateChanged (ItemEvent event) {
        transition.doTime (true);
    }
});
```

Figure 1.6 shows an explode effect from the program.

FIGURE 1.6 *The explode effect frome the QT Effects program*



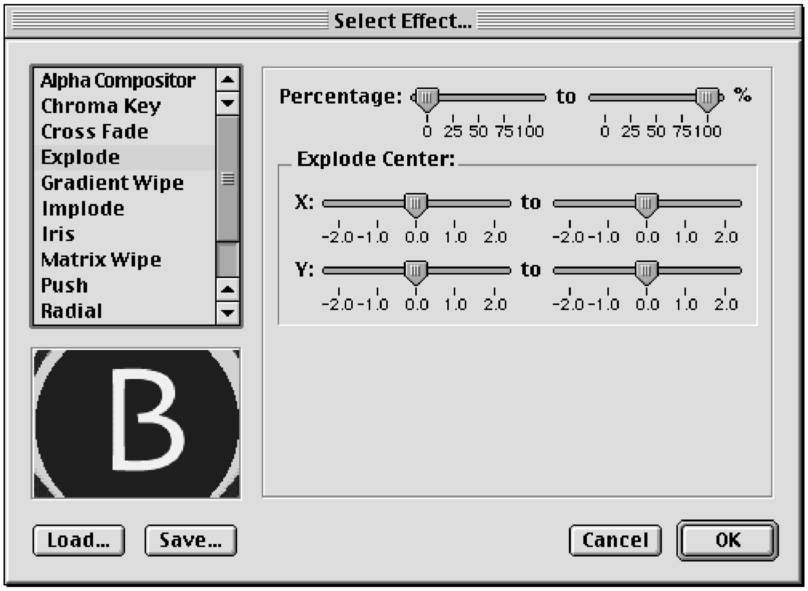
Effect Length (msecs): 4000 [✓ Time]

Total Frames: 8 Frames

Frames Per Second: 2

To make an effect, you need to either build one in code or the user can choose an effect from QuickTime's Choose Effects dialog box, as shown in Figure 1.7.

FIGURE 1.7 *QuickTime's Choose Effects dialog box*



You set up an atom container to use a SMPTE effect. Using SMPTE effects, you set the `WhatAtom` to SMPTE, ensure the endian order is big-endian, and use SMPTE effect number 74:

```
public AtomContainer createSMPTEEffect (int effectType,
                                              int effectNumber) {
    AtomContainer effectSample = new AtomContainer();

// We are using SMPTE Effects so set the what atom to smpt
    effectSample.insertChild (new Atom(kParentAtomIsContainer),
        kEffectWhatAtom,
        1,
        0,
        EndianOrder.flipNativeToBigEndian32(kWipeTransitionType));

// We are using SMPTE effect number 74  - start at 0%, stop at 100%
    effectSample.insertChild (new Atom(kParentAtomIsContainer),
                effectType,
                1,
                0,
                EndianOrder.flipNativeToBigEndian32(effectNumber));

    return effectSample;
}
```

In this example, the dialog box is configured to show only effects that expect two sources, but it can be configured to show effects applied or requiring only a single source.

```
    public static void showDialog (QTEffect ef) throws QTException {
        AtomContainer effectSample = ParameterDialog.showParameterDialog
                                        (new EffectsList (2, 2, 0), 0);
        ef.setEffect (effectSample);
    }
```

You use the `doTransition()` method, which is part of the `QTTransion` class in the `quicktime.app.image` package, to set parameters to control the rendering behavior of the effect.

```
    QTTransition ef = new QTTransition ();
    ef.setTime (800);
    ef.setSourceImage (sourceImage);
    ef.setDestinationImage (destImage);
    ef.setEffect (createSMPTEEffect (kEffectWipe,
                                        kRandomWipeTransitionType));
```

Your application can also directly control the rendering of each frame of an effect through setting the frame directly and redrawing the effect.

ANIMATION AND COMPOSITING

With animation, you are working essentially with a time-based script of composited images, which simply means that the images will change over time. With compositing, you are using images from different sources and layering them and generating a composited image.

USING THE SWCOMPOSITOR

The SWCompositor class provides the capability to compose a complex image from disparate image sources and then treat the result as a single image, which is presented to the user. It also provides a time base that controls the rendering cycle and allows your application to attach time-based behaviors or actions.

The SWCompositor uses the QuickTime SpriteWorld internally to perform its compositing tasks and its TimeBase for its timing services. All of the actual drawing of the members of a SWCompositor is handled through the interaction between the SpriteWorld and Sprite classes of QuickTime.

The SpriteWorld itself is wrapped by the SWCompositor class, and to represent the Sprite class it uses the TwoDSprite. The TwoDSprite is a presenter, that is it presents image information. The presentation of image information within the context of the SWCompositor SpriteWorld is determined by the matrix, graphics mode, layer, and visibility of the Sprite object.

To create a Sprite, you need a valid SpriteWorld. To create a SpriteWorld, you need a valid QDGraphics destination. Depending on whether a SWCompositor is visible, you may or may not have a valid destination QDGraphics. The interaction between the SWCompositor and its TwoDSprite presenters handles the saving and creating of SpriteWorld and Sprite objects—your application does not need to deal specifically with this issue.

The SWCompositor object presents the functionality of the SpriteWorld within the context of the QTDisplaySpace interface. The TwoDSprite object wraps the QuickTime Sprite object, giving it a Transformable interface so that its visual characteristics can have matrix transformations applied to them. It also implements the Compositable interface, as graphics modes can be applied when a sprite's image data is rendered. You can save the current display state of an individual sprite and recreate a sprite from the information you saved; the TwoDSpriteInfo object is a helper class created for this purpose.

The SWCompositor supports two important characteristics of members. If the member implements a DynamicImage, this indicates to the Compositor that the image being presented by the member is apt to change. For example, the member is a MoviePresenter, in which case the member needs to create a special object—an Invalidator—that will invalidate the sprite that is presenting the member, so that the composite cycle will redraw that sprite, and you will see the changing image data of a movie as it plays back. This is discussed in more detail below.

A further service that the SWCompositor renders to its new members is the handling of members that implement the Notifier interface. If a member implements this interface, the Compositor establishes the connection between the Notifier (the new member) and its NotifyListener (the new member's TwoDSprite). When the new member's image data is complete, it can then automatically notify its registered NotifyListener—its TwoDSprite. This is used with QTImageDrawer members where the QTImageDrawer won't have valid image data until the Java offscreen image is created. It could also be used with images that may reside on a remote server, or interactively where the user may choose or draw an image that is part of an animation.

The SWController

The SWController deals with SpriteWorldHitTest calls on the SpriteWorld that are contained by the SWCompositor's subclasses. By default, it performs hit-testing on the actual sprite image itself. However, your application can set the hit-test flags to support any mode of hit-testing appropriate. As a subclass of the MouseController, it will return any member of the SWCompositor that is hit only if isWholespace() is true. If isWholespace() is false, the hit sprite must be a member of the SWController itself.

The Compositor

The Compositor class is a subclass of SWCompositor, with its primary role being the relaxation of membership requirements of its members. The member object of a Compositor is only required to implement the ImageSpec interface, in which case, the Compositor creates the TwoDSprite that presents that image data in its display space. If a TwoDSprite itself is added to the Compositor, it is added directly as a member, since TwoDSprite also implements the ImageSpec interface.

If the new member implements the Transformable interface, then its current matrix will be set on its TwoDSprite presenter. If the new member implements the Compositable interface, then its current GraphicsMode is applied to its TwoDSprite presenter. These actions simplify the adding of members to the Compositor. However, once added, your application will need to deal directly with the member's TwoDSprite presenter to alter the Matrix or GraphicsMode of the sprite.

Note that most of the functionality of the Compositor is within the SWCompositor abstract class. It is only in the membership requirements that the Compositor specializes the SWCompositor—specifically in the creation, removal, and retrieval of a member's TwoDSprite presenter.

COMPOSITED EFFECTS

The example code in this section, which is available in the SDK, shows the use of a Compositor to create a composited image out of effects, sprites, and Java text. It also shows you how to use this as a backdrop for a QuickTime movie that draws directly to the screen. You construct a composited image containing the layering of an image file, a ripple effect, an animation and some Java text. Over this, you place a movie and its movie controller, which is drawn in front of the composited image.

The Compositor is used to combine multiple-image objects, including a CompositableEffect (the ripple effect), into a single image that is then blitted on screen. Both the Compositor and the QTPlayer are added as members of the top-level DirectGroup client of a QTCanvas. The code also shows you how timing hierarchies can be established when spaces are contained within each other.

The media required for this sample code include

- ShipX.pct files (where X is a number indicative of a frame order)
- Water.pct

- jumps.mov

The compositing services of the Compositor (that is, transparent drawing, alpha blending, and so on) are not available with a DirectGroup. A DirectGroup does allow for its member objects to be layered. Thus, the movie can draw in front of the Compositor unheeded. It shows the embedding of a Compositor space in a parent Compositor, and then the embedding of this Compositor in a parent DirectGroup display space.

We create the parent Compositor that will contain the background image, ripple effect, Java text, and the spaceship compositor. The spaceship compositor is created similarly to preceding examples:

```
Dimension d = new Dimension (kWidth, kHeight);
QDRect r = new QDRect(d);
QDGraphics gw = new QDGraphics (r);
Compositor comp = new Compositor (gw, QDColor.green,
                                        new QDGraphics (r), 10, 1);
```

We add the background image, setting it to the same size as the Compositor. The ripple effect will ripple the pixels that this image draws:

```
QTFile bgFile = new QTFile(
                    QTFactory.findAbsolutePath("pics/water.pct"));
GraphicsImporterDrawer if1 = new GraphicsImporterDrawer (bgFile);
if1.setDisplayBounds (r);
ImagePresenter background =
                ImagePresenter.fromGraphicsImporterDrawer (if1);
comp.addMember (background, Layerable.kBackMostLayer);
```

The ripple effect is layered to apply on top of the background image, and its bounds are set to only the top part of the compositor's display bounds. A ripple effect is applied only to what is behind it, not to sprites or text drawn in front of it. The QuickTime ripple effect codec works by moving pixels around on the destination QDGraphics that it is set to. By placing it in a Compositor, your application can control which part of an image it ripples—in this case, the water picture that is behind it.

```
CompositableEffect e = new CompositableEffect ();
AtomContainer effectSample = new AtomContainer();
effectSample.insertChild (new Atom(kParentAtomIsContainer),
        kEffectWhatAtom,
        1,
        0,
        EndianOrder.flipNativeToBigEndian32(kWaterRippleCodecType));
e.setEffect (effectSample);
e.setDisplayBounds (new QDRect (0, kHeight - 100, kWidth, 100));
comp.addMember (e, 2);
```

We add the contained `Compositor`. Yellow is set as the background color, which is then not drawn, as we set the graphics mode of the `Compositor` to transparent with yellow as the transparent color.

We also add a `Dragger` so that members of this compositor can be dragged around when any modifier key is pressed when the `mousePressed` event is generated. We also add a `Dragger` to the parent `Compositor` so that we can drag any of its top-level members when no modifier keys are pressed:

```
Compositor sh = new Compositor (
                        new QDGraphics (new QDRect(160, 160)),
                        QDColor.yellow, 8, 1);
addSprites (sh);
sh.setLocation (190, 90);
sh.setGraphicsMode (new GraphicsMode (transparent, QDColor.yellow));
sh.getTimer().setRate(1);
sh.addController(new SWController (new Dragger (

MouseResponder.kAnyModifiersMask,

MouseResponder.kAnyModifiers), true));
comp.addMember (sh, 1);

comp.addController(new SWController (
            new Dragger (MouseResponder.kNoModifiersMask), true));
```

You use the `QTImageDrawer` object using the Java-drawing APIs to draw the Java text, which is then given a transparency so that only the text characters themselves are displayed by QuickTime. Note that you set the background color to white, so the Java text appears transparent. White provides a reliable transparent background for different pixel depths.

```
myQTCanvas.setBackground (Color.white);
```

You add the Java text in front of the background image and ripples and set its transparency to the background color of the `QTCanvas`, so that only the text is seen.

```
QTImageDrawer qid = new QTImageDrawer (jt, new Dimension (110, 22),

Redrawable.kSingleFrame);
Paintable jt = new JavaText ();
qid.setGraphicsMode (new GraphicsMode (transparent, QDColor.white));
qid.setLocation (200, 20);
comp.addMember (qid, 1);
```

The code here provides a good demonstration of the timing hierarchy that is built using the display spaces of the `Compositor` and `DirectGroup`.

We make a `DirectGroup` as the top-level container space, adding both the containing `Compositor` as a member of this group and the movie `jumps.mov`. Finally, we set the rates of both the `Compositor` and the `DirectGroup` to 1 so that they are playing when the window is shown:

```
DirectGroup dg = new DirectGroup (d, QDColor.white);
dg.addMember (comp, 2);

QTFile movieFile = new QTFile (
                        QTFactory.findAbsolutePath ("jumps.mov"));
QTDrawable mov = QTFactory.makeDrawable (movieFile);
mov.setDisplayBounds (new QDRect(20, 20, 120, 106));
dg.addMember (mov, 1);

myQTCanvas.setClient (dg, true);

comp.getTimer().setRate(1);
dg.getTimer().setRate(1);
```

The top DirectGroup is the master time base for all of its members; the rate at which its time base is set (the top text box) determines the overall rate of its members. The members can have their own rates that become offset based on the rates of their parent groups. To start the program, you set the top rate to 1.

TRANSITION EFFECTS

We end this document with a program that demonstrates how to use the QuickTime effects architecture and apply it to a character in an animation scene.

The code in this section shows how you can build a transition effect and apply it to a character in a realistic animation of a UFO encounter! In the program, a fading effect is applied to transition to the spaceship image. The image fades in and out as per the rate and the number of frames set for the transition.

It shows the usage of effects and effects presenters. The kCrossFadeTransitionType effect is applied to the source and the destination images, which that makes them fade as per the number of frames set for the transition.

The QTEffectPresenter is used to embed the QTTransition effect and present it to the Compositor, which draws it to the canvas. Note that the QTTransition effect cannot be directly added to the Compositor; instead, it is given to the QTEffectPresenter, which is added to the Compositor. If a filter were applied, it would have the same limitations as a transition when added to a Compositor. It must be added using the QTEffectPresenter.

A ripple effect is applied to the water image (in front of the water image taking up the same location), using the CompositableEffect class. Zero sourced effects, such as the ripple effect, can be added directly to a Compositor.

```
CompositableEffect ce = new CompositableEffect ();
AtomContainer effectSample = new AtomContainer();
effectSample.insertChild (new Atom(kParentAtomIsContainer),
        kEffectWhatAtom,
                    1,
                    0,
                    EndianOrder.flipNativeToBigEndian32(kWaterRippleCodecType));

                ce.setEffect (effectSample);
                ce.setDisplayBounds (new QDRect(0, 220, 300, 80));
                comp.addMember (ce, 3);
```

The `Fader` class is used to create the `QTTransition` and return the `QTEffectPresenter` that will supply the pixel data that becomes the image data for this member's sprite:

```
class Fader implements StdQTConstants {
    Fader() throws Exception {
        File file = QTFactory.findAbsolutePath ("pics/Ship.pct");
        QTFile f = new QTFile (file.getAbsolutePath());

        QDGraphics g = new QDGraphics (new QDRect (78, 29));
        g.setBackColor (QDColor.black);
        g.eraseRect(null);
        ImagePresenter srcImage = ImagePresenter.fromGWorld(g);
        Compositable destImage = new GraphicsImporterDrawer (f);

        ef = new QTTransition ();
        ef.setRedrawing(true);
        ef.setSourceImage (srcImage);
        ef.setDestinationImage (destImage);
        ef.setDisplayBounds (new QDRect(78, 29));
        ef.setEffect (createFadeEffect (kEffectBlendMode, kCrossFadeTransitionType));
        ef.setFrames(60);
        ef.setCurrentFrame(0);
    }

    private QTTransition ef;

    public QTEffectPresenter makePresenter() throws QTException {
        QTEffectPresenter efPresenter = new QTEffectPresenter (ef);
        return efPresenter;
    }

    public QTTransition getTransition () {
        return ef;
    }

    AtomContainer createFadeEffect (int effectType, int effectNumber)
                                                            throws QTException {
        AtomContainer effectSample = new AtomContainer();
        effectSample.insertChild (new Atom(kParentAtomIsContainer),
                        kEffectWhatAtom,
                        1,
                        0,
                        EndianOrder.flipNativeToBigEndian32(kCrossFadeTransitionType));

        effectSample.insertChild (new Atom(kParentAtomIsContainer),
                                effectType,
                                1,
                                0,
                                EndianOrder.flipNativeToBigEndian32(effectNumber));
        return effectSample;
    }
}
```

We then create the `QTEffectPresenter` for the transition and add it as a member of the `Compositor`:

```
Fader fader = new Fader();
QTEffectPresenter efp = fader.makePresenter();
efp.setGraphicsMode (new GraphicsMode (blend, QDColor.gray));
efp.setLocation(80, 80);
comp.addMember (efp, 1);

comp.addController(new TransitionControl (20, 1,
fader.getTransition()));
```

The controller object implements the `TicklishController` and subclasses the `PeriodicAction` class that has a `doAction()` method, which gets invoked on every tickle call.

```
class TransitionControl extends PeriodicAction
                                implements TicklishController {
    TransitionControl (int scale, int period, QTTransition t) {
        super (scale , period);
        this.t = t;
    }
```

The doAction() call is overidden to set the current frame and redraw the TransitionEffect. The source and the destination images of the transition effect are swapped when the number of set frames is reached. The transition's controller then rests for a few seconds before it is awakened again and reapplied. The incoming time values to the doAction method (called by PeriodicAction.tickle()) are used to calculate the rest and transition, ensuring that if the rate of playback changes, the transition controller will react to these changes.

When the transition is quiescent, we set the redrawing state of the QTEffectPresenter to false. This ensures that when the Compositor invalidates this presenter it will not invalidate the sprite, as we are not currently drawing into the QTEffectPresenter. When the transition is being applied, that is when the current frame is set, then the isRedrawing method will return true. The Invalidator for the QTEffectPresenter will then redraw the effect and invalidate its sprite presenter. Thus, this controller is also able to control the redrawing of both itself and its sprite through the use of the redrawing state and ensure that the Compositor only renders the sprite that presents this QTEffectPresenter when it actually changes its pixel data—that is, the image data of the effect's presenter.

```
protected void doAction (float er, int tm) throws QTException {
if (waiting) {
if ((er > 0 && ((startWaitTime + waitForMsecs) <= tm))
|| (er < 0 && ((startWaitTime - waitForMsecs) >= tm))) {

waiting = false;
t.setRedrawing(true);
} else
return;
}
int curr_frm = t.getCurrentFrame();
curr_frm++;
t.setCurrentFrame(curr_frm);
if (curr_frm > t.getFrames()) {
curr_frm = 0;
t.setRedrawing(false);
t.setCurrentFrame(curr_frm);
t.swapImages();
waiting = true;
startWaitTime = tm;
}
```