Estratto della Part 2, Chapter 7, 8 e 9

Java™ Media APIs: Cross-Platform Imaging, Media, and Visualization
(link a Safari bookstore online)

By Alejandro Terrazas, John Ostuni, Michael Barlow

Editor: Sams Publishing

The Java Media Framework

-Dr. Michael "Spike" Barlow

The second major section of the book, Chapters 7, 8 and 9, covers time-based media (that is, video and audio) and the JMF—Java Media Framework, a Java API dedicated to the processing of time-based media.

Fundamentally, the JMF is an extension to Java for handling audio and video (audio and video being the two primary forms of time-based media). More rigorously, the JMF API (Java Media Framework Application Programming Interface) is one of the official Java Optional Packages from Sun Microsystems that extends the functionality of the core Java Platform. Included in 2D Graphics and Imaging on the Java Platform this group of Optional Packages are others that are covered in the book: Java 3D and Java Advanced Imaging (JAI).

The JMF comprises some 200 odd additional classes pertaining to the handling of time-based media. Handling is used in the broadest sense to include playback, capture, processing, and transmission, for either local media or media from a remote site, and as part of either an applet or application. Among the possibilities the API affords are platform (hardware and OS) independent video conferencing, complete audio and video editing suites, empowering the latest mobile computing such as cellular phones and PDAs (Personal Digital Assistants), and when taken in conjunction with the other media APIs, completely integrated multimedia applications written entirely in Java and running on any platform.

JMF Coverage in the Book

The three chapters in this section of the book follow a progression of simple out-of-the-box utilization of the API to sophisticated usage, such as in combination with other specialized features and APIs of Java. Hence, a linear progression through the material is recommended as the default. However, those of you possessing a familiarity with time-based media or parts of the API might want to skip some of the introductory material.

The structure of the three chapters is as follows:

- Chapter 7, "Time-Based Media and the JMF: An Introduction"— The first chapter of the section on the JMF serves as both an introduction to time-based media in general and to the JMF API. Some of the fundamental concepts and issues for both digital audio and video are introduced. Midway through the chapter is an introduction to the JMF API in terms of its features, promise, central concepts, and main classes.
- Chapter 8, "Controlling and Processing Media with JMF"— This chapter serves as the core chapter of the JMF section, covering the key features of the JMF API. The topics covered include managers, data sources and sinks, multiplexing and demultiplexing, codecs, format conversion, effects, and the capture of media from devices.
- Chapter 9, "RTP and Advanced Time-Based Media Topics"— This chapter covers some of the more advanced features of the JMF API. Chief among these topics is the Real-Time Transport Protocol (RTP) support within JMF and the corresponding ability to transmit or receive streaming media such as over the Internet. Also covered are issues such as extending the API and utilizing other APIs in conjunction with JMF.

Obtaining and Installing the JMF

The JMF extends the functionality of the Java platform and is an official Optional Package. As such, it is a free download available from Sun Microsystems' Java site: http://java.sun.com. Following the Products & APIs link will present the browser with a wealth of APIs; among them, the JMF can be found under the Optional Packages heading at the bottom of the page.

Alternatively, and more directly, Sun maintains a central Web page regarding the JMF: http://java.sun.com/products/java-media/jmf/index.html. You should definitely bookmark this URL: It not only has links for downloading the latest version of the JMF, but it also links to documentation and example programs, as well as the latest JMF-related news.

Sun provides several different versions of the JMF for download. These differ in the OS platform they are intended to run on. The current version, as of the time of writing, is v2.1.1a. At the previously mentioned central site, Sun provides links for a cross-platform Java version, a Windows Performance Pack, and a Solaris SPARC Performance Pack. A link is also provided to Blackdown's JMF implementation for Linux. All versions require JDK 1.1.6 or later for full functionality. Those of you who want to obtain the JMF without possessing the JDK should download and install that first.

Although the cross-platform version is pure byte code and will run on any machine supporting Java, it is recommended that you download and install the OS specific versions that matches your OS. This is because these implementations have been optimized with native code where appropriate, and hence should run faster than the cross-platform version. Thus, those of you who are running Windows 95, 98, or NT should download the Windows Performance Pack, those of you who are running on one of Sun's UNIX machines should download the Solaris SPARC Performance Pack, and those of you who are under Linux should download Blackdown's version of the JMF for Linux. Those of you who are not employing any of these (for example, on a Macintosh) should download the cross-platform version.

Sun provides detailed and specific instructions regarding the download and installation process. Those instructions are tailored to the specific version downloaded. Following the download links will take the browser through those instructions. Thus, specific download and installation instructions are not repeated here. Installation of any version of the JMF is quite simple, consisting of self-installing executables or the equivalent. However, those of you who want detailed installation instructions can find them at http://java.sun.com/products/java-media/jmf/2.1.1/setup.html.

Following the installation process, you should check that the JMF is available for usage. One means of checking this is to attempt to run the JMStudio demonstration program that is provided as part of the JMF. Discussed further in Chapter 7, JMStudio is a powerful application that demonstrates many of the capabilities of the JMF, such as playback, capture, and processing. Running JMStudio is as simple as typing `java JMStudio` at your command prompt. If the JMF installed properly, a small JMStudio window will pop up from which the various functions can be selected.

An alternative means of checking whether the JMF installed correctly is to point your browser at http://java.sun.com/products/java-media/jmf/2.1.1/jmfdiagnostics.html, Sun's JMF diagnostic page. As part of the installation process, the JMF is made available to your Web browser so that JMF-based applets can be run. The preceding URL tests this feature. Similarly, http://java.sun.com/products/java-media/jmf/2.1.1/samples/index.html contains JMF-based applets that will play movie trailers, providing

that the JMF is installed on your machine, and it is arguably a more exciting means of testing the functionality of the newly installed JMF.

Additional JMF-Related Resources

A number of resources pertaining to the JMF are available on the Web. Sun's central JMF page, http://java.sun.com/products/java-media/jmf/index.html, acts as a clearing house for many, but not all, of these additional resources.

Two key resources that anyone undertaking serious JMF programming should possess are the API (class) documentation and the Programmer's Guide from Sun. The API documentation is a class-by-class description of the API. The Programmer's Guide is a comprehensive introduction to the API from its authors. Both these documents can be browsed online or downloaded to a user's machine. Both the online and downloadable version of these documents can be found linked from Sun's central JMF page.

Other resources at Sun's site include excellent sample programs, source code for the JMF itself and JMStudio, as well as user guides for JMStudio and JMFRegistry.

Sun maintains a free mailing list: jmf-interest, for those wanting to discuss the JMF. The details for subscribing to and posting to the list can be found at the following URL: http://java.sun.com/products/java-media/jmf/support.html. (It is also linked from Sun's main JMF site.) Joining the list is highly recommended for those undertaking programming in the JMF—the list is a small but helpful community with relatively low traffic (typically fewer than a dozen messages a day) with Sun engineers periodically monitoring and posting on the list. The list's past archives, found at http://archives.java.sun.com/archives/jmf-interest.html, contain a wealth of information.

Finally, it is worth noting that although the JMF comes with many audio and video codecs (the compression schemes that are used for audio and video and which dictate its format), further codecs can be installed. These additional codecs then expand the functionality of the JMF—JMF is then able to handle media of that format. Two popular codecs of note, MPEG-4 and DivX, can be incorporated into the JMF in this manner. IBM, through its AlphaWorks division, has provided an implementation of MPEG-4 for the JMF at http://www.alphaworks.ibm.com/tech/mpeg-4. DivX support, currently a popular video format on the Internet because its high compression and good visual quality, can be incorporated into the JMF by downloading the DivX codec from the DivX home page: http://www.divx.com/.

Chapter 7. Time-Based Media and the JMF: An Introduction

IN THIS CHAPTER

- Time-Based Media
- Processing Media
- Audio Primer
- Video Primer
- What Is the JMF?
- Java and Time-Based Media: A Short History
- Media Formats and Content Types Supported by JMF
- Levels of Usage of the JMF API
- Programming Paradigms When Using JMF
- Structure of the API
- Time—A Central Concept
- Bare Bones Player Applet—A First Applet Using JMF

This section of the book covers time-based media (that is, video and audio) and the JMF (Java Media Framework)—a Java API dedicated to the processing of time-based media.

The section is broken into three chapters—this one, Chapter 8, "Processing Media with JMF," and Chapter 9, "RTP and Advanced JMF Topics"—that follow a progression of simple out of the box utilizations of the API to sophisticated usage such as in combination with other specialized features and APIs of Java. Hence, a linear progression through the material is recommended as the default. However, those of you possessing familiarity with time-based media or parts of the API might want to skip some of the introductory material.

In particular the structure of the three chapters is as follows:

Chapter 7, "Time-Based Media and the JMF: An Introduction," serves as both an introduction to time-based media in general and to the JMF API. In particular, some of the fundamental concepts and issues for both digital audio and video are introduced. Midway through the chapter, that is followed by an introduction to the JMF API in terms of its features, promise, central concepts, and main classes.

Chapter 8, "Processing Media with JMF," serves as the core chapter of Part II, "Time-Based Media: The Java Media Framework and Java Sound," covering the key features of the JMF API. The topics include managers, data sources and sinks, multiplexing and demultiplexing, codecs, format conversion, effects, and capture of media from devices.

Chapter 9, "RTP and Advanced JMF Topics," covers some of the more advanced features of the JMF API. Chief among these covered topics is the RTP (Real-Time Transport Protocol) support within JMF and the corresponding ability to transmit or receive streaming media such as over the Internet. Also covered are issues such as extending the API and utilizing other APIs in conjunction with JMF.

This chapter serves as a general overview of time-based media followed by an introduction to the JMF API. The area of time-based media is not only a broad and involved topic, but also one that is continually changing as new approaches, formats, and standards are introduced. The first section introduces time-based media as well as some of its key concepts and considerations—particularly those

with a direct bearing on the JMF API. Those of you who want a more detailed coverage of time-based media are directed to the plenitude of material in book form as well as on the Web.

Midway through the chapter, the JMF API is introduced. The main features and potential of the API are illustrated. Next is a more detailed introduction that covers the main classes of the API, programming approaches to using the API, and the central concept of time before concluding with an example applet that shows how simple using the API can be.

Time-Based Media

The term media possesses a number of meanings and connotations to most people: from modern print and broadcast press to the inclusion of terms such as multimedia.

Time-based media, from the perspective of the JMF API and Java, is broadly defined as any data that varies in a meaningful manner with respect to time.

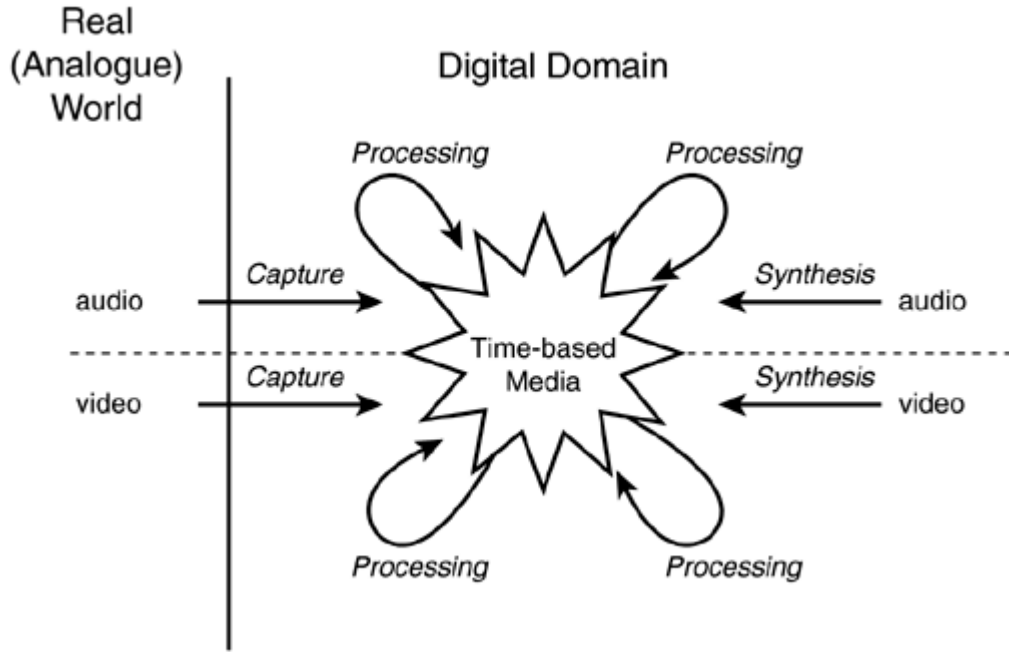
Implicitly, two further properties are understood to be possessed by time-based media:

- The data is intended for presentation (perhaps not immediately but at some, possibly future, stage) to a human being. With current technologies, that is understood to be through vision or hearing.
- The data is in a digital format. Typically this involves capturing (digitizing) analogue (real-world) data such as from a microphone. Alternatively, the media might inherently be digital such as speech synthesized by a computer.

Thus, time-based media is generally understood to be video, audio, or a combination of both.

Both categories, audio and video, can be subdivided into naturally captured media (for example, microphone or video camera) and synthetically produced media (for example, 3D animation sequences). However, the boundaries between natural and synthesized media aren't clear, becoming less so daily. Even naturally captured media is subject to post-capture processing such as to enhance or add features that weren't in the original. (The movie industry practice of blue screening to merge matte painted backgrounds with film of actors recorded in a studio is a classic example of this.) Figure 7.1 shows this breakdown of the types of time-based media. Indeed, the blurring of the distinction between natural and synthetic media is almost a direct result of the fact that after the media is digitized (if that was even necessary), it can be processed in any manner imaginable. In the most general sense of processing that includes capturing, presenting, transmitting, as well as converting, compressing, and so on, controlling and processing time-based media is exactly what the JMF is intended for.

Figure 7.1. Origins and types of time-based media.



Typical examples of time-based media include TV broadcasts, the data captured from a microphone or video camera attached to a PC, an MP3 file on a hard disk, a video conference across the Internet, and webcasts.

Throughout this and the following chapters, time-based might be dropped from the term media. Such usage indicates time-based media as previously defined, and not any other meaning ascribable to the word media.

Time-Based Media on a Computer

In the past decade, there has been a revolution in terms of access to and in particular generation of time-based media, most notably digital media. Previously, only large production companies in the form of movie studios, TV and radio stations, and other such specialists, were capable of producing high-quality media. Correspondingly, dedicated devices or venues were required to present such media for an audience: a TV set for TV broadcasts, a radio for radio broadcasts, and a cinema for movies.

Changes in computing, both in terms of technology and penetration of daily life, have fundamentally altered the paradigm from production only by specialists and presentation only on dedicated devices to production by anyone (with a computer) and a very versatile presentation option (the computer).

The technological advances that have driven this change include the ongoing and significant increases in both processor power and storage capacity of the PC, together with improvements in networking and telecommunication. These hardware advances have gone hand-in-hand with software developments that have made it possible to harness the greater power afforded by the average PC. Advances in processor power have meant that the various compressed formats used to store video and audio could

be processed in real time. Thus, a PC could be used to present and even save the media. Advances in storage capacity, as first witnessed by the advent of the CD-ROM as a standard peripheral, increasingly large (in terms of storage capacity) hard disks, and more recently the DVD (digital versatile disc) have meant that inherently large media files can be stored on a PC. Correspondingly, network advances have meant that it is now possible, and commonplace, to access media stored or generated remotely.

Socially, the computer has transitioned from being seen as a specialist device for computation and calculation to a general-purpose household item with wide applicability in many areas—not the least of which is communication. This is particularly illustrated by the World Wide Web (WWW) in which users not only see it as commonplace to surf the Net (pulling in content from all over the world), but also increasingly expect or demand that the content be dynamic and entertaining—often with time-based media! The JMF was designed, at least in part, with this purpose in mind, and is centrally placed: Java has been a key-enabler of the Web and in particular Web-interactivity since its earliest days—JMF further enhances the Web support power of Java.

Bandwidth, Compression, and Codecs

Time-based media, in its raw form suitable for presentation through speakers or on a display, is particularly large—high in bandwidth. That poses a particular challenge in the area of storing (for example, on a hard disk) and transmitting (for example, over a modem) media and introduces the idea of compression.

The following sections on audio and video go into further detail, but for a moment consider the size of a typical three-minute audio track on a music CD, bearing in mind that raw video is even more demanding (often about 100 times more).

The raw audio format is known as PCM (Pulse Code Modulation). CD audio is particularly high quality: It covers the entire range of human hearing (which ranges up to about 20KHz: twenty thousand hertz). With such accuracy in representation, most people cannot discern the difference between the original and the stored signal. To achieve that detailed representation, 44,100 samples are taken each second for each of the left and right audio channels. Each sample is 16 bits (two bytes: a range of some 65,536 possible values). That equates to 176,400 bytes or 1,411,200 bits of information per second. For the three-minute piece of music, that equates to 31,752,000 bytes (over 30 megabytes) of information.

A modern PC's hard disk will soon fill with a few hundred such audio files. More significant and sobering is the transfer rate required to stream that audio data across a network so that it can be played in real-time: over one million bits per second. Contrasting that with a 56K modem(peak performance not reaching 56,000 bits per second) that most home users have as their means of connecting to the Internet, it can be seen that compromises are necessary: the required transfer rate exceeds that possible by a factor greater than 20 times.

The need for compression is obvious. Although the particulars of modern compression algorithms are complicated, the fundamentals of all approaches are the same. The media is kept in a compressed format while being stored or transmitted. The media is decompressed only immediately prior to presentation or if required for processing (for example, to add an effect).

The components that perform this task of compression and decompression are known as codecs (COmpression/DECompression) and can work in hardware or software. For each audio and video,

there is a range of codecs that vary in their compression capabilities: the quality of the resulting media, amount of processing required, and the support they receive from the major companies working in the multimedia arena.

Most codecs are lossy, meaning that they don't perfectly preserve the original media: Some quality of the original media is lost when it is compressed and is thereafter unrecoverable. Although this is unfortunate, appropriate design of the codec can result in some or most of the losses not being perceptible to a human audience. Examples of such losses might be the blurring of straight edges (for example, text) in a video image or the addition of a slight buzz to the audio. Regardless of the undesirability of these losses in quality, no known lossless codecs are capable of achieving anywhere near the compression necessary for streaming high quality audio and video over a typical (home) user of today's connection to the wider network.

All codecs employ one or more of the following three general strategies in order to achieve significant compression:

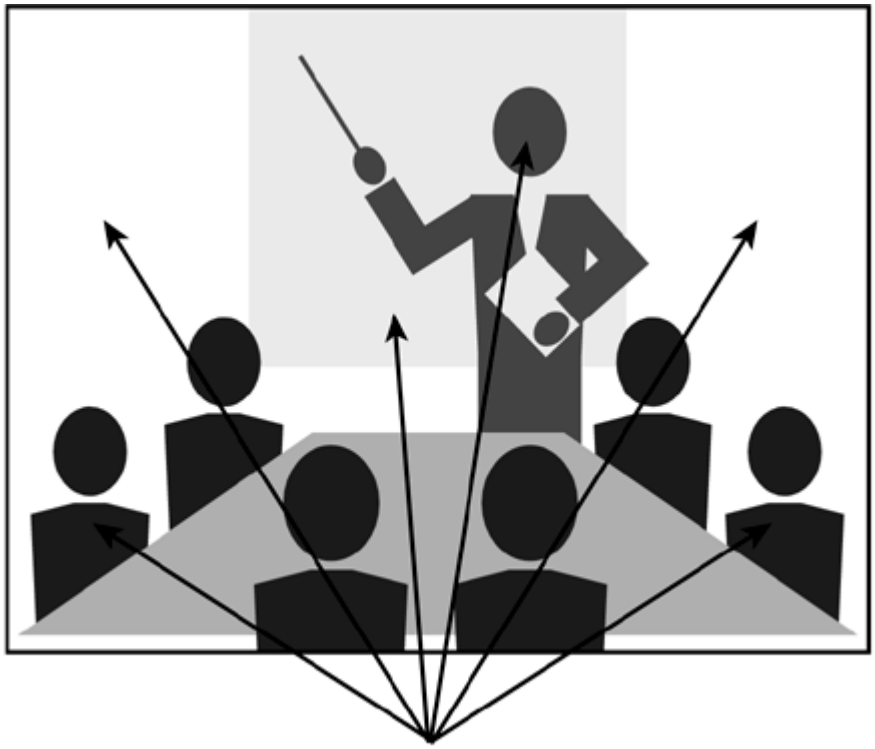
Spatial redundancy— These schemes exploit repetition within the current frame (sample) of data. Although not applicable for audio encoding in which each frame is a single value, significant savings can be made for typical video images. Most images have regions of a solid color—backgrounds such as a blue sky, the beige walls of a house, or individual subject elements such as a white refrigerator or a solid-color shirt. Basically, such schemes can be thought of as recording the recurring color and the region of the image that it ranges over, rather than keeping multiple copies (one for each pixel that composes the solid color block) of the same thing.

Temporal redundancy— These schemes exploit the fact that the difference between successive video frames or successive audio samples is generally small relative to the size of the frame or sample itself). Rather than transmit or store a completely new frame or sample, only the difference from the previous sample needs to be stored or transmitted. For both audio and video, this approach is generally very effective. Although there are instances, such as a new scene in a video, in which that isn't true. A strong example of the benefits of such approaches include video of a news anchorperson: Most of the image is static, and only relatively minor changes occur from frame to frame—the anchorperson's head and facial movements. Even far more dynamic video (for example, a football match) still has considerable static (from frame to frame) regions, and significant savings are still achieved. Similarly, most sound—whether speech, music, or noise—is tightly constrained in a temporal sense. Temporal encoding based schemes pose challenges for non-linear editing. (A frame is defined in terms of its predecessor, but what if that predecessor is removed or, even more challenging, altered?) Such schemes tend to degrade in compression performance and quality over a long period time. For both reasons, these schemes periodically (for example, once per second) transmit a completely new frame (known as a key-frame).

Features of human perception— The human visual and auditory systems have particular idiosyncrasies that might be exploited. These include non-linearity across the spectrum being perceived as well as more complex phenomenon such as masking. Visually, humans distinguish some regions of the color spectrum less keenly, whereas in the auditory domain, human perception strongly emphasizes the lower frequency (deeper) components of a sound at the expense of those higher frequency components. Clever coding schemes can exploit these coarser regions of perception and dedicate fewer resources to their representation. Strategies based on human perception differ fundamentally to the two pervious schemes because they are based on subjective rather than objective measures and results.

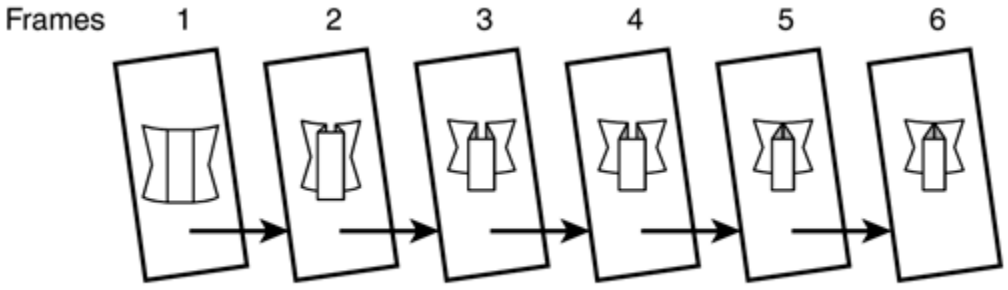
Figure 7.2 shows the concept of spatial compression, whereas Figure 7.3 shows temporal compression. Figure 7.4 shows the non-linearity of human perception in the auditory domain: the range of human hearing (in Hertz on the horizontal axis) is shown against the perceptually critical bands (bark) found through psychoacoustic experiments. Such known relationships can be exploited by audio compression schemes.

Figure 7.2. Spatial compression opportunities.



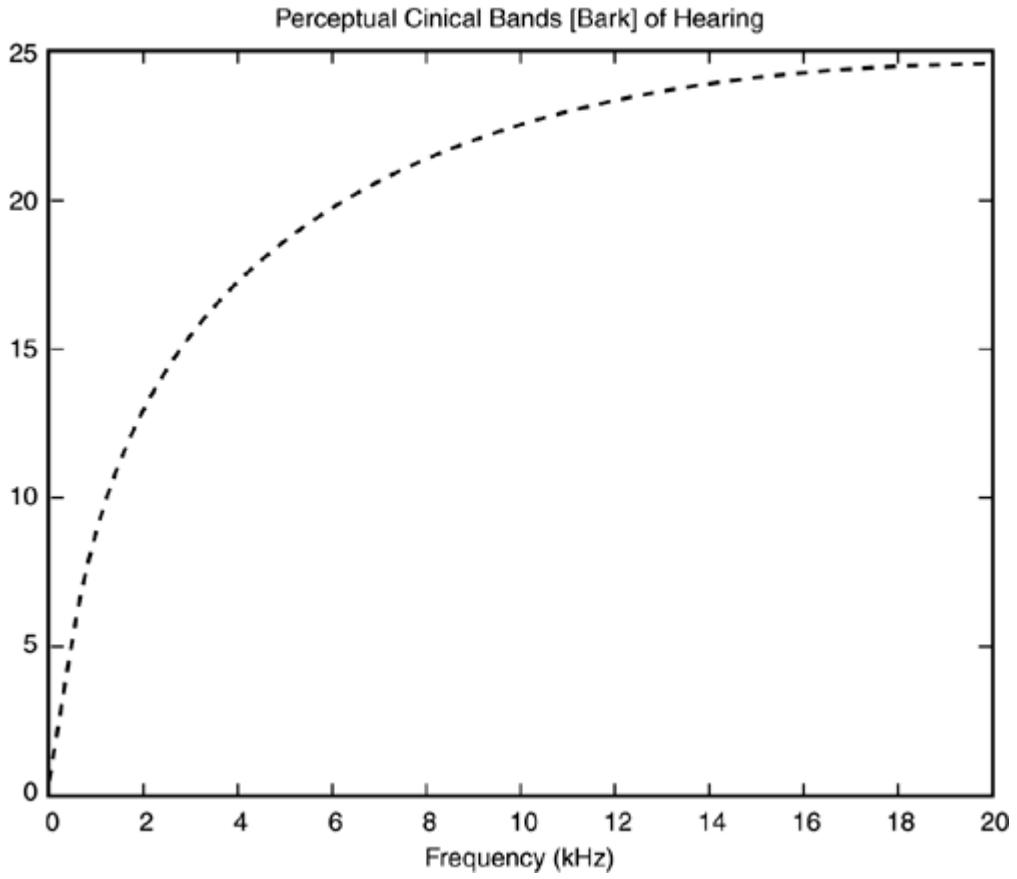
Regions (blocks) of solid color in the image that could be spatially compressed

Figure 7.3. Temporal compression could be used to record only the differences from the previous frame.



Successive frames in an image.

Figure 7.4. Perceptually critical bands (bark) of human hearing matched against the frequency range of human hearing.



Format, Content Type, and Standards

The codec used to encode and decode a media stream defines its format. Thus the format of a media stream describes the actual low-level structure of the stream of data. Examples of formats include cinepak, H.263, and MPEG-1 in the video domain and Mu-Law, ADPCM, and MPEG-1 in the audio domain.

Sitting atop a media's format, and often being confused with it, is known as the content type or sometimes the architecture of the media. The content type serves as a type of super-structure allowing the specification of codecs and other details such as file structure of the total API. Examples of content types include such well-known names as AVI, QuickTime, MPEG, and WAV.

As an illustration of the distinction between media format and content type, it is worth noting that most content types support multiple possible formats. Thus the QuickTime content type can employ Cinepak, H.261, and RGB video formats (among others), whereas the WAV (Wave) content type might be A-law, DVI ADPCM, or U-Law (among others). Hence an alternative model is to see the various content types as media containers; each can hold media in a number of different formats.

An obvious question, given the apparent profusion of formats and content types, is where are the standards? Why are there so many formats and content types, and are they all really necessary?

International standards do exist in the form of the various MPEG versions. (It's currently at three, although the latest version is known as MPEG-4 because no MPEG-3 standard exists.) MPEG stands for the Motion Picture Expert Group and is a joint committee of the ISO (International Standards Organization) and IEC (International Electrotechnical Commission). These standards are of very high quality: well designed and with high compression. However, because of a number of interrelated factors that include commercial interests, differences in technology, historic developments, as well as differing requirements from formats, these standards are yet to dominate the entirety of the audio and video fields.

Perhaps the most important reason that various formats exist is that each is designed with a different purpose in mind. Although some are clearly better than others (particularly older formats) in a number of dimensions, none dominate in all aspects. The most important aspects of differentiation are degree of compression, quality of the resulting media, and processing requirements. These three aspects aren't mutually exclusive, but are competing factors: For instance, higher compressions are likely to require greater processing and result in more loss of quality. Various formats (codecs) weight these factors differently, resulting in formats with diverse strengths and weaknesses. It becomes clear then that there is no single best format; the best can only be defined in terms of the constraints and requirements of a particular application.

On the other hand, the different content types are chiefly attributable to commercial and historical developments. Some content types such as QuickTime and AVI, although now almost cross-platform standards, were traditionally associated with a particular PC platform: the Macintosh in the case of QuickTime and the Windows PC in the case of AVI. The advent of the WWW and more powerful PCs have seen a second generation of content type such as RealMedia (RealAudio/RealVideo), which is specifically targeted at streaming media across the Internet.

Tracks and Multiplexing/Demultiplexing

Time-based media often consists of more than one channel of data. Each of these channels is known as a track. Examples include the left and right channels for traditional stereo audio or the audio and video track on an AVI movie. Recent standards, such as the MPEG-4 content type, support the concept of a multitude of tracks composing a single media object.

Each track within a media object has its own format. For instance, the AVI movie could possess a video track in MJPG (Motion JPEG) format and an audio track in ADPCM (Adaptive Differential Pulse Code Modulation) format. The media object, however, has a single content type (in our example, AVI). Such multitrack media are known as multiplexed.

Creation of multiplexed media involves combining multiple tracks of data, a process known as multiplexing. For instance, the audio track captured from a microphone would be multiplexed with the video track captured from a video camera in order to create a movie object. Similarly, the processing of existing media might result in further multiplexing as additional tracks (for instance, a text track of subtitles for a movie) are added to the media.

The corollary operation of separating individual tracks from a multitrack media object is known as demultiplexing. This is necessary prior to presentation of the media so that each track can have the appropriate codec applied for decompression and the resulting raw media sent to the correct output device (for example, speakers for audio track, display for the video track).

If processing of a media object is required, the appropriate tracks would need to be demultiplexed so that they could be treated in isolation, processed (such as to add an effect), and then multiplexed back into the media object. This processing can also result in the generation of new tracks, which then need to be multiplexed into the media object. An example of this might be adding subtitles to a movie: the audio track is demultiplexed and processed automatically by a speech recognizer to generate a transcription as a new track. That new track is then multiplexed back in with the original video and audio.

Figures 7.5, 7.6 and 7.7 show the various roles of multiplexing and demultiplexing in media creation, processing, and presentation.

Figure 7.5. Role of demultiplexer in playback of media.

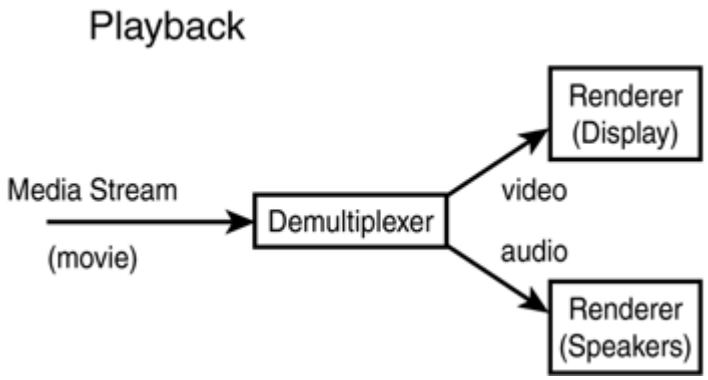


Figure 7.6. Role of multiplexer in capture of media.

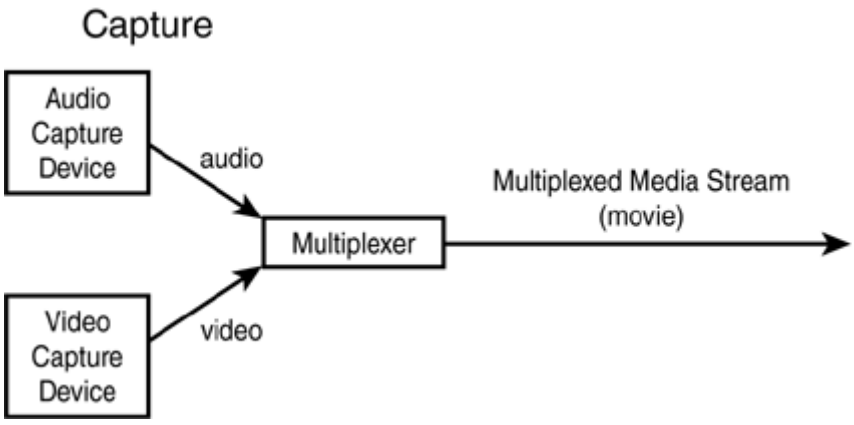
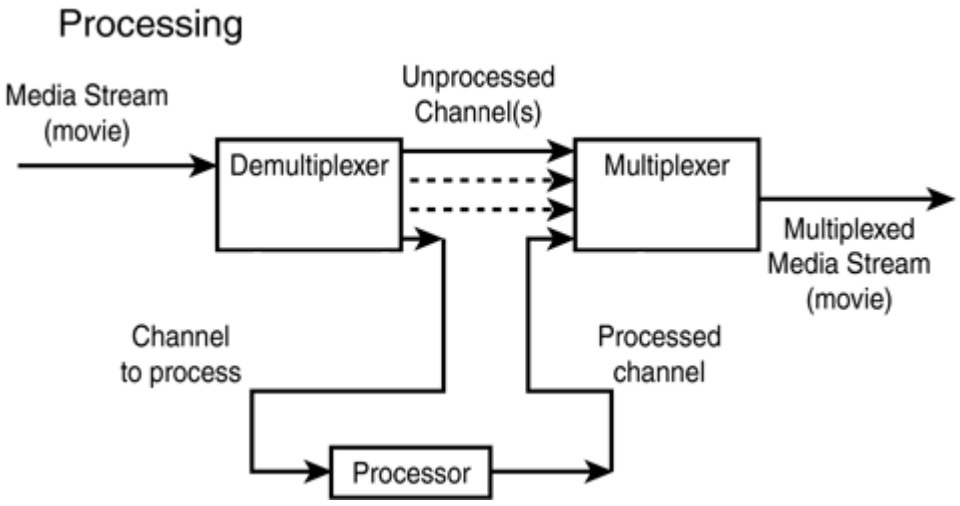


Figure 7.7. Role of demultiplexer and multiplexer in the processing of media.



Streaming

The origins of time-based media on the computer lie in applications where media was stored on devices such as a CD-ROM and played from that local source. These forms of applications are still commonplace and important. They were enabled by emerging technologies, such as higher storage capacity devices in the form of CD-ROMs; similarly, Internet technology (combined with increasing computing power) has led to challenging new areas of application for time-based media.

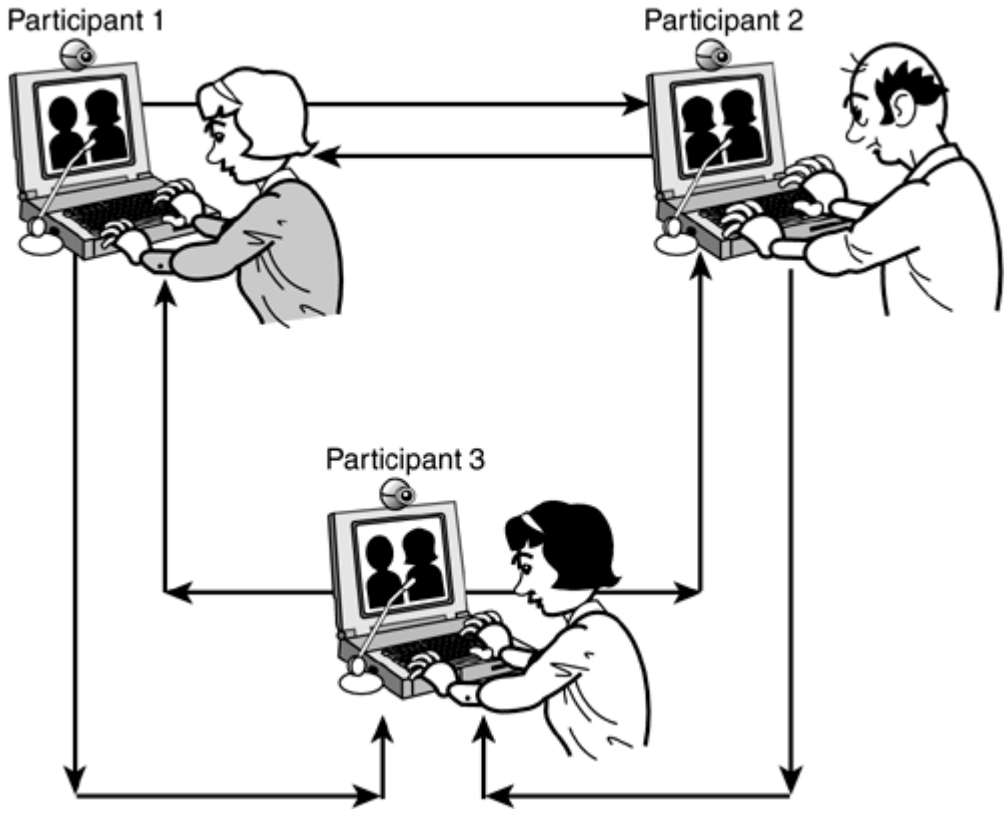
True streaming (also called real-time streaming) of media is the transfer and presentation, as it arrives, of media from a remote site (in the typical case, across the Internet). Examples of such streaming of media can be found in the numerous webcasts that have proliferated on the Web, including numerous radio stations and national TV broadcasters such as the BBC.

A hybrid form of streaming known as progressive streaming also exists, which is less technically challenging than true streaming and quite common on the Web today. Progressive streaming is employed where it is expected or known that the bandwidth requirements of the media (in order to play in real-time) exceed the available bandwidth for transfer. With progressive streaming, the media is downloaded to your system's hard disk. However, the rate of transfer and portion downloaded is monitored. When the estimated (based on current transfer rate) time to complete the transfer drops below the time required to play the entire media, play is begun. This ensures that play of the media is begun as soon as possible while guaranteeing (as long as transfer rate doesn't drop) that the presentation will be continuous.

In this passive reception aspect, streaming, in its end result, is little different from the already familiar forms of radio and TV. The aspect that really empowers the potential of streaming is that media creation (not just reception) is possible for each user. This enables new levels of communication between users when audio and video can be streamed between sites in real-time. The "killer application" of this technology is the video conference: all participants in the conference stream audio

and video of themselves to all other participants while simultaneously receiving and playing or viewing the streams received from other participants. Figure 7.8 shows a typical video conferencing scenario.

Figure 7.8. Typical video conferencing scenario.



Streaming affords considerable technical challenges, many of which still haven't been overcome adequately. Not the least of these challenges is the already discussed bandwidth requirements for time-based media. Streaming across the low-bandwidth connections to the Internet possessed by today's typical user—a 56K modem—can be achieved only by the application of the most extreme compression codecs, resulting in severe quality loss (typically a few pixilated or blurred frames per second). The situation is less extreme for audio but still not perfect. The situation is only exacerbated by the fact that simultaneous, bi-directional streaming is required for applications such as video conferencing: both sites transmitting and receiving media simultaneously.

The challenges don't stop simply at bandwidth limitations but more generally stem from data transmission across a network, typically a Wide Area Network (WAN) such as the Web. The data that forms the media stream, typically in fixed sized packets, suffers a delay, known as latency, between its transmission and receipt. That latency can and typically does vary between packets as network load and other conditions change. Not only does this pose a problem for the timely presentation of the media, but also the latency might vary so much between packets that they are received out of order whereas others might simply be lost (never received) or corrupted. Both ends of the media stream, the transmitter and receiver (or source and sink), have no control over these conditions when operating across a network such as the Internet. Transmission using appropriate protocols for communications

such as RTP (Real-time Transfer Protocol) and RTCP (RTP Control Protocol) can aid the monitoring and, hence, detection of and possible compensation for such network induced problems. However it cannot fix them.

Processing Media

The great advantage of possessing media in a digital format is that it opens endless possibilities. No longer is media simply captured and then stored or transmitted prior to presentation. After it has been captured (or generated), it becomes digital data that might be processed in any number of ways including filtering, adding effects, being compressed or decompressed, as well as being combined with other media. Figure 7.9 shows the traditional analogue approach to media capture, editing, and transmission. Figure 7.10 shows the digital approach and the additional opportunities it affords.

Figure 7.9. Traditional analogue approach to media capture, editing, and transmission.

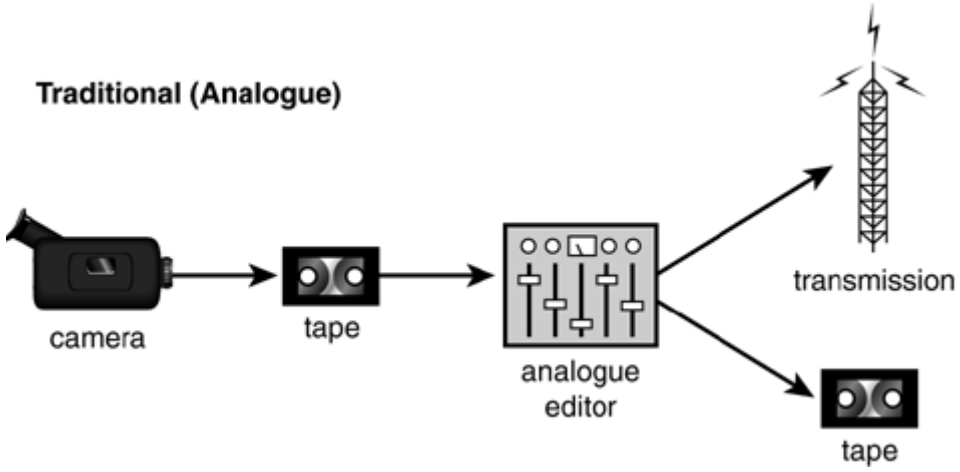
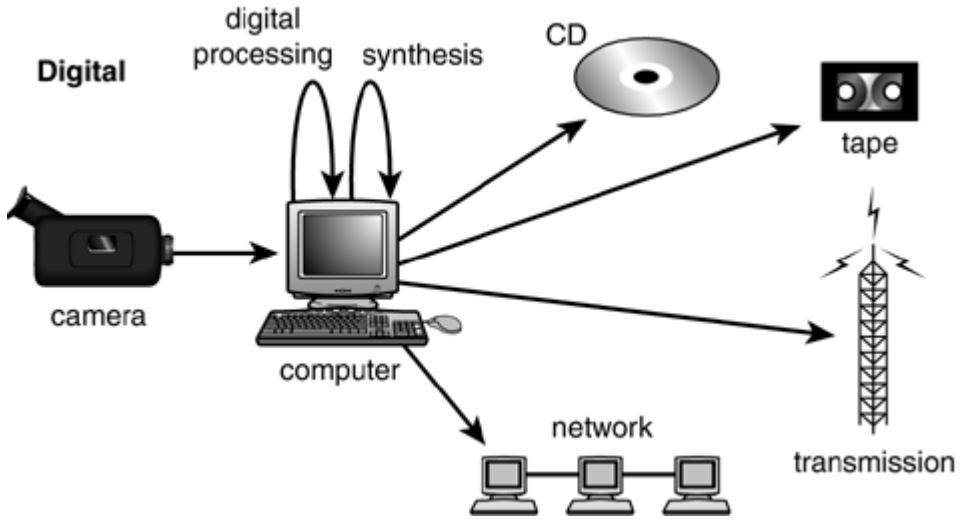


Figure 7.10. Digital approach to media capture, processing, and distribution.



Traditionally, the act of processing media is known as editing, or alternatively post production. With the advent of digital media, this became known as Non-Linear Editing (NLE). The media could be edited (for example, composited) at an arbitrary location (time) along its length rather than be constrained to linearly editing (from start to finish) the media due to representation and storage restrictions (for example, on a tape).

However, both the terms editing (whether linear or non-linear) and post production have a strong association with direct human manipulation of the media. Typically, video and audio editing software such as Adobe's Premiere, Ulead's Media Studio, Sonic Foundry's Sound Forge, or Goldwave is used by a person to manually segment, splice, transform, and so on the media. Editing then is a subset of processing because processing not only includes direct human control of the operation, but also includes automated processing entirely under software control. Direct software processing of media without the need for human intervention opens a host of exciting possibilities and new applications: for instance, automatic subtitling, motion tracking, and image enhancement.

The following list defines some of the more common audio and video processing (editing) operations that are carried out:

Capture— Recording audio or video content directly in a digital format. Alternatively, transferring it from an analogue medium (for example, VCR) into a digital form.

Compositing— Blending two media objects together to combine them. It is the same technique as Superimposing. Examples include adding captions to a video sequence.

Cropping— Removing a portion of the media. This term is typically used for audio content.

Fading In/Out— Smoothing the transition between two different images or sounds to convey a sense of continuity. It is a common form of Transition.

Filter— Using this dedicated tool to modify either video or audio by adding an effect. Audio might be filtered to remove noise at a particular frequency band, whereas a motion-blur filter might be applied to video for effect.

Logging— Viewing the original media material and determining the sections that will be employed in the project being constructed.

Morphing— Transforming one image into another across time by mapping features of one image to another. Typical examples include morphing one person's face to another's.

Printing— Saving a video object from the computer back to a more traditional format such as a VCR or camcorder.

Resampling— Changing the sample frequency for some audio. Typically done as a means of reducing the size of the object by resampling to a lower frequency (downsampling).

Superimposing— Laying one media object over the top of the other in order to combine them. For instance, the image of actors in a studio can be superimposed over an outdoor backdrop.

Transition— Moving between two dissimilar video images or audio samples rather than simply juxtaposing them. Examples include fading, wiping, and scrolling.

Audio Primer

Sound occurs because of a vibration of molecules that arrives at our ears as a wave. Typically the molecules vibrating are air, but sound also propagates through other mediums including liquids and solids.

The rate at which the molecules vibrate determines the pitch of the sound, whereas the amount (amplitude) of vibration determines the volume. The rate of vibration is known as the frequency and is measured in hertz. One hertz represents one complete cycle or vibration per second. A person with unimpaired hearing is able to perceive sound from around 20Hz to around 20KHz (20,000 Hertz). However, human perception isn't evenly distributed across that frequency range: Far more attention or emphasis is given to the lower frequency range that, perhaps not surprisingly, matches the frequency contained in human speech. Transforming the frequency scale to a log representation provides a reasonable first approximation of the "weight" given to different frequencies by our hearing system. Figure 7.4 (shown earlier) shows the perceptually critical bands of hearing.

Nearly every sound, with the exception of pure tones generated musically or automatically, is a complex amalgam of vibrations at different frequencies. It is the sum of these individual vibrations and their amplitudes (strengths or volumes) that make up the sound. Thus, not only can a sound be described, but also composed or generated by detailing the individual frequencies (and their amplitudes) that compose it. Similarly, a sound can be altered by changing the frequency or amplitude of one or more of the pure tones that compose it. This type of functionality is available in some of the more sophisticated audio studio applications.

Normal sounds such as speech, music, and much of what we consider noise (for example, traffic or office sounds) aren't static and unvarying but constantly changing in their component frequency and amplitude characteristics. Indeed it is that fundamental time varying property that allows us to generate speech as a sequence of sounds (phonemes) and music as a sequence of notes.

Sound, arriving as it does, is inherently an analogue quantity. Digitization is the process of transforming an analogue sound into a digital representation. Dedicated hardware, such as a PC's soundcard, is required to perform this task of analogue-to-digital (A-to-D) conversion as well as the inverse digital-to-analogue (D-to-A) conversion when a digital sound is to be presented (sent to speakers).

In performing digitization, two choices must be made, which both significantly impact the quality of the recorded (in the computer) sound and the size of the resulting media object (file if it is saved or conversely bandwidth required if it is being transmitted). These are the sampling frequency and the quantization level.

The first choice is the sampling rate (frequency)—the number of times per second that the sound will be captured (turned into a number). It is vital for the sound to be sampled frequently enough to capture its ever-changing nature and the frequency of the individual components of each sound. The Nyquist Theorem exactly describes this relationship between sampling frequency and frequency of the signal being captured. If a signal is being sampled at frequency f_n, only signals up to $f_n/2$ will be accurately represented. For instance, the sampling rate used for audio CDs is 44.1KHz (44,100 Hertz), meaning that all sounds up to 22.05KHz will be reliably captured: quite sufficient for the human ear. However if a lower sampling rate is used (as is often done), the higher frequency components of the sound won't be represented correctly. For instance, if sampling at 11,025Hz (a submultiple of 44.1KHz that is often used), nothing above 5.5KHz would be correctly represented. Not only could this result in the loss of an important part of the sound, but also it tends to adversely affect perceptions of naturalness because nearly all sounds have resonances that extend into the higher frequencies.

Signal frequencies above the Nyquist frequency (half sampling rate) aren't lost but folded back into the lower frequency domain in a process similar to taking the modulus of a number. This is known as aliasing. It is a familiar visual phenomenon with the rotors on helicopters and planes, and even the spokes on wheels, appearing to be stationary or going backward on film (the interaction of the frequency of the rotation of the rotor, blade, or spoke and the much lower sampling frequency at which the film was shot). Such an outcome will result in significant corruption of the signal, manifesting as a hiss or other noise, if there were strong signals above the Nyquist frequency. For this reason, low-pass filters are normally used to eliminate these high frequency components prior to sampling.

The second choice in the digitization process is the quantization level: the number of bits used to represent each sample. The greater the number of bits employed, the better dynamic range or sound resolution that occurs because of being able to more accurately define the amplitude at that point in time. The choice of an adequate number of bits (for example, 16 that is used in CD audio) will ensure that quieter passages aren't lost. Too few bits make the audio signal sound fuzzy such as through a poor telephone.

Choices of sampling rate and quantization not only affect the quality of the resulting audio, but also directly determine the bandwidth (size) of that audio object. This is a very important factor when considering streaming audio over a network with a bandwidth limitation. The following formula

illustrates the relationship whereas Table 7.1 shows the bandwidth for one second of audio at some of the more common sampling rate and quantization level combinations:

Bits per Second = # Channels x Sampling rate x quantization level

Table 7.1. Bandwidth Requirements for Audio at Different Sampling Rates and Quantization Levels

GuidelineExamples of Quality	Sampling Rate	Quantization Level	Number of Channels	Kilobytes/Second
CD Audio	44.1KHz	16	2 (stereo)	176.4
FM Radio	22.05KHz	16	2 (stereo)	88.2
Stereo 1 - Acceptable	11.025KHz	16	2 (stereo)	44.1
AM Radio	11.025KHz	16	1 (mono)	22.05
Stereo 2 - Grainy	11.025KHz	8	2 (stereo)	22.05
Old hand-held game machine	11.025KHz	8	1 (mono)	11.025

Clearly, a choice of lower sampling rates, quantization levels, and the number of channels can significantly reduce bandwidth requirements, but at the expense of a potentially significant reduction in quality. Some of the most commonly employed codecs (compression schemes) for audio coding will be discussed in the next section. These audio codecs can significantly reduce the bandwidth requirements.

Speech and Music

The two most commonly processed forms of audio data are speech and music, each of which has its own unique characteristics.

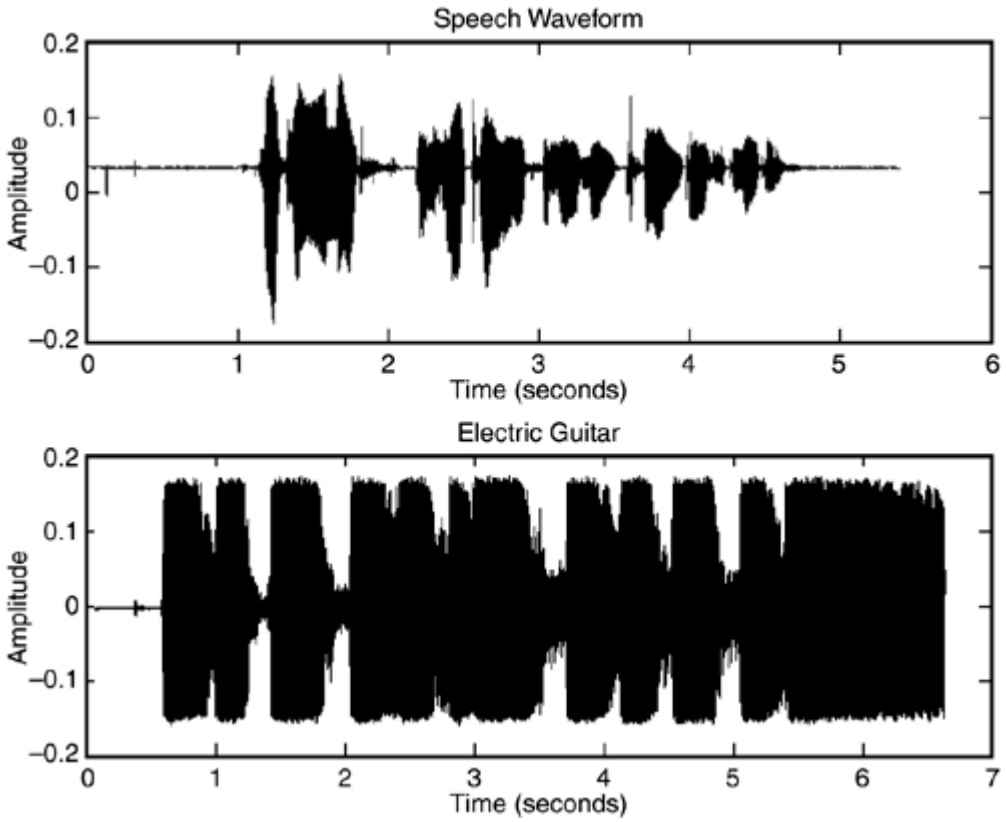
Speech is produced by the human vocal apparatus in which placement of the articulators— the lips, tongue, jaw, and velum (soft palate that includes the uvula)—form the shape of the passage through which air flows. The shape of this passage determines its resonant frequencies and hence the sound produced from the lips as air escapes.

One property of speech sounds lends itself well to compression—most of the signal's energy is concentrated in a frequency range from 100Hz to under 5KHz (varies depending on the sound and speaker). This isn't to say that higher frequency components to the sound don't exist because they certainly do. Rather, most of the information that people use to determine the sound as well as other information such as speaker gender and identity can be found in this region. This can be exploited by sampling at a frequency to capture the vital information, but not to preserve the total sound. Although the digitized speech might not sound exactly like the original, most of the vital information will still be preserved, and at a considerable bandwidth saving. For example, speech sampled at 11KHz is still easily intelligible.

Music is such an encompassing category, being dependent on the form of music and type of instruments used, that it is difficult to make generalizations about its properties. However, music is far more likely to cover a wider frequency range than speech and, hence, suffer more from sampling at lower frequencies.

Figure 7.11 shows the time waveform of a short passage of a speech, "The JMF; an API for Handling Time-based Media," from an adult male in the top plot, as well the first cords of "Smoke on the Water" played on an electric guitar in the bottom plot.

Figure 7.11. Contrast between two sound waveforms.



An alternative form of encoding music is known as MIDI (pronounced "mid-ee") for Musical Instruments Digital Interface. This is a digital format for recording the instruments and notes that are being played in a piece of music—not the sounds themselves. As such, it is an extremely compact format when contrasted with digitized sound. On the down side, MIDI doesn't guarantee the same level of fidelity in reproduction that sampling can—it is dependent on the quality of the playback instrument (often a computer soundcard but originally synthesizers and other such instruments) in its capability to use its voices (different sampled or synthesized instruments) to reproduce the sound appropriately.

Content Types, Formats, and Codecs

The origins of the three major audio content types are associated with a particular computer platform—Wave (WAV) from the Windows platform, AIFF from the Macintosh platform, and AU from Unix. All

three have grown in parallel such that they roughly provide similar functionality in terms of supported formats. The JMF provides support for all three as well as MIDI, GSM, and the various MPEG schemes.

Until recently, the dominant codecs in the audio arena have had their origins in the telecommunication area, being codecs for compressing speech over telephone lines. Among this group are codecs such as ADPCM, A-Law, and U-Law. A common approach among such codecs is known as companding. The basis of companding is to use a non-linear quantization scale: fewer bits are allocated to the higher values (somewhat analogous to transforming to the log domain).

New codecs have appeared based on perceptual compression and designed for music (more challenging than speech). The MPEG schemes are the best known of these. In particular, MP3 (MPEG Layer 3, not MPEG-3 because such an entity doesn't exist) is famous because of its use to encode music on the Internet. The MPEG compression scheme is frequency domain based. Sound is transformed into a number of (for example, 32) frequency channel values. The frequency dependence of the threshold of hearing (minimum volume for a sound to be heard) is combined with masking effects (loud sounds at one frequency raise the hearing threshold for other frequencies) so that the minimum number of bits are used to encode each channel and hide quantization noise. The MP3 scheme is well known for achieving roughly a 10:1 compression while also maintaining a (perceptually) high quality.

The following lists some of the better-known audio codecs:

ADPCM (Adaptive Differential Pulse Code Modulation)— A temporal based compression scheme that looks at the difference between successive samples. The scheme is further strengthened (but complicated) by predicting what the next sample should be and transmitting or storing only the difference between the predicted and actual difference. A non-linear scheme is employed to record this value. ADPCM is supported by the JMF.

A-Law— A companding compression scheme, A-Law is a standard from the ITU that is closely related to G7.11 (U-Law), and used in those countries where U-Law isn't found. Based on the compression of speech over phone lines, it is able to reduce 12-bit samples to 8-bit quantization. A-Law is supported by the JMF.

G.711 (U-Law)— A companding compression scheme, G7.11 is an ITU standard employed in Japan and North America, as well as being used commonly on the Web and by Sun and NeXT machines. It is related to the A-Law scheme. G.711 compression is supported by the JMF.

GSM— An international standard for mobile digital telephones, GSM is based on linear predictive coding: the prediction of future samples based on (a weighted sum of) those that have already been seen. The scheme achieves significant compression but at a noticeable loss of quality. GSM is supported by the JMF.

MPEG Layer I, II, and III— From the MPEG-1 and MPEG-2 standards, the three layers represent an increasingly (from 1 to 3) sophisticated compression scheme based on perception (see the previous discussion). Layer I corresponds to a data rate of 192Kbps, Layer II a data rate of 128Kbps, and Layer III (MP3: the most famous and widely used) corresponds to an upper-bound on data rate of 64Kbps. The JMF supports all three layers.

RealAudio— From Real Networks and famous because of its widespread exposure and usage on the Internet. RealAudio is a codec designed to support the real-time streaming of audio. RealAudio is a proprietary codec.

To illustrate the differences in terms of degree of compression and audio quality between different codecs, the book's Web site (www.samspublishing.com) has a number of versions of the same audio sample. The audio sample is a short piece in four segments. The first segment is a speech from an adult male speaker of Australian English and serves as an introduction. The three remaining segments are all instrumental music. The first musical piece is an organ playing a few bars of "California Dreaming." The second musical piece is a guitar playing a few, well known, bars of "Smoke on the Water." The third and final musical piece is a five second segment of a Didgeridoo being played: a traditional woodwind instrument of the Australian Aborigine. The same original audio sample has been transcoded using a number of different codecs so that they can be contrasted. The name of each file identifies the codec, sampling rate, and quantization level used:

<codec>_<sampling rate>_<quantisation>.<content_type>

For instance, `GSM_8_16.wav` is a Wave file encoded using GSM at 8KHz sampling and a quantization level of 16 bits. Sampling rates that aren't exact multiples of one thousand (for example, 22.05KHz and 11.025KHz) are rounded as such for the purposes of filenames only. Thus, `Linear_22_16.wav` is a Wave file with linear encoding sampled at 22.05KHz with 16-bit quantization.

Video Primer

Persistence of vision is the name of the phenomenon that enables humans to see a succession of still frames, projected at sufficient speed, as a smooth moving picture. Both video and animation rely on this property of the human visual system. The fusion frequency is the rate at which the frames must be projected in order for them to "fuse" into a perceptually continuous stream. The particular frequency varies between individuals (and the amount of motion between frames) with around 40 frames per second ensuring a flicker-free perception of smooth motion. However persistence of vision isn't a binary (all-or-nothing) effect: lower frame rates still convey the illusion of motion, although with worse flicker and jerkiness as the frame rate drops. However, anything below about 10 frames per second is perceived for what it really is: a succession of still frames.

The roots of video lie in the television industry and its various standards, although some of the limitations that shaped television standards, such as the low screen refresh rates of televisions, no longer hold for modern computer-based technology. The three analogue broadcast standards are known as NTSC (National Television Systems Committee) used in the United States and Japan, PAL (Phase Alternating Line) used in Europe and Australia, and SECAM (Sequential Couleur Avec Memoire) used in France. Although the particular horizontal and vertical resolutions, as well as frame rate, differ somewhat among the three standards, all follow a similar approach for encoding the signal. Because of bandwidth considerations at the time (some concerns clearly don't change), each frame is divided into two fields, one consisting of the even lines in the frame, and the other consisting of the odd lines. These are transmitted in succession, and the frame is composed by interlacing the fields.

On the other hand, each frame in a raw (digital) video sequence is a separate and complete image. These raw frames are invariably kept as bitmaps—an image composed of a number of picture elements (pixels). The number of pixels is defined by the horizontal and vertical resolution of the image. The more pixels there are, the more sharp, clear, and detailed the image is. Each pixel records the color intensities at that point of the image. Color might be recorded as RGB (Red, Green, Blue) or Luminance/Chrominance values. Regardless, a number of bits are employed to represent that color value at each pixel. The more bits employed, the truer the colors of the resulting image. A far more complete discussion of 2D images can be found in Chapter 10, "3D Graphics, Virtual Reality, and Visualization."

Just as for audio, choices of number of frames per second and quantization not only affect the quality of the resulting video, but also directly determine the bandwidth (size) of that video object. This is a very important factor when considering storing video or even more constraining, streaming over a network. The following formula illustrates the relationship while Table 7.2 shows the bandwidth for one second of video at some of the more common frame rate and quantization level combinations.

Bits per Second = Frame rate x Horizontal Resolution

```
x Vertical Resolution x Bits per Pixel
```

Table 7.2. Bandwidth Requirements for Video at Different Resolutions, Frame Rates, and Color Quantization Levels

Typical Example	Frame Rate	Horizontal Resolution	Vertical Resolution	Bits per Pixel	Kilobytes/Second
NTSC	~30	640	480	24	27,000
PAL	25	768	576	24	32,400
"Quarter Screen" TV	24	320	240	24	5,400
Video Conference 1	12	320	240	16	1,800
Video Conference 2	12	160	120	16	450

Contrasting Table 7.2 with Table 7.1, it can be seen just how greedy video is with regard to bandwidth. Even the lowest quality settings from Table 7.2—something that would result in little more than a very small image of low quality in one corner of the screen—consumes nearly three times the bandwidth of CD quality audio. To achieve (current) television quality video, a bandwidth over 150 times greater than that required by an audio CD is required. Compression is an absolute necessity for video if it is to be used with today's computers and networks.

Content Types, Formats, and Codecs

Two content types (architectures) have long dominated the video arena, becoming de facto standards. These are QuickTime and AVI. Although originally associated with a single platform (the Macintosh for QuickTime, and Windows PC for AVI), they are now cross-platform. Each supports a number of video (and audio) codecs within its architecture: in fact chiefly the same ones. Both of these content types are strongly supported in the JMF.

A third significant, but far more recent, name in the video content type area is RealVideo. Both a content type and format, RealVideo from Real Networks is targeted at streaming of video over networks, and has become the Web leader in this area.

Unlike audio, which is far less demanding of bandwidth, significant compression is required in order to play video on a computer, including from a CD-ROM. For this reason, the area of video codecs has and continues to receive considerable attention and effort from international bodies, the private sector, and academia. An example of this ongoing development is the relatively recent release of the MPEG-4 standard.

Invariably the codecs in common usage at the moment are lossy. Most are based on a block compression scheme in which the individual frame image is subdivided into a number of fixed-sized blocks. A common size for such blocks is eight-by-eight pixels. Two techniques are commonly used to compress these square blocks—Vector Quantization (VQ) and Discrete Cosine Transforms (DCT). The full details of each approach are beyond the scope of this book.

However, VQ builds a codebook of different possible blocks—similar to color swatches. Each image block is then encoded (quantized) as the number of the codebook element that it most resembles (is closest to). On the other hand, those schemes using DCT transform each block into the frequency domain (the DCT is analogous to the Fourier transform). Savings (compression) can then be made by utilizing fewer bits to represent higher frequency components because these are known not to contribute as significantly to the perceptual quality of an image.

A number of codecs are asymmetric, taking different amounts of time to compress versus decompress the same stream. In all cases, the compression takes longer. This is due to the nature of the task—compression is simply more time-consuming because of all the calculations required—and partly due to design choices. It is generally assumed that the equipment dedicated to compressing video might be specialized and powerful, whereas playback might have to occur across a range of equipment. Under such an assumption, easing the task of decompression at the expense of compression is a good choice.

Some of the better known video codecs are as follows:

Cinepak— A very common format spanning multiple PC platforms (originally designed for Apple's QuickTime) and even game consoles. Cinepak is perhaps the most popular means currently employed to encode video in multimedia applications. Cinepak employs temporal and spatial compression in a lossy scheme that uses VQ and blocks. The scheme is intended for software implementation with compression, taking considerably more time than decompression. Cinepak performs well with video that contains substantial motion, but can have problems with static images. The Cinepak codec is supported in the JMF 2.1.1.

DivX— An open-source codec based on the MPEG-4 (see later) standard, DivX is gaining wide popularity on the Internet because of its free availability for most platforms and the quality of its compression.

H.261— An international standard targeted at the video-conferencing area with bandwidths in the 16-48 kilobytes-per-second range, H.261 is a lossy scheme using block DCT and motion compensation. It has some similarity to MPEG-1, which it predates. The H.261 codec is supported in the JMF 2.1.1.

H.263— Another international standard and an advance on H.261, H.263 is also designed for video conferencing applications at low bit rates. Its compression algorithms are superior to H.261 (block based DCT), and it should be used in preference to that standard when bandwidth is critical. The H.263 codec is supported by the JMF.

Indeo— A codec from Intel, Indeo is now available on a number of platforms. Indeo employs both spatial and temporal compression in a lossy scheme that uses VQ and blocks. Indeo takes longer to compress than decompress video. Indeo v32 and v50 are supported by JMF 2.1.1.

MJPG— Motion-JPEG is a scheme directly based on the JPEG (Joint Picture Experts Group) approach of compressing individual still images. MJPG employs spatial compression only, considering each frame in isolation. This is not optimal in a compression sense, but it does make stream editing easier. The scheme is widely used by video capture cards. The approach is lossy and based on a block-oriented DCT. The MJPG standard is supported in JMF 2.1.1.

MPEG-1— The first standard issued by the Motion Picture Expert Group, MPEG-1 is a lossy scheme employing spatial compression and a more sophisticated (than Cinepak, for instance) temporal compression system. MPEG-1 is the standard on which the Video CD is based. The scheme is lossy and uses DCT for block-oriented compression. MPEG-1 was designed (in 1988) to be carried out in hardware (particularly compression), although modern PC systems are more than capable of decoding MPEG-1 in real-time and can also perform compression (with acceptable delays). MPEG-1 is supported in JMF 2.1.1.

MPEG-2— An extension of the MPEG-1 standard to take it from 30 frames per second to 60 frames per second of high quality video and used in applications requiring such quality (for example, broadcast transmissions over satellite). MPEG-2 is the standard on which products such as Digital Television set-top boxes and the DVD are based. Initially (the standard was ratified in 1994), MPEG-2 required specialized hardware, particularly for the compression side. However, all modern PC systems are capable of rendering MPEG-2 in real-time and can perform compression with acceptable delays.

MPEG-4— The latest international standard from the MPEG team, MPEG-4 is more than a video compression scheme. The video compression scheme holds much promise, yielding high-quality images at low bit rates, and is closely related to the H.263 standard. MPEG-4 follows the MPEG family of codecs approach of block-based DCT compression. MPEG-4 is supported in JMF-2.1.1 via extensions provided by IBM. These are discussed in Chapter 9.

RealVideo— A proprietary codec from Real Networks, RealVideo is currently probably the most commonly found codec on the Web for streaming video. One of the features of RealVideo is that several different versions of a movie can be provided in order to match the bandwidth limitations of different users (for example, T1 versus cable modem versus 28.8Kbps version).

Sorensen— A software codec, the same as Indeo and Cinepak, the Sorensen codec employs spatial and temporal compression in a lossy scheme based on vector quantization of blocks. A newer codec than Indeo and Cinepak, Sorensen employs a more sophisticated temporal compression scheme that includes motion compensation, and can therefore achieve better results.

To illustrate the differences in terms of degree of compression and artifacting (losses or artifacts in the images because of the compression scheme) between different codecs, the book's Web site (www.samspublishing.com) has a number of versions of the same video. The video is a short piece in three segments. The first segment is a "talking head"—a static shot of me talking to the camera. The second segment is outdoor and dynamic—me riding a bicycle within camera range; whereas the third segment is a short synthetic (generated, not captured with a camera) sequence. The same original video has been transcoded using a number of different codecs so that they can be contrasted. Figure 7.12 shows four images from the video, where each image is from a different encoding: the top-left panel is Cinepak, the top right is IV32, bottom left is RGB, and bottom right is motion JPEG. Each version differs because of the codec used to compress it. However, the static screen shot shouldn't be used as the basis of comparison because of artifacts of the screen capture.

Figure 7.12. JMStudio playing four versions of the same sample file.

The name of each file identifies the codec and screen resolution found in that sample.

<codec>_<horizontal>x<verticaln>.<content_type>

For instance, the file `MJPG_320x240.mov` is a QuickTime (.mov) file encoded with Motion JPEG at a resolution of 320x240.

What Is the JMF?

Fundamentally, the JMF is an extension to Java for handling audio and video. More rigorously, the JMF API (Java Media Framework Application Programming Interface) is one of the Official Java optional APIs that extends the functionality of the core Java Platform. Included in this group of optional APIs, freely available from Sun, are others such as Java 3D and Java Advanced Imaging (JAI).

JMF, as its name implies, is a collection of classes to enable the processing of (time-based) media objects. Sun Microsystems' JMF 2.1.1 Programmer's Documentation introduces the JMF as

Java Media Framework (JMF) provides a unified architecture and messaging protocol for managing the acquisition, processing, and delivery of time-based media data. JMF is designed to support most standard media content types, such as AIFF, AU, AVI, GSM, MIDI, MPEG, QuickTime, RMF, and WAV.

Sun's main JMF page has the following to say of the API:

The Java Media Framework API (JMF) enables audio, video and other time-based media to be added to Java applications and applets. This optional API, which can capture, playback, stream and transcode multiple media formats, extends the multimedia capabilities on the J2SE platform, and gives multimedia developers a powerful toolkit to develop scalable, cross-platform technology.

Thus, the JMF is a collection of classes aimed at extending the Java Platform in the areas of video and audio processing, whether locally or across a network, and for both applets and applications.

Features of the JMF API

Amongst the key features of the API are

- Platform independence. There is a reference implementation that will run anywhere Java runs.
- Integrated and uniform handling of Audio and Video as media objects.
- Support for a significant number of the major audio and video content types and codecs.
- Playback of media.
- Saving of media (to a file).
- Capture of media from devices such as cameras and microphones.
- Receipt of media streams transmitted across the network.
- Transmission of media streams (across the network).
- Multiplexing/Demultiplexing (combining and splitting) of media.
- Transcoding (altering to a different format) media.

- A unified processing framework that supports all operations on media (for example, effects) as processing.
- Extensibility to support further formats and plug-ins.
- Seamless integration with the existing Java API.

The Promise of JMF

Enumerating the features of JMF provides a rather bland view of the API. Only after the potential applications implemented using the JMF are considered can the true possibilities become clear.

Among the exciting possibilities, are the following:

- Video conferencing across a range of platforms and networks
- A complete video and audio editing suite
- Empowering the latest mobile computing such as cellular phones and PDAs (Personal Digital Assistants)
- Integrated multimedia applications entirely in Java and hence running on any platform

Video conferencing is often considered a "killer app," bringing together a number of technologies in order to allow people to visually and verbally communicate in real-time. The availability of a video conferencing system of reasonable quality and independent of both hardware (particular cameras, microphones, hardware codecs) and software (particular operating systems) constraints would likely have a major impact in the conduct of both business and private life. The JMF, and Java more broadly, is a framework in which that can be achieved. The challenge remains bandwidth; but newer codecs (all of which can be incorporated into the JMF) and network services continue to whittle away at this hurdle.

The strength of the JMF lies not so much in the functionality of the API itself but in the broader context of the complete Java platform. This brings not only portability but also seamless integration with a number of other APIs and suites. For instance, a complete video and audio editing suite could be developed by using the JMF for handling the raw media, in combination with Java's AWT and Swing sets for presenting a GUI, and the JAI (Java Advanced Imaging) API for performing a number of the (video) effects.

Similarly, Java is a perfect solution in the consumer and embedded technologies area such as mobile phones, personal digital assistants, and TV-set-top boxes (for example, digital television set top boxes). Indeed one of the design goals of Java was to meet the security and portability demands of such a range of devices. The Java Micro Edition and related technologies deliver on that need. The advent of the next generation of these devices has seen the availability of increased processing power coupled with the demand for more sophisticated interfaces and content. The JMF is perfectly suited for these needs and is already being used to, among other things, stream video to the latest mobile phones.

With that said, it is worth remembering that the JMF is in its adolescence right now and, not being fully mature, it has some shortcomings. In particular a number of formats, including the important MPEG-2 and MPEG-4 standards, currently aren't part of the JMF distribution. That is expected to be addressed in the next JMF release, and Sun representatives have said that they have a continuing dedication to supporting the latest open standards.

Another catch for the unwary JMF programmer is that the JMF is a separate download and not part of the standard Java platform—a vanilla JVM (Java Virtual Machine) isn't capable of running a JMF program. This has implications for those people wanting to write applets using features of the JMF. To ensure that the widest audience can run them, the author must provide either instructions for downloading and installing the JMF (a difficult task for many users) or an automated mechanism for installing the necessary subset of the JMF classes.

Java and Time Based Media: A Short History

Although the Java platform (standard edition) is a powerful tool for many applications, including some aspects of multimedia, its support of time-based media has never been strong. Until recently (SDK1.3), the only class within the core Java platform that dealt directly with time-based media was `AudioClip`, a relatively simple class that supported the loading and play of (Sun) AU audio files, and little else.

The Java Media Framework was designed to extend the functionality of Java into the arena of time-based media. It has gone through two major versions with the current release number being v2.1.1.

JMF v1.0 was known as the Java Media Player and provided playback functionality. Two reference implementations were released: one for Windows and one for Solaris. Version 1.1 was a platform independent (or Pure Java) release. Version 1 of the JMF API was developed by Sun Microsystems, Silicon Graphics, Inc., and Intel Corporation.

JMF 2.0 dramatically extended the capabilities of JMF 1.0 by adding streaming, multiplexing and demultiplexing, media capture, transcoding, a unified processing framework, and an extensible plug-in design. It was designed by Sun Microsystems and IBM. Three implementations of JMF2.0 were released: a Pure Java Reference version, as well as an optimized version for Windows and one for Solaris. Version 2.1 of the API added support for Linux as well as increasing support for various streaming video servers. Version 2.1.1, current as of the time of writing, has improved the RTP API as well as added support for the H.263 codec.

Media Formats and Content Types Supported by JMF

The JMF provides support for a number of the most important and popular content types and formats in both the audio and video arenas. In the area of content types, that includes names such as QuickTime, AVI, Wave, and MPEG. In the area of formats or codecs, that includes MPEG (for example, MP3), U-law, Cinepak, MJPG, and H.263. Further, as witnessed by the history of the various versions of JMF, that support has continued to increase (for example, H.263 added in the most recent version).The two most important formats currently absent from the JMF are MPEG-2 and MPEG-4. These are significant omissions! However the JMF development team at Sun has set support for these two formats as its highest priority. As such, it is expected that support for MPEG-2 and MPEG-4 will be found in the next major release of the JMF.

That broad coverage of content types and formats allows the JMF not only to claim platform and format independence, but also to provide new opportunities to programmers. Programmers can select the appropriate format for the task at hand and even transcode between formats as needed.

In the area of protocols, the JMF supports the file, http, ftp, and rtp protocols.

Tables 7.3 and 7.4 show the media format and content type support of the current version (v2.1.1) of the JMF. Table 7.3 shows support for audio content types, and Table 7.4 shows support for video content types. There are three implementations of JMF2.1.1: the cross-platform (Cross) pure Java version, the Solaris (Sol) performance version, and the Windows (Win) performance version. They differ slightly in their support of formats. Most formats supported by the JMF can be both read (decoded) and write (encoded); however, in some cases that isn't true. Hence these tables have entries that list content type, format, which implementations can decode that format, and which implementations can encode it.

Table 7.3. Audio Content Types and Formats (Codecs) Supported by the JMF 2.1.1 Implementations

Content Type	Format	Decode/Read	Encode/Write
AIFF (.aiff)	8-bit mono/stereo linear	Cross, Sol, Win	Cross, Sol, Win
	16-bit mono/stereo linear	Cross, Sol, Win	Cross, Sol, Win
	G.711 (U-law)	Cross, Sol, Win	Cross, Sol, Win
	A-law	Cross, Sol, Win	
	IMA4 ADPCM	Cross, Sol, Win	Cross, Sol, Win
GSM (.gsm)	GSM mono audio	Cross, Sol, Win	Cross, Sol, Win
MIDI (.mid)	Type 1 & 2 MIDI	Sol, Win	
MPEG Layer II Audio (.mp2)	MPEG layer 1,2 audio	Cross, Sol, Win	Sol, Win
MPEG Layer III Audio (.mp3)	MPEG layer 1, 2 or 3 audio	Cross, Sol, Win	Sol, Win
Sun Audio (.au)	8-bit mono/stereo linear	Cross, Sol, Win	Cross, Sol, Win
	16-bit mono/stereo linear	Cross, Sol, Win	Cross, Sol, Win
	G.711 (U-law)	Cross, Sol, Win	Cross, Sol, Win
	A-law	Cross, Sol, Win	
Wave (.wav)	8-bit mono/stereo linear	Cross, Sol, Win	Cross, Sol, Win
	16-bit mono/stereo linear	Cross, Sol, Win	Cross, Sol, Win
	G.711 (U-law)	Cross, Sol, Win	Cross, Sol, Win
	A-law	Cross, Sol, Win	
	GSM mono	Cross, Sol, Win	Cross, Sol, Win

Table 7.3. Audio Content Types and Formats (Codecs) Supported by the JMF 2.1.1 Implementations

Content Type	Format	Decode/Read	Encode/Write
	DVI ADPCM	Cross, Sol, Win	Cross, Sol, Win
	MS ADPCM	Cross, Sol, Win	
	ACM	Win	Win

Table 7.4. Video Content Types and Formats (Codecs) Supported by the JMF 2.1.1 Implementations

Content Type	Format	Decode/Read	Encode/Write
AVI (.avi)	Audio: 8-bit mono/stereo linear	Cross, Sol, Win	Cross, Sol, Win
	Audio: 16-bit mono/stereo linear	Cross, Sol, Win	Cross, Sol, Win
	Audio: DVI ADPCM compressed	Cross, Sol, Win	Cross, Sol, Win
	Audio: G711 (U-law)	Cross, Sol, Win	Cross, Sol, Win
	Audio: A-law	Cross, Sol, Win	
	Audio: GSM mono	Cross, Sol, Win	Cross, Sol, Win
	Audio: ACM	Win	Win
	Video: Cinepak	Cross, Sol, Win	Sol
	Video: JPEG (411, 422, 111)		
	Cross, Sol, Win	Sol, Win	
	Video: RGB	Cross, Sol, Win	Cross, Sol, Win
	Video: YUV	Cross, Sol, Win	Cross, Sol, Win
	Video: VCM	Win	Win
Flash (.swf, .spl)	Macromedia Flash 2	Cross, Sol, Win	
HotMedia (.mvr)	IBM HotMedia	Cross, Sol, Win	
MPEG-1 Video (.mpg)	Multiplexed System stream	Sol, Win	
	Video-only stream	Sol, Win	
MPEG-4 Video		IBM	IBM
QuickTime (.mov)	Audio: 8-bit mono/stereo linear	Cross, Sol, Win	Cross, Sol, Win
	Audio:16-bit mono/stereo linear	Cross, Sol, Win	Cross, Sol, Win

Table 7.4. Video Content Types and Formats (Codecs) Supported by the JMF 2.1.1 Implementations

Content Type	Format	Decode/Read	Encode/Write
	Audio: G711 (U-law)	Cross, Sol, Win	Cross, Sol, Win
	Audio: A-law	Cross, Sol, Win	
	Audio: GSM mono	Cross, Sol, Win	Cross, Sol, Win
	Audio: IMA4 ADPCM	Cross, Sol, Win	Cross, Sol, Win
	Video: Cinepak	Cross, Sol, Win	Sol
	Video: H.261	Sol, Win	
	Video: H.263	Cross, Sol, Win	Sol, Win
	Video: JPEG (411, 422, 111)	Cross, Sol, Win	Sol, Win
	Video: RGB	Cross, Sol, Win	Cross, Sol, Win

An additional feature of JMF 2.0 and later is that it is user extensible in the area of protocols, content type, and formats supported. A number of Interfaces are supplied, which users can implement with their own classes. The multiformat support of JMF should continue to grow, not only through the releases of Sun, but also through third party and individual development.

Chapter 8 includes a sample class that queries the JMF Manager class in order to determine the types of media supported for the particular version of JMF and the platform it is running on. That and Sun's JMF site can be used to determine the level of support for various media offered by the JMF.

Levels of Usage of the JMF API

The JMF affords the user a range of programming opportunities. These extend from using the JMF without ever writing a single line of code (using `JMFStudio`), through simple player programming as found in the example at the end of this chapter, all the way to extending the capabilities of the JMF by adding new formats, effects, or codecs (as discussed in Chapter 9).

This has two implications. First, it is possible to take a minimalist approach in learning the JMF: learning only the necessary features for the application required while still achieving the desired effect. Second, it is possible to learn the JMF in layers—starting with the easier concepts and applications and slowly delving into the underlying structure and complexity as and when it becomes desirable.

This part's chapters follow an approach of moving from a simple to a more complicated utilization of the API. The following subsections identify some of the most common levels of utilization that occur, though typically an individual's usage and knowledge don't correspond exactly to any of the following four categories.

Out of the Box with `JMStudio`

As a demonstration of the capabilities inherent in the JMF API, Sun has included an application known as `JMStudio` (Java Media Studio) in the JMF 2.1.1 bundle. The class file is found with all the classes from the JMF API in the file jmf.jar. Running the application is as simple as `java JMStudio`.

Despite its innocuous appearance (see Figure 7.13), `JMStudio` is a powerful application that supports playback, capture, transmission, and transcoding. In these later aspects of capture, transmission, and transcoding, it far exceeds the capabilities of free players, although as noted previously the JMF doesn't support all possible video and audio formats. (In particular, the important formats of RealMedia, Sorensen, and divX aren't to be found, whereas MPEG-2 and MPEG-4 are expected to appear in the next release.)

Figure 7.13. The innocuous appearing, yet extremely versatile, `JMStudio` application that comes as part of the JMF 2.1.1 distribution.

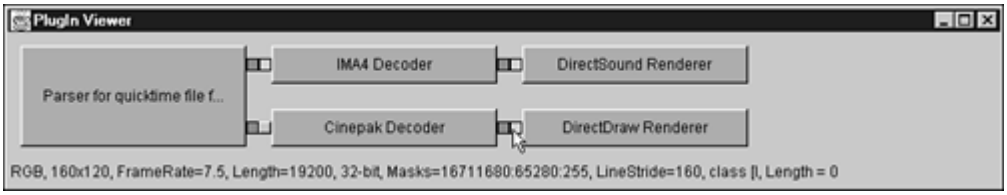
A complete discussion of `JMStudio` is beyond the scope of this part. Sun maintains documentation on the application at http://www.java.sun.com/products/java-media/jmf/2.1.1/jmstudio/jmstudio.html. Although `JMStudio` is chiefly a proof of concept, it is worth enumerating the functionality it provides because it is often more convenient to use `JMStudio` to perform a task (such as transcode between two formats) than write a one-off piece of code to carry out the same thing.

The features of `JMStudio` include

- Support for multiple content types (architectures) and codecs.
- Play audio and video.
  - From local file
  - From URL
  - From an RTP stream
- Capture audio or video from devices connected to computer.
- Export (transcode) media to a file.
- Transmit media via RTP.
- View the plugins (and progress there of) currently processing a media object.

In addition to all the functionality listed previously, this last feature is particularly useful for those wanting to learn about the API itself. By invoking the PlugIn Viewer, it is possible to see the individual objects chained together to form the processing, as well as observe such statistics as the frame rate and size. Figure 7.14 shows the PlugIn Viewer as a QuickTime video (encoded with the Cinepak codec) is decoded for play.

Figure 7.14. JMF's PlugIn Viewer in action.



Those only wanting to use the JMF indirectly through `JMStudio` will likely have sufficient knowledge by reading this chapter. However, reading the main portions of at least the following chapter will convey a far deeper understanding of how the JMF API works.

Simple Players

One of the common usages of the JMF is to incorporate audio or video play into an applet or application. As you will see with the example at the end of this chapter, that is a relatively painless and straightforward task. JMF provides a centralized manager from which can be obtained a player capable of dealing with a particular media object (content type and format). The player operates under a similar model to that of a modern VCR/DVD player: It can be started and stopped, as well as having its rate of play changed.

Programming at this level requires an understanding of the key concepts of the JMF (such as its model of time) together with the knowledge of some of the central classes in the API.

Those wanting to program the JMF at this level will likely only need to consult this and the earlier portions of the Chapter 8. However the deeper the knowledge of the API, the more subtle and complete control can be exercised over any players created. Further, if streaming capabilities are required, the first half of Chapter 9 will also need to be consulted.

Processing: Getting Under the Hood

The JMF provides powerful features for processing time-based media, including multiplexing, demultiplexing, transcoding, saving to a file, and so on. Although many users of the JMF will start off initially satisfied with simpler player applications, it is likely that a significant portion will move into these areas of more sophistication.

Processing is the topic of Chapter 8, and hence it serves as core material for those wanting to utilize the JMF in this manner. In addition, it is likely that most of the material covered in Chapter 9 will also be relevant to users with such applications in mind.

Extending the API, and Interfacing with Other APIs

The most sophisticated levels of usage of the JMF fall into two categories. The first category involves extending the API. The JMF is designed to be extensible so that users can write their own codecs, formats, effects, and so on and thus expand the JMF to suit their needs and constraints. This level of programming requires a strong familiarity with the API as well as the specialized knowledge concerning the feature being added. It is one of the topics covered in Chapter 9.

The second category involves employing the JMF as part of a larger, integrated application in which the processing can be chained or synchronized. An example of such an application includes feeding a JMF video into a Java 3D virtual world. Such applications have great potential, but require the programmer to have familiarity with all the APIs concerned. This synergy between APIs is covered in part (from a JMF perspective) in Chapter 9, and is the topic of Chapter 14, "Integrating Across the Java Media API."

Programming Paradigms When Using JMF

JMF based applets and applications tend to require particular programming approaches that other general calculation programs don't usually possess. This section addresses those approaches.

The JMF deals with time-based media. That requires not only a sophisticated model of time, but also support for the idiosyncratic and asynchronous behavior of a range of hardware devices and networks. Different capture devices might take considerably different times to become ready, network streams might drop out, and file systems might become full through saving large media files. All these eventualities and many, many more should be dealt with robustly and appropriately by a well-written JMF program.

Event Driven

Event-driven programming lies at the heart of most JMF programs. Graphical user interfaces programmed in the AWT or Swing set must wait and respond to user actions (for example, a button press) that occur asynchronously (that is, the program has no knowledge of when they'll occur). Also, a JMF program that is playing or processing must wait and respond to the various timing events and actions arising from a player or processor.

Those of you who aren't familiar with the concepts and practice of event-driven programming should consider acquiring such practice before delving too deeply into the JMF API. Such knowledge is necessary because of the central role of events in controlling the API.

The later section within this chapter dealing with time and the next chapter cover the major listener interfaces and events of the API. However, it is extremely typical to see lines like those found in Listing 7.1 in a JMF program.

Listing 7.1 Skeleton Example of the Type of Event Driven Programming Used in Conjunction with the JMF

```
public class MyJMFProgram implements ControllerListener {
:
:
  player.addControllerListener(this);
:
:
public synchronized void controllerUpdate(ControllerEvent e) {
:
}
}
```

Such a class is listening to, and will be sent events from, the player object.

Threading

The devices responsible for controlling and transporting time-based media (for example, networks, renderers, and capture devices) are asynchronous. As in many applications, when this is combined with the control of multiple streams, channels, tracks, or sources and destinations of media, it is necessary to delegate control of individual items to separate threads so that the whole program won't suffer a bottleneck or be brought to an unresponsive halt by a single recalcitrant subtask.

Java provides strong and fundamental support for threads through the `Thread` class and `Runnable` interface. User classes that are to run as threads can either extend `Thread` or implement `Runnable`. Because of Java's single inheritance, it is often better for a class to implement `Runnable` (which only consists of the single `run()` method) than the subclass `Thread`.

An application employing threads tends to consist of a main, controller program that creates the threads for the individual tasks and both monitors and communicates with them as necessary. The classes that are to act as the threads must possess a `run()` method. This is started when the thread is started, and the thread is alive only while the method hasn't returned.

A threaded program usually possesses code similar to that found in Listing 7.2.

Listing 7.2 Typical Structure of Code Starting Up a New Thread

```
// Need to create a thread to handle a sub-task. First create
// a new instance of the class that will do the work. Then
// pass that object to the Thread constructor. Finally, start
// up the thread.

MyThreadedController controlObj = new MyThreadedController(...);
Thread theThread = new Thread(controlObj);
theThread.start();
```

Exceptions

Exceptions in Java represent unusual, abnormal, and unexpected results that halt the normal flow of program execution. For JMF programs, exceptions are a very real possibility that a well-written and robust program must be capable of dealing with or at least exiting gracefully and with the maximum amount of information for the user. Examples of such exceptions within the context of a JMF program include the inability to create a player or processor for the media object specified, a number of time related exceptions (for example, trying to invoke a method on a processor that isn't yet in a state to support that action), as well as IO exceptions (for example, attempting to open a file that doesn't exist).

Hence, it is common to find a number of `try {...} catch { ...}` blocks in a program, whereby the code that could potentially throw an exception is enclosed in the try block; whereas the one or more catch blocks contain code for dealing with the exceptions that might arise. If you are unfamiliar with exceptions and the mechanism for handling them, refer to a general Java textbook or reference. A typical example of this type of processing is found in Listing 7.3.

Listing 7.3 Typical Usage of `try{ }, catch{ }` Blocks to Deal with Thrown `Exceptions`

```
try {
  Player player = Manager.createPlayer(locator);
}
catch (NoPlayerException e) {
  System.err.println("Unable to create a player for..." + e);
}
```

URLs and Networks

One of the important features of Java is the integration of network support into the heart of the language. That theme of integrating networking support extends to the JMF, where for instance it is not only possible to play a file across the network, but also it is relatively simple. One central class of the API is the `MediaLocator`, which specifies the location of a media object and is closely related to Java Platform's URL class.

Integration of networking features into the JMF extends into support for RTP (Real-time Transport Protocol), the communication protocol employed to stream media across networks. That topic is covered in Chapter 9.

Structure of the API

The JMF API (v2.1.1) comprises a total of 209 classes, of which 85 are interfaces, divided among 11 APIs.

In Java, APIs serve both to group related classes while also acting as a means of controlling the visibility of the attributes and methods of those classes. In order to employ a class that is the member of an API, it, or the entire API, must be imported.

The 11 APIs that comprise the JMF, together with their domain, are listed as follows:

`javax.media`— The main, top-level API comprising most of the classes and also most of the important ones such as `Time`, `Manager`, `Processor`, and `Player`.

`javax.media.bean.playerbean`— A collection of seven classes that provide Java Bean encapsulation for a Player. `MediaPlayer` is the most important class in the API.

`javax.media.control`— An API comprised of 18 Interfaces defining the different types of controls. Examples include `FrameRateControl` and `FormatControl`.

`javax.media.datasink`— An API of one interface and three events defining a listener for `datasink` events.

`javax.media.format`— An important API of 10 classes (one of which is an exception) defining the different formats that JMF is capable of processing. Examples of the classes include `AudioFormat` and `H263Format`.

`javax.media.protocol`— An important API of 25 classes (15 being interfaces) providing support for communication with datasources and capture devices. Among the important classes included in this API are `DataSource` and `CaptureDevice`.

`javax.media.renderer`— An API of two interfaces defining a renderer (for video content).

`javax.media.rtp`— The top-level of the three APIs dealing with RTP (Real-time Transport Protocol) it comprises 26 classes (most interfaces) dealing with streaming content with RTP.

`javax.media.rtp.event`— An API of 23 events that might result when using RTP.

`javax.media.rtp.rtcp`— An API of five classes (four being interfaces) defining usage of RTCP (RTP Control Protocol) within the JMF.

`javax.media.util`— An API of two highly useful classes: `BufferToImage` and `ImageToBuffer` for converting between JMF buffers and AWT images.

Similar to all the Java APIs, and indeed the larger APIs within the core Java Platform, it takes considerable time to gain a thorough familiarity with the entire structure of the API. However, each class and interface in the API has been created for a purpose, and time spent studying the API isn't wasted, and indeed can save considerable effort or frustration.

Further, if this and the following chapters on the JMF don't mention a particular class or functionality within JMF, it doesn't mean that such a class doesn't exist within the API. In three chapters, it is impossible to cover all 209 classes in the API while also providing sufficient coverage of the most important aspects of the API. When in doubt and no JMF-related resource appears to have an answer, one of the first places to start should be with the JMF API Specification: http://java.sun.com/products/java-media/jmf/2.1.1/apidocs/.

Key Classes in the API

Although all classes in the API have a role to play, some are more central than others. These central classes can be found again and again in JMF programs and provide the backbone of those programs.

The next chapter discusses each of the classes in depth. However the following list serves as a reference to many of those backbone classes and what roles they serve in programs, without becoming overly cluttered with details.

`AudioFormat`— Information about an audio format including sampling rate and quantization level.

`CaptureDevice`— Interface defining behavior that all capture devices (for example, cameras) must possess.

`CaptureDeviceInfo`— Information about a particular capture device, including the formats supported.

`CaptureDeviceManager`— Manager aware of all the capture devices on the system and capable of providing information about them or, for example, a list that supports a particular format.

`Clock`— Interface defining JMF's fundamental time model. Key classes such as `Players` and `Processors` implement this interface.

`Codec`— Interface supporting the processing of media data from one format into (typically) another.

`Controller`— Interface built on `Clock` that defines the five states of stopped time (see the next section).

`ControllerListener`— Interface defining a listener for `Controller` generated events. Because `Controllers` include `Players` and `Processors`, this is a vital interface that is implemented somewhere in just about every JMF program.

`Controls`— An interface specifying a means of obtaining a control for an object.

`DataSink`— Interface for accepting data and rendering it to some source such as a file.

`DataSource`— Class providing a simple protocol for managing media arriving from a particular source (for example, a file).

`Demultiplexer`— Interface defining a processing unit that accepts a single input stream and outputs the demultiplexed tracks that composed the stream.

`Effect`— An interface defining a media processing unit that accepts a buffer of data, processes it in some way (but doesn't change its format), and outputs the processed buffer. The `Effect` interface supports many types of processing.

`FileTypeDescriptor`— Defines the different content type (architectures) supported.

`Format`— An abstraction of the format of a media object without all the encoding specific details.

`Manager`— Central manager or access point for obtaining resources such as `Players`, `Processors`, `DataSources`, and `DataSinks`.

`MediaEvent`— A `Parent` event class for all media events (for example, `ControllerEvent`).

`MediaLocator`— A means of specifying the location of media content. Used in the creation of players and data sources and sinks.

`Multiplexer`— A processing unit that accepts multiple input tracks and interleaves them to produce a single output container format.

`Participant`— A participant in an RTP session: a sender or receiver.

`Player`— An object for rendering (playing) and controlling (for example, stopping, changing rate of play) a media object.

`PlugIn`— An interface defining a generic plug-in that processes media data in some manner.

`Processor`— An extension to the `Player` interface, the `Processor` defines an object capable of processing and controlling a media object.

`PullDataSource`— A media source from which the data must be pulled (for example, a file).

`PushDataSource`— A media source from which the data is streaming (for example, an RTP session).

`Time`— An object that defines time to nanosecond precision.

`TimeBase`— A constantly ticking source of time.

`VideoFormat`— Format information about video data including frame rate.

Time—A Central Concept

Not surprisingly, after all it is called time-based media, time is a key concept in the JMF. Thus it is important that programmers who employ the API have a good understanding of the model of time that JMF uses.

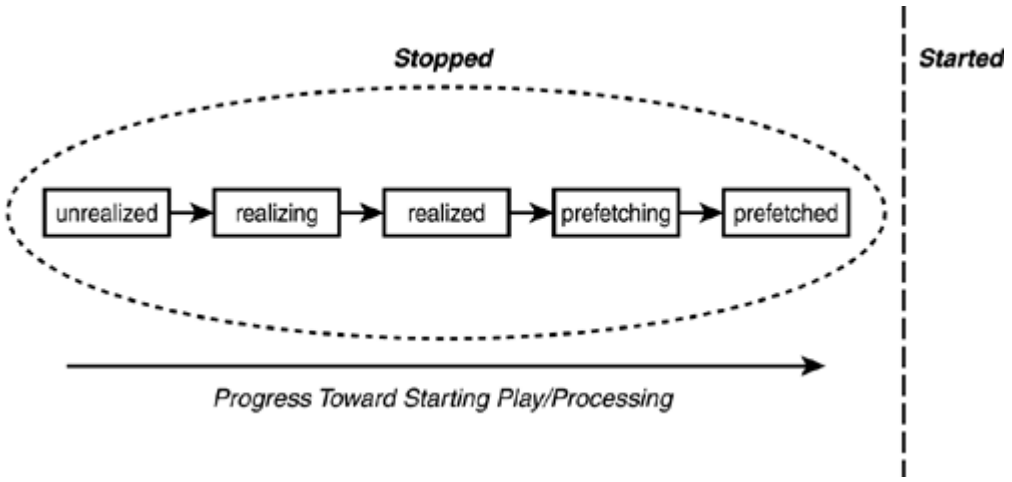
JMF provides a layered model of time. At the bottom-most layer exists an exact representation of an instant in time at the nanosecond level. The next layers support the concept of a constantly ticking source of time: a clock. The highest layers support abstractions of time as being started or stopped and being in one of a number of potential states. These layers allow a programmer to exercise timing control (for example, alter play rate) as well as keep track of time-based processes at the level

appropriate for the task. In particular, the abstracted higher-level models of time, as being one of a number of states, are particularly important for most JMF programs.

This section broadly discusses the high-level, abstract models of time because these are the most important in gaining a conceptual understanding of the functioning of a JMF program. The next chapter contains a far more detailed discussion of the time model of the JMF and the classes that support that model.

Controlling the processing (for example, playing or transcoding) of time-based media might, at first glance, appear to be either a process that is started (ongoing or happening) or stopped (yet to begin or finished). Certainly that division between started and stopped is true. However, initiating control is generally not instantaneous; indeed, it can be quite time-consuming. Resources generally need to be gathered. For instance, a file might need to be read, a socket opened, or a buffer filled. For this reason, the JMF divides stopped time into five exclusive states. In that way, JMF programs are better capable of managing the asynchronous and perhaps lengthy preparation tasks involved in handling time-based media. Figure 7.15 shows those five states and their relationships to one another.

Figure 7.15. The five (stopped) states a `Controller` transitions goes through before it is ready to start.



These five states represent the life cycle of a media `Controller` (for example, `Player`) from creation to being ready to start. Transitions that forward toward a Prefetched (ready to start) state are under program control, whereas errors or other events might lead to a transition in the other direction. Programs (objects) might be informed of state changes by adding themselves as listeners for such events. In that manner, a program can initiate operations as well as respond timely and appropriately as those operations move through their various stages.

The five controller states that subdivide stopped time are

Unrealized— The controller has been created but hasn't even begun to perform its task by gathering any resources. All controllers start in this state.

Realizing— The controller is acquiring information about the resources it needs to function. This transition state should result in the controller becoming Realized, although the time taken is unknown.

Realized— The controller has gathered information about all the resources it needs to perform its task. Indeed it is likely to have acquired all those resources necessary, which would not entail tying up an exclusive system resource (for example, grabbing a hardware device). Realized is a steady state; it is moved past when the controller has prefetching initiated.

Prefetching— The processor is performing start-up processing such as filling buffers or acquiring hardware resources. This is a transition state that should result in the controller becoming Prefetched, although the time taken is unknown.

Prefetched— The controller has acquired all necessary resources, performed all prestartup processing, and is ready to be started.

Bare Bones Player Applet—A First Applet Using JMF

This final section introduces a simple bare bones applet that illustrates how simply features of the JMF can be used.

`BBPApplet` (Bare Bones Player Applet) is an applet that will play a single media object. The name of the media object (file) is specified as a parameter in the HTML file (within the applet tag) that contains the applet. The applet is written in a minimalist fashion: Only those necessary features of the JMF are employed, and nothing fancy is done with regard to GUI design. Subsequent examples in the next chapter will be more sophisticated.

Listing 7.4 is the source of the applet, which can also be found on the book's companion Web site (www.samspublishing.com). What should be clear is how small the applet is: It consists of just two methods, each of fewer than a dozen lines. Yet the result is an applet capable of playing any number of a range of different video and audio formats while also giving the user direct control over that playback through a control panel. Figure 7.16 shows the applet in action playing a synthetically generated video. Also shown are the Media Properties and PlugIn Viewer windows that were raised through the BBPApplet's controls.

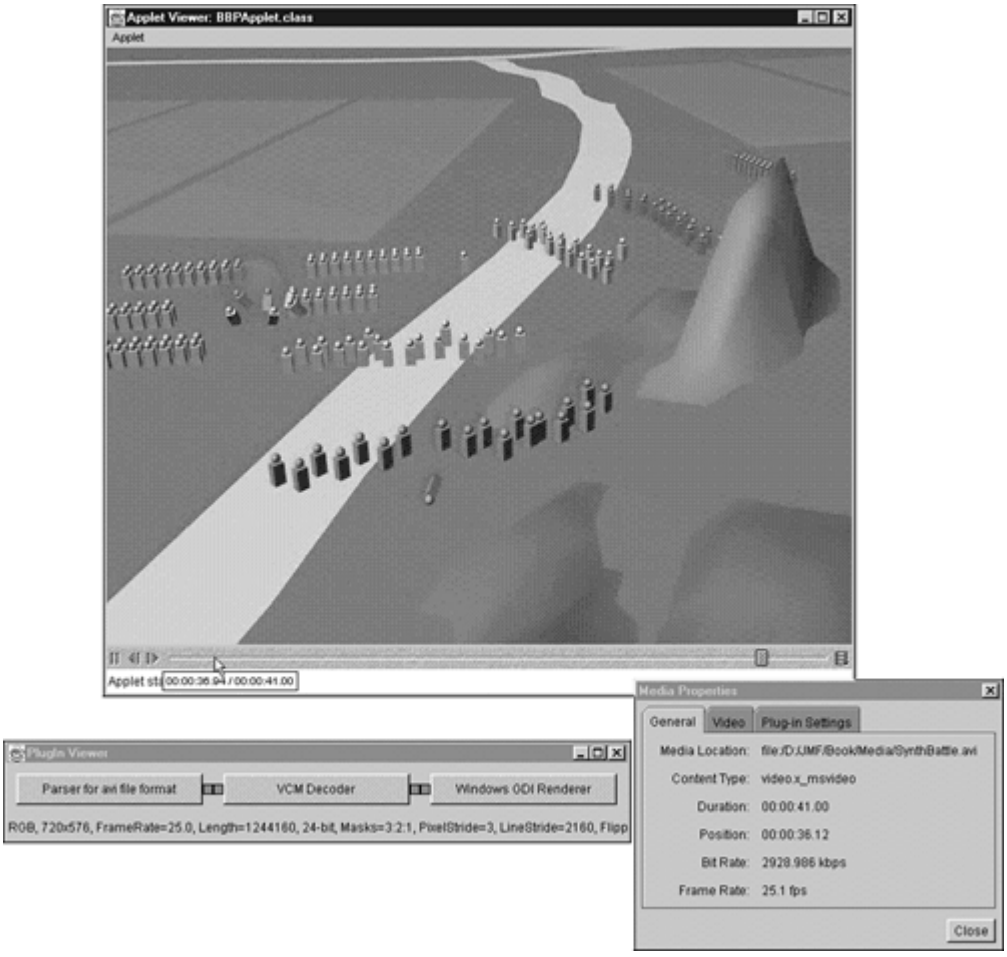
Listing 7.4 BBPApplet (Bare Bones Player Applet)

```
/**************************************************************
* A "Bare Bones" Player Applet (BBP Applet) that will play
* the media object indicated in the "media2Play" property
* of the Applet tag.
*
*<p>The applet demonstrates the relative ease with which
* the JMF can be employed, particularly for playing. The
* applet is a minimal player, placing the controls for the
* player and the visual component for the played object
* within the Applet. The object plays once, but can be
* controlled by the user through the control panel provided.
*
```

```
*<p>The tag for the Applet should look something like:
*
* <!-- Sample HTML
* <applet code="BBPApplet.class" width=300 height=400>
* <param name="media2Play" value="myVideo.mpg">
* </applet>
* -->
*
*@author Spike Barlow
**********************************************************/

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;
import java.util.*;
import javax.media.*;
public class BBPApplet extends Applet implements ControllerListener {

    /**************************************
    * Object to play the media. Only attribute
    * that the Applet really needs.
    **************************************/
    protected Player    player;

    /**************************************
    * The name of the media to be played.
    **************************************/
    protected String    nameOfMedia2Play;

    /**********************************************
    * Name of the Property field within the
    * applet tag indicating the name of the media
    * to play.
    **********************************************/
    private static final String   MEDIA_NAME_PROPERTY = "media2Play";

    /**********************************************
    * Object describing the location of the media to
    * be played.
    **********************************************/
    protected MediaLocator      locator;



/*************************************************************
* Initialise the applet by attempting to create and start a
* Player object capable of playing the media specified in the
* applet tag.
*************************************************************/
public void init() {

    setLayout(new BorderLayout());
    setBackground(Color.lightGray);
    try {
        nameOfMedia2Play = (new URL(getDocumentBase(),
                            getParameter(MEDIA_NAME_PROPERTY)
                            )).toExternalForm();
```

```
        locator = new MediaLocator(nameOfMedia2Play);
        player = Manager.createPlayer(locator);
        player.addControllerListener(this);
        player.start();
    }
    catch (Exception e) {
        throw new Error("Couldn't initialise BBPApplet: "
                + e.getMessage());
    }
}



/*************************************************************
* Respond to ControllerEvents from the Player that was created.
* For the bare bones player the only event of import is the
* RealizeCompleteEvent. At that stage the visual component and
* controller for the Player can finally be obtained and thus
* displayed.
*************************************************************/
public synchronized void controllerUpdate(ControllerEvent e) {

    if (e instanceof RealizeCompleteEvent) {
        add(player.getVisualComponent(),"North");
        add(player.getControlPanelComponent(),"South");
        validate();
    }

}

}
```

Figure 7.16. The `BBPApplet` (Bare Bones Player Applet) playing a synthetically generated video.



Several objects from the JMF API are employed in the applet. Chief among these is the `Player` object `player`. It is the only real attribute that the applet must possess because both methods need to refer to it. In order to create the `Player` object, a `MediaLocator` object `locator` is constructed from the user-specified filename that contains the media to be played. After that object is constructed, it is passed to the `Manager` class so that the `Player` can be created.

Several features typical of JMF programming can be found in this small example. The applet employs event-driven programming in order to determine when the `Player` is realized, and thus when a visual component and control panel for that `Player` can be obtained. Further, `try/catch` blocks are used to enclose the code in the `init()` method that could conceivably throw an exception, although for this short example nothing clever is done about an exception.

The fundamental algorithm of the applet can be written as follows:

1. Obtain the name of media file to play (using the parameter/property tag).

2. Convert filename to a `MediaLocator` object.

3. Create a `Player` object for the `MediaLocator`.

4. Listen to the `Player` object for events.

5. Start the `Player` object.

6. Wait for the `Player` object to become realized.

7. At the time of realization:

    - Obtain the `Player` object's visual component and place at top of applet.

    - Obtain the `Player` object's control component and place at bottom of applet.

Steps 1–5 all occur within the `init()` method, which is called when the applet is initialized. Step 6 is an expression of the event-driven nature of the program: There is no loop waiting or checking for realization to occur. Rather, it will be signaled by an event generated by the `player`. Step 7 occurs within the `controllerUpdate()` method, which is called when the `Player` object generates an event. If that event is a `RealizeCompleteEvent`, the `Player` object's components are obtained and added to the applet.

Two particulars of Listing 7.4 are worth further explanation. First, the initial line inside the `try` block of the `init()` method chains several steps together that result in a fully qualified name for the media object to play. As a first step in that process the value of the applet property that specifies the name of the file is obtained. That is combined with the document base of the applet, typically the same directory in which the applet is found, to produce a `URL` (object) that is then transformed back to a `String` (suitable for the `MediaLocator` constructor). Second, a layout manager—`BorderLayout`—is employed to ensure reasonable positioning of the controls and display on the screen. This allows the visual component of the player to be added to the top of the applet, whereas the controls are added to the bottom.

It is worth noting that a more complete example would likely override such methods as `start()` and `stop()` in order to control or free resources (that is, the `Player` object) as appropriate. Further, detecting and responding to the other types of events generated by the `Player` object could be used to provide additional functionality (for instance, looping play).

Summary

This chapter serves as an introduction to time-based media and the Java Media Framework (JMF), setting the stage for the next two chapters, which delve into the details of the JMF API.

The first half of the chapter provides a general and broad introduction to the concepts and practice of time-based media. The common features of all time-based media are covered before audio and video

are addressed separately. A recurring theme is the high bandwidth demands of time-based media and hence the needs for compression. The alternatives in content types (architectures) and codecs for both audio and video were discussed.

The second half of the chapter introduces the JMF API. The potential of and support provided by the API is broached first. That is followed by an overview of the different levels of complexity at which the JMF can be employed together with common programming approaches when using the API. Finally, the key classes of the API are surveyed along with a synopsis of the JMF model of time. The chapter concludes with a short applet that plays media files, showing how simple it can be to write JMF programs.

Chapter 8. Controlling and Processing Media with JMF

IN THIS CHAPTER

- Detailed Time Model
- The Control and Processing Chains
- Managing the Complexity
- It's All About Control
- Sourcing Media and Media Format
- `MediaHandler`
- Playing Media
- Conserving Media
- `PlugIns`
- Processing Media
- Media Capture

This chapter covers control of time-based media with the JMF (Java Media Framework) API. It serves as the core chapter of three chapters that cover time-based media: Chapter 7, "Time-Based Media and the JMF: An Introduction," serves as an introduction, and Chapter 9, "RTP and Advanced Time-Based Media Topics," covers advanced and specialized topics.

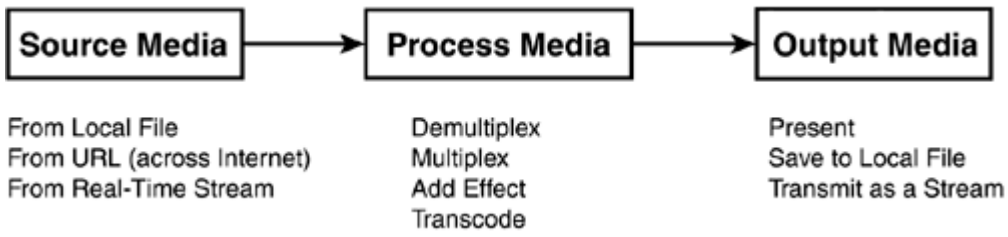
In the context of this chapter, control is used in the broadest sense to cover all actions concerning time-based media. That encompassing definition includes what is traditionally considered processing: decoding, encoding, transcoding (decoding from one format and encoding as another), effects, filters, multiplexing, and demultiplexing, as well as sourcing the data itself: capturing, reading from a file, and outputting (presentation or saving).

In fact, the fundamental approach to control can be seen as falling into three steps:

1. Source the media.
2. Process the media.
3. Output the media.

Figure 8.1 shows these three steps in control.

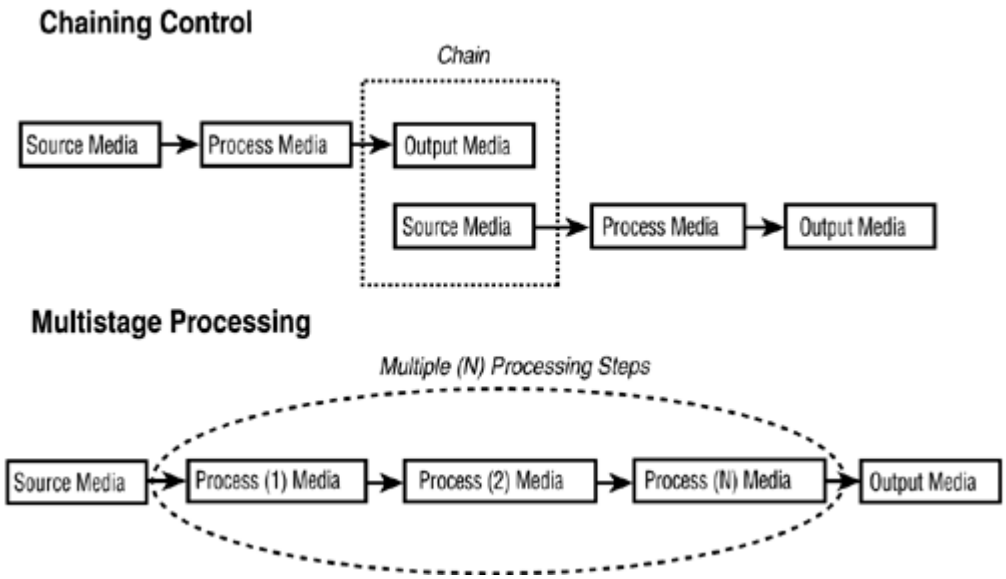
Figure 8.1. Three steps in control of media with the JMF.



This control can be chained: The output of one control stage can serve as the input to another. For instance, one control segment might secure a movie streaming across the Internet and demultiplex it into separate audio and video tracks that are saved to separate files. A second control module might then take that output audio data and add a reverb effect to one segment before multiplexing it back with the video track and saving the resulting movie.

Step 2, processing, can be an involved multistaged action, perhaps involving effects, codecs, multiplexers, and demultiplexers. Figure 8.2 is a schematic diagram showing multistage processing and chaining of control.

Figure 8.2. Chaining the processing of media.



The chapter falls into three broad modules, each consisting of the following sections:

- Major steps and classes with roles in the control chain
- Processing
- Media Capture

The first module is a grab bag of topics covering the approach used by JMF in achieving control over media and is a necessary prelude to the details of the later modules. In particular, it includes a more detailed discussion of the JMF model of time (than that presented in the previous chapter) and an introduction to the key manager classes, as well as discussions of Controls, DataSources, and DataSinks.

The second module concerns processing and begins with a discussion of the expanded stop-time categories that the JMF employs for processing. The topics of the earlier sections of the chapter are then illustrated with a number of processing examples (such as transcoding).

The final module covers the topic of media capture—sampling audio or video directly from devices attached to the computer.

Detailed Time Model

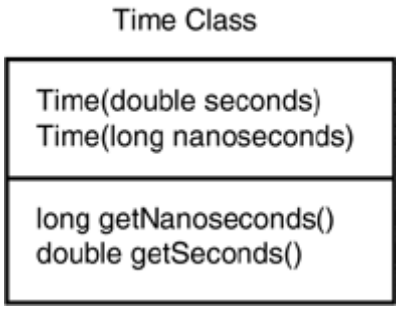
As discussed in Chapter 7, the JMF employs a layered approach to its representation of time. At the low-level end of the time model are classes for representing time to nanosecond accuracy. At the high-level end of the model, the JMF sees controllers as being in one of a number of states that are transitioned between under program control.

Low-Level Time: `Time` and `TimeBase` Classes

Two classes, `Time` and `SystemTimeBase`, and one interface, `TimeBase`, detail the JMF's low-level model of time.

At the bottom of the hierarchy and also perhaps the most fundamental is the `Time` class. A `Time` object represents a particular instant in time to one nanosecond accuracy. Figure 8.3 shows the `Time` class's constructors and public methods.

Figure 8.3. The `Time` class.



`Time` objects are often returned by methods used to query the temporal status of another object; for instance, the amount of elapsed play time on some media.

Similarly, `Time` objects can be employed to alter the temporal properties of an object. For instance, to specify a particular point in time from which to start playback of media, a `Time` object might be employed as follows:

1. Construct a `Time` object with a value, specified in seconds or nanoseconds, as the offset from the start of the media at which play should commence.

2. Pass that object to the `Player` object's `setMediaTime()` method.

The `Time` class also possesses a special constant `TIME_UNKNOWN`. This constant finds applications in contexts in which an object might be asked the duration of the media it is associated with, but its length hasn't, or cannot, be ascertained.

The `Time` class specifies a single instant in time, whereas time-based media, by its nature, is dynamic and time varying. JMF's support for ticking (at 1 nanosecond per tick) time comes in the form of the `TimeBase` interface. The `TimeBase` interface is an important one, and one that is implemented by a number of important classes. (More accurately, it is subsumed in other key interfaces such as `Controller` and `Player`, which extend the interface.) The `TimeBase` interface defines only two methods, both of which are used to query the current time of the `TimeBase` object. The `getTime()` method returns a `Time` object. An alternate means of obtaining the same information is the `getNanoseconds()` method, which returns a long. There is no provision in a `TimeBase` for altering time: It can only be queried regarding its current state.

As a default implementation of the `TimeBase` interface, JMF provides the class `SystemTimeBase`. `SystemTimeBase` has a single empty constructor and only the two methods defined in the `TimeBase` interface. Alternatively, the system time base can be obtained through the `Manager` class's `getSystemTimeBase()` method.

The `Clock` Interface

Those classes that implement the `TimeBase` interface provide a constantly ticking, unalterable source of time. However, controlling media means providing control over the temporal properties of that media: being able to start or stop the media at arbitrary locations as well as control its rate (for example, fast forward or rewind on a player). The `Clock` interface is the means by which that is achieved, and it is implemented by objects that support the JMF model of time.

In many ways, the `Clock` interface is pivotal both as the cement between the low-level and high-level time models and as a core interface of the API. The `Controller`, `Player`, and `Processor` interfaces, all central to the functionality of the JMF, extend `Clock`.

`Clocks` are typically associated with a media object. Indeed, control over media such as playing or processing entails having a `Clock` associated with that media. (Because `Controller`, `Player`, and `Processor` interfaces extend `Clock`, the `Player` or `Processor` object is also the clock.) A `Clock` serves as both the timekeeper for its media and also a means of altering and adjusting the time of that media. The time a clock keeps is known as the media time.

Clocks achieve their dual task of monitoring and altering the time flow of their associated media by employing a `TimeBase`. As noted, `TimeBase` objects represent constantly ticking and unalterable time. Therefore, the `Clock` provides a remapping or transform from the `TimeBase` time to that associated with the media. This is a simple linear transform requiring three parameters: the rate (for example, of play), the media start time, and the time base start time. From these, the media time can be determined as follows:

```
media_time = media_start_time + rate x (time_base_time - time_base_start_time)
```

The meanings of the previous terms are as follows:

Media time— The media's own position in time. For instance, if an audio clip was one minute in length, its media time would range between 0 and 60 seconds.

Media start time— The offset within the media from which play is started. If play starts from the beginning of the media, this value is `0`. If it was started seven and a half seconds in, this value would be `7.5`.

Rate— The rate of time passage for the media. A rate of `1` represents normal forward passage (for example, play), whereas a value of `-5` would represent a fast rewind.

Time base time— The time of the `TimeBase` object that the `Clock` incorporates. This starts ticking (increasing) as soon as the `Clock` object is created and never stops.

Time base start time— The time of the `TimeBase` object at which the `Clock` is started and synchronized with the `TimeBase`. For instance, the `Clock` might be started 3.2 seconds after the `Clock` was created (and hence the `TimeBase` was also created and started ticking). Hence, the time base start time would be 3.2 seconds.

A `Clock` is in one of two possible states: `Started` or `Stopped`. A clock is started by making the `syncStart()` method call. The `syncStart()` method accepts a single argument being the time base start time from which the `Clock` should be started. Once the `Clock`'s `TimeBase` object reaches that time, the clock will synchronize with the `TimeBase` and enter the `Started` state. This mechanism allows a `Clock` to be set to start at some future time (or at the current time by passing the `syncStart()` method the `Clock`'s own `TimeBase` object's current time). Any changes to the media (start) time and rate must be performed before a `Clock` enters the `Started` state. Attempting to use the methods that carry out these operations on a `Clock` in the `Started` state will result in a `ClockStartedError` being thrown. Thus, the usual steps in starting a clock are

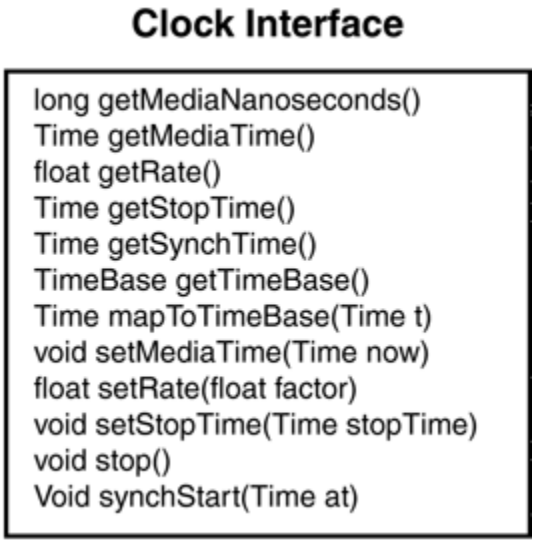
1. Stop the clock if it is currently started.
2. Set the media (start) time of the `Clock`.
3. Set the rate of the `Clock`.
4. `syncStart()` the `Clock`.

A `Clock`'s initial state is `Stopped`. After a clock is `Started`, it can be stopped in one of two ways. It can either be stopped immediately with the `stop()` method, or a media stop time can be set: Once the media time reaches (or if it has already exceeded) that time, the `Clock` will stop.

Finally, it is worth noting that it is possible to synchronize two or more `Clocks` by setting them to use the same `TimeBase` object. The `Clock` interface exposes methods for getting and setting the `TimeBase` object associated with the clock.

Figure 8.4 shows all the methods of the `Clock` interface. Besides those already discussed, often used methods are `getMediaTime()` and `getMediaNanoseconds()`. For instance, these might be called repeatedly as media is being played in order to provide some feedback on elapsed time for the viewer. Similarly, `setRate()` and `setMediaTime()` are used to provide user control in playback scenarios, but might only be called on a stopped clock.

Figure 8.4. The `Clock` interface.



High Level Time: The `Controller` Interface

The `Controller` interface directly extends `Clock` in three areas:

- Extends the concept of `Stopped` into a number of states concerning resource allocation, so that time-consuming process can be better tracked and controlled.
- Provides an event mechanism by which the states can be tracked.
- Provides a mechanism by which objects providing further control over the controller can be obtained.

It is on top of the interface that the commonly used `Player`—which is used in the last example in Chapter 7—and `Processor` interfaces sit.

As we explained in the previous chapter, achieving the state where the control, processing, or play of media can be started isn't an instantaneous operation. Resources need to be gathered in order to support that control. Tasks involved in resource gathering include opening files for reading, filling buffers, or gaining exclusive control of hardware devices (for example, a hardware decoder). This point is illustrated in the next subsection in which the time taken to gather resources so that a video can be played is shown.

The `Controller` interface subdivides the `Stopped` category of `Clock` into five stages that reflect the state of preparedness of the `Controller`: how close it is to being capable of being started. Those five states, in order of least prepared through prepared to start, are

`Unrealized`— A `Controller` that has been created but hasn't undertaken any resource gathering.

`Realizing`— A transition state reflecting the fact that the `Controller` is gathering information about the resources needed for its task, as well gathering resources themselves.

`Realized`— A steady state reflecting a `Controller` that has gathered all the nonexclusive resources needed for a task.

`Prefetching`— A transition state reflecting the fact that the `Controller` is gathering all resources needed for its task that weren't obtained in the realizing state. Typically this means acquiring exclusive usage resources such as hardware.

`Prefetched`— The `Controller` has acquired all necessary resources, performed all pre-startup processing, and is ready to be started.

The `Controller` interface provides program control for the movement between these states via a set of methods. Similarly, the `Controller` interface allows for program monitoring of those transitions via an event system. Objects can implement the `ControllerListener` interface and thus be sent events as the `Controller` transitions between the various states. Figure 8.5 shows the methods and associated transitions between states. Figure 8.6 shows the events that are generated as a `Controller` transitions between its states.

Figure 8.5. `Controller` methods that cause state transitions.

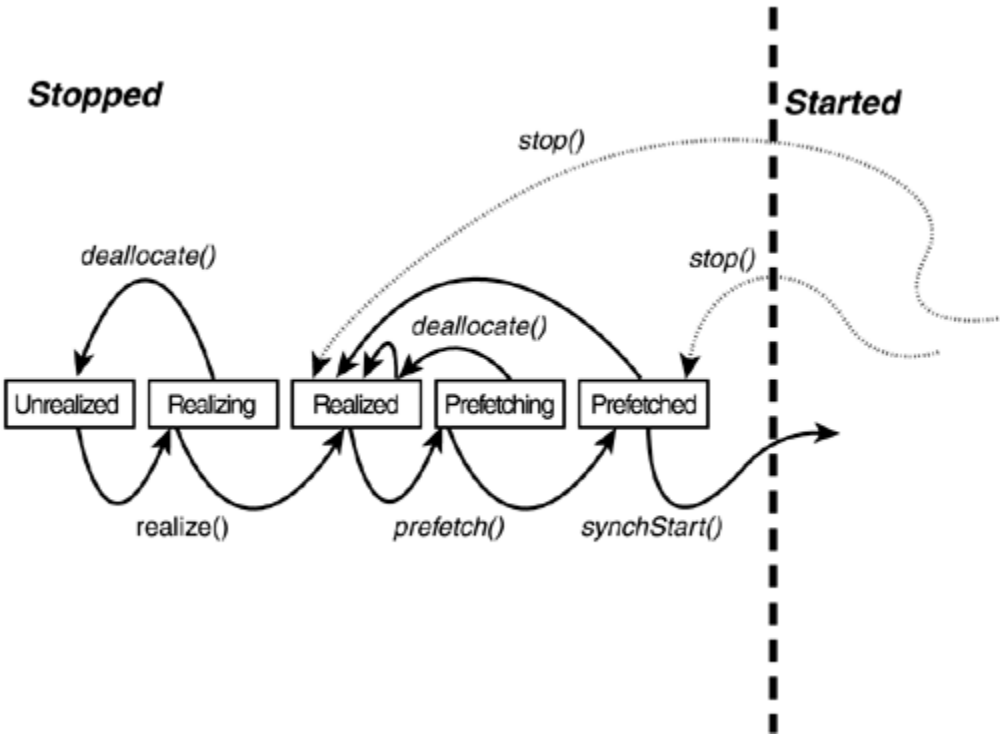
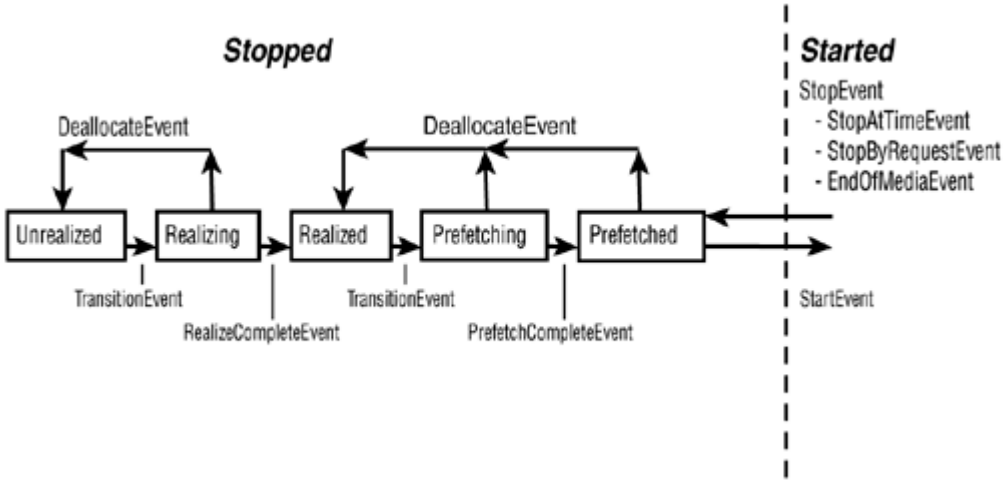


Figure 8.6. `Events` generated as a `Controller` transitions between states.



As shown in Figure 8.5, a `Controller` has five methods for controlling transition between states. The forward motion methods are `realize()`, `prefetch()`, and `synchStart()` (from the `Clock` interface) for moving the `Controller` into a more prepared, and finally `Started`, state. They are asynchronous—they return immediately, but the engendered action generally takes some time to complete. When the transition is complete (or interrupted by a `stop()` or other such call), an event is posted. The reverse direction methods are `stop()` and `deallocate()`. These are synchronous methods. `stop()` is used to stop a started `Controller`. The `Controller` transitions to `Prefetched` (or in some cases where resources must be relinquished: `Realized`) and might subsequently be restarted. `deallocate()` frees the resources consumed by a `Controller` and should be used for that purpose (for example, in the `stop()` method of an applet). `deallocate()` cannot be called on a `Started Controller`; it must be stopped first. `deallocate()` returns a `Controller` to the `Realized` state if it is in that state or greater (a state closer to `Started`); otherwise, the `Controller` returns to `Unrealized`.

A `Controller` posts events about its state changes. Those objects wanting to be informed about `Controller` events must implement the `ControllerListener` interface. The `ControllerListener` interface consists of a single method:

```
public synchronized void controllerUpdate(ControllerEvent e)
```

Objects communicate their desire to be sent a `Controlleris` events by calling that `Controlleris` `addControllerListener()` method.

The events posted by a `Controller` fall into one of four categories:

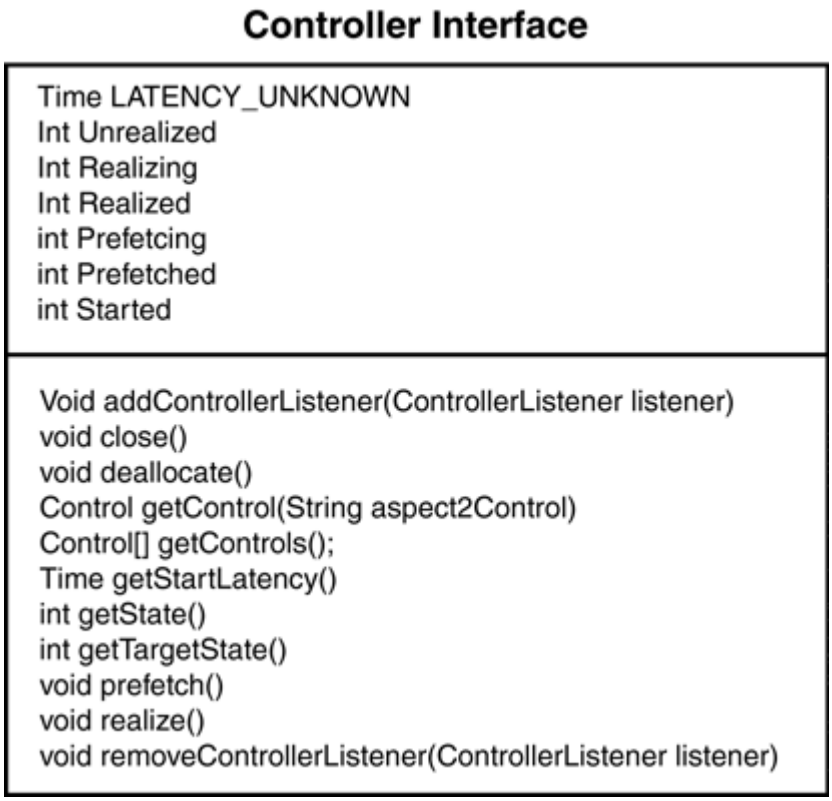
- Life cycle transitions
- Method acknowledgements

- Status change information
- Error notification

The `TransitionEvent`, or a subclass such as `EndOfMediaEvent`, is a `Controller`'s means of reporting state changes. The method acknowledgement events `RealizeCompleteEvent`, `PrefetchCompleteEvent`, `StartEvent`, `DeallocateEvent`, and `StopByRequestEvent` are used to communicate the fulfillment of the corresponding methods—for example, `realize()`—called on the `Controller`. There are three status change events: `RateChangeEvent`, `StopTimeChangeEvent`, and `MediaTimeSetEvent` that inform the listener of changes in rate, stop time, and when a new media time is set. Errors fall under the `ControllerErrorEvent` class and include `ResourceUnavailableEvent`, `DataLostErrorEvent`, and `InternalErrorEvent`. Other errors are thrown as exceptions. For instance, attempting to `syncStart()` a `Controller` before it achieves the `Prefetched` state will result in a `NotPrefetchedError` being thrown.

Figure 8.7 shows all the methods and constants of `Controller` that aren't inherited from the `Clock` or `Duration` interfaces. Among the important methods of the interface not discussed previously are `close()`, `getStartLatency()`, `getControl()`, and `getControls()`. `close()` is used to release all resources and cease all activity associated with a `Controller`. The `Controller` can no longer be employed (its methods called) after it has been closed. The `getStartLatency()` method returns an estimate of the amount of time required in a worst-case scenario before the first frame of data will be presented. It is used to provide an estimate for `syncStart()` calls. The estimate is more accurate if the `Controller` is in the `Prefetched` state. The `getControl()` and `getControls()` methods provide a means for obtaining Control objects. These can be used to alter the behavior of the `Controller`. Controls and `Controllers` are two different things despite their unfortunate similarity in name. Controls are discussed in a subsequent section.

Figure 8.7. The `Controller` interface.



Timing a Player

In order to illustrate the concepts covered in this section and the inter-relationship between the high- and low-level models of time that the JMF supports, the simple player applet `BBPApplet` from the previous chapter was modified slightly.

The modification consisted of time-stamping and printing every `Controller` event that was received by the applet. This provides a map through time of the course of the player from the instant it is started, until the time the media being played (an mpeg video) finishes.

The modification itself is simple, and rather than reproducing the code of the entire applet again, only the changes made will be discussed. A `SystemTimeBase` object was constructed immediately prior to the `Player` being started. Each time the `controllerUpdate()` method was entered, the `SystemTimeBase` object was queried as to the time, and that value plus the event that was received were printed to the screen. The three steps were

1. Where other attributes such as Player are declared, declare an additional attribute—an object of type `SystemTimeBase`.

```
2.   // The object to be used to timestamp Controller events.
3.   protected SystemTimeBase    timer;
```

2. Immediately prior to the `player.start()` asynchronous call, create the `SystemTimeBase` object in the `init()` method. From that instant forward, the timer will continue to tick with (potentially) nanosecond accuracy.

```
3.   timer = new SystemTimeBase();
4.   player.start();
```

3. In the `controllerUpdate()` method, immediately query the timer object as to its time, and print that (converted into seconds) and the event that was received.

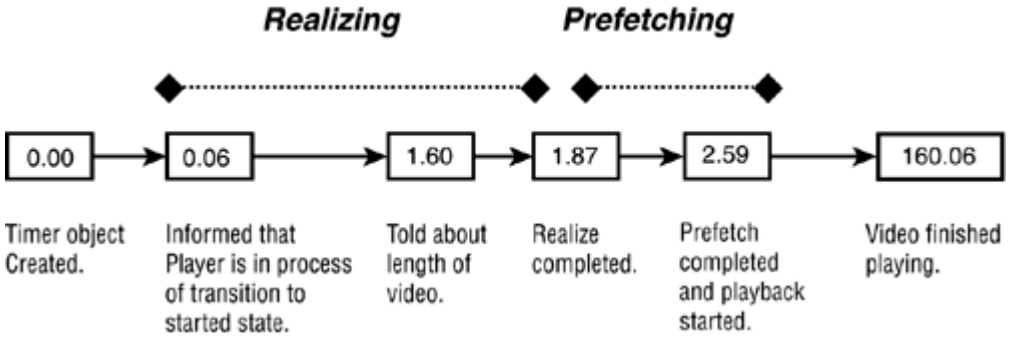
```
4.   // Print the time the event was received, together with the event,
5.   // to the screen.
6.   System.out.println(""+(double)timer.getNanoseconds()/Time.ONE_SECOND
7.       + ": " + e);
```

The timing data output when the applet played an mpeg video somewhat over two minutes in length is reproduced in Listing 8.1. Each Controller event received by the class (generated by the Player) has been time stamped and output. The events occur in chronological order with a timestamp (in seconds) being the first piece of information on the line, that being followed by the event itself. Figure 8.8 shows the state transitions through time, distilling the most important information.

Listing 8.1 Timing information output by the modified BBPApplet (Bare Bones Player Applet)

```
0.06: javax.media.TransitionEvent[source=com.sun.media.content.video.mpeg.Handle
r@275d39,previous=Unrealized,current=Realizing,target=Started]

1.6: javax.media.DurationUpdateEvent[source=com.sun.media.content.video.
mpeg.Handler@275d39,duration=javax.media.Time@10fe28
1.87: javax.media.RealizeCompleteEvent[source=com.sun.media.content.video.mpeg.H
andler@275d39,previous=Realizing,current=Realized,target=Started]

2.59: javax.media.TransitionEvent[source=com.sun.media.content.video.mpeg.Handle
r@275d39,previous=Realized,current=Prefetching,target=Started]

2.59: javax.media.PrefetchCompleteEvent[source=com.sun.media.content.video.mpeg.
Handler@275d39,previous=Prefetching,current=Prefetched,target=Started]

2.59: javax.media.StartEvent[source=com.sun.media.content.video.mpeg.Handler@275
d39,previous=Prefetched,current=Started,target=Started,mediaTime=javax.media.Tim
e@36e39f,timeBaseTime=javax.media.Time@19dc16]

160.06: javax.media.EndOfMediaEvent[source=com.sun.media.content.video.mpeg.Hand
ler@275d39,previous=Started,current=Prefetched,target=Prefetched,mediaTime=javax
.media.Time@60a26f]

160.06: javax.media.DurationUpdateEvent[source=com.sun.media.content.video.mpeg.
Handler@275d39,duration=javax.media.Time@484a05
```

Figure 8.8. Timeline for the events a `Player` received when presenting a video.



What is clear from the output and figure is that the steps in preparing to play media are lengthy, particularly from the perspective of a computer that executes millions of instructions per second. The realizing step took over a second and a half, whereas the prefetch step required nearly three quarters of a second, all for media stored locally on the hard disk.

The Control and Processing Chains

Four key classes play central roles in all JMF control and processing. These four classes form links in a chain who's first element is always a source of media. Depending on the particular task that media is then handled (such as playing or processing), that handling might be the end result itself (such as a `Player`); or the handling might result in a new data source or even a persistent media object (for example, a new media file).

Figure 8.9 shows the key classes involved in playing media. The media is sourced through a `DataSource` object. That is used to create a `Player` object, which renders the media to the appropriate hardware devices.

Figure 8.9. Playing media with the JMF: The media is sourced through a `DataSource` and played with a `Player`.

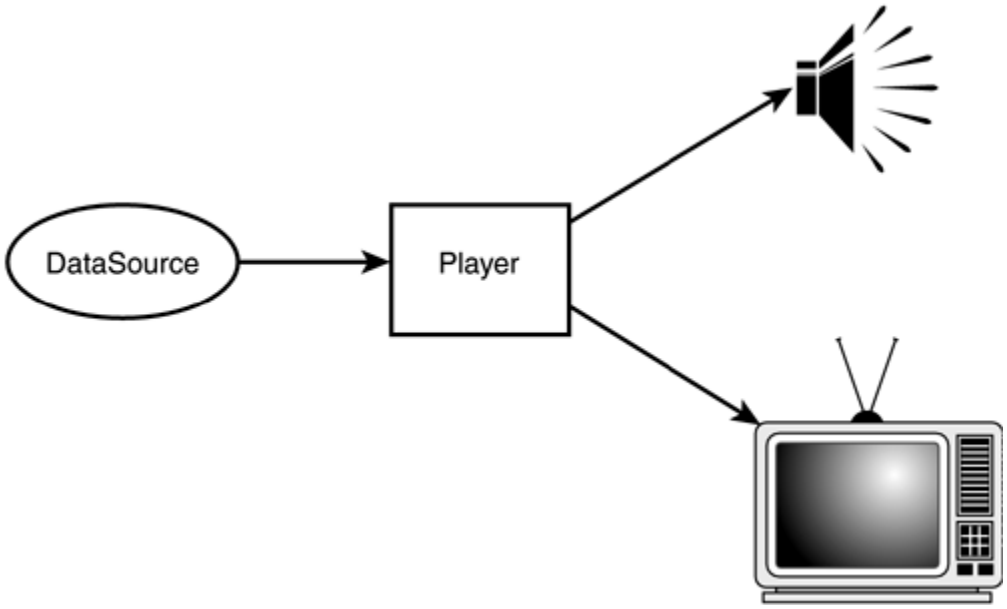


Figure 8.10 shows the key classes involved in processing media. As with the playing of media, the first step in processing is sourcing the media through a `DataSource` object. That is then processed—an encompassing term that includes multiplexing/demultiplexing, effects, and transcoding. The result of the processing is another `DataSource`. Illustrating that processing can be chained, the `DataSource` produced by a first stage of processing can serve as the input `DataSource` for a second round of processing. Similarly, processing could be performed and the results fed to a `Player` object for final rendering (display).

Figure 8.10. Employing a `Processor`: Input arrives from a `DataSource` and the `Processor` produces a `DataSource` as the result of its processing.

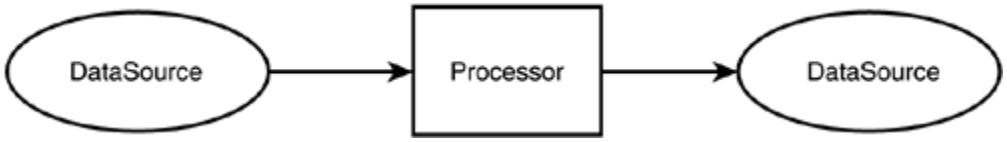


Figure 8.11 shows the key classes concerned when time-based media is to be produced that exists past the termination of the JMF program that produced it. As in all cases, the original media is sourced through a `DataSource` object. That is then typically processed, such as transcoding to a different format, to produce a new `DataSource`. A `DataSink` for that second `DataSource` is obtained so that the media can be preserved (for example, written to a file).

Figure 8.11. Employing a `DataSink` to output data. The `DataSource` produced by a `Processor` serves as the input to the `DataSink`.

The `Player`, `Processor`, `DataSource`, and `DataSink` classes are all `MediaHandlers`. Both the `MediaHandler` interface and the four classes are discussed in their own sections later in this chapter. The means of obtaining instances of these classes under the JMF is centralized through a manager class, which is discussed next.

Managing the Complexity

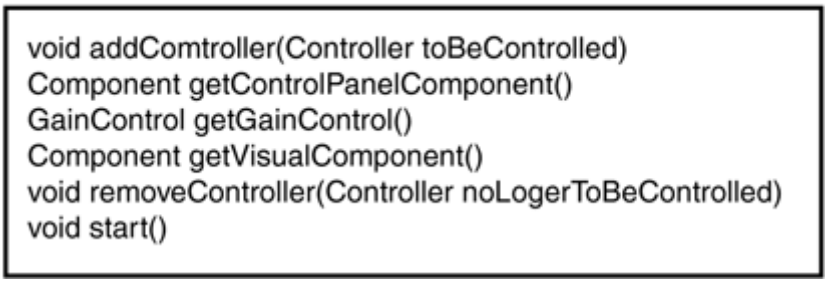
As graphically illustrated in the previous chapter, time-based media is a broad category encompassing not only different types of media (for example, audio and video), but also different content types (for example, QuickTime and AVI), and different formats for compression (for example, MPEG and Cinepak). This leads to a plethora of diverse media that differ at the conceptual level (visual or aural) down to the bit sequence by which they are encoded. Further complexity is added by the multitude of hardware devices from which media can be captured, and to which it can be rendered.

On the other hand, the goal of the JMF is to present a uniform, platform independent interface to controlling, processing, capturing, and rendering media. That means, for example, a single program to play media regardless of particulars of its encoding; not a different program for each type (category x content_type x encoding_scheme) of media.

The JMF successfully resolves these two conflicting items by providing four manager classes who's prime role is to track the numerous classes required to support media handling, while shielding the user from that complexity through the provision of a simple and consistent interface. Effectively, the managers act as brokers or intermediaries between user code and the necessarily complex functionality provided by the JMF. This provides user code with a simple, abstracted model; freeing it of unwanted complexity. Figure 8.12 illustrates that conceptual role of the managers. The `BBPApplet` (Bare Bones Player Applet) from Chapter 7, "Time-based Media and the JMF: An Introduction," provides a good example of this abstraction afforded by the managers. The applet simply requests that the manager create an object (`Player`) capable of playing the media it has indicated. The manager responds with an object suitable for the task. The actual object provided will depend on the content type and format of the media in question. However, from the applet's perspective there is simply a Player object which will do the task. The manager hid all the details of finding and constructing an instance of the appropriate class.

Figure 8.12. Role of manager classes (for example, `Manager`) as registry and shield for user code from the necessary complexity of classes in order to support the multitude of media formats.

Player Interface



The JMF has four manager classes— each with the word Manager in their name, appropriately enough. Each of these classes exposes a number of static methods through which they provide their service. The four classes are as follows:

`Manager`— The central management class from which `Players`, `Processors`, `DataSources`, and `DataSinks` are obtained. This manager is discussed in the next subsection.

`CaptureDeviceManager`— Manager encapsulating knowledge of the capture devices (for example, sound or video capture cards) attached to the machine. This manager is discussed toward the end of the chapter.

`PackageManager`— A manager providing knowledge of and control over the packages that contribute to JMF's functionality. This manager is discussed in Chapter 9, "RTP and Advanced Time-Based Media Topics."

`PlugInManager`— A manager encapsulating knowledge of installed plug-ins, as well as a means of registering new ones. The JMF model of a plug-in incorporates multiplexers, demultiplexers, codecs, effects, and renderers. This manager is discussed in Chapter 9.

Although there are empty constructors for all except the `Manager` class, all methods exposed by the managers are static: They are invoked using the class name, and don't need an instance of the class constructed before they can be invoked.

Hence, for instance, to obtain information about a particular named `CaptureDevice`, the code should be written as follows:

```
String    deviceName = "...";    // Set to the actual name of the device
CaptureDeviceInfo captureDInfo = CaptureDeviceManager.getDevice(deviceName);
```

rather than

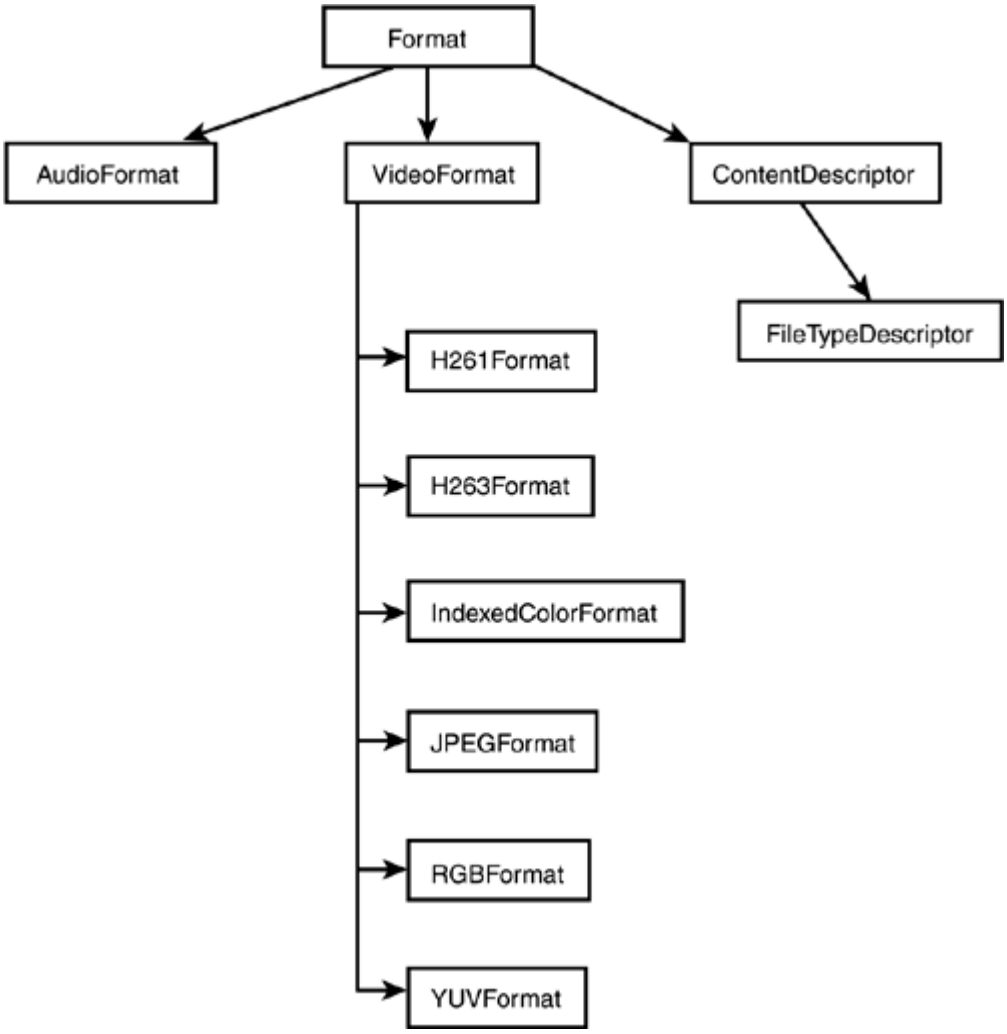
```
String deviceName="...";    // Set to the actual name of the device
CaptureDeviceManager captureDManager = new CaptureDeviceManager();
CaptureDeviceInfo captureDInfo = captureDManager.getDevice(deviceName);
```

The `Manager` Class

The `Manager` class is the single most important manager class, and arguably the most important class in the JMF given its role in creation of `Players`, `Processors`, `DataSinks`, and `DataSources`.

Figure 8.13 shows the methods of the `Manager` class. As can be seen, `Manager` possesses methods for the creation of `DataSources`, `DataSinks`, `Players`, and `Processors`. It has already been noted in the previous section that these four classes play the primary roles in the control and processing of media. Given their significance, the importance of `Manager` as the sole agent of their creation shouldn't be underestimated.

Figure 8.13. The `Manager` class.



Although subsequent sections discuss the `Player`, `Processor`, `DataSource`, and `DataSink` classes, their creation through the `Manager` class is discussed here.

Creating a `Player`

Objects implementing the `Player` interface are used in controlling the playback of media. The `Manager` class provides six methods for the creation of a `Player`: Three create an `Unrealized Player`: `createPlayer()`, whereas the other three create a `Realized` (that is, already partially resourced and more along the path to being ready to start) `Player`: `createRealizedPlayer()`.

The three `createRealizedPlayer()` versions are provided as a means to accelerate the creation of a `Player`. The method calls blocks (that is, the next line of code doesn't execute until the `Player` is `Realized`). This has the advantage that the `Realized` event doesn't need to be listened for by the invoker of the method (and hence methods such as `getVisualComponent()` can be called as the next line).

The three versions of `Player` creation methods (whether for an `Unrealized` or `Realized Player`) accept either a `MediaLocator`, `URL`, or `DataSource` as the single parameter. The steps in creating a `Player` that the `Manager` follows are as follows:

1. Convert the `URL` to a `MediaLocator` (if `URL` based method is used).
2. Create a `DataSource` for the `MediaLocator` (if `DataSource` based method isn't used).
3. Obtain the `Player` that can handle the `DataSource`.
4. Attach the `DataSource` to the `Player`.
5. Return the `Player` object.

The `URL` and `MediaLocator` based creation methods are the more commonly used because they correspond to the typical scenario in which a `Player` is employed: The media is in some location (for example, a file) and needs only to be played. On the other hand, the `DataSource` based method is useful when playing is the end result of a chain that involved other processing (that produced a `DataSource`) as earlier steps.

Thus the typical usage of `Manager` to create a `Player` is as follows:

```
MediaLocator locator = new MediaLocator(...);    // Specify media location;
Player player = Manager.createPlayer(locator);
```

Alternatively, a `Realized Player` could be created with the `createRealizedPlayer()` method as follows:

```
MediaLocator = new MediaLocator(...);    // Specify the media location;
Player player = Manager.createRealizedPlayer(locator);
```

In creating a `Player` object, `Manager` follows a set algorithm of searching through the various `Player` classes looking for one capable of handling the content type that the `DataSource` (constructed as an earlier part of the process if the `DataSource` wasn't supplied to the method) specifies. The process is simple linear iteration through the list of constructed classnames until one is found capable of handling the media.

If `Manager` cannot create a `Player`, it might throw an `IOException` (that is, file doesn't exist), a `NoPlayerException` (that is, a content-type that the JMF does not handle), or in the case of the `createRealizedPlayer()` methods, a `CannotRealizeException` (that is, couldn't obtain the resources necessary).

`Players` will be discussed further in a subsequent section.

Creating a `Processor`

Objects implementing the `Processor` interface are used in controlling the processing of media. The `Manager` class provides four methods for the creation of a `Processor`: Three create an `Unrealized Processor`: `createProcessor()` , whereas the final method creates a `Realized` (that is, already partially resourced and more along the path to being ready to start) `Processor`: `createRealizedProcessor()`.

The `createRealizedProcessor()` method is provided as a means of accelerating the creation of a `Processor`. The method call blocks (that is, the next line of code doesn't execute until the `Processor` is `Realized`). This has the advantage that the `Realized` event doesn't need to be listened for by the invoker of the method. The `createRealizedProcessor()` method accepts a single parameter that is a `ProcessorModel` which fully specifies the input or output format of the media; thus the processing to be performed.

The three variants of `createProcessor()` have the same form as for `Player` creation: accepting either a `MediaLocator`, `URL`, or `DataSource` as the single parameter. Similarly, the `Manager` follows the same process in creation of a `Processor` as it does in the creation of a Player:

1. Convert the `URL` to a `MediaLocator` (if `URL` based method is used).
2. Create a `DataSource` for the `MediaLocator` (if `DataSource` based method isn't used).
3. Obtain the `Processor` that can handle the `DataSource`.
4. Attach the `DataSource` to the `Processor`.
5. Return the `Processor` object.

The `URL` and `MediaLocator` based creation methods should be used in the case where the processing is the first step in the chain of control. Alternatively, if it is a subsequent step (for example, chained processing) the `DataSource` based method should be used.

Thus the typical usage of `Manager` to create a `Processor` is as follows:

```
MediaLocator locator = new MediaLocator(...);    // Specify media location;
Processor processor = Manager.createProcessor(locator);
```

Alternatively, a `Realized Processor` could be created with the `createRealizedProcessor()` method as

```
//Specify the model for processing
ProcessorModel model = new ProcessorModel(...);
// Create the processor
Processor processor = Manager.createRealizedProcessor(model);
```

In creating a `Processor` object, `Manager` follows a set algorithm of searching through the various `Processor` classes that correspond to the same approach as that used for `Player` creation.

If `Manager` cannot create a `Processor`, it might throw an `IOException` (that is, file doesn't exist), a `NoProcessorException` (that is, a content-type that the JMF doesn't handle), or in the case of the `createRealizedProcessor()` methods, a `CannotRealizeException` (that is, couldn't obtain the resources necessary).

`Processors` will be discussed further in a subsequent section.

Creating a `DataSource`

`DataSources` are the means by which `Players`, `Processors`, or `DataSinks` obtain their data. Creation of a `Player`, `Processor`, or `DataSink` always involves a `DataSource`; whether provided explicitly to the creation method (as in the creation of a `DataSink`), or created as part of the larger process (as in the creation of a `Player` where a `MediaLocator` is provided). Details of `DataSources` are fully discussed in a subsequent section.

There are two methods for creating a `DataSource` from a location specification (`URL` or `MediaLocator`) using `createDataSource()`. There are also two methods for creating specialized `DataSources`: merging `DataSource` (one that combines two or more `DataSources`) with `createMergingDataSource()` and creating a cloneable `DataSource` (one that can be cloned to be processed or played by different systems simultaneously) with `createCloneableDataSource()`.

Listing 8.2 shows a number of the `DataSource` creation methods being used in a hypothetical scenario in which two `DataSources` are created, one is cloned (so it can be dual processed), and then the two sources are combined.

Listing 8.2 Hypothetical `DataSources` Scenario

```
MediaLocator firstLocation = new MediaLocator(...);    //Location of 1st media
MediaLocator secondLocation = new MediaLocator(...);    // Location of 2nd media
try {
  // Create the two datasources.
  DataSource firstSrc = Manager.createDataSource(firstLocation);
  DataSource secondSrc = Manager.createDataSource(secondLocation);
  // Create cloneable version of 2nd src, then clone it.
  DataSource cloneableSrc = Manager.createCloneableDataSource(secondSrc);
  DataSource cloneA = cloneableSrc.createClone();
  // Create a merged DataSource combining the 1st and 2nd  DataSources.
  DataSource srcArray = new DataSource[2];
  srcArray[0] = firstSrc;
  srcArray[1] = cloneA;
  DataSourced mergedSrcs = Manager.createdMergingDataSource(srcArray);
}
```

`DataSources` are identified by the protocol they support. In creation of a `DataSource`, the `Manager` class follows a similar approach as that employed for `Player` and `Processor` creation. A list of classes

supporting the protocol specified (by the `URL` or `MediaLocator`) is compiled and that list is linearly searched until a class capable of sourcing the media is found.

Failure to create a `DataSource` from a `URL` or `MediaLocator` will result in the method throwing an `IOException` or `NoDataSourceException`. Failure to create a merging `DataSource` will result in an `IncompatibleSourceException` being thrown.

Creating a `DataSink`

`DataSinks` are used to take the media from a `DataSource` and render it to a particular location (for instance, a file). The `Manager` class provides a single method, `createDataSink()`, for the creation of a `DataSink`. The method accepts two parameters—a `DataSource` from which the media is sourced, and a `MediaLocator` that specifies the destination location.

The steps that the `Manager` class follows when creating a `DataSource` instance are similar to those used in the creation of a `DataSource`—with the protocol of the `MediaLocator` used to compile a list of `DataSink` classes that support the protocol. That list is then searched in order to find an appropriate class for which an instance can be created.

Failure to create a `DataSink` will result in a `NoDataSink` exception being thrown.

`DataSinks` are discussed in a later section of this chapter. However as an example of their usage, Listing 8.3 shows a portion (the object creation) of the process in which a `DataSink` could be used to create a copy of a media file. (Obviously it would be far more efficient to simply copy the file using the operating system commands.)

Listing 8.3 Some of the Major Steps in Using a `DataSink` Object

```
String    origin = "file:...";
String    destination = "file:...";
MediaLocator    originLocation = new MediaLocator(origin);
MediaLocator    destinationLocation = new MediaLocator(destination);
Processor     p = Manager.createRealizedProcessor(null);
DataSource    src = p.getDataOutput();
DataSink    dest = Manager.createDataSink(src,destinationLocation);
:        :
```

Querying the `Manager`

Besides the methods for creating `Players`, `Processors`, `DataSources`, and `DataSinks`, the `Manager` class provides a number of information methods—methods that provide information about the configuration and support of the installed version of the JMF.

These information methods are the various `get*()` methods:

`getHint()`— Obtains information about hints provided to JMF.

`getCacheDirectory()`— Determines what directory the JMF uses for temporary storage.

`getDataSourceList()`— Determines what `DataSource` classes support a particular protocol.

`getHandlerClassList()`— Determines what `Player` classes support a particular content type.

`getProcessorClassList()`— Determines what `Processor` classes support a particular content type.

The `getHint()` method should be passed one of the constants defined in the `Manager` class (for example, `CACHING`) and returns the setting for that as an instance of `Object`.

The `getCacheDirectory()` method has no parameters and returns a `String`.

The three `get*List()` methods `getDataSourceList()`, `getHandlerClassList()`, and `getProcessorClassList()` each accept a `String` specifying the content type or protocol for which support is being queried. The methods return a `Vector`. The elements of the `Vector` are `Strings`, and those `Strings` are the fully qualified names of the classes that provide that support.

Listing 8.4 is an application `ManagerQuery`, which employs the information gathering methods of `Manager` in order to provide the user with details about the JMF. In running the program, users can query JMF as to its support (`Players`, `Processors`, or `DataSources`) for different formats and protocols. Alternatively, a complete picture can be produced by not specifying any particular formats or protocols: In this case all classes supporting all known formats are listed. The application can be found on the book's companion Web site. Similarly, a graphical application `GUIManagerQuery`, based on `ManagerQuery`, is available from the book's companion Web site. It provides the same functionality as `ManagerQuery` but with a graphical user interface.

Listing 8.4 The `ManagerQuery` Application Used to Discover Details About JMF

```
/*****************************************************************
* ManagerQuery - Query the manager class about the configuration and
* support of the installed JMF version. ManagerQuery is a text-based
* application that provides a report on the support of the JMF for
* Players, Processors and DataSinks.
*
* Without any command-line arguments ManagerQuery prints a complete
* (LONG) list of Player, Processor, and DataSource classes that
* support the various formats, protocols, and content types.
*
* Alternatively it is possible to provide command-line arguments
* specifying the format or protocol for which support is to be
* checked. The means of calling is as follows:
*   java ManagerQuery [ [-h|-p|-d] support1 support2 ... supportN]
* The -h flag specifies handlers (Players) only.
* The -p flag specifies Processors only.
* The -d flag specifies DataSources only.
* Leaving off the flag defaults behaviour to checking for Players
* only.
*
* For instance:
*   java ManagerQuery -h mp3 ulaw
* would list the classes capable of Playing the MP3 (MPEG, Layer 3)
* and U-Law formats (codecs).
*
```

```
* ManagerQuery always prints the version of JMF, caching directory,
* and hints prior to any other output.
*
* @author Spike Barlow
*****************************************************************/
import javax.media.*;
import javax.media.protocol.*;
import javax.media.format.*;
import java.util.*;

public class ManagerQuery {
  ////////////////////////////////////////////////
  // Constants to facilitate selection of the
  // approprite get*List() method.
  ////////////////////////////////////////////////
  public static final int HANDLERS = 1;
  public static final int PROCESSORS = 2;
  public static final int DATASOURCES = 3;
  ////////////////////////////////////////////////////
  // Array containing all the content types that JMF2.1.1
  // supports. This is used when the user provides no
  // command-line arguments in order to generate a
  // complete list of support for all the content types.
  ////////////////////////////////////////////////////
  private static final String[] CONTENTS = {ContentDescriptor.CONTENT_UNKNOWN,
      ContentDescriptor.MIXED, ContentDescriptor.RAW,
ContentDescriptor.RAW_RTP, FileTypeDescriptor.AIFF,
FileTypeDescriptor.BASIC_AUDIO, FileTypeDescriptor.GSM,
      FileTypeDescriptor.MIDI, FileTypeDescriptor.MPEG,
FileTypeDescriptor.MPEG_AUDIO, FileTypeDescriptor.MSVIDEO,
FileTypeDescriptor.QUICKTIME, FileTypeDescriptor.RMF,
FileTypeDescriptor.VIVO, FileTypeDescriptor.WAVE,
      VideoFormat.CINEPAK, VideoFormat.H261, VideoFormat.H263,
VideoFormat.H261_RTP, VideoFormat.H263_RTP,
VideoFormat.INDEO32, VideoFormat.INDEO41, VideoFormat.INDEO50,
      VideoFormat.IRGB, VideoFormat.JPEG, VideoFormat.JPEG_RTP,
VideoFormat.MJPEGA,  VideoFormat.MJPEGB, VideoFormat.MJPG,
VideoFormat.MPEG_RTP, VideoFormat.RGB, VideoFormat.RLE, VideoFormat.SMC,
VideoFormat.YUV, AudioFormat.ALAW,
      AudioFormat.DOLBYAC3, AudioFormat.DVI, AudioFormat.DVI_RTP,
AudioFormat.G723, AudioFormat.G723_RTP, AudioFormat.G728,
AudioFormat.G728_RTP, AudioFormat.G729, AudioFormat.G729_RTP,
AudioFormat.G729A, AudioFormat.G729A_RTP, AudioFormat.GSM,
      AudioFormat.GSM_MS, AudioFormat.GSM_RTP, AudioFormat.IMA4,
AudioFormat.IMA4_MS, AudioFormat.LINEAR, AudioFormat.MAC3,
AudioFormat.MAC6, AudioFormat.MPEG, AudioFormat.MPEG_RTP,
AudioFormat.MPEGLAYER3, AudioFormat.MSADPCM,
AudioFormat.MSNAUDIO, AudioFormat.MSRT24,
AudioFormat.TRUESPEECH, AudioFormat.ULAW, AudioFormat.ULAW_RTP,
AudioFormat.VOXWAREAC10, AudioFormat.VOXWAREAC16,
AudioFormat.VOXWAREAC20, AudioFormat.VOXWAREAC8,
AudioFormat.VOXWAREMETASOUND, AudioFormat.VOXWAREMETAVOICE,
AudioFormat.VOXWARERT29H, AudioFormat.VOXWARETQ40,
      AudioFormat.VOXWARETQ60, AudioFormat.VOXWAREVR12,
AudioFormat.VOXWAREVR18};
  ////////////////////////////////////
  // The protocols that JMF supports.
  ////////////////////////////////////
  private static final String[] PROTOCOLS = { "ftp", "file", "rtp", "http"};
```

```
/**************************************************************
* Return a String being a list of all hints settings.
**************************************************************/
public static String getHints() {

  return "\tSecurity: " + Manager.getHint(Manager.MAX_SECURITY) +
      "\n\tCaching: " + Manager.getHint(Manager.CACHING) +
      "\n\tLightweight Renderer: " +
Manager.getHint(Manager.LIGHTWEIGHT_RENDERER) +
      "\n\tPlug-in Player: " +
Manager.getHint(Manager.PLUGIN_PLAYER);
}


/**************************************************************
* Produce a list of all classes that support the content types or
* protocols passed to the method. The list is returned as a formatted
* String, while the 2nd parameter (which) specifies whether it is
* Player (Handler), Processor, or DataSource classes.
**************************************************************/
public static String getHandlersOrProcessors(String[] contents,
int which) {
  String  str="";
  Vector  classes;
  int     NUM_PER_LINE = 1;
  String  LEADING = "\t    ";
  String  SEPARATOR = "  ";

  if (contents==null)
    return null;

  ////////////////////////////////////////////////////////////////
  // Generate a separate list for each content-type/protocol
  //specified.
  ////////////////////////////////////////////////////////////////
  for (int i=0;i<contents.length;i++) {
    str=str + "\t" + contents[i] + ":\n";
    if (which==HANDLERS)
      classes = Manager.getHandlerClassList(contents[i]);
    else if (which==PROCESSORS)
      classes = Manager.getProcessorClassList(contents[i]);
    else
      classes = Manager.getDataSourceList(contents[i]);
    if (classes==null)
      str = str + "\t    <None>\n";
    else
      str = str + formatVectorStrings(classes,LEADING,NUM_PER_LINE,
                    SEPARATOR);
  }
  return str;
}


/**************************************************************
* Get a list of all Handler (Player) classes that support each of the
* formats (content types).
**************************************************************/

public static String getHandlers() {
```

```
  return getHandlersOrProcessors(CONTENTS,HANDLERS);
}


/**************************************************************
* Get a list of all Processor classes that support each of the
* formats (content types).
**************************************************************/
public static String getProcessors() {
  return getHandlersOrProcessors(CONTENTS,PROCESSORS);
}


/**************************************************************
* Get a list of all DataSources classes that support each of the
* protocols.
**************************************************************/

public static String getDataSources() {
  return getHandlersOrProcessors(PROTOCOLS,DATASOURCES);
}

/**************************************************************
* Format the Vector of Strings returned by the get*List() methods
* into a single String. A simple formatting method.
**************************************************************/
public static String formatVectorStrings(Vector vec, String leading,
int count, String separator) {
  String  str=leading;


  for (int i=0;i<vec.size();i++) {
    str = str + (String)vec.elementAt(i);
    if ((i+1)==vec.size())
      str = str + "\n";
    else if ((i+1)%count==0)
      str = str + "\n" + leading;
    else
      str = str + separator;
  }
  return str;
}


/**********************************************************
* Produce a list showing total support (i.e., Player,
* Processors, and DataSinks) for all content types and
* protocols.
**********************************************************/
public static void printTotalList() {
  System.out.println("\nPlayer Handler Classes:");
  System.out.println(getHandlers());
  System.out.println("\nProcessor Class List:");
  System.out.println(getProcessors());
  System.out.println("\nDataSink Class List: ");
  System.out.println(getDataSources());
}
/**************************************************************
```

```
* Main method. Produce a version and hints report. Then if no command
* line arguments produce a total class list report. Otherwise process
* the command line arguments and produce a report on their basis.
**********************************************************************/
public static void main(String args[]) {

  System.out.println("JMF: " + Manager.getVersion());
  String cacheArea = Manager.getCacheDirectory();
  if (cacheArea==null)
    System.out.println("No cache directory specified.");
  else
    System.out.println("Cache Directory: " + cacheArea);
  System.out.println("Hints:");
  System.out.println(getHints());

  // No command-line arguments. Make a total report.
  if (args==null || args.length==0)
    printTotalList();
  else {

    // Command-line. Process flags and then support to be
    // queried upon in order to generate appropriate report.
    String  header="";
    int     whichCategory = 0;
    String[]  interested;
    int     i;
    int     start;
    if (args[0].equalsIgnoreCase("-h")) {
      header = "\nPlayer Handler Classes: ";
      whichCategory = HANDLERS;
    }
    else if (args[0].equalsIgnoreCase("-p")) {
      header = "\nProcessor Class List:";
      whichCategory = PROCESSORS;
    }
    else if (args[0].equalsIgnoreCase("-d")) {
      header = "\nDataSink Class List: ";
      whichCategory = DATASOURCES;
    }
    if (whichCategory==0) {
      whichCategory = HANDLERS;
      header = "\nPlayer Handler Classes: ";
      interested = new String[args.length];
      start = 0;
    }
    else {
      interested = new String[args.length-1];
      start = 1;
    }
    for (i=start;i<args.length;i++)
      interested[i-start] = args[i];
    System.out.println(header);
    System.out.println(getHandlersOrProcessors(interested,whichCategory));
  }
}
}
```

In order to specify a particular query, the formats or protocols in question are provided as command-line arguments to the application. An initial flag specifier supports `Processors` (`-p`), `DataSources` (`-d`), or `Players` (handlers, hence `-h`), which are being examined. Failure to specify a flag is interpreted as being a query about `Players`, whereas a lack of any command-line arguments is interpreted as a query about `Player`, `Processor`, and `DataSource` support for all formats and protocols. Listing 8.5 shows two runs of the program. The first in which Processor classes supporting the `mpg` (MPEG) and `avi` (AVI) content types are listed. The second in which handlers (Players) classes supporting `http` (Hypertext Transfer Protocol) are listed.

Listing 8.5 Two Runs of the `ManagerQuery` Application Show How It Can Be Employed and the Output Produced

```
D:\JMF\Book\Code>java ManagerQuery -p mpg avi
JMF: 2.1.1
Cache Directory: C:\WINDOWS\TEMP
Hints:
        Security: false
        Caching: true
        Lightweight Renderer: false
        Plug-in Player: false

Processor Class List:
        mpg:
            media.processor.mpg.Handler
            javax.media.processor.mpg.Handler
            com.sun.media.processor.mpg.Handler
            com.ibm.media.processor.mpg.Handler
        avi:
            media.processor.avi.Handler
            javax.media.processor.avi.Handler
            com.sun.media.processor.avi.Handler
            com.ibm.media.processor.avi.Handler

D:\JMF\Book\Code>java ManagerQuery -h http
JMF: 2.1.1
Cache Directory: C:\WINDOWS\TEMP
Hints:
        Security: false
        Caching: true
        Lightweight Renderer: false
        Plug-in Player: false

Player Handler Classes:
        http:
            media.content.http.Handler
            javax.media.content.http.Handler
            com.sun.media.content.http.Handler
            com.ibm.media.content.http.Handler
```

It's All About `Control`

The various objects provided by the JMF such as `Players`, `Processors`, `DataSources`, `DataSinks`, and plug-ins have complex and configurable behavior. For instance, it is desirable to allow the frame that a `Player` starts playing from to be set, or the bit rate for a `codec` to be specified. The JMF provides a uniform model for controlling the behavior of objects through the `Control` interface.

Many JMF objects expose `Control` objects through accessor (get) methods. These can be obtained and used to alter the behavior of the associated objects. Indeed, there is an interface known as `Controls` that many objects implement as a uniform means of providing `Control` objects that specify their behavior.

Hence the standard approach in tailoring a `Processor`, `Player`, `DataSource`, or `DataSink` to match a particular need is as follows:

1. Create the `Player`, `Processor`, `DataSource`, or `DataSink`.
2. Obtain the `Control` object appropriate to the behavior to be configured.
3. Use methods on `Control` object to configure the behavior.
4. Use the original `Player`, `Processor`, `DataSource`, or `DataSink`.

Because of some unfortunate naming choice for classes and interfaces in the JMF, the following classes bear similar names: `Control`, `Controls` (x2), and `Controller`, as well as a package known as control. It is worth taking this opportunity to delineate the differences among these confusingly named classes:

`Control`— An interface describing an object that can be used to control the behavior of a JMF object such as a `Player` or `Processor`. The `Control` interface is discussed in this section. It is extended to a number of specialized `Control` Interfaces, such as `FramePositioningControl`.

`Controller`— An interface upon which `Player` and `Processor` are built and which is intimately associated with the timing model of the JMF. The `Controller` interface was discussed in an earlier section of this chapter.

`Controls` (`javax.media.Controls` and `javax.media.protocol.Controls`)— An interface implemented by objects that provide a uniform mechanism for obtaining their `Control` objects(s).

`javax.media.control`— A package within the JMF API that contains 18 interfaces that extend the basic `Control` interface. Examples include `FramePositioningControl`, `TrackControl`, and `FormatControl`.

The `Control` interface itself is particularly simple, possessing a single method only, with the real control functionality being specified in the various 18 `Control` interfaces that extend `Control` in the JMF API. Each of those interfaces has a specific functionality as detailed by its name and methods. Instances of these interfaces are the means of configuring the behavior of the object from which they were obtained. The following list quickly summarizes each of them:

`BitRateControl`— A control for specifying and querying the bit rate settings, such as the encoding bit rate for a compressor (`codec`).

`BufferControl`— A control for querying and specifying buffer thresholds and sizes.

`FormatControl`— A control for querying the format support as well as setting the format for the associated object.

`FrameGrabbingControl`— A control for enabling the grabbing of still video frames from a video stream.

`FramePositioningControl`— A control to allow the precise positioning of a video stream as either a frame number of time (from start).

`FrameProcessingControl`— A control to specify the parameters employed in frame processing.

`FrameRateControl`— A means of querying as well as setting the frame rate.

`H261Control`— A control for specifying the parameters of the H.261 video codec.

`H263Control`— A control for specifying the parameters of the H.263 video codec.

`KeyFrameControl`— A control for specifying or querying the `KeyFrame` interval: the interval between transmission of complete (keyframes) rather than delta frames in codecs (such as mpeg) that use temporal based compression.

`MonitorControl`— A control for specifying the degree of monitoring (viewing or listening to) of media as it is captured.

`MpegAudioControl`— A control for specifying the parameters of MPEG Audio encoding.

`PacketSizeControl`— A control for specifying the packet size parameters.

`PortControl`— A control to access the input and output ports of a device (such as a capture device).

`QualityControl`— A control for specifying the parameters of quality (higher quality generally comes at the expense of higher processing demands).

`SilenceSuppressionControl`— A control for specifying the parameters of silence suppression. Silence suppression is an audio compression scheme whereby silent passages aren't transmitted.

`StreamWriterControl`— A control by which the maximum size for an output stream (for example, DataSink or Multiplexer) can be set as well as the size queried.

`TrackControl`— A control to query, manipulate, and control the data of individual media tracks in a Processor.

Each of these Control interfaces can be found in the javax.media.control package, which necessitates the import of that package if they are to be employed.

As mentioned previously, each interface possesses methods specific to its functionality. Some are simple such as FrameGrabbingControl with its single method grabFrame() that returns a Buffer object; others such as MpegAudioControl have more than a dozen methods plus associated constants. However, most interfaces are small, with 3–5 methods, and quite easy to understand.

Visual Control for the User

Time-based media, as defined in the Chapter 7, is intended for presentation to a human being. It is natural, then, to provide the viewer or listener with maximum control over that experience. To that end, many Control objects have an associated visual Component. That Component can be obtained and added to the graphical user interface provided for the user. Actions upon the Component result in method calls on the associated Control object that hence alter the behavior of the associated Player, Processor, DataSource, or DataSink.

The Control interface that is the superclass of the previous 18 possesses a single method, getControlComponent(), that returns an AWT Component—a graphical component that can be added to a graphical user interface and through which the user can directly and intuitively set the control parameters.

However, not all Controls have an associated graphical Component. Those that don't have a Component return null to the getControlComponent() method call. Thus code using the method should check to ensure that a non-null reference was returned before attempting to add the Component to an AWT Container (for example, Applet, Frame, or Panel). Adding a null Component to a Container will result in an exception being thrown.

Getting Control Objects

There are two methods by which Control objects can be obtained. These methods, getControl(), and getControls() are defined in the Controls interface. The Controls interface is extended by many important interfaces including DataSink, Codec, Renderer, and the various pull and push data and buffer streams. Other important classes such as Controller (that is, the superclass of Player and Processor) also provide the two methods.

The getControl() method is used to obtain a single Control object. The method is passed the complete name of the Control class as a String and returns an object implementing that interface. The fully qualified name of the class must be passed, thus listing the package "path" to the class. Then the method returns an object of class Control, necessitating it to be cast to the type of Control before its methods can be employed. For example,

```
BitRateControl bitControl = (BitRateControl)
processor.getControl("javax.media.control.BitRateControl");
```

The second means of obtaining `Control` objects is through the `getControls()` method. The method accepts no arguments and returns (supposedly) all the `Control` objects, as an array, associated with the object on which `getControls()` was called. Look at the following example:

```
Control[] allControls = player.getControls();
```

As an example of the `Control` objects associated with a `Player`, Listing 8.6 shows the `Control` objects obtained from the `Player` used in the `BBPApplet` example of Chapter 7. The `getControls()` method of the `Player` object (actually inherited from `Controller`) was called once the `Player` was realized and the objects returned were printed.

Listing 8.6 The 11 Control Objects Obtained on a Particular `Player` Object when `getControls()` Was Called

```
11 controls for a Player @ REALIZED:
1: com.ibm.media.codec.video.mpeg.MpegVideo
2: com.sun.media.codec.video.colorspace.YUVToRGB
3: com.sun.media.renderer.video.DDRenderer
4: com.sun.media.renderer.audio.DirectAudioRenderer$MCA
5: com.sun.media.renderer.audio.AudioRenderer$BC
6: com.sun.media.PlaybackEngine$BitRateA
7: com.sun.media.PlaybackEngine$1
8: com.sun.media.controls.FramePositioningAdapter
9: com.sun.media.BasicJMD
10: com.sun.media.PlaybackEngine$PlayerTControl
11: com.sun.media.PlaybackEngine$PlayerTControl
```

It is worth noting several points in connection with Listing 8.6. First, the `11 controls` case corresponds to the `Manager` being instructed to create `Players` that support plug-ins for demultiplexing, codecs, and so on. In the case where the created `Player` wasn't plug-in enabled, only three `Controls` were obtained. Enabling plug-in based `Players` was achieved as follows:

```
Manager.setHint(Manager.PLUGIN_PLAYER,new Boolean(true));
```

Second, all the `Controls` returned by the previous call are from the Sun and IBM packages (com.sun.media and com.ibm.media) and hence aren't documented in the API. Further, only the `BasicJMD Control` had a visual component, that being the PlugIn Viewer. That combination of undocumented classes plus lack of visual components makes the `getControls()` approach of exerting control not particularly helpful and next to useless, at least in this case.

However, it appears that controller's `getControls()` method doesn't actually return all possible `Control` objects for a `Player`. As noted previously, rather than asking for all `Control` objects, it is possible to ask for them by name. When that was done, it was possible to obtain a number of `Control` objects including `FrameRateControl` and `FrameGrabbingControl` among others as follows:

```
FrameGrabbingControl frameControl = (FrameGrabbingControl)
player.getControl("javax.media.control.FrameGrabbingControl");
```

As in the previous case, these `Controls` were only available when the `Manager`'s `PLUGIN_PLAYER` hint had been set to `true`.

Thus, the surest and safest means of obtaining the appropriate `Control` objects and using them is with the following algorithm:

```
If using a Player
       Set Manager's PLUGIN_PLAYER hint to true prior to creating the Player
For each Controller needed
        Get it by name (full class name to getControl() method).
        If object returned not null
               "Use it"
               If want to provide direct user control
                       Call getControlComponent() on Controller object
                       If object returned is not null
                              Add that to GUI
                       else
                              Implement own GUI interface
```

Sourcing Media and Media `Format`

Controlling and handling media always begins with its sourcing and is achieved by specifying its location, protocol, and format. With that information, the media stream can be obtained and the appropriate `Controller` (for example, `Player` or `Processor`) or `DataSink` instances can be created (see the earlier section on the `Manager` class).

Hence, the first significant step in media control is creating a `DataSource` object, although sometimes this isn't done explicitly by the user but implicitly by the `Manager` class.

`DataSource` isn't the only class with relevance to this vital first stage. Several other classes are involved in the sourcing of media:

`MediaLocator`— Specifies the location and protocol for the media. Closely related to the `URL` class. Needed to create a `DataSource`.

`Manager`— Creates a `DataSource` either explicitly or implicitly (for example, when a `Player` or `Processor` is created).

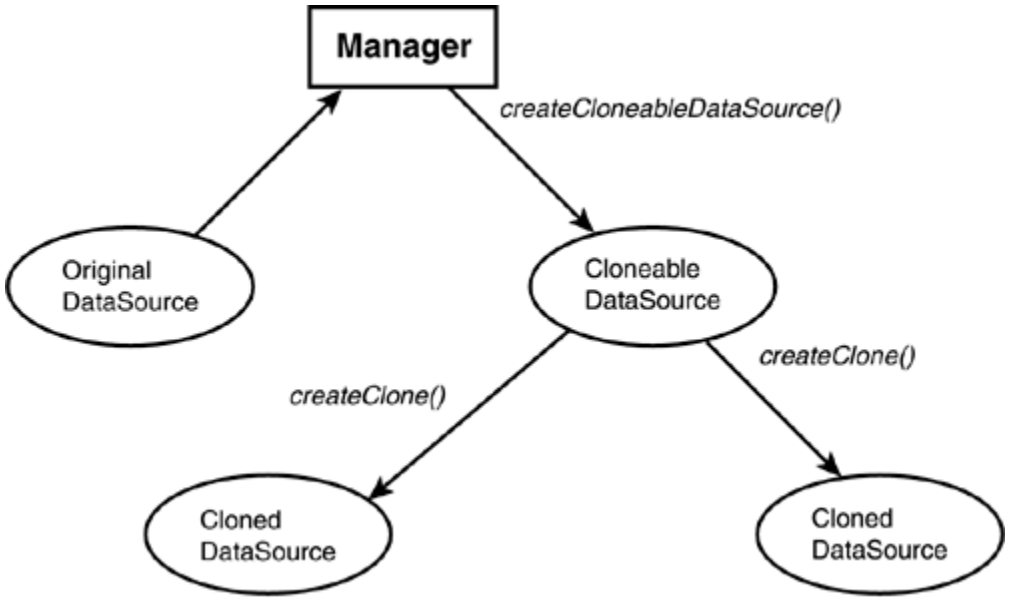
`DataSource`— A manager for media transfer. Required to create `Players`, `Processors`, or `DataSinks`. One of the most central classes in the whole JMF.

`SourceStream`— A stream of data. `SourceStreams` are managed by `DataSources`. A number of subclasses of `SourceStream` depend on whether the stream is push or pull and the format is low level.

`Format`— An abstraction of the format of media. This class is extended by the more specialized classes `ContentDescriptor`, `FileTypeDescriptor`, `AudioFormat`, and `VideoFormat`. Each `SourceStream` of a `DataSource` has an associated `ContentDescriptor` (`Format`).

Figure 8.14 shows the relationships between these classes, with `DataSource` cast as the central participant because of its significance in the creation of `Players` or `Processors`.

Figure 8.14. The relationship of other data description classes to that of `DataSource`.



`DataSource`

Media processed by the JMF can be a video file on the local file system, an audio track on CD, a videoconference streaming across the Internet, or a range of other possibilities. All these are instances of media sources, and the JMF provides a uniform means of describing them and managing their transfer via the `DataSource` class.

The `DataSource` class provides an abstracted, simple model of the media that can then be employed as part of the processing or control chain: creation of a `Player`, `Processor`, or `DataSink` by the `Manager` class involves the prior creation of a `DataSource` (either by the caller, or implicitly as part of the creation process).

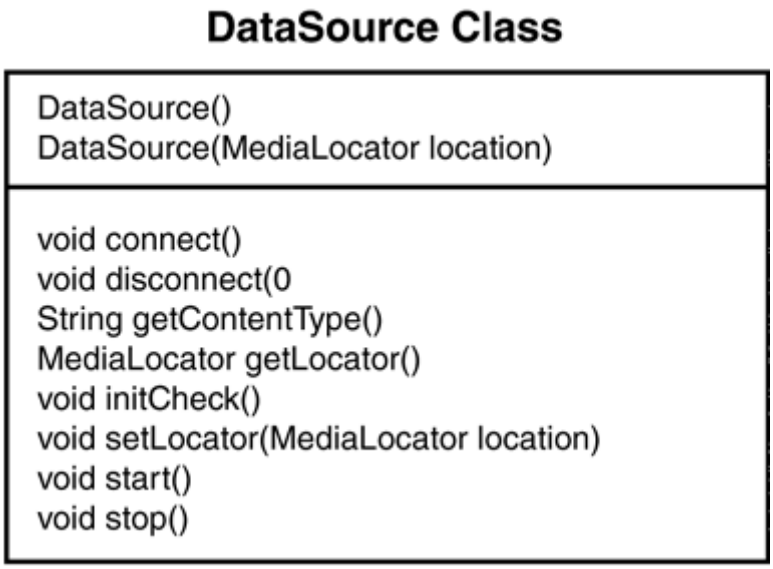
`DataSources` can be seen as managing the transfer of media content. Their creation requires the specification of a protocol and a location from which the media can be obtained. With that specification, the `Manager` class follows an iterative process of finding a `DataSource` class that supports the protocol and data format.

`DataSource` instances manage media transfer by managing one or more `SourceStreams`—the media streams.

A key feature of all DataSources is that they cannot be reused. For instance, a DataSource used by a Player cannot later be used by a Processor.

From a user's perspective, a DataSource object is generally rather passive. Not only is it often created as a (again, from the user's perspective) by-product of creating a Player or Processor, but also it isn't common to use the methods on a DataSource object because these are invoked by the associated Player or Processor. Figure 8.15 shows the methods of DataSource.

Figure 8.15. The DataSource class.



Although it is possible to construct a DataSource directly with one of its two constructors, it is next to worthless because the base DataSource provides no functionality. Only those subclasses constructed by Manager provide the required abilities. Hence, the Manager class should be used to create DataSources. As discussed in the earlier section on the Manager, there is an overloaded createDataSource() method of Manager—it accepts either a URL or a MediaLocator and returns a DataSource. There are also two methods of Manager for creating specialist DataSources—merging and cloneable. These will be discussed shortly.

The key methods of the class are connect(), disconnect(), start(), and stop(). As their names imply, the connect() and disconnect() methods open (or close) a connection to the media source that was specified by the MediaLocator used to create the DataSource. Similarly start() initiates a data transfer whereas stop() halts it.

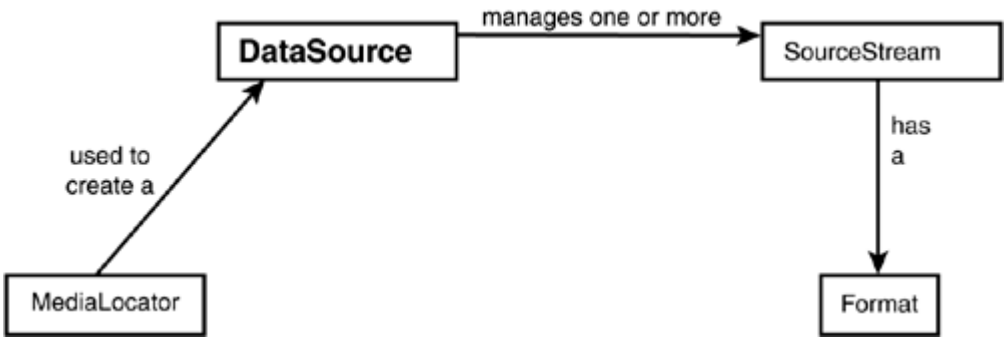
DataSources can be classified on two axes— one axis being how data transfer is initiated (push or pull) and the other being the unit of transfer (raw bytes of a Buffer object). Pull DataSources have the transfer initiated and controlled by the client. Examples of these protocols include http (Hypertext Transfer Protocol), ftp (File Transfer Protocol), and file. Push DataSources have the transfer controlled by the server. Examples of push media include broadcast media, Web casts (multicasts), and video on demand. Two categories for each of the two axes leads to four subclasses of DataSource:

PullDataSource, PullBufferDataSource, PushDataSource, and PushBufferDataSource. Each subclass extends DataSource by providing a getStreams() method that returns an array of the appropriate SourceStream (for example, PullSourceStream for a PullDataSource). Clearly, the type of DataSource dictates the type of operations that can be supported on it, whereas a video obtained from a file can be replayed, played in reverse, or positioned. The same isn't true for a broadcast video.

Cloneable and Merged DataSources

Two special DataSources, cloneable and merged, are created through the Manager class (see the earlier section on the Manager). Not surprisingly, the cloneable DataSource can be cloned so that, for instance, multiple versions of the same media stream can be processed simultaneously. The DataSource returned by Manager's createCloneableDataSource() has an additional method createClone(). After a DataSource has been cloned, the original (not cloneable) version shouldn't be employed. Figure 8.16 shows the process of creating and using a cloneable DataSource.

Figure 8.16. Steps involved in cloning a DataSource object.



On the other hand, the merged data source is simply the combination of two or more DataSources. Manager's createMergingDataSource() is passed an array of DataSources and returns a single DataSource that is their merged combination. All DataSources in the array must be of the same type (for example, all PullBufferDataSources); otherwise, an IncompatibleSourceException will be thrown.

MediaLocator

The MediaLocator class is the JMF's means of describing the location of media. Closely related to the Java Platform's URL class, a MediaLocator can be constructed from a String (for example, MediaLocator ml = new MediaLocator("file://media/example.mov") or from a URL.

MediaLocator objects serve little purpose other than providing the necessary information—the media's location and protocol—for the construction of a DataSink, Player, or Processor. The object possess five accessor style methods:

`getProtocol()`— Returns a `String` such as `"http"`

`getRemainder()`— Returns a `String` that is all except the protocol

`getURL()`— Returns a URL object

`toExternalForm()`— Returns a `String` representation suitable for constructing a `MediaLocator`

`toString()`— Returns a `String` version suitable for printing

`SourceStream` and `Buffer`

A `SourceStream` abstracts a single (source) stream of media data. Each `DataSource` manages one or more `SourceStream` objects.

`SourceStreams` represent a lower level of detail than many JMF programs interact with directly. For the greater portion of JMF applications, the `DataSource` class provides sufficient detail with the desired abstraction from the complexity and detail of the underlying data.

Just as for `DataSource`, the `SourceStream` interface is extended by four interfaces that classify the method of transfer initiation (pull or push), and the unit of transfer (raw data or buffer): `PullBufferStream`, `PullSourceStream`, `PushBufferStream`, and `PushSourceStream`.

The `SourceStream` interface specifies three methods as well as the two methods resulting from the extension of `Controls`. The three methods are as follows:

`endOfStream()`— a Boolean method that returns `true` if the end of stream has been reached

`getContentDescriptor()`— Returns the `ContentDescriptor` (format) for the stream

`getContentLength()`— Returns the length of the stream in bytes (or `SourceStream.LENGTH_UNKNOWN` if it cannot be ascertained)

The two additional methods are the `getControl()` and `getControls()` methods of the `Controls` interface. All the subinterfaces provide a `read()` method for actually transferring data.

As seen with both the `DataSource` and `DataStream classes`, objects of these two types can either be raw data or buffered. The `Buffer` class is the JMF representation of these buffers or containers that transfer data from one processing stage to the next in a `Player` or `Processor`, or between a buffer source stream and its handler.

The `Buffer` object not only carries the media data, but also metadata such as the media format, timestamps, length, and other header information.

The `Buffer` class is pivotal in allowing the JMF to be combined with other Java APIs such as JAI (Java Advanced Imaging) because the JMF provides methods to convert between `Buffer` objects and AWT `Image` objects. This combination of APIs and deliberate low-level processing of media data are the two

chief reasons for employing a `Buffer` class: Most JMF applications don't need to delve to this level of detail.

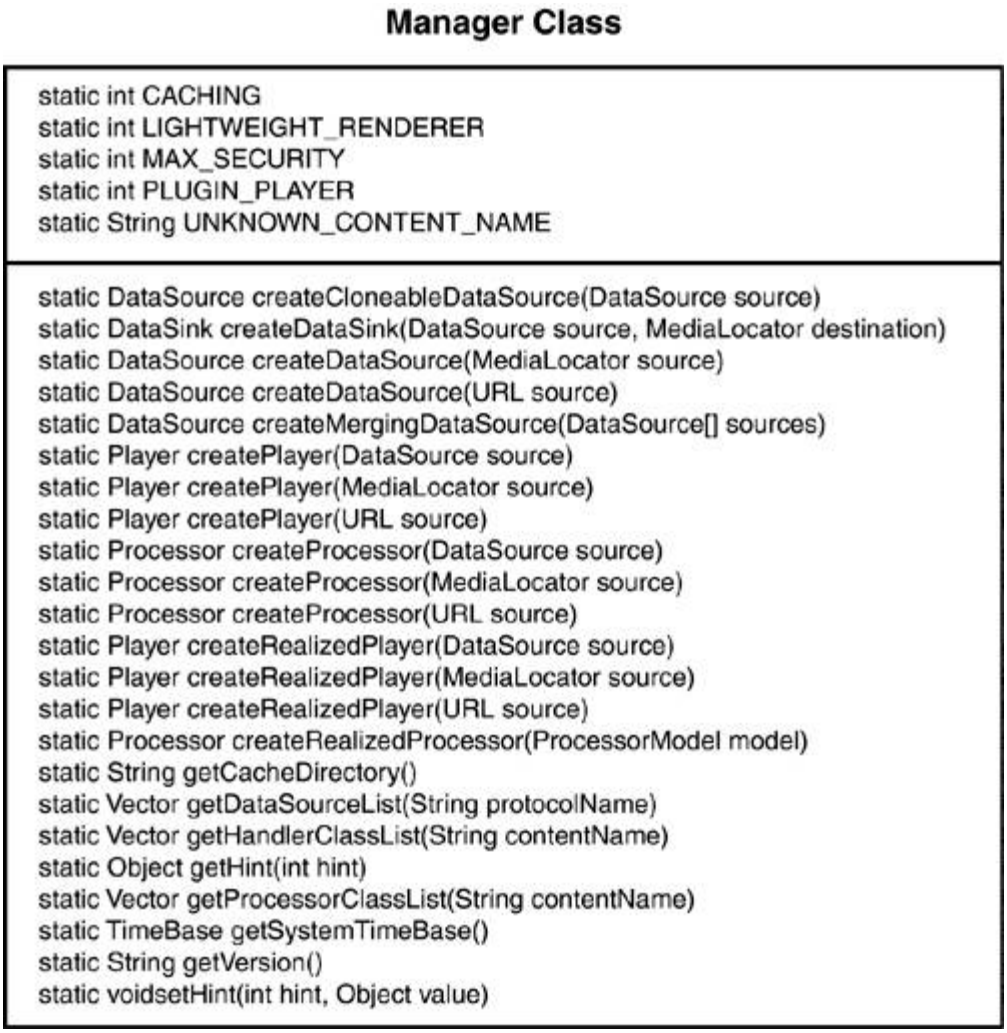
The `Buffer` class itself is complex with a large number of fields (most constants) and methods. Among the more important methods are `getData()`, `getHeader()`, `getLength()`, `getFormat()`, and `getSequenceNumber()`. The `Buffer` class will be discussed further in Chapter 9.

`Format`

The `Format` class is provided as the means of describing the format of media in an abstract sense: `Format` objects carry no information about encoding specific or timing specific information.

The `Format` class is extended by three more specific classes: `AudioFormat`, `VideoFormat`, and `ContentDescriptor`. The `VideoFormat` class itself is extended by a number of codec specific classes (for example, `H263Format`, `YUVFormat`). Similarly, the `ContentDescriptor` class is extended by the `FileDescriptor` class. Figure 8.17 shows the hierarchy of relationships between the `Format` classes, whereas the list that follows the figure indicates the chief purpose of each of the classes.

Figure 8.17. The class hierarchy stemming from the `Format` class.



`Format` is an abstraction of an exact media format. It is the superclass of all other format classes.

- `AudioFormat`— Format information specific to audio media such as number of channels, sampling rates, or quantization level.
- `ContentDescriptor`— Format information about media data containers: raw, raw RTP, and mixed.

  `FileTypeDescriptor`— Fundamentally an enumeration (listing) of all content types that are file based (for example, QuickTime, AVI, Wave, and so on).

- `VideoFormat`— An enumeration of the various video formats (codecs) supported by JMF (for example, motion JPEG, run-length encoding, and YUV) as well as video parameters such as frame rate and encoding type.

  `H261Format`— Format information specific to the H261 codec

  `H263Format`— Format information specific to the H263 codec

  `IndexedColorFormat`— Format information specific to the indexed color codec

- `JPEGFormat`— Format information specific to the JPEG codec and decimation schemes
- `RGBFormat`— Format information specific to the RGB codec
- `YUVFormat`— Format information specific to the YUV codec

Each `SourceStream` (managed by a `DataSource`) has an associated `ContentDescriptor` that can be obtained with the `getContentDescriptor()` method of `SourceStream`. Similarly, each `Buffer` object has an associated Format that can be obtained with the `getFormat()` method.

As with `SourceStreams`, many JMF programs providing significant functionality remain blissfully free of the details inherent in the various `Format` classes. The `ManagerQuery` application from earlier in the chapter employs the various constants (static final attributes) of the `Format` classes in order to generate a comprehensive list of all formats supported by the JMF.

The most important methods of the `Format` class are `getDataType()`, which returns a `Class` that is the type of data (for example, a byte array); `getEncoding()`, which returns a `String` uniquely identifying the encoding; and `matches()`, which is a `boolean` method accepting another `Format` and returning `true` if they match.

`MediaHandler`

`MediaHandler` is the centralized interface for objects that read and control media delivered from a `DataSource` object. Interfaces that extend `MediaHandler` include `Player`, `Processor`, `DataSink`, and `Demultiplexer`, which are all key classes commonly used in JMF programs.

The `MediaHandler` interface consists of a single method `setSource()`. The method accepts a `DataSource` and associates or links that `DataSource` with the object in question: that is the location in which the data to be handled will be obtained.

The creation of Players, Processors, or DataSinks by the Manager class requires a DataSource that the Manager needs to create as a prestep if it isn't supplied with a DataSource as part of the call. Part of the creation process by the Manager class is the invocation of the setSource() method on the Player, Processor, or DataSink. That is why the method generally isn't called from user code. Indeed, the setSource() method can throw either an IOException or IncompatibleSourceException. The presence or lack of an exception is how the Manager class searches the list of possible candidate classes until one that supports a particular DataSource (and hence media format) is found.

Playing Media

One of the most common uses of the JMF is for playing media: audio or video. As already illustrated in the previous chapter with BBPApplet, playing media with the JMF is a relatively simple process because of the abstraction from format details provided by the JMF.

The JMF provides the Player class for playing media. Player creation is achieved through the Manager class, as described in a previous section, by specifying the location of the media. A Player object provides methods and additional associated objects (Control objects) for controlling the playback: starting and stopping, setting the rate of play, and so on. Indeed the Player class exposes graphical (AWT) Components for control and visualization of the playback. These can be added directly to a GUI, preventing the need for the user to construct a large number of GUI components and linking those to actions (methods) on the Player object.

Player Interface

A Player is a MediaHandler object for controlling and rendering media. It is the chief class within the JMF for combined handling and rendering and provides an abstracted, high-level model of the process to user code that separates it from lower-level considerations such as the format of the media tracks. Hence, code employing a Player sees a single object with the same methods and attributes, regardless of the particulars of the media being played.

The Player interface directly extends that of Controller. Thus, a Player not only provides all the methods detailed for a Controller, but also supports the five-state model of stopped time (Unrealized, Realizing, Realized, Prefetching, Prefetched) of Controller. Understanding the Player interface requires an understanding of the Controller interface. However only the extensions of Player to the Controller interface will be discussed here; readers needing to reacquaint themselves with the features of Controller should revisit the first section of this chapter.

Creation of a Player (with the exception of the MediaPlayer Bean, which is discussed in the next subsection) is achieved through the central Manager class. As discussed in the earlier section on the Manager, there are six methods for creating a Player; three for creating an Unrealized Player and three for creating a Realized Player. A Player can be created on the basis of a URL, MediaLocator, or DataSource. The following code fragment is typical of Player creation:

```
String mediaName = "ftp://ftp.cs.adfa.edu.au/media/someExample.wav";
MediaLocator locator = new MediaLocator(mediaName);
Player thePlayer = Manager.createPlayer(locator);
```

Keeping track of the status and maintaining control of a `Player` is achieved through the same mechanism as that for `Controller` discussed earlier. Objects can implement the `ControllerListener` interface and be added as listeners to the `Player`. The `Player` posts events (discussed in the `Controller` section of the chapter) indicating state transitions (for example, Realizing to Realized), errors, and other relevant media events (for example, end of media reached, duration of media now known).

A number of the methods of `Player`, most inherited from `Controller`, exert direct control over the state of a `Player`. Others can alter the state as a side effect. The following list shows those methods:

`realize()`— Moves the `Player` from `Unrealized` to `Realized`. Asynchronous (returns immediately).

`prefetch()`— Moves the `Player` from `Realized` to `Prefetched`. Asynchronous (returns immediately).

`start()`— Moves the `Player` from whatever state it is in currently to `Started`. The `Player` will transition through any intermediary states and send notice of such transitions to any listeners. Asynchronous (returns immediately).

`stop()`— Moves the `Player` from `Started` state back to either (depending on the type of media being played) `Prefetched` or `Realized`. Synchronous (returns only after `Player` is actually stopped).

`deallocate()`— Deallocates the resources acquired by the `Player` and moves it to the `Realized` state (if at least `Realized` at the time of call) or to `Unrealized` (if in `Realizing` or `Unrealized` at time of call). Synchronous.

`close()`— Closes down the `Player` totally. It will post a `ControllerClosedEvent`. The behavior of any further calls on the `Player` is unspecified: The `Player` shouldn't be used after it is closed. Synchronous.

`setStopTime()`— Sets the media time at which it will stop. When the media reaches that time, the `Player` will act as though `stop()` was called. Asynchronous (returns immediately and stops play when time reaches that specified).

`setMediaTime()`— Changes the media's current time. This can result in a state transition. For instance, a started `Player` might return to `Prefetching` while it refills buffers for the new location in the stream. Asynchronous (in that any state changes triggered aren't waited for).

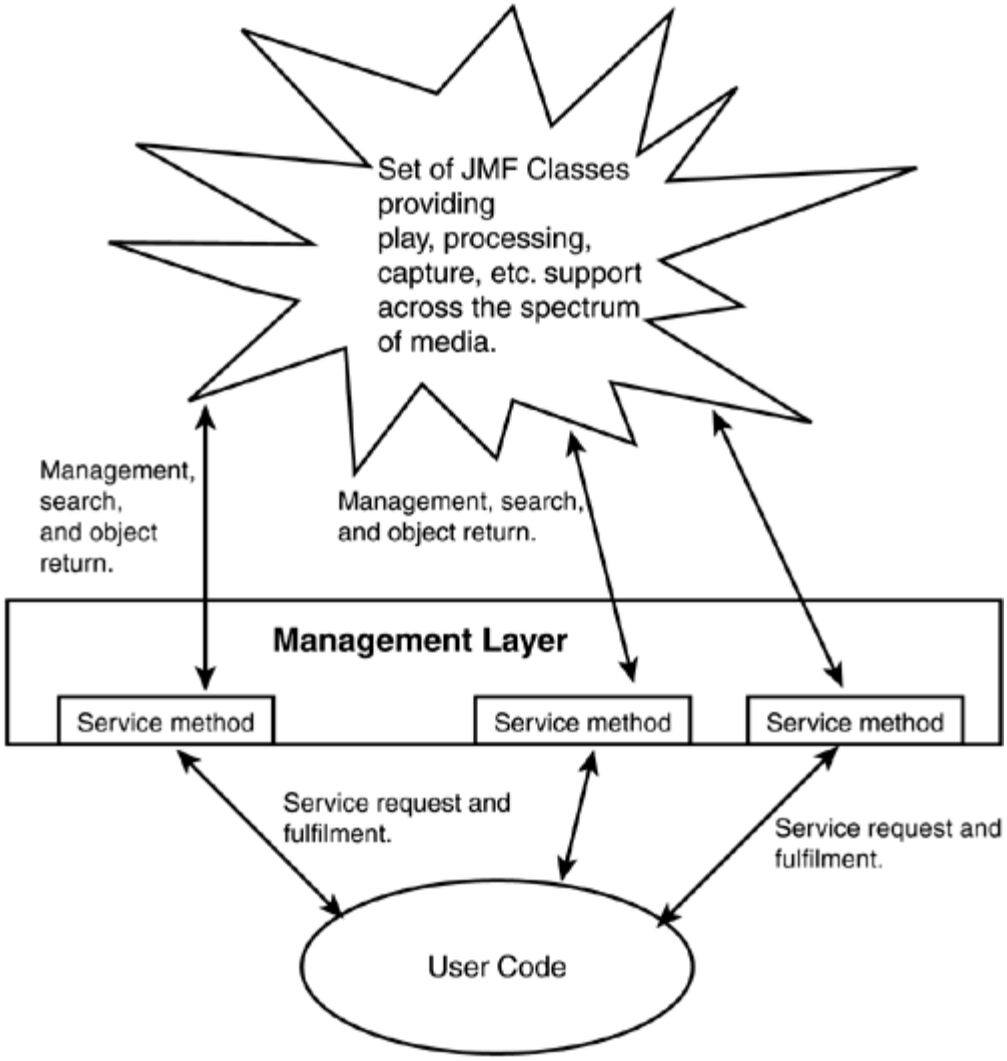
`setRate()`— Sets the rate of playback. Can result in a state change on a started `Player` for similar reasons to those for `setMediaTime()`. Asynchronous (in that any state changes triggered aren't waited for).

As you can see, many (of the most important) methods are asynchronous, returning to the caller immediately but before any state transitions can occur. This is particular true of starting a `Player`: Gathering the resources necessary to play media can be a time-consuming process.

This in turn illustrates the need to listen to a `Player`: User code must listen to the events generated by a `Player` so that it can determine, for example, when to place control components on the screen or when to show (and remove) download progress bars.

The `Player` class extends `Controller` with the addition of six methods as shown in Figure 8.18. One method is extremely useful and commonly employed. The `start()` method will transition a `Player` object through all intermediary stopped states, regardless of starting state (for example, `Unrealized`), to `Started`. This is the most common way to start a `Player` because it lifts the burden of calculating an appropriate synchronization time to be passed to the `syncStart()` methods. State transitions are still reported to all `ControllerListener`'s listening to the `Player`. Thus it is still possible to initiate appropriate actions (for example, obtain control and visual `Components`) as the `Player` passes through the appropriate states.

Figure 8.18. The `Player` interface.



Two other methods of Player, getControlPanelComponent() and getVisualComponent(), find high usage. These are used to obtain AWT Components suitable for controlling and displaying the output (playback) of the Player, respectively. These methods are only valid on a Player that is in a Realized, or higher (closer to Started) state. Calling them on an Unrealized player will result in a NotRealizedError exception being thrown. Hence, it is necessary to listen to the Player's events to ascertain when these Components can safely be obtained. Further, the methods return null if there isn't a valid Component, such as is the case with a visual Component for purely audio media. The user should always check that the returned object isn't null before it is used (generally added to a GUI).

Finally, the GainControl object obtained with getGainControl() can be used to control the gain (volume) of playback. The addController() and removeController() methods are provided for cases in which a single Player controls the synchronized handling (generally other Players) of multiple media. This is discussed briefly in a subsequent subsection.

The MediaPlayer Bean

The JMF directly exposes one class that implements the Player interface: MediaPlayer. MediaPlayer is a fully featured JMF Player encapsulated as a Java Bean.

Playing media with the MediaPlayer Bean is quite simple and painless because not only are various methods provided for configuring the behavior of the Bean, but it also handles the transition from playing one media to another when a different location is set. Set the PlayerOfMedia example application in the following section for user code to deal with that issue.

The following code fragment shows how simple setting up and starting a MediaPlayer can be. In this case, the fragment sets the MediaPlayer to play the media in videos/example1.mov once.

```
Player mp = new MediaPlayer();
mp.setLoop(false);
mp.setMediaLocation("file://videos.example1.mov");
mp.start();
```

Controlling Multiple Players/Controllers

It is possible through the addController() and removeController() methods of Player to employ a single Player object to manage a number of Controllers. This is an approach that allows playback (or indeed processing) among a number of Players to be synchronized by being driven by a single Player.

There are a number of details and caveats to the approach, and users considering such synchronization should carefully read the Player class specification, which includes a lengthy discussion of this topic. However, the fundamental model of the approach is that Controllers added to a Player fall under the control of that Player. Method calls on the Player are also made on the added Controllers, and events from the Controllers can propagate back to those listening to the Player object.

Application: Player of Media

This subsection discusses an example application, `PlayerOfMedia`, that illustrates a number of the features of the `Controller`, `Player`, and other classes already discussed.

As its name indicates, `PlayerOfMedia` is an application capable of rendering time-based media. Through a simple menu system, the user might select to open (for play) any media. He might then enter the URL of the media or browse the local filesystem to select a file. If the application is unable to create a `Player` for the media, the user is informed via a pop-up dialog box. Otherwise, the application adds the appropriate visual and control components to the GUI, resizing itself appropriately, and hands control to the user (through the control component). When the end of the media is reached, play restarts from the beginning.

In terms of core functionality, the ability to play media, `PlayerOfMedia`, isn't dissimilar to the `BBPApplet` example from Chapter 7. However, `PlayerOfMedia` provides further features in its ability for users to specify the actual media they want played. That requires the handling of not only arbitrary media, but also more importantly graceful handling of errors or exceptions as well as resource management. If a new `Player` object is created, the old `Player` object (if one exists) and its associated resources must be freed up.

Listing 8.7 shows the source of the application, which can also be found on the book's companion Web site. It is worth noting that the greater portion of the code doesn't concern the JMF per se, but rather the graphical user interface (simple as it is) that the program provides. The creation of menus and dialog boxes is one of the main reasons for the size of the code. Figure 8.19 shows the application as the user is playing one media (a video) and about to replace it with another (a WAVE file of didgeridoo play).

Figure 8.19. The `PlayerOfMedia` GUI application running and displaying a video while the user is about to open a different media file.

The two methods, `createAPlayer()` and `controllerUpdate()`, directly concern the JMF. The `createAPlayer()` method is responsible for creating a new `Player` and moving it to the `Realized` state. It is passed a `String`, which is the name and location of the media to be played. However, before a new `Player` can be created, the method ascertains if there is an existing Player object. If so, it is stopped and any resources it had acquired are freed with the `close()` call. Similarly, the old `Player` is no longer listened to (events generated from it won't be received). A `MediaLocator` is constructed and passed to the `Manager` class in order to create the new `Player`. Any resulting exceptions are caught and the user informed. Otherwise, if the process of creation was successful, the application starts listening to the new `Player` as well as directing it to become `Realized`.

The `controllerUpdate()` method is required for any class that implements the `ControllerListener` interface and is where nearly all the control of the `Player` object occurs. The method is called with events generated by the currently listened to `Player` object. These typically represent state transition events, such as from `Realizing` to `Realized`, but can also include errors or information about the duration of the media. The method reacts to the particular type of event received and acts accordingly, which perhaps alters its user interface, moves the `Player` along to the next appropriate state, or even posts an error message. In particular, a `RealizeComplete` event means that visual and control `Components` can be obtained for the new `Player` object. Any old `Components` (from the previous `Player`) or download progress bars should first be discarded; then the new ones should be obtained and

added to the `GUI`. Reaching the end of play (signaled by an `EndOfMedia` event) is dealt with by setting the media's own time to zero (back to the start) and restarting the `Player`.

Listing 8.7 The `PlayerOfMedia` GUI application that Allows the User to Select the Media He Wants to Play

```
import java.awt.*;
import java.awt.event.*;
import javax.media.*;
import javax.media.protocol.*;
import javax.media.control.*;
import java.io.*;
/**************************************************************
* A Graphical application allowing the user to choose the media
* they wish to play.
* PlayerOfMedia presents a Dialog in which the user may enter
* the URL of the media to play, or select a file from the local
* system.
*
* @author Spike Barlow
**************************************************************/
public class PlayerOfMedia extends Frame implements ActionListener,
      ControllerListener {

  /** Location of the media. */
  MediaLocator  locator;

  /** Player for the media */
  Player        player;

  /** Dialog for user to select media to play. */
  Dialog        selectionDialog;

  /** Buttons on user dialog box. */
  Button        cancel,
                open,
                choose;

  /** Field for user to enter media filename */
  TextField     mediaName;

  /** The menus */
  MenuBar       bar;
  Menu          fileMenu;
  /** Dialog for informing user of errors. */
  Dialog        errorDialog;
  Label         errorLabel;
  Button        ok;

  /** Graphical component for controlling player. */
  Component     controlComponent;

  /** Graphical component showing what isbeing played. */
  Component     visualComponent;

  /** Graphical component to show download progress. */
  Component     progressBar;
```

```
  /** Sizes to ensure Frame is correctly sized. */
  Dimension     controlSize;
  Dimension     visualSize;
  int           menuHeight = 50;

  /** Directory user last played a file from. */
  String        lastDirectory = null;

  /** Flags indicating conditions for resizing the Frame. */
  protected static final int  VISUAL = 1;
  protected static final int  PROGRESS = 2;


/*************************************************************
* Construct a PlayerOfMedia. The Frame will have the default
* title of "Player of Media". All initial actions on the
* PlayerOfMedia object are initiated through its menu
* (or shotcut key).
*************************************************************/
PlayerOfMedia() { this("Player of Media"); }

/*************************************************************
* Construct a PlayerOfMedia. The Frame will have the title
* supplied by the user. All initial actions on the
* PlayerOfMedia object are initiated through its menu
* (or shotcut key).
*************************************************************/
PlayerOfMedia(String name) {

  super(name);
  ////////////////////////////////////////////////////////
  // Setup the menu system: a "File" menu with Open and Quit.
  ////////////////////////////////////////////////////////
  bar = new MenuBar();
  fileMenu = new Menu("File");
  MenuItem openMI = new MenuItem("Open...",
new MenuShortcut(KeyEvent.VK_O));
  openMI.setActionCommand("OPEN");
  openMI.addActionListener(this);
  fileMenu.add(openMI);
  MenuItem quitMI = new MenuItem("Quit",
new MenuShortcut(KeyEvent.VK_Q));
  quitMI.addActionListener(this);
  quitMI.setActionCommand("QUIT");
  fileMenu.add(quitMI);
  bar.add(fileMenu);
  setMenuBar(bar);

  ////////////////////////////////////////////////////
  // Layout the frame, its position on screen, and ensure
  // window closes are dealt with properly, including
  // relinquishing the resources of any Player.
  ////////////////////////////////////////////////////
  setLayout(new BorderLayout());
  setLocation(100,100);
  addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
      if (player!=null) { player.stop(); player.close();}
```

```
      System.exit(0); } });

  ////////////////////////////////////////////////////
  // Build the Dialog box by which the user can select
  // the media to play.
  ////////////////////////////////////////////////////
  selectionDialog = new Dialog(this,"Media Selection");
  Panel pan = new Panel();
  pan.setLayout(new GridBagLayout());
  GridBagConstraints gbc = new GridBagConstraints();
  mediaName =new TextField(40);
  gbc.gridx = 0; gbc.gridy = 0; gbc.gridwidth=2;
  pan.add(mediaName,gbc);
  choose = new Button("Choose File...");
  gbc.ipadx = 10; gbc.ipady = 10;
  gbc.gridx = 2; gbc.gridwidth= 1; pan.add(choose,gbc);
  choose.addActionListener(this);
  open = new Button("Open");
  gbc.gridy = 1; gbc.gridx = 1; pan.add(open,gbc);
  open.addActionListener(this);
  cancel = new Button("Cancel");
  gbc.gridx = 2; pan.add(cancel,gbc);
  cancel.addActionListener(this);
  selectionDialog.add(pan);
  selectionDialog.pack();
  selectionDialog.setLocation(200,200);

  ////////////////////////////////////////////////////
  // Build the error Dialog box by which the user can
  // be informed of any errors or problems.
  ////////////////////////////////////////////////////
  errorDialog = new Dialog(this,"Error",true);
  errorLabel = new Label("");
  errorDialog.add(errorLabel,"North");
  ok = new Button("OK");
  ok.addActionListener(this);
  errorDialog.add(ok,"South");
  errorDialog.pack();
  errorDialog.setLocation(150,300);

  Manager.setHint(Manager.PLUGIN_PLAYER,new Boolean(true));
}

/**************************************************************
* React to menu selections (quit or open) or one of the
* the buttons on the dialog boxes.
**************************************************************/
public void actionPerformed(ActionEvent e) {

  if (e.getSource() instanceof MenuItem) {
    ////////////////////////////////////////////////
    // Quit and free up any player acquired resources.
    ////////////////////////////////////////////////
    if (e.getActionCommand().equalsIgnoreCase("QUIT")) {
      if (player!=null) {
        player.stop();
        player.close();
      }
      System.exit(0);
```

```
    }
    ////////////////////////////////////////////////////
    // User to open/play media. Show the selection dialog box.
    ////////////////////////////////////////////////////
    else if (e.getActionCommand().equalsIgnoreCase("OPEN")) {
      selectionDialog.show();
    }
  }
  //////////////////////
  // One of the Buttons.
  //////////////////////
  else {
    ////////////////////////////////////////////////////////////
    // User to browse the local file system. Popup a file dialog.
    ////////////////////////////////////////////////////////////
    if (e.getSource()==choose) {
      FileDialog choice = new FileDialog(this,
"Media File Choice",FileDialog.LOAD);
      if (lastDirectory!=null)
        choice.setDirectory(lastDirectory);
      choice.show();
      String selection = choice.getFile();
      if (selection!=null) {
        lastDirectory = choice.getDirectory();
        mediaName.setText("file://"+ choice.getDirectory() +
selection);
      }
    }
    ////////////////////////////////////////////
    // User chooses to cancel opening of new media.
    ////////////////////////////////////////////
    else if (e.getSource()==cancel) {
      selectionDialog.hide();
    }
    ////////////////////////////////////////////////////
    // User has selected the name of the media. Attempt to
    // create a Player.
    ////////////////////////////////////////////////////
    else if (e.getSource()==open) {
      selectionDialog.hide();
      createAPlayer(mediaName.getText());
    }
    ////////////////////////////////////////
    // User has seen error message. Now hide it.
    ////////////////////////////////////////
    else if (e.getSource()==ok)
      errorDialog.hide();
  }
}


/**************************************************************
* Attempt to create a Player for the media who's name is passed
* the the method. If successful the object will listen to the
* new Player and start it towards Realized.
**************************************************************/
protected void createAPlayer(String nameOfMedia) {

  ////////////////////////////////////////////////////////
```

```
  // If an existing player then stop listening to it and free
  // up its resources.
  ////////////////////////////////////////////////////////////
  if (player!=null) {
    System.out.println("Stopping and closing previous player");
    player.removeControllerListener(this);
    player.stop();
    player.close();
  }

  ////////////////////////////////////////////////////////////
  // Use Manager class to create Player from a MediaLocator.
  // If exceptions are thrown then inform user and recover
  // (go no further).
  ////////////////////////////////////////////////////////////
  locator = new MediaLocator(nameOfMedia);
  try {
    System.out.println("Creating player");
    player = Manager.createPlayer(locator);
  }
  catch (IOException ioe) {
    errorDialog("Can't open " + nameOfMedia);
    return;
  }
  catch (NoPlayerException npe) {
    errorDialog("No player available for " + nameOfMedia);
    return;
  }

  ////////////////////////////////////////////////////////////
  // Player created successfully. Start listening to it and
  // realize it.
  ////////////////////////////////////////////////////////////
  player.addControllerListener(this);
  System.out.println("Attempting to realize player");
  player.realize();
}


/************************************************************
* Popup a dialog box informing the user of some error. The
* passed argument isthe text of the message.
************************************************************/
protected void errorDialog(String errorMessage) {

  errorLabel.setText(errorMessage);
  errorDialog.pack();
  errorDialog.show();
}


/************************************************************
* Resize the Frame (window) due to the addition or removal of
* Components.
************************************************************/
protected void resize(int mode) {
  ////////////////////////////////////////
  // Player's display and controls in frame.
  ////////////////////////////////////////
```

```
  if (mode==VISUAL) {
    int maxWidth = (int)Math.max(controlSize.width,visualSize.width);
    setSize(maxWidth,
controlSize.height+visualSize.height+menuHeight);
  }
  ////////////////////////////////
  // Progress bar (only) in frame.
  ////////////////////////////////
  else if (mode==PROGRESS) {
    Dimension progressSize = progressBar.getPreferredSize();
    setSize(progressSize.width,progressSize.height+menuHeight);
  }
  validate();
}

/**************************************************************
* React to events from the player so as to drive the presentation
* or catch any exceptions.
**************************************************************/
public synchronized void controllerUpdate(ControllerEvent e) {

  ////////////////////////////////////
  // Events from a "dead" player. Ignore.
  ////////////////////////////////////
  if (player==null)
    return;

  ////////////////////////////////////////////////////////
  // Player has reached realized state. Need to tidy up any
  // download or visual components from previous player. Then
  // obtain visual and control components for the player,add
  // them to the screen and resize window appropriately.
  ////////////////////////////////////////////////////////
  if (e instanceof RealizeCompleteEvent) {
    ////////////////////////////////////////////////
    // Remove any inappropriate Components from display.
    ////////////////////////////////////////////////
    if (progressBar!=null) {
      remove(progressBar);
      progressBar = null;
    }
    if (controlComponent!=null) {
      remove(controlComponent);
      validate();
    }
    if (visualComponent!=null) {
      remove(visualComponent);
      validate();
    }
    ////////////////////////////////////////////////////
    // Add control and visual components for new player to
    // display.
    ////////////////////////////////////////////////////
    controlComponent = player.getControlPanelComponent();
    if (controlComponent!=null) {
      controlSize = controlComponent.getPreferredSize();
      add(controlComponent,"Center");
    }
    else
```

```
      controlSize = new Dimension(0,0);
    visualComponent = player.getVisualComponent();
    if (visualComponent!=null) {
      visualSize = visualComponent.getPreferredSize();
      add(visualComponent,"North");
    }
    else
      visualSize = new Dimension(0,0);
    ////////////////////////////////////////////////////
    // Resize frame for new components and move to prefetched.
    ////////////////////////////////////////////////////
    resize(VISUAL);
    System.out.println("Player is now pre-fetching");
    player.prefetch();
  }
  ////////////////////////////////////////////////////
  // Provide user with a progress bar for "lengthy" downloads.
  ////////////////////////////////////////////////////
  else if (e instanceof CachingControlEvent &&
player.getState() <= Player.Realizing && progressBar==null) {
    CachingControlEvent cce = (CachingControlEvent)e;
    progressBar = cce.getCachingControl().getControlComponent();
    if (progressBar!=null) {
      add(progressBar,"Center");
      resize(PROGRESS);
    }
  }
  ////////////////////////////////////////////////
  // Player initialisation complete. Start it up.
  ////////////////////////////////////////////////
  else if (e instanceof PrefetchCompleteEvent) {
    System.out.println("Pre-fetching complete, now starting");
    player.start();
  }
  ////////////////////////////////////////////////////
  // Reached end of media. Start over from the beginning.
  ////////////////////////////////////////////////////
  else if (e instanceof EndOfMediaEvent) {
    player.setMediaTime(new Time(0));
    System.out.println("End of Media – restarting");
    player.start();
  }
  ////////////////////////////////////////////////////////////
  // Some form of error. Free up all resources associated with
  // the player, don't listen to it anymore, and inform the
  // user.
  ////////////////////////////////////////////////////////////
  else if (e instanceof ControllerErrorEvent) {
    player.removeControllerListener(this);
    player.stop();
    player.close();
    errorDialog("Controller Error, abandoning media");
  }

}
```

```
/**************************************************************
* Create a PlayerOfMedia object and pop it up on the screen for the
* user to interact with.
**************************************************************/
public static void main(String[] args) {

  PlayerOfMedia  ourPlayer = new PlayerOfMedia();
  ourPlayer.pack();
  ourPlayer.setSize(200,100);
  ourPlayer.show();
}
}
```

Playing Media with a Processor

One of the desirable extensions to the `PlayerOfMedia` application would be for it to provide a number of statistics about the media it is playing: frame rate, size, duration, content type, and codec. However, using a `Player`, most of that information simply isn't available because a `Player` provides no control over any of the processing that it performs on the media, nor over how it renders the media itself. The very abstraction that makes it relatively simple to write programs that play media also makes it impossible to determine much information about the media being played.

An alternative to using a `Player` object to play (render) media is to use a `Processor` object. `Processor` objects are discussed in detail in a later section of this chapter. Details aside, a `Processor` is really just a specialized type of `Player` that allows control over the processing that is performed on the input media stream. The output of a `Processor` is either a `DataSource` or it is rendered (played).

In order to force a `Processor` to render the media rather than outputting it, `Processor's setContentDescriptor()` method should be passed `null`. This will be revisited in the `Processor` section, but passing the `null` reference implies do not create an output `DataSource` (that is, render the media).

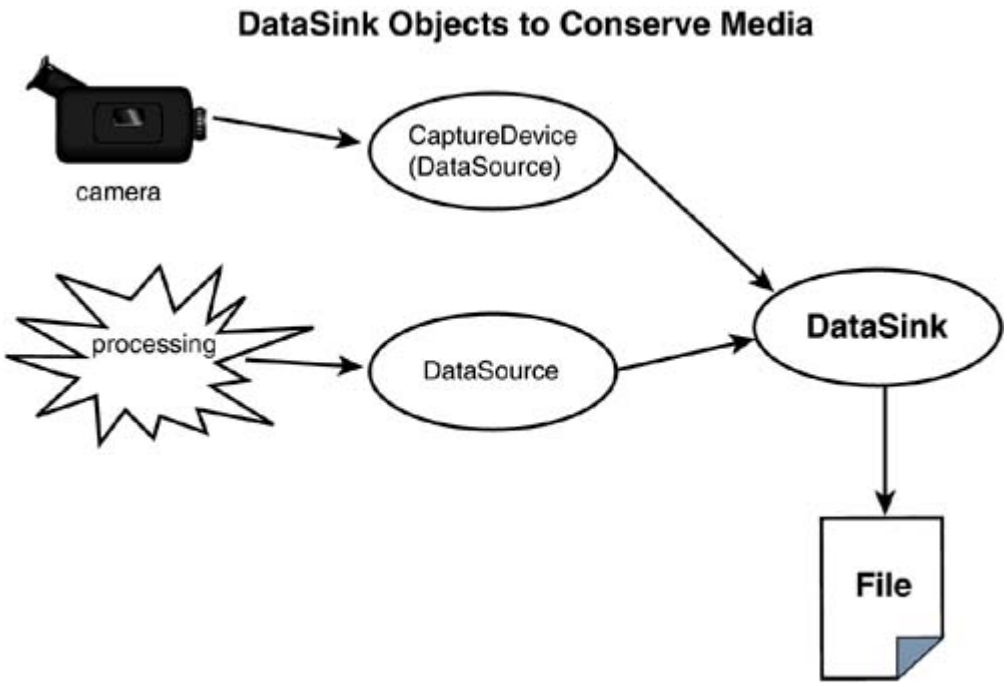
`Processors` provide all the user interface control features of `Players` as well as the ability to access the individual tracks that compose the `DataSource`. Through this mechanism, their formats can be ascertained and such a report generated. The subsequent example `MediaStatistics` shows how such a report can be ascertained. A program to play media that also reported statistics on the media would then employ a `Processor` (rather than `Player`) and combine the features of `PlayerOfMedia` and `MediaStatistics`.

Conserving Media

It is often desirable to keep a permanent copy of media by saving it to a file. The media is then available for subsequent playback, processing, or broadcast. The original media to be saved might be captured from a microphone or camera, the result of some processing (such as transcoding), or a broadcast streaming across the network. In JMF, all these instances are represented as `DataSources`,

and the class used to save media is known as a `DataSink`. Figure 8.20 shows these possible applications of a `DataSink`.

Figure 8.20. Uses of a `DataSink` object to save media to a file.



DataSink

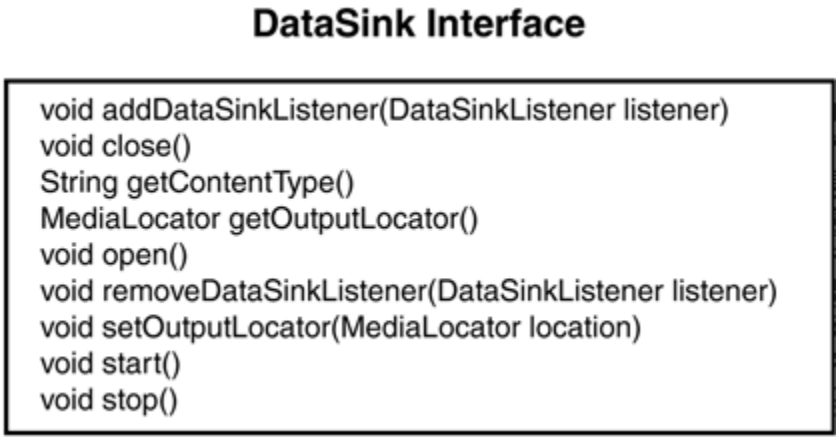
The `DataSink` interface specifies an object that accepts media from a `DataSource` and renders it to some destination. Most commonly, that destination is a local file, but it could equally be writing or broadcasting across the network. Hence `DataSinks` are important objects, and are often seen in JMF programs.

As discussed earlier, a `DataSink` object is created through the `Manager` class with the static `createDataSink()` method. The method expects two parameters: the `DataSource` to which the `DataSink` will be connected and a `MediaLocator` specifying the destination that is the sink. The `Manager` class returns a `DataSink` object or throws a `NoDataSinkException` if it was unable to create the `DataSink`. The following code fragment shows the typical creation process:

```
DataSource source;
MediaLocator destination;
DataSink sink;
:        :
// Code that would see source and destination with valid values
:        :
try { sink = Manager.createDataSink(source,destination); }
catch (NoDataSinkException nde) { // Do something }
```

Figure 8.21 shows the methods of `DataSink` (excluding those inherited from `MediaHandler` and `Controls` which `DataSink` extends). Transfer is managed through the `open()`, `start()`, `stop()`, and `close()` methods. The `open()` method opens a connection to the destination (specified by the `MediaLocator` when the `DataSource` was created). The method might throw an `IOException` or a `SecurityException` (for example, not allowed to write to file system when an applet). After an output connection has been established with `open()`, transfer can be initiated with the `start()` method. This method also might throw an `IOException`. Transfer is halted with the `stop()` method (which can throw an `IOException`), whereas all resources are freed and the connection closed down with the `close()` method.

Figure 8.21. The `DataSink` interface.



`DataSink` objects generated `DataSinkEvent` events and can be listened to by those classes that implement `DataSinkListener`. The methods associated with events are `addDataSinkListener()` and `removeDataSinkListener()`. These events generated by `DataSink` objects are discussed in the next subsection.

`setOutputLocator()` and `getOutputLocator()`, as their names imply, are used as a means of specifying or obtaining the output `MediaLocator`—where the `DataSink` writes its output. The `setOutputLocator()` method is rarely used by user programs because this action is performed by the `Manager` class as part of the `DataSink` creation process. It is an error (an error is thrown) to call the method more than once. The `getContentType()` method returns a `String` specifying the content type of the media that is being consumed by the `DataSink`.

Employing a `DataSink` usually follows a number of simple steps:

1. Create the `DataSink` (from a `DataSource`).
2. Listen for events from the `DataSink`.
3. `open()` and `start()` the `DataSink`.

4. When the transfer is complete (for example, end of media reached), `stop()` and `close()` the `DataSink`.

`DataSink` Events

`DataSink` objects generate `DataSinkEvent` events in order to communicate the status of the `DataSink`. `DataSinkEvent` objects have two subclasses that indicate the two types of events a `DataSink` generates: `DataSinkErrorEvent` and `EndOfStreamEvent`. As should be evident from their names, these events either indicate an error with the `DataSink` (`DataSinkErrorEvent`) or the `DataSource` feeding the `DataSink` has signaled an end-of-stream (no more data).

Those objects wanting to receive events from a `DataSink` must implement the `DataSinkListener` interface. The interface consists of a single method:

```
void dataSinkUpdate(DataSinkEvent e)
```

Listing 8.8 shows a typical use of a `DataSink` object (sink) to preserve media coming from a `DataSource` object (source). Note the use of an anonymous class to listen to events generated by the `DataSink` and close it when the end of media stream has been detected. The anonymous listener class also performs error detection.

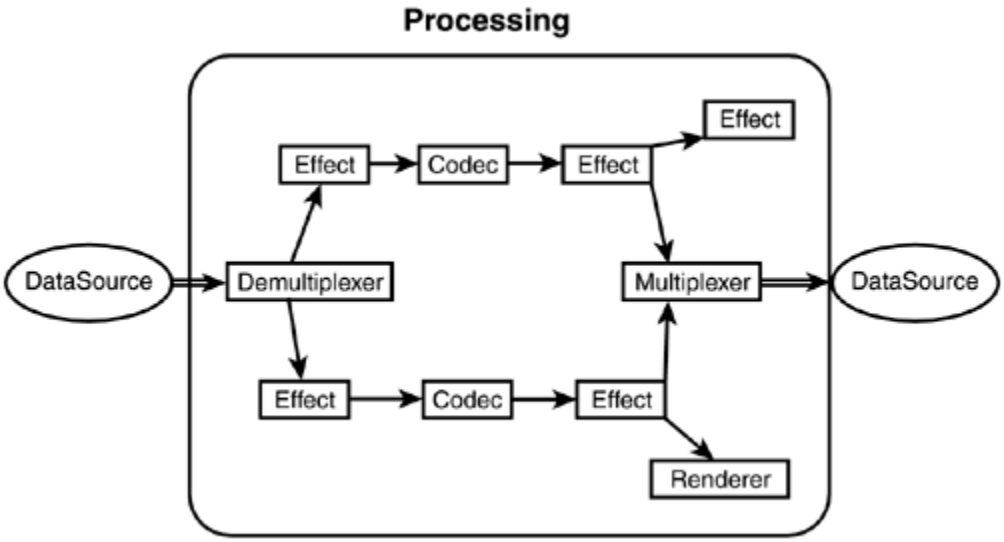
Listing 8.8 Use of a `DataSink` and Its Event Handling

```
DataSource    source;           // Assumed to already exist.
DataSink    sink;
MediaLocator    destinationLocation = ...;  // Create destination appropriately
try { sink = Manager.createDataSink(source,destinationLocation); }
catch (NoDataSinkException nde) {
  // Print an appropriate error message then rethrow exception
  throw nde;
}
sink.addDataSinkListener(new DataSinkListener() {
    public void dataSinkUpdate(DataSinkEvent e) {
      if (e instanceof EndOfStreamEvent) {
        sink.close();        // Will also stop the sink first
        source.disconnect();
      }
      else if (e instanceof DataSinkErrorEvent) {
        if (sink!=null)
          sink.close();
        if (source!=null)
          source.disconnect();
      }
    }    // End of dataSinkUpdate() method
  });    // End of addDataSinkListener() method
```

PlugIns

PlugIns are a powerful feature of the JMF that allow fine control over the processing of media. Fundamentally, as shown by Figure 8.22, the PlugIn model breaks processing into five component stages that can be chained together.

Figure 8.22. Position of the various PlugIns in the processing chain.



The components that correspond to each of these five stages are

Demultiplexer— Splits media into its constituent tracks

Codec— Compresses or decompresses a media track

Effect— Performs special effects processing on a media track; a generic category for any form of manipulation of the media data

Renderer— Plays (renders) a media track such as video to the screen or audio to speakers

Multiplexer— Combines media tacks into a single media stream

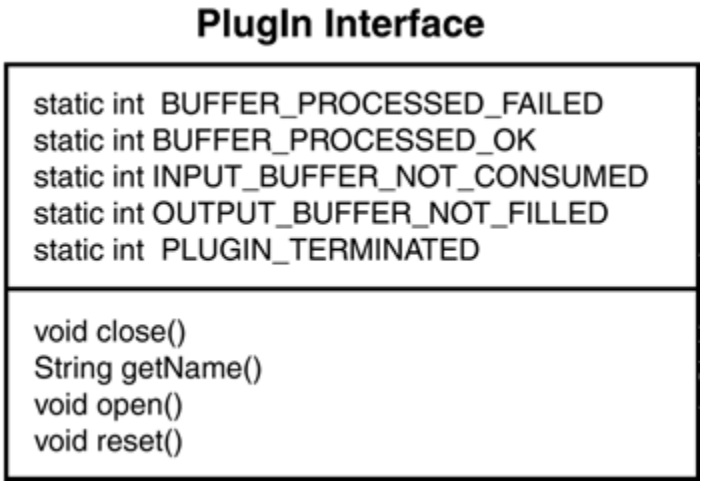
These forms of processing are discussed in greater detail, but independent of the JMF, in the previous chapter.

Processors (discussed in the next section) that support TrackControls allow the specification and order of the PlugIns. The PlugIns' order will operate on the individual tracks that compose a media item. Thus, the PlugIn model provides a programmer with fine control over the processing performed on media. Indeed, as discussed in Chapter 9, programmers can write their own classes that implement the various PlugIn interfaces.

One hindrance is worth mentioning here: It isn't mandatory that `PlugIns` be supported in all JMF implementations. Although the current implementations of version 2.1.1 (Sun reference, Windows performance, Solaris Performance, and Linux performance) all support `PlugIns`, it is possible that some future implementation might not. As an example, this could be the case with a JMF implementation for mobile computing (that is, mobile phones and PDAs) in which resource constraints are tight.

As shown in Figure 8.23, the `PlugIn` interface is relatively simple. All key functionality lies in the `Demultiplexer`, `Codec`, `Effect`, `Renderer`, and `Multiplexer` interfaces that extend `PlugIn`. Hence, a `PlugIn` is a generic processing unit that accepts media in a particular format and processes or presents that data. The `open()` and `close()` methods serve to prepare or terminate the `PlugIn` activity and acquire or free the resources required by the `PlugIn`, respectively. The `reset()` method resets the state of the `PlugIn`, whereas `getName()` returns a human-readable `String` defining the name of the `PlugIn`.

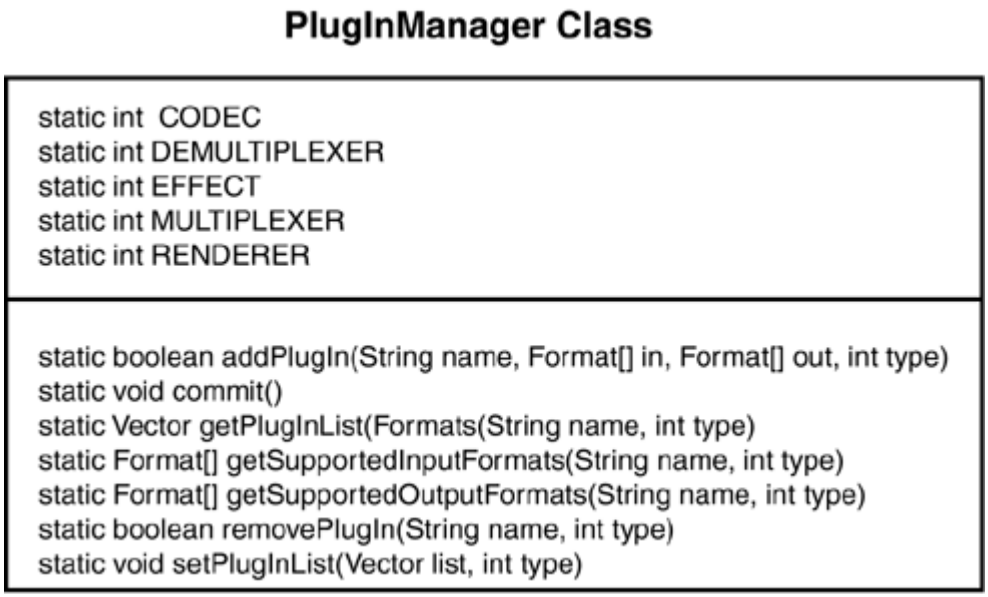
Figure 8.23. The `PlugIn` interface.



`PlugInManager`

Given the potential plethora of `PlugIns`, it isn't surprising that the JMF provides a manager class, `PlugInManager`, for maintaining a register of all `PlugIns` that can be both queried and altered. Figure 8.24 shows the methods and class variables of `PlugInManager`. As for the other manager classes, all methods are static (invoked using the class name).

Figure 8.24. The `PlugInManager` class.



`PlugInManager` finds particular use in those situations in which the list of `PlugIns` is being altered. It will be discussed more in Chapter 9, which deals with, among other things, extending the functionality of the JMF.

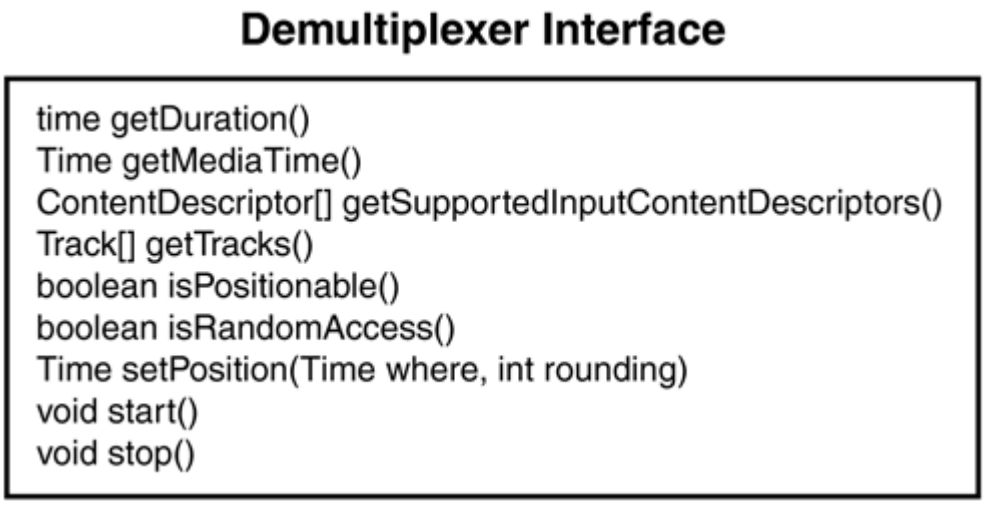
`PlugInManager` can also be used to query what `PlugIns` are available. The `getPlugInList()` method returns a `Vector` of `PlugIns` that are of a particular type (for example, `PlugInManager.CODEC`), support a particular input format, and produce a particular output format. Alternatively, given a particular `PlugIn`, the `getSupportedInputFormats()` and `getSupportedOutputFormats()` methods can be employed to discover what `Formats` the plug-in in question supports.

`Demultiplexer`

The first `PlugIn` in any processing chain is a `Demultiplexer`. Its task is to separate a media stream into its individual tracks. Hence, it has a single input and multiple (the number of tracks) outputs.

The `Demultiplexer` plug-in extends the `PlugIn`, `MediaHandler`, and `Duration` interfaces; in addition, it defines the methods as shown in Figure 8.25.

Figure 8.25. The `Demultiplexer` interface.



The single most important method of the interface is `getTracks()`, which returns an array of `Track` objects. This method might either throw a `BadHeaderException` (if the header information in the media is incomplete or inappropriate) or an `IOException`. The `start()` method must be called before `Track`'s `readFrame()` method will be called on any of the `Tracks` returned by `getTracks()`. The `stop()` method should be called when no further `Frames` are to be read.

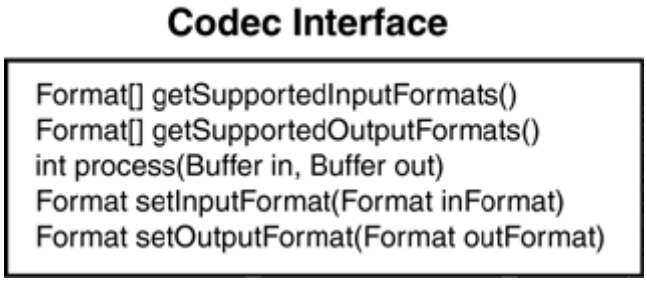
The interface also provides a number of informative methods: Among these is `getSupportedInputContentDescriptors()`, which returns an array of `ContentDescriptor` objects that are supported by the demultiplexer.

`Codec`

A `Codec` is a `PlugIn` that performs processing on an input `Buffer` and produces an output `Buffer`. This is a surprisingly broad definition because a codec is typically understood to be a compressor or decompressor: a processing unit that converts from one format to another. However, the JMF `Codec` interface is more encompassing: Any form of processing from input `Buffer` to output `Buffer` falls under the category of `Codec` (although see the next subsection on `Effects`).

`Codec` extends the `PlugIn` interface and thus includes all its methods. Figure 8.26 shows the methods of `Codec` itself. `Codecs` work in one of two modes known as frame based and stream based. Frame-based codecs can handle data of any size: Each processing call results in the consumption of the input `Buffer` and the production of an output `Buffer`. Stream based `Codecs` don't have the same synchronization between input and output `Buffers`. Each processing call might result in only a portion of the input `Buffer` being consumed (processed). Alternatively, an output `Buffer` might not be produced after each processing call.

Figure 8.26. The `Codec` interface.



The key method of the class is `process()`, which accepts an input `Buffer` and returns an output `Buffer` as a parameter. The supported input and output `Formats` of a `Codec` can be queried, whereas the particular `Formats` to be employed for input and output can also be set through the methods of the class.

`Effect`

The `Effect` interface is an empty interface that extends `Codec`: It represents objects that process media data in `Buffers` but don't alter its `Format`. For instance, an audio `Effect` might add reverb to a media track of a particular, or a number of, format(s).

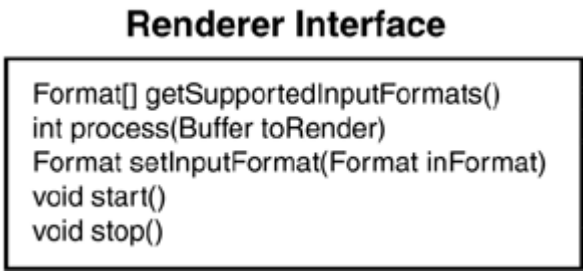
Hence, the `Effect` interface possesses all the methods of `Codec`, but no others.

`Renderer`

The `Renderer` interface defines a processing unit that renders (plays) a single track of media to a predefined device such as the display or speakers. Being a final link in the processing chain, it has a single input and no outputs.

Figure 8.27 shows the methods of `Renderer`. The single most important method is `process()`, which is provided with a `Buffer` that must be rendered. The `start()` method initiates the rendering process, whereas `stop()` halts it. The `Formats` supported by a `Renderer` can be queried, whereas the particular `Format` to use in order to render can be set through the appropriate methods.

Figure 8.27. The `Renderer` interface.

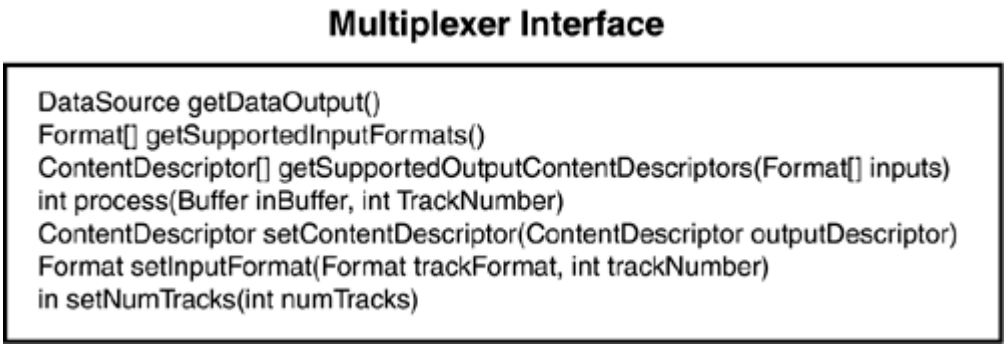


`Multiplexer`

The `Multiplexer` interfaces define a processing unit that combines one or more (typically more) media tracks into a single output content type (`ContentDescriptor`). The interleaved tracks are available as an output `DataSource` object.

Figure 8.28 shows the methods that the `Multiplexer` interface adds to `PlugIn`. Although, as with other `PlugIns`, the `process()` method is central, several other methods are also vital to achieve any multiplexing task. The `process()` method is provided with a `Buffer` that corresponds to a particular track number. Setting up a `Multiplexer` requires informing the `Multiplexer` of the number of tracks (using the `setNumTracks()` method) and the format of each of those tracks (using the `setInputFormat()` method), as well as specifying the required output content type (using the `setContentDescriptor()` method). The resulting `DataSource` is obtained with the `getDataOutput()` method.

Figure 8.28. The `Multiplexer` interface.



The following psuedo-code shows the typical steps involved in using a `Multiplexer`.

```
Set the output ContentDescriptor
Set the number of tracks
For each track
       Set its format
while there is more data to multiplex
       for each track
              Process that track's current Buffer
Get the output DataSource
```

Processing Media

In the context of the JMF, processing is a broad and encompassing term that includes all manipulation of time-based media. Examples of processing include compressing and decompressing; transcoding, changing between compression formats and adding digital effects; demultiplexing, splitting the media into tracks; multiplexing, combining tracks into a single stream; and rendering, playing back media. It

shouldn't be surprising that these examples match the five plug-in categories discussed in the previous section: `PlugIn`s are processing units within the processing chain.

Processing lies at the heart of all programs written to handle media. Although sourcing media with `DataSource` and `MediaLocator` objects is a necessity, and perhaps `DataSink` objects will also be used, processing is the reason for or purpose of the programming. Even playing media, a very common form of JMF program, falls under the umbrella of processing because to play media is to render it.

Not surprisingly then, understanding the JMF model of processing is a necessity for anyone who intends to carry out any significant JMF based programming. At the core of processing sits the `Processor` class, which will be discussed next. However, the `Processor` class extends `Player`, which extends `Controller`, which extends `Clock`—all topics covered earlier in the chapter. Similarly, controlling tasks such as transcoding or multiplexing requires knowledge of the JMF classes employed to represent format (both content type and track). Hence to understand processing fully requires understanding not only this section but also the earlier half of the chapter in which these various topics and classes were covered, primarily as preparation to understand processing.

`Processor` Timescale

Extending `Controller` as it does indirectly by extending `Player`, which extends `Controller`, the `Processor` class has a similar model of time. Time is stopped or started. But in recognition of the fact that transitioning from stopped to started isn't instantaneous, requiring significant resource acquisition (as it often does), stopped time is subdivided into a number of states. These states reflect the preparedness (or lack thereof) of the `Controller` to start.

The `Controller` interface divides stopped time into five states which, in order of least prepared (to start) to prepared are known as Unrealized, Realizing, Realized, Prefetching, and Prefetched.

Because of the nature of processing and, in particular, the need to determine the format of the tracks that compose the media, as well as to specify the processing that will occur on those tracks, the `Processor` class subdivides stopped time into seven states. The two additional states, known as Configuring and Configured, are added between the Unrealized and Realized states. That leads to a sequence as follows: Unrealized, Configuring, Configured, Realizing, Realized, Prefetching, Prefetched.

A brief summary of each of those states is as follows:

`Unrealized`— A `Processor` that has been created but hasn't undertaken any resource gathering.

`Configuring`— A transition state reflecting the fact that the input (to the `Processor`) `DataSource` is being analyzed as to its format and that of its individual tracks.

`Configured`— A steady-state reflecting a `Processor` that has successfully gathered format information about the input `DataSource`. It is in this state that a `Processor` should be programmed with its processing task.

`Realizing`— A transition state reflecting the fact that the `Processor` is gathering information about the resources needed for its task, as well as gathering resources themselves.

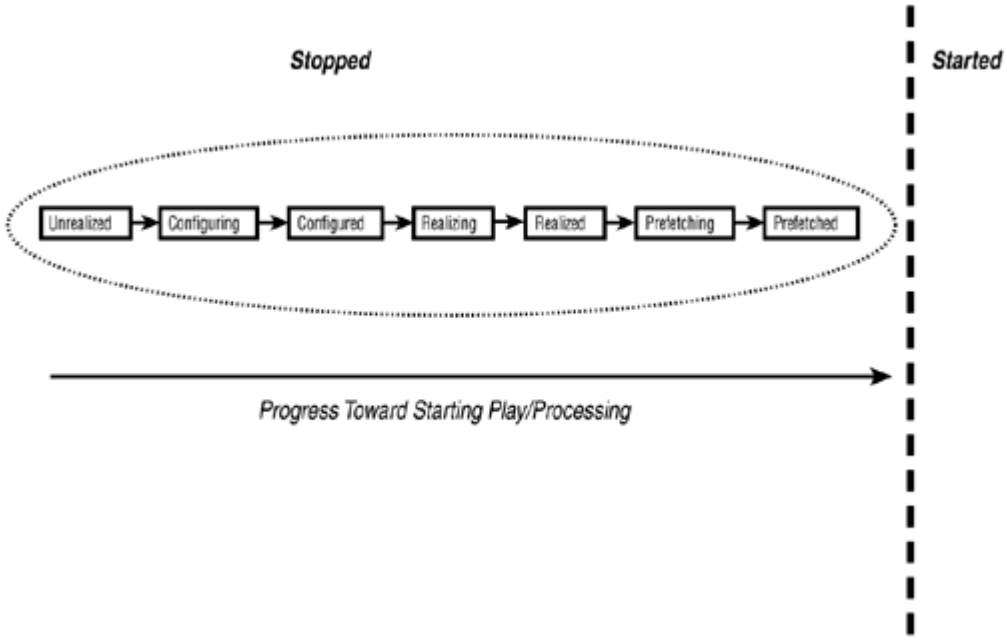
`Realized`— A steady-state reflecting a `Processor` that has gathered all the non-exclusive resources needed for a task.

`Prefetching`— A transition state reflecting the fact that the `Processor` is gathering all resources needed for its task that weren't obtained in the realizing state. Typically, this means acquiring exclusive usage resources such as hardware.

`Prefetched`— The `Processor` has acquired all necessary resources, performed all pre-startup processing, and is ready to be started.

Figure 8.29 shows the seven states and their relationship in terms of preparedness to start. The next section details the methods for transitioning between the states as well as the events that a `Processor` generates as it makes those transitions.

Figure 8.29. The seven states of a `Processor` as they measure how prepared a `Processor` is to start.



The addition of the Configuring and Configured states reflects querying the `DataSource` about its format and that of its constituent tracks. This not only provides finer granularity of stopped time, but also provides the particular instant in which the processor can be programmed. A `Processor` must be programmed (configured) for what processing it will undertake before it transitions to Realizing because that reflects commitment to a particular type of processing. A `Processor` cannot be programmed to the task it will undertake until all relevant information about the source media has been obtained. Hence, the Configured state is the only time at which a `Processor` can be programmed.

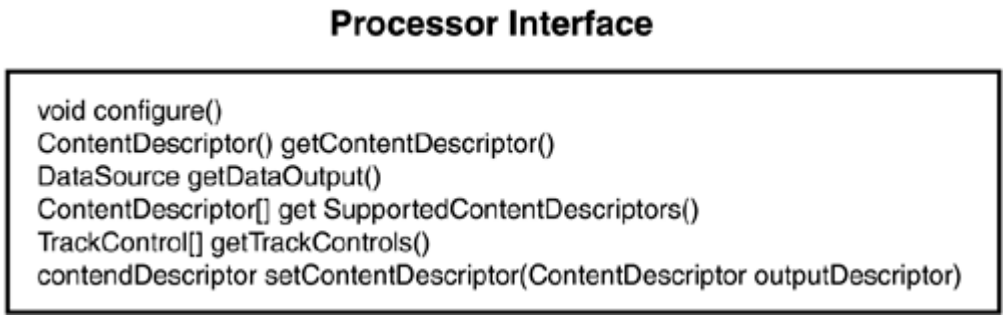
`Processor` Interface

A `Processor` is a `MediaHandler` object that extends the `Player` interface. In many ways, `Processor` follows a similar abstracted-from-media-details approach as that of `Player`. Given a particular set of requirements (in this case, processing media as opposed to simply playing it), a suitable `Processor` object can be obtained by way of the `Manager` class. From the user-code perspective, this object looks the same as all other `Processor` objects, possessing the same set of methods. This uniformity of interface, regardless of media particulars, leads to versatile programs capable of processing multiple types of media.

However, media particulars are also where the `Processor` differs significantly, in a creation sense, from `Player`. What is entailed in playing media is well understood and requires little specification beyond the media to be played. However, processing is far broader, indeed all but infinite, in variability (for instance, consider different digital effects). The potential with the type of processing that can be performed requires a far tighter prescription from the user at the time of `Processor` configuration so that the desired form of processing can be achieved. In a nutshell, creating and configuring a `Processor` is far more complicated than the equivalent task for a `Player`. The means of `Processor` creation and configuration, both important and involved, are detailed in a separate subsection.

Figure 8.30 shows the six methods of `Processor` that are above and beyond those inherited from `Player`. Three of the methods are vital in the programming of a `Processor`—`configure()`, `getTrackControls()`, and `setContentDescriptor()`. Two are general information or accessor methods—`getContentDescriptor()` and `getSupportedContentDescriptors()`. One, `getDataOutput()`, is the means of obtaining the output or result of the processing that was performed.

Figure 8.30. The `Processor` interface.



The `configure()` method asynchronously brings a `Processor` to the Configured state. As discussed in the next subsection, this is a vital step in programming a `Processor`. Only when a `Processor` is in the Configured state can it be programmed for the processing it will carry out.

The `setContentDescriptor()` method sets the `ContentDescriptor` of the `DataSource` that the `Processor` outputs. A content type such as AVI, QUICKTIME, or WAVE of the processor's output is set using this method. A content type is also known as meta-format or media container. The method returns the actual `ContentDescriptor` that was set, which might be the closest matching `ContentDescriptor` supported by the `Processor`. The method might also return `null` if no

ContentDescriptor could be set, as well as throwing a NotConfiguredError exception if the Processor is either Unrealized or Configuring.

The getTrackControls() method returns an array of TrackControl objects—one for each track in the DataSource that is being processed. As described in the next subsection, the TrackControl objects can then be employed to program the processing that is performed on the associated track of media. The method throws a NotConfiguredError exception if the Processor isn't at least Configured at the time of calling.

Processors produce a DataSource as the result of the processing they perform. That DataSource can be used as the input to a subsequent Processor or Player. The getDataOutput() method is the means of obtaining the DataSource that a Processor produces. The method throws a NotRealizedError exception if the Processor isn't at least Realized at the time of invocation.

The getContentDescriptor() method returns the currently set ContentDescriptor that will be used for the output of the processor. The getSupportedContentDescriptors() returns an array of all ContentDescriptors that the Processor can output. Both methods throw a NotConfiguredError exception if the Processor isn't at least Configured at the time of calling.

A number of the methods of Processor, most inherited from Controller or Player, exert direct control over the state of a Processor. Others can alter the state as a side effect. The following list shows those methods:

configure()— Moves the Processor from Unrealized to Realized. This is a key step in Processor management because the Configured stage is the only time that a Processor can be programmed as to its task.

realize()— Moves the Processor from either Unrealized or Configured to Realized. Asynchronous (returns immediately). Generally, realize() should only be called after the Processor has reached Configured and has been programmed as to its task.

prefetch()— Moves the Processor from Realized to Prefetched. Asynchronous (returns immediately).

start()— Moves the Processor from its current state to Started. The Processor will transition through any intermediary states and send notice of such transitions to any listeners. As for the realize() method, start() should only be called after a processor has been programmed. Asynchronous (returns immediately).

stop()— Moves the Processor from the Started state back to either Prefetched or Realized, depending on the type of media being processed and the processing task. Synchronous (returns only after Processor is actually stopped).

deallocate()— Deallocates the resources acquired by the Processor and moves it to the Realized state (if at least Realized at the time of call) or to Unrealized (if in Realizing or Unrealized state at time of call). Synchronous.

`close()`— Closes down the `Processor` totally. It will post a `ControllerClosedEvent`. The behavior of any further calls on the `Processor` is unspecified: the `Processor` shouldn't be used after it is closed. Synchronous.

`setStopTime()`— Sets the media time at which it will stop. When the media reaches that time, the `Processor` will act as though `stop()` was called. Asynchronous (returns immediately and stops play when time reaches that specified).

`setMediaTime()`— Changes the media's current time. This might result in a state transition. For instance, a started `Processor` might return to Prefetching while it refills buffers for the new location in the stream. Asynchronous (in that any state changes triggered aren't waited for).

`setRate()`— Sets the rate of processing. It might result in a state change on a started `Processor` for similar reasons to those for `setMediaTime()`. As for the `Player` interface, there is no guarantee that a `Processor` will support any rate other than 1.0. If a requested rate isn't supported, the closest matching rate is set. Asynchronous (in that any state changes triggered aren't waited for).

Figure 8.31 shows the states of a `Processor` with the methods that force transitions between those states. Figure 8.32 shows the events that a `Processor` generates as it transitions between states.

Figure 8.31. Methods of `Processor` that trigger state changes.

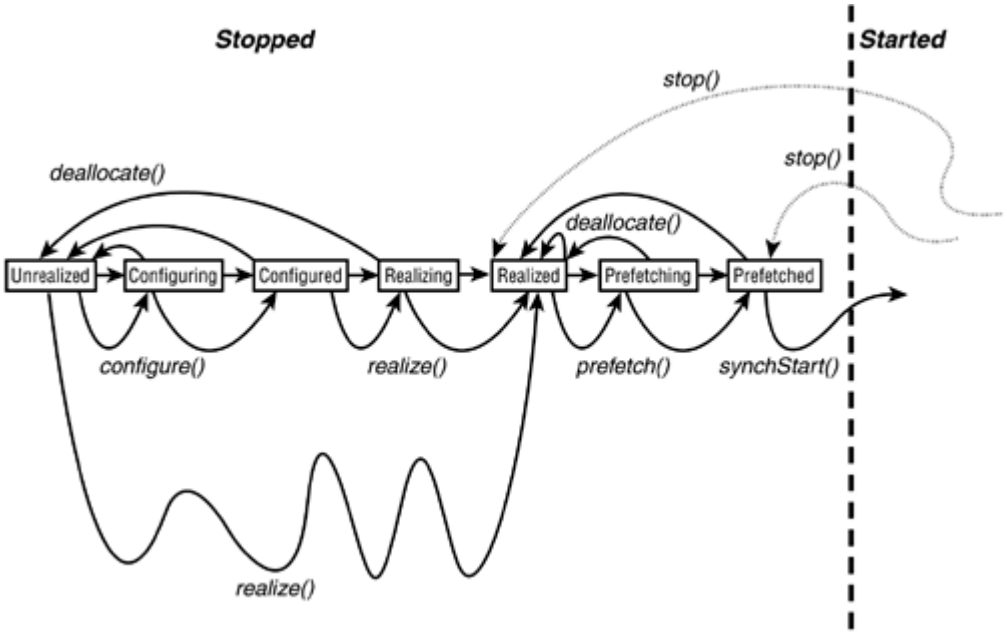
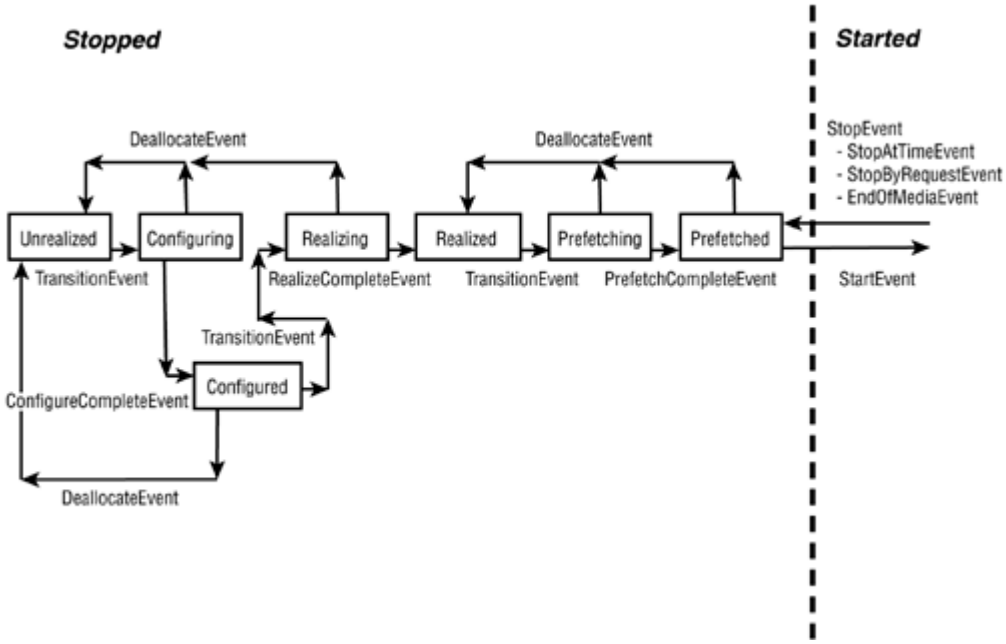


Figure 8.32. `Events` generated when a `Processor` transitions between states.



Creating and Programming `Processors`

The key initialization step for a `Processor` isn't simply its creation but its programming. The specific task that it must perform needs to be defined prior to `Processor` realization because realization, and the prefetching that follows, involves obtaining the resources necessary to perform the task. However, programming a `Processor` requires the prior knowledge of the format of the media that will be processed. This apparent dilemma is resolved through the extended model of stopped time provided by `Processor`: The Configured state provides the opportunity at which the format of the input media is known but prior to the realizing (resource gathering) step.

The JMF provides two primary approaches by which the task of a `Processor` can be programmed. One, perhaps simpler, approach is to encapsulate all necessary processing as a `ProcessorModel` object and have the `Manager` class create a `Processor` object that will perform exactly that task. The second, arguably more complex but certainly more versatile, approach is to have the `Manager` class create a `Processor` but leave programming entirely under user control. The user code must bring the `Processor` to the Configured state and then use methods and objects such as `TrackControls` to define the task of the `Processor`. With that completed, the `Processor` can be moved through to Started.

The `Manager` class provides four methods for creating a `Processor`. Three `createProcessor()` methods accept the specification of the input media (as a `DataSource`, `URL`, or `MediaLocator`) and return an unrealized `Processor`. It is the user's responsibility to bring that `Processor` to the Configured state (with the `configure()` method), carry out the programming, and then start the `Processor`. The three `createProcessor()` methods can throw `IOException` and `NoProcessorException` exceptions.

The alternative means of creating a `Processor` is with `Manager`'s `createRealizedProcessor()` method. The method accepts a `ProcessorModel` object that describes all processing requirements—the input and output formats as well as the individual tracks. The method is synchronous (blocking)—it won't return until the `Processor` is realized. This can make programming considerably easier. As for the other `Processor` creation methods of `Manager`, `createRealizedProcessor()` can throw both `IOException` and `NoProcessorException`. It can also throw a `CannotRealizeException`.

`ProcessorModel`

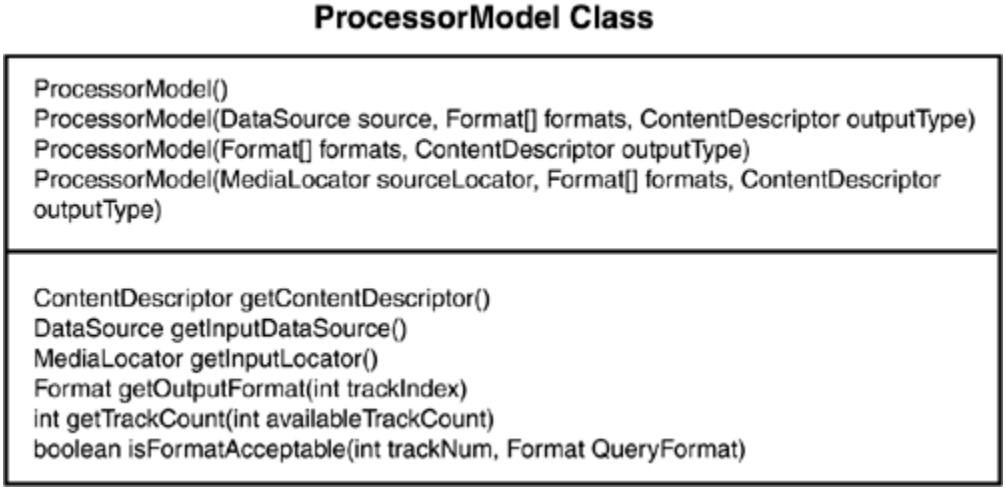
The simplest way to program a `Processor` is with a `ProcessorModel` object at the time of the `Processor's` creation using `Manager`'s `createRealizedProcessor()` method. The `ProcessorModel` object encapsulates all the processing that the `Processor` is to perform. In particular, it specifies the input media, the required output content type, and the format of each track of the output.

The task of using a `ProcessorModel` to obtain an appropriate Processor is relatively simple:

1. Construct the objects needed by the `ProcessorModel` constructor (for example, `DataSource` and `Format` objects).
2. Construct the `ProcessorModel`.
3. Use `Manager`'s `createRealizedProcessor()` method to obtain an appropriate `Processor`.

Figure 8.33 shows the constructors and methods of `ProcessorModel`. As you can see, the methods of `ProcessorModel` are simply information queries about the object's attributes. The chief role of the class is to encapsulate all necessary information about the processing task to be performed. Hence the class constructors are the most important elements of the class.

Figure 8.33. The `ProcessorModel` class.



There are four constructors, although the no-argument constructor is simply a placeholder. The two most commonly used constructors accept either a `MediaLocator` or `DataSource` as the specification of the input to the `Processor`. That is then followed by an array of `Format` objects—each element of that

array specifying the desired format for the particular track. The final argument is a `ContentDescriptor` specifying the output content type (media container or meta-format). The fourth constructor accepts no specification of the input media: only output track formats and content type. This constructor is used for capturing media (finding a capture device that will meet the output demands of such a processor).

Listing 8.9 shows the use of a `ProcessorModel` in the apparent transcoding of a movie. A `MediaLocator` is created specifying the source media, an output content type of MSVIDEO (AVI) is created, and formats are specified for the two tracks: Cinepak for the video and Linear for the audio. Those objects are used to construct a `ProcessorModel`, which in turn is employed in the creation of a `Processor`.

Listing 8.9 Use of a `ProcessorModel` Object in the Apparent Transcoding of a Movie

```
// Construct objects needed to make ProcessorModel
MediaLocator src = new MediaLocator(
"file://d:\\jmf\\book\\media\\videoexample\\iv50_320x240.mov");
formats = new Format[2];
formats[0] = new VideoFormat(VideoFormat.CINEPAK);
formats[1] = new AudioFormat(AudioFormat.LINEAR);
 container = new FileTypeDescriptor(FileTypeDescriptor.MSVIDEO);
// Construct the ProcessorModel
ProcessorModel model = new ProcessorModel(src,formats,container);
// Create the realized Processor using the ProcessorModel
Processor p = Manager.createRealizedProcessor(model);
```

The one drawback of the `ProcessorModel` approach to `Processor` programming is that fine control is removed from the user's hands. In particular, the user has no say over how the processing is performed in terms of what classes are employed. The user cannot specify which `PlugIns` to employ, and this is a particular drawback in two areas. First, it means that no `Effect` processing can be performed through a `ProcessorModel` because a `ProcessorModel` is only aware of output formats, not manipulations within a format. Second, it might be desirable to perform processing as a chain of `PlugIns` more complex than `Demultiplexer`, `Codec`, and `Multiplexer`. This is where the second approach to programming a `Processor`, through its `TrackControl` objects, comes into its own.

`TrackControl`

The second means of programming the processing task for a `Processor` is directly through the `TrackControl` interface that corresponds to each track composing the media to be processed. The approach is somewhat more involved than using a `ProcessorModel` but is far more flexible, allowing the user to specify codec chains and renderers to be applied to individual tracks—something not possible with the `ProcessorModel` approach.

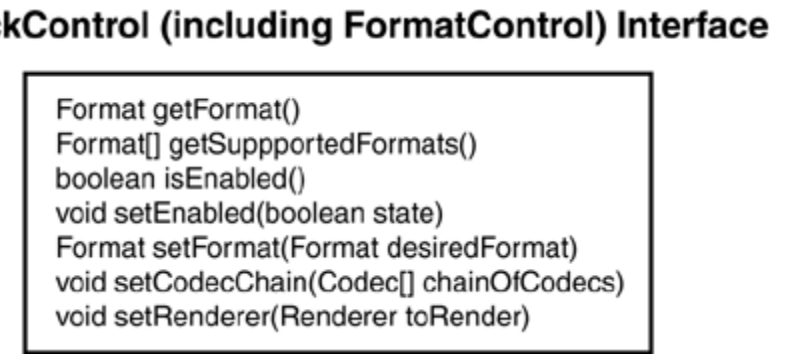
The basic algorithm for pursuing the `TrackControl` based approach to programming a `Processor` is as follows:

```
Create a Processor through the Manager class
Bring the Processor to the Configured state using configure()
Set the required ContentDescriptor on the Processor which specifies the content
```

```
  type of the media that will be produced
Obtain the TrackControls of the Processor

For each TrackControl
       Set the required output Format
       If codecs/effects are to be used for the track
              Set the TrackControl's codec chain
       If the track is to be rendered
              Set the TrackControl's renderer
Start the processor
```

`TrackControl` is an interface that extends both the `FormatControl` and `Controls` interfaces. Figure 8.34 shows the methods of `TrackControl` plus those inherited from `FormatControl` (an interface not discussed to date). In terms of programming a `Processor`, the most important methods are `setEnabled()`, `setFormat()`, `setCodecChain()`, and `setRenderer()`.

Figure 8.34. The `TrackControl` interface.



The `setEnabled()` method is used to determine whether a track will be processed. Passing the `setEnabled()` method a value of `false` means that the track won't be processed and won't be output by the Processor.

The `setFormat()` method is passed a `Format` object specifying the desired output `Format` that the `Processor` will produce for that track. The method returns the `Format` that was actually set (the closest matching `Format` if the requested `Format` wasn't supported), or `null` if no `Format` could be set.

The `setCodecChain()` method is passed an array of `Codec PlugIns` that are to be applied to the track as a chain—that is, in the order they are found in the array. This is a powerful mechanism to exactly controlling the order of compression/decompression and effect processing upon each track. It is important to note that the `Effect PlugIn` is a subclass of `Codec`, so these can also be passed as elements of the array. The method might throw an `UnSupportedPlugInException` or a `NotConfiguredError` exception.

The `setRenderer()` method is used to specify the `Renderer PlugIn` that is to play a particular track. By default, `Processors` don't render the tracks that they are processing but rather only produce an output `DataSource`.

It is worth mentioning that the `TrackControl` interface extends `Controls`; thus it is possible to obtain the `Control` objects associated with the track. Among these will be `Control` objects for any codecs being employed. Thus, it is possible to control the particulars of the compression/decompression on a per-track basis by employing such `Control` objects.

Listing 8.10 shows the programming of a `Processor` to perform the same task as the one in Listing 8.9. However, in this case, the `Processor` is programmed through the `TrackControl` approach `ProcessorModel`. As you can see, the setup cost for simple transcoding—what both pieces of code are doing—is higher with the `TrackControl` approach. What isn't being shown in the fragment is the versatility offered in terms of specifying a codec chain or renderer for each track.

Listing 8.10 Programming of a `Processor` Through the `TrackControls` Approach

```
MediaLocator src = new MediaLocator(
"file://d:\\jmf\\book\\media\\videoexample\\iv50_320x240.mov");
Processor p = Manager.createProcessor(src);
p.addControllerListener(this);
p.configure();
:
:
public void synchronized controllerUpdate(ControllerEvent e) {

if (e instanceof ConfigureCompleteEvent) {
  p.setContentDescriptor(new
        FileTypeDescriptor(FileTypeDescriptor.MSVIDEO));
  TrackControl[] controls = processor.getTrackControls();
  for (int i=0;i<controls.length;i++) {
    if (controls[i].getFormat() instanceof VideoFormat)
      controls[i].setFormat(new VideoFormat(VideoFormat.CINEPAK));
    else
      controls[i].setFormat(new AudioFormat(AudioFormat.LINEAR));
  }
  p.start();
}
else if (e instanceof ...) { ...}
```

Utility Class: `MediaStatistics`

This subsection discusses a utility class, `MediaStatistics`, who's source can be found in Listing 8.11 as well as on the book's companion Web site. `MediaStatistics` employs a `Processor` in order to acquire information about the format of the individual tracks that compose a media object. A derivative graphical application `GUIMediaStatistics`, which provides the same functionality but with a graphical user interface, can also be found at the book's companion Web site.

The class itself illustrates a number of the features and classes of JMF that have been discussed in the chapter to date: `Processor` and `Format` classes, `Controller` events, and the asynchronous time model of the JMF.

A `MediaStatistics` object is constructed by providing the name and location (a `String` suitable for constructing a `MediaLocator`) of the media in question. A `Processor` is created for that media and

brought to the Configured state. At that stage, the `Format` of the individual tracks can be obtained and reported on.

The class also contains a `main()` method so that it can run as a standalone application. Any command-line options are treated as media names: A `MediaStatistics` object is constructed and used to generate a report on the format of the tracks of the media. Listing 8.12 shows a run of the application for an MPEG video file. Users should note (as in the example) that the media name must consist of both the protocol (file:// if a local file) and the location of the media (the fully qualified pathname if a local file).

Several features of the class are worth noting. Because discovering the format of tracks requires a Configured `Processor`, the format information might not be available immediately. To deal with this, the class keeps track of its own state as well as providing methods such as `getState()` and `isKnown()` (which returns `true` when the formats are finally known) so that the user can check when a report is available. Alternatively, and more powerfully, the class could generate its own events and send those to listener classes. Although not difficult to implement, it would obscure the main purpose of the example and hence is omitted.

Two constructors are provided; one returns instantly but makes no guarantees that the format information is available. The second constructor blocks until the format information is known or a prescribed timeout period has transpired. This has the advantage that the user code employing `MediaStatistics` won't need to wait an unknown period before querying about formats.

The `MediaStatistics` object keeps track of its own state by catching any exceptions that occur in the `Processor` creation as well as listening to the `Processor`. Listening to the `Processor` in this instance is far simpler than for `Players` or more sophisticated `Processors`. Only the `ConfigureCompleteEvent` is of interest; in which case, the `TrackControls` should be obtained as a means of obtaining the `Format` objects for each track. The resources tied up in the `Processor` can then also be released.

The class contains a number of information (accessor) methods including

`getNumTracks()`— Gets the number of tracks possessed by the media

`getState()`— Determines the state of the object

`getReport()`— Gets a `String` detailing the format of all tracks

`getTrackFormat()`— Returns the `Format` object for the specified track number

`isAudioTrack()` and `isVideoTrack()`— Return `true` if the specified track number is an audio or video track

`isKnown()`— Returns `true` if the format information is known (if the object's state is `KNOWN`).

Listing 8.11 The `MediaStatistics` Application that Produces Statistics About the Particular Format of Media

```
import javax.media.*;
import javax.media.control.*;
import javax.media.format.*;

/**************************************************************
* A Class to determine statistics about the tracks that compose
* a media object. Given the name (URL/location) of media a
* Processor is constructed and brought to the Configured state.
* At that stage its TrackControls are obtained as a means of
* discovering the Formats of the individual tracks.
*
* Because reaching Configured can take time, the MediaStatistics
* object keeps track of its own state and provides methods for
* determining that state. Only when it reaches the KNOWN state
* can statistics be obtained. Similarly there are 2 constructors:
* one creating a Processor and starting it toward Configured but
* returning immediately. The other is a blocking constructor, it
* won't return until the Processor reaches Configured or the
* specified time-out expires. This has the advantage that the
* object can be used immediately (rather than polling it to
* determine when it enters the KNOWN state).
*
* The chief information gathering method is getReport() which
* returns a String reporting on the Format of all tracks of
* the media. Alternatively the Format of individual tracks can
* also be ascertained.
*
* @author Spike Barlow
**************************************************************/
public class MediaStatistics implements ControllerListener {

  /** State: Yet to create the Processor. */
  public static final int NOT_CREATED = 0;

  /** State: Unable to create the Processor. */
  public static final int FAILED = -1;

  /** State: Processor is Configuring. */
  public static final int CONFIGURING = 1;

  /** State: Details of media are Known. */
  public static final int KNOWN = 2;

  /** Number of tracks is Unknown. */
  public static final int UNKNOWN = Integer.MIN_VALUE;

  /** Period in milliseconds to sleep for before
  * rechecking if reached KNOWN state. */
  protected  static final int WAIT_INTERVAL = 20;

  /** Number of tracks possessed by the media. */
  protected int             numTracks = UNKNOWN;

  /** Formats of the individual tracks. */
  protected Format[]        trackFormats;
```

```
  /** Processor needed to ascertain track information. */
  protected Processor       processor;

  /** State that the object is currently in. A reflection
  * of the state the Processor is in. */
  protected int             state = NOT_CREATED;

  /** The name of the media on which stats are being compiled. */
  protected String          nameOfMedia;

/*************************************************************
* Construct a MediaStatistics object for the media with the
* passed name. This is a blocking constructor. It returns
* only when it is possible to obtain the track statistics or
* when the specified time-out period (in milliseconds) has
* transpired.
*************************************************************/
MediaStatistics(String mediaName, int timeOutInMilliseconds) {

  nameOfMedia = mediaName;
  // Construct the Processor
  try {
    MediaLocator locator = new MediaLocator(mediaName);
    processor = Manager.createProcessor(locator);
  }
  // Any exception is a failure.
  catch (Exception e) {
    state = FAILED;
    return;
  }
  // Listen to and start configuration of the Processor.
  processor.addControllerListener(this);
  state = CONFIGURING;
  processor.configure();
  ////////////////////////////////////////////////////////
  // Wait till the Processor reaches configured (the object
  // reaches KNOWN) or the specified time-out interval has
  // transpired, by looping, sleeping,and rechecking.
  ////////////////////////////////////////////////////////
  if (timeOutInMilliseconds>0) {
    int waitTime = 0;
    while (waitTime<timeOutInMilliseconds && !isKnown()) {
      try { Thread.sleep(WAIT_INTERVAL); }
      catch (InterruptedException ie) { }
      waitTime += WAIT_INTERVAL;
    }
  }
}

/*************************************************************
* Construct a MediaStatistics object for the media with the
* passed name. This is not a blocking constructor: it returns
* immediately. Thus calling getReport() immediately may result
* in "Still parsing media" report. The isKnown() method should
* be used to check for this condition.
*************************************************************/
MediaStatistics(String mediaName) {
  this(mediaName,-1);
```

```
}

/*************************************************************
* Respond to events from the Porcessor. In particular the
* ConfigureComplete event is the only one of interest. In this
* case obtain the TrackControls anduse these to obtain the
* Formats of each track. Also modify the state and close down
* the Processor (free up its resources).
*************************************************************/
public synchronized void controllerUpdate(ControllerEvent e) {

  if (e instanceof ConfigureCompleteEvent) {
    TrackControl[] controls = processor.getTrackControls();
    // As long as there are TrackControls, get each track's format.
    if (controls.length!=0) {
      numTracks = controls.length;
      trackFormats = new Format[controls.length];
      for (int i=0;i<controls.length;i++) {
        trackFormats[i] = controls[i].getFormat();
     }
      state = KNOWN;
    }
    else {
      state = FAILED;
    }
    // Close down the Processor.
    processor.removeControllerListener(this);
    processor.close();
    processor = null;
  }
}


/*************************************************************
* Determine what state the object is in. Returns one of the
* class constants such as KNOWN, FAILED or CONFIGURING.
*************************************************************/
public int getState() {
  return state;
}


/*************************************************************
* Determine the number of tracks possessed by the media. If
* that is unknown, either due to the processor creation
* failing or because the processor is not yet Configured then
* the class constant UNKNOWN is returned.
*************************************************************/
public int getNumTracks() {
  return numTracks;
}

/*************************************************************
* Obtain the Format for the specified track number. If the
* track doesn't exist, or it has yet to be determined how
* many tracks the media possesses, null is returned.
*************************************************************/
public Format getTrackFormat(int track) {
```

```
  if (track<0 || track>=numTracks)
    return null;
  return trackFormats[track];
}

/**************************************************************
* Is the object in the KNOWN state? The KNOWN state reflects
* the fact that information is known about the number and
* Format of the tracks. The method can be used to ascertain
* whether a report is available (meaningful) or not.
**************************************************************/
public boolean isKnown() {
  return state==KNOWN;
}


/**************************************************************
* Returns true if the specified track number is an audio track.
* If the track doesn't exist, the number of tracks is yet
* unknown, or it isn't audio then false is returned.
**************************************************************/
public boolean isAudioTrack(int track) {

  if (track<0 || track>=numTracks)
    return false;
  return trackFormats[track] instanceof AudioFormat;
}

/**************************************************************
* Returns true if the specified track number is a video track.
* If the track doesn't exist, the number of tracks is yet
* unknown, or it isn't video then false is returned.
**************************************************************/
public boolean isVideoTrack(int track) {

  if (track<0 || track>=numTracks)
    return false;
  return trackFormats[track] instanceof VideoFormat;
}


/**************************************************************
* Returns a report, as a String, detailing thenumber and format
* of the individual tracks that compose the media that this
* object obtained statistics for. If the object is not in the
* KNOWN state then the report is a simple String, indicating
* this.
**************************************************************/
public String getReport() {
  String  mess;

  if (state==FAILED)
    return "Unable to Handle Media " + nameOfMedia;
  else if (state==CONFIGURING)
    return "Still Parsing Media " + nameOfMedia;
  else if (state==KNOWN) {
    if (numTracks==1)
      mess = nameOfMedia + ": 1 Track\n";
    else
```

```
      mess = nameOfMedia + ": " + numTracks + " Tracks\n";
    for (int i=0;i<numTracks;i++) {
      if (trackFormats[i] instanceof AudioFormat)
        mess += "\t"+(i+1)+" [Audio]: ";
      else if (trackFormats[i] instanceof VideoFormat)
        mess += "\t"+(i+1)+" [Video]: ";
      else
        mess += "\t"+(i+1)+" [Unknown]: ";
      mess += trackFormats[i].toString() + "\n";
    }
    return mess;
  }
  else
    return "Unknown State in Processing " + nameOfMedia;
}


/*************************************************************
* Simple main method to exercise the class. Takes command
* line arguments and constructs MediaStatistics objects for
* them, before generating a report on them.
*************************************************************/
public static void main(String[] args) {

  MediaStatistics[] stats = new MediaStatistics[args.length];

  for (int i=0;i<args.length;i++) {
    stats[i] = new MediaStatistics(args[i],200);
    System.out.println(stats[i].getReport());
    stats[i] = null;
  }
  System.exit(0);
}
}
```

Listing 8.12 Output of a Run of the `MediaStatistics` Program on a Particular MPEG Movie File

```
C:\Spike\JMF\Code>java MediaStatistics file://d:\media\badday.mpeg
file://d:\media\badday.mpeg: 2 Tracks
        1 [Video]: MPEG, 176x120, FrameRate=29.9, Length=31680
        2 [Audio]: mpegaudio, 44100.0 Hz, 16-bit, Mono, LittleEndian,
                 Signed, 4000.0 frame rate,  FrameSize=16384 bits
```

Utility Class: `Location2Location`

This subsection discusses a utility class, `Location2Location`, found in Listing 8.13 as well as on the book's companion Web site. `Location2Location` is a class that both transcodes media to a different format and saves/writes the resulting media to a new location.

In order to perform its task, `Location2Location` combines many of the most important classes and features of the JMF, and is the most comprehensive example of the JMF features to date in the book. Key classes involved include `Processor`, `Manager`, `DataSource`, `DataSink`, `ProcessorModel`, and

MediaLocator; whereas the event-driven asynchronous nature of the JMF is illustrated through the class implementing ControllerListener while also using an anonymous class to listen for events from the DataSink object.

The class is employed through the mechanism of the user constructing a Location2Location object that specifies the origin and destination of the media (either Strings or MediaLocators), the desired format for the individual tracks that compose the media (an array of Formats), and the content type (ContentDescriptor) for the resulting media container. Construction automatically initiates the transcoding portion of the task, whereas transfer to the new destination is achieved by calling the transfer() method.

The transcoding tasks— transforming the format of the media—is achieved via the processor class, whereas the writing to a new location is achieved through a DataSink. The Processor object is created through the specification of a ProcessorModel that includes the Format specification for the individual tracks as well as the content type (ContentDescriptor) of the resulting container (for example, Quicktime or AVI).

The DataSink is constructed using the DataSource that is the output of the Processor together with the user specified destination.

Both the Processor and DataSink perform their respective tasks (transcoding and rendering to a destination) asynchronously. The class illustrates means of dealing with this pervasive feature of the JMF. The class listens for events from both the Processor and DataSink and maintains its own internal model of its current state. This is exposed to the user through a number of class constants and a method getState(), which is used to query the current state. Further, the transfer() method can be invoked asynchronously, which will return immediately, but the user will need to check periodically for when the process is complete. It can also be invoked synchronously, which will block until the process completes. An even better solution would be for Location2Location to generate its own events and allow classes to register themselves as listeners for those events. Such an approach isn't difficult to implement, but requires the writing of an additional interface (the Listener interface) and class (the Event class) as well as the code to maintain the list of listeners. We feel that this further detail would obscure the purpose of the example, which is to illustrate features of the JMF.

Listing 8.13 The Location2Location Utility Class Capable of Transferring and Transcoding Media

```
import javax.media.*;
import javax.media.datasink.*;
import javax.media.protocol.*;

/**************************************************************
* Transfer media from one location to another carrying out the
* specified transcoding (track formats and content type) at the
* same time.
*<p>Users specify a source and destination location, the
* Formats (to be realised) of the individual tracks, and a
* ContentDescriptor (content type) for output.
*<p>A Processor is created to perform and transcoding and its
* output DataSource is employed to construct a DataSink in
* order to complete the transfer.
```

```
*<p>The most important method of the class is transfer() as
* this opens and starts the DataSink. The constructor builds
* both the Processor (which is starts) and the DataSink.
*<p>The object keeps track of its own state, which can be queried
* with the getState() method. Defined constants are FAILED,
* TRANSLATING, TRANSFERRING, and FINISHED. The process is
* asychronous: transcoding largish movies can take a long time.
* The calling code should make allowances for that.
************************************************************/
public class Location2Location implements ControllerListener {

  /** Output of the Processor: the transcoded media. */
  protected DataSource  source;

  /** Sink used to "write" out the transcoded media. */
  protected DataSink  sink;

  /** Processor used to transcode the media. */
  protected Processor processor;

  /** Model used in constructing the processor, and which
  * specifies track formats and output content type */
  protected ProcessorModel  model;

  /** State the object is in. */
  protected int   state;

  /** Location that the media will be "written" to. */
  protected MediaLocator  sinkLocation;

  /** The rate of translation. */
  protected float translationRate;

  /** Process has failed. */
  public static final int FAILED = 0;

  /** Processor is working but not finished. DataSink is yet
  * to start. */
  public static final int TRANSLATING = 1;

  /** DataSink has started but not finished. */
  public static final int TRANSFERRING = 3;

  /** Transcoding and transfer is complete. */
  public static final int FINISHED = 4;

  /** String names for each of the states. More user friendly */
  private static final String[] STATE_NAMES = {
    "Failed", "Translating", "<UNUSED>", "Transferring",
    "Finished"};

  /** Period (in milliseconds) between checks for the blocking
  * transfer method. */
  public static final int WAIT_PERIOD = 50;

  /** Wait an "indefinite" period of time for the transfer
  * method to complete. i.e., pass to transfer() if the
  * user wishes to block till the process is complete,
  * regardless of how long it will take. */
```

```
  public static final int INDEFINITE = Integer.MAX_VALUE;

/**************************************************************
* Construct a transfer/transcode object that transfers media from
* sourceLocation to destinationLocation, transcoding the tracks as
* specified by the outputFormats. The output media is to have a
* content type of outputContainer and the process should (if
* possible) run at the passed rate.
**************************************************************/
Location2Location(MediaLocator sourceLocation,
    MediaLocator destinationLocation, Format[] outputFormats,
    ContentDescriptor outputContainer, double rate) {

  ////////////////////////////////////////////
  // Construct the processor for the transcoding
  ////////////////////////////////////////////
  state = TRANSLATING;
  sinkLocation = destinationLocation;
  try {
    if (sourceLocation==null)
      model = new ProcessorModel(outputFormats,outputContainer);
    else
      model = new ProcessorModel(sourceLocation,
                  outputFormats,outputContainer);
    processor = Manager.createRealizedProcessor(model);
  }
  catch (Exception e) {
    state = FAILED;
    return;
  }

  translationRate = processor.setRate((float)Math.abs(rate));
  processor.addControllerListener(this);

  ////////////////////////////////////////////////////////
  // Construct the DataSink and employ an anonymous class as
  // a DataSink listener in order that the end of transfer
  // (completion of task) can be detected.
  ////////////////////////////////////////////////////////
  source = processor.getDataOutput();
  try {
    sink = Manager.createDataSink(source,sinkLocation);
  }
  catch (Exception sinkException) {
    state = FAILED;
    processor.removeControllerListener(this);
    processor.close();
    processor = null;
    return;
  }
  sink.addDataSinkListener(new DataSinkListener() {
      public void dataSinkUpdate(DataSinkEvent e) {
        if (e instanceof EndOfStreamEvent) {
          sink.close();
          source.disconnect();
          if (state!=FAILED)
            state = FINISHED;
        }
        else if (e instanceof DataSinkErrorEvent) {
```

```
          if (sink!=null)
            sink.close();
          if (source!=null)
            source.disconnect();
          state = FAILED;
        }
      }
  });
  // Start the transcoding
  processor.start();
}


/*************************************************************
* Alternate constructor: source and destination specified as
* Strings, and no rate provided (hence rate of 1.0)
*************************************************************/
Location2Location(String sourceName, String destinationName,
    Format[] outputFormats, ContentDescriptor outputContainer) {

  this(new MediaLocator(sourceName), new MediaLocator(destinationName),
    outputFormats, outputContainer);
}


/*************************************************************
* Alternate constructor: No rate specified therefore rate of 1.0
*************************************************************/
Location2Location(MediaLocator sourceLocation,
    MediaLocator destinationLocation, Format[] outputFormats,
    ContentDescriptor outputContainer) {

  this(sourceLocation,destinationLocation,outputFormats,outputContainer,1.0f);
}


/*************************************************************
* Alternate constructor: source and destination specified as
* Strings.
*************************************************************/
Location2Location(String sourceName, String destinationName,
    Format[] outputFormats, ContentDescriptor outputContainer,
    double rate) {

  this(new MediaLocator(sourceName), new MediaLocator(destinationName),
    outputFormats, outputContainer, rate);
}


/*************************************************************
* Respond to events from the Processor performing the transcoding.
* If its task is completed (end of media) close it down. If there
* is an error close it down and mark the process as FAILED.
*************************************************************/
public synchronized void controllerUpdate(ControllerEvent e) {

  if (state==FAILED)
    return;
```

```
  // Transcoding complete.
  if (e instanceof StopEvent) {
    processor.removeControllerListener(this);
    processor.close();
    if (state==TRANSLATING)
      state = TRANSFERRING;
  }
  // Transcoding failed.
  else if (e instanceof ControllerErrorEvent) {
    processor.removeControllerListener(this);
    processor.close();
    state = FAILED;
  }
}



/***********************************************************
* Initiate the transfer through a DataSink to the destination
* and wait (block) until the process is complete (or failed)
* or the supplied number of milliseconds timeout has passed.
* The method returns the total amount of time it blocked.
***********************************************************/
public int transfer(int timeOut) {

  // Can't initiate: Processor already failed to transcode
  ////////////////////////////////////////////////////////
  if (state==FAILED)
    return -1;

  // Start the DataSink
  /////////////////////
  try {
    sink.open();
    sink.start();
   }
  catch (Exception e) {
    state = FAILED;
    return -1;
  }
  if (state==TRANSLATING)
    state = TRANSFERRING;
  if (timeOut<=0)
    return timeOut;

  // Wait till the process is complete, failed, or the
  // prescribed time has passed.
  //////////////////////////////////////////////////
  int waited = 0;
  while (state!=FAILED && state!=FINISHED && waited<timeOut) {
    try { Thread.sleep(WAIT_PERIOD); }
    catch (InterruptedException ie) { }
    waited += WAIT_PERIOD;
  }
  return waited;
}
```

```
/*************************************************
* Initiate the transfer through a DataSink to the
* destination but return immediately to the caller.
*************************************************/
public void transfer() {

  transfer(-1);
}


/*************************************************
* Determine the object's current state. Returns one
* of the class constants.
*************************************************/
public int getState() {

  return state;
}

/*************************************************
* Returns the object's state as a String. A more
* user friendly version of getState().
*************************************************/
public String getStateName() {

  return STATE_NAMES[state];
}

/*************************************************
* Obtain the rate being used for the process. This
* is often 1, despite what the user may have supplied
* as Clocks (hence Processors) don't have to support
* any other rate than 1 (and will default to that).
*************************************************/
public float getRate() {

  return translationRate;
}

/*************************************************
* Set the time at which media processing will stop.
* Specification is in media time. This means only
* the first "when" amount of the media will be
* transferred.
*************************************************/
public void setStopTime(Time when) {

  if (processor!=null)
    processor.setStopTime(when);
}

/*************************************************
* Stop the processing and hence transfer. This
* gives user control over the duration of a
* transfer. It could be started with the transfer()
* call and after a specified period stop() could
* be called.
*************************************************/
```

```
public void stop() {

  if (processor!=null)
    processor.stop();
}
}
```

As a means of illustrating how the `Location2Location` class might be employed, the simple example `StaticTranscode` found in Listing 8.14 is provided. The class provides no user interaction, performing the same task each time. That task is to transcode two media files into two others. The first media is a short piece of audio (someone playing an electric guitar) that is transcoded from linear, wave format into MP3. The second media is a movie consisting of both audio (GSM format) and video (Indeo 5.0 format) in a Quicktime meta-format, which is transcoded into AVI meta-format with linear audio and Cinepak video. Users wanting to perform transcoding could use `StaticTranscode` as a starting template, modifying filenames and formats appropriately. Alternatively, for one-off tasks, JMStudio could be used interactively.

As you can see by viewing the source, employing the `Location2Location` class requires the prior construction of `Format` and `ContentDescriptor` objects that detail the format for the transcoding that will occur.

`StaticTranscode` is a trivial example, simply illustrating how `Location2Location` might be used. However it is relatively easy to write applications that build on top of the functionality provided by `Location2Location`. For example, an audio ripper (or its inverse)—a program that converts CD audio to MP3—could easily be written so that given a directory, it processes all files found there and converts them into MP3.

Listing 8.14 The `StaticTranscode` Class, a Simple Example of How the `Location2Location` Class Might Be Employed

```
import javax.media.*;
import javax.media.protocol.*;
import javax.media.format.*;

/**************************************************************
* Simple example to show the Location2Location class in action.
* The Location2Location class transfer media from one location to
* another performing any requested tanscoding (format changes)
* at the same time.
*
* The class is used twice. Once to transform a short wave audio
* file of an electric guitar (guitar.wav) into MP3 format.
* The second example converts of Quicktime version of the example
* video from chapter 7, encoded with the Indeo 5.o codec and
* GSM audio into an AVI version with Cinepak codec for the video
* and linear encoding for the audio.
**************************************************************/
public class StaticTranscode {

public static void main(String[] args) {
  String  src;
  String  dest;
```

```
  Format[]  formats;
  ContentDescriptor container;
  int     waited;
  Location2Location dupe;

  ////////////////////////////////////////////////////////////
  // Transcode a wave audio file into an MP3 file, transferring it
  // to a new location (dest) at the same time.
  ////////////////////////////////////////////////////////////
  src = "file://d:\\jmf\\book\\media\\guitar.wav";
  dest = "file://d:\\jmf\\book\\media\\guitar.mp3";
  formats = new Format[1];
  formats[0] = new AudioFormat(AudioFormat.MPEGLAYER3);
  container = new FileTypeDescriptor(FileTypeDescriptor.MPEG_AUDIO);

  dupe = new Location2Location(src,dest,formats,container);
  System.out.println("After creation, state = " + dupe.getStateName());
  waited = dupe.transfer(10000);
  System.out.println("Waited " + waited + " milliseconds. State is"
+ " now " +dupe.getStateName());


  ////////////////////////////////////////////////////////////
  // Transcode a Quicktime version of a movie into an AVI version.
  // The video codec is altered from Indeo5.0 to Cinepak,the audio
  // track is transcoded from GSM to linear, and is result is saved
  // as a file "qaz.avi".
  ////////////////////////////////////////////////////////////
  src = "file://d:\\jmf\\book\\media\\videoexample\\iv50_320x240.mov";
  dest = "file://d:\\jmf\\book\\media\\qaz.avi";
  formats = new Format[2];
  formats[0] = new VideoFormat(VideoFormat.CINEPAK);
  formats[1] = new AudioFormat(AudioFormat.LINEAR);
  container = new FileTypeDescriptor(FileTypeDescriptor.MSVIDEO);
  dupe = new Location2Location(src,dest,formats,container,5.0f);
  System.out.println("After creation, state = " + dupe.getStateName());
  waited = dupe.transfer(Location2Location.INDEFINITE);
  int state = dupe.getState();
  System.out.println("Waited " + (waited/1000) + " seconds. State is"
     + " now " +dupe.getStateName() + ", rate was " + dupe.getRate());
  System.exit(0);
}
}
```

The output of `StaticTranscode` is shown in Listing 8.15. As you can see, the audio file, 6.5 seconds in length, took just over 2.5 seconds to transcode and save. On the other hand, the movie of less than 1 minute (57 seconds) took more than 10 minutes to transcode and save. This was on a Pentium IV system with 256MB of RAM. Clearly these processes can take a long time to complete. Furthermore, the actual time required varies depending on other factors such as system load at the time. Running the same program again saw variations of as much as 20% in time to complete.

Listing 8.15 Output of the `StaticTranscode` Program Showing Time Needed to Transcode and Sink the Media

```
After creation, state = 1
Waited 2660 milliseconds. State is now 4
After creation, state = 1
Waited 618 seconds. State is now 4, rate was 1.0
```

Media Capture

One of the more exciting aspects of the JMF, particularly when combined with transmission over a network as discussed in Chapter 9, is the ability to capture media directly from devices attached to the computer. The stereotypical examples of these devices are microphones, video cameras, and video capture boards.

Thus it is possible to directly record sound or video through the appropriate hardware connected to the PC and either save that to a file, play it back, process and transcode it, or even transmit it.

The JMF model of media capture sees the capture device as a `DataSource`. With the appropriate initialization steps, detailed later, media capture falls within the precincts of playing or processing media: When a `DataSource` has been found, it can be handled in any way.

Several classes have a role to play in media capture. They are as follows:

`CaptureDeviceManager`— Manages the central registry of capture devices known to the JMF. Provides a means for querying and updating that registry as well as a means of obtaining a particular capture device's information (`CaptureDeviceInfo`).

`CaptureDeviceInfo`— Information about a particular capture device, including the Formats it supports. Most importantly, it has a `MediaLocator` describing the device from which a `DataSource` can be created (hence the media obtained).

`CaptureDevice`— A further specialization of the `DataSource` produced by a capture device to include appropriate methods for control of the device and its output.

The process of media capture via the JMF tends to proceed as follows:

1. Obtain a `CaptureDeviceInfo` object for the device from which media will be captured (typically obtained by querying `CaptureDeviceManager`).

2. Get the `CaptureDeviceInfo`'s `MediaLocator`.

3. Create a `DataSource` from the `MediaLocator`.

4. Create a `Player` or `Processor` using the `Manager` class and the `DataSource` from the previous step.

5. Perform any necessary configuration or programming (for example, `Processor` programming or creation of `DataSink`).

6. Start the `Player` or `Processor`.

The following subsections provide more details of the process and classes involved.

`JMFRegistry` and `JMStudio`

It is worth mentioning that the two utilities, `JMFRegistry` and `JMStudio` that come as part of the JMF 2.1.1 distribution, provide direct control over and information regarding capture.

The `JMFRegistry` application provides, among other features, the ability to query what capture devices are available on a system. Among the most important information that `JMFRegistry` provides is the name of the device as it is known to the JMF. This is the name by which the user can obtain that device from the `DeviceManager`. Similarly, if a new capture device is added to a system, the `JMFRegistry` can be used to update the list of capture devices known to the JMF. The `JMFRegistry` is simply invoked as Java `JMFRegistry` at the command prompt.

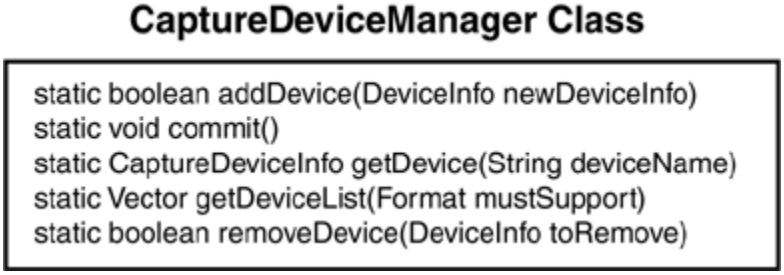
Media capture can be performed directly through the `JMStudio` application. This is often convenient as an alternative to writing code for one-off capture of audio and video. Capture is performed through the Capture option on the File menu, which presents all known capture devices. Such media can also be exported (saved).

`CaptureDeviceManager`

The `CaptureDeviceManager` class is the first stop in writing JMF code that captures media. This is because the class is the means by which the chain that leads to a `DataSource` coming from a capture device is started. The `CaptureDeviceManager` can return a `DeviceInfo` object corresponding to a named capture device or all devices that support a particular `Format`.

Figure 8.35 shows the methods of `CaptureDeviceManager`. Similar to the other manager classes, all methods are static (invoked using the classname). Of the five methods, three—`addDevice()`, `removeDevice()`, and `commit()`—are concerned with updating JMF's knowledge of connected capture devices. Although the methods provide automatic (program) control over the addition or removal of a device, the same operations can be performed easily through the `JMRegistry` application.

Figure 8.35. The `CaptureDeviceManager` class.



The `getDevice()` and `getDeviceList()` methods are the two key methods of the class in terms of initiating a capture task. Given the name of a capture device as a `String`, `getDevice()` returns a `CaptureDeviceInfo` object that matches the device. `CaptureDevices` have names such as `"vfw:Logitech USB Video Camera:0"` or `"DirectSoundCapture"`, and must be known to the user. The method returns `null` if the matching capture device couldn't be found.

The alternative means of initiating capture is to specify the desired `Format` of the captured data and find a capture device that can produce data in that format. This capability is provided through the `getDeviceList()` method. The method returns a `Vector` of `CaptureDeviceInfo` objects that support the `Format` in question. Passing a `null Format` object to the method results in it returning a `CaptureDeviceInfo` object for all capture devices known. This feature is used by the simple `ListCaptureDevices` application found in Listing 8.15, as well as on the book's companion Web site. The application simply prints a list of all capture devices on the system that the JMF is aware of.

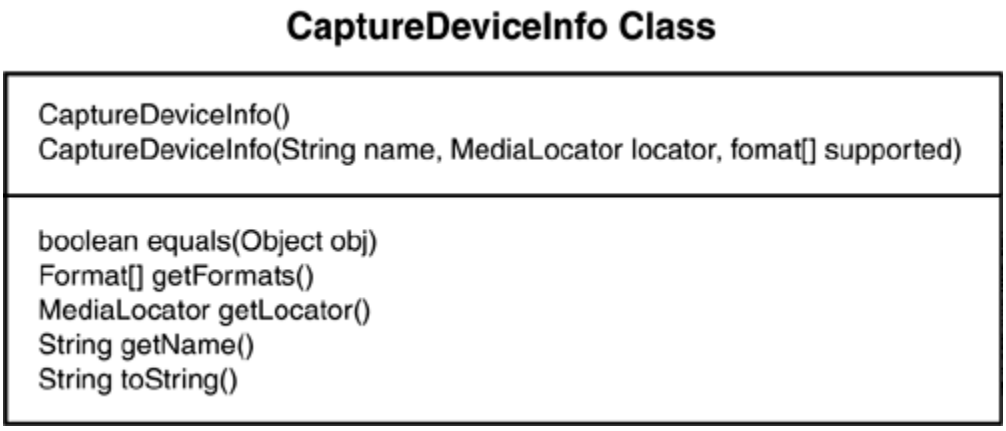
Listing 8.15 The `ListCaptureDevices` Application that Lists All Capture Devices on the Current Machine

```
import javax.media.*;
import java.util.*;

/**************************************************************
* Simple application to list all capture devices currently
* known to the JMF. The CaptureDeviceManager is queried as to
* known devices and its output printed to the screen.
*
* @author Michael (Spike) Barlow
**************************************************************/
public class ListCaptureDevices {

public static void main(String[] args) {

  ////////////////////////////////////////////////////////
  // Query CaptureDeviceManager about ANY capture devices (null
  // format)
  Vector info = CaptureDeviceManager.getDeviceList(null);
  if (info==null)
    System.out.println("No Capture devices known to JMF");
  else {
    System.out.println("The following " + info.size() +
               " capture devices are known to the JMF");
    for (int i=0;i<info.size();i++)
      System.out.println("\t"+(CaptureDeviceInfo)info.elementAt(i));
  }
}
}
```

`CaptureDeviceInfo`

The `CaptureDeviceInfo` class is the JMF mechanism for describing a capture device. It encapsulates the device's name, supported `Formats`, and a unique `MediaLocator` that can be used to source the data the device produces. Figure 8.36 shows the methods of the class.

Figure 8.36. The `CaptureDeviceInfo` class.



Although the class has constructors, these are for those third-party developers who are extending the JMF by writing drivers for a new device. The chief means of obtaining a `CaptureDeviceInfo` object is through the `CaptureDeviceManager` class with either the `getDevice()` or `getDeviceList()` methods.

The key method of `CaptureDeviceInfo` is `getLocator()`, which returns a `MediaLocator` for the device in question. Through the `Manager` class, that `MediaLocator` can then be used to create a `DataSource`—the data being captured by the device.

`CaptureDevice`

The `CaptureDevice` interface is a specialization of a `DataSource` that includes appropriate capture device functionality. This interface isn't typically employed by the programmer because its functionality is covered by the `Processor` or `Player` that is handling the captured media. However, the interface does provide the ability to control the `Format` of individual streams originating from the device as well as controls for starting, stopping, connecting, and disconnecting.

Audio or Video Capture with the `SimpleRecorder` Application

As an illustration of the process of audio or video capture, the `SimpleRecorder` application in Listing 8.16 (also on the book's companion Web site) is provided.

Listing 8.16 The `SimpleRecorder` Application Illustrating the Media Capture Process

```
import javax.media.*;
import javax.media.format.*;
import javax.media.protocol.*;
```

```
import java.util.*;

/**********************************************************************
* A simple application to allow users to capture audio or video
* through devices connected to the PC. Via command-line arguments
* the user specifies whether audio (-a) or video (-v) capture,
* the duration of the capture (-d) in seconds, and the file to
* write the media to (-f).
*
* The application would be far more useful and versatile if it
* provided control over the formats of the audio and video
* captured as well as the content type of the output.
*
* The class searches for capture devices that support the
* particular default track formats: linear for audio and
* Cinepak for video. As a fall-back two device names are
* hard-coded into the application as an example of how to
* obtain DeviceInfo when a device's name is known. The user may
* force the application to use these names by using the -k
* (known devices) flag.
*
* The class is static but employs the earlier Location2Location
* example to perform all the Processor and DataSink related work.
* Thus the application chiefly involves CaptureDevice related
* operations.
*
* @author Michael (Spike) Barlow
**********************************************************************/
public class SimpleRecorder {

////////////////////////////////////////////////////////
// Names for the audio and video capture devices on the
// author's system. These will vary system to system but are
// only used as a fallback.
////////////////////////////////////////////////////////
private static final String AUDIO_DEVICE_NAME = "DirectSoundCapture";
private static final String VIDEO_DEVICE_NAME =
                            "vfw:Microsoft WDM Image Capture:0";

////////////////////////////////////////////////////////
// Default names for the files to write the output to for
// the case where they are not supplie by the user.
////////////////////////////////////////////////////////
private static final String DEFAULT_AUDIO_NAME =
                                "file://./captured.wav";
private static final String DEFAULT_VIDEO_NAME =
                               "file://./captured.avi";

////////////////////////////////////////
// Type of capture requested by the user.
////////////////////////////////////////
private static final String AUDIO = "audio";
private static final String VIDEO = "video";
private static final String BOTH = "audio and video";

////////////////////////////////////////////////////////////////
// The only audio and video formats that the particular application
// supports. A better program would allow user selection of formats
// but would grow past the small example size.
```

```
//////////////////////////////////////////////////////////////
private static final Format AUDIO_FORMAT =
                  new AudioFormat(AudioFormat.LINEAR);
private static final Format VIDEO_FORMAT =
                  new VideoFormat(VideoFormat.CINEPAK);


public static void main(String[] args) {

  ////////////////////////////////////////////////////
  // Object to handle the processing and sinking of the
  // data captured from the device.
  ////////////////////////////////////////////////////
  Location2Location capture;

  ///////////////////////////////////
  // Audio and video capture devices.
  ///////////////////////////////////
  CaptureDeviceInfo audioDevice = null;
  CaptureDeviceInfo videoDevice = null;

  ////////////////////////////////////////////////////////////
  // Capture device's "location" plus the name and location of
  // the destination.
  ////////////////////////////////////////////////////////////
  MediaLocator      captureLocation = null;
  MediaLocator      destinationLocation;
  String            destinationName = null;

  ////////////////////////////////////////////////////////////
  // Formats the Processor (in Location2Location) must match.
  ////////////////////////////////////////////////////////////
  Format[]          formats = new Format[1];

  ////////////////////////////////////////////
  // Content type for an audio or video capture.
  ////////////////////////////////////////////
  ContentDescriptor audioContainer = new
                   ContentDescriptor(FileTypeDescriptor.WAVE);
  ContentDescriptor videoContainer = new
                  ContentDescriptor(FileTypeDescriptor.MSVIDEO);
  ContentDescriptor container = null;

  ////////////////////////////////////////////////////////////////
  // Duration of recording (in seconds) and period to wait afterwards
  ////////////////////////////////////////////////////////////////
  double            duration = 10;
  int               waitFor = 0;

  //////////////////////////
  // Audio or video capture?
  //////////////////////////
  String            selected = AUDIO;

  ////////////////////////////////////////////////////
  // All devices that support the format in question.
  // A means of "ensuring" the program works on different
  // machines with different capture devices.
  ////////////////////////////////////////////////////
```

```
Vector           devices;

//////////////////////////////////////////////////////
// Whether to search for capture devices that support the
// format or use the devices whos names are already
// known to the application.
//////////////////////////////////////////////////////
boolean          useKnownDevices = false;


//////////////////////////////////////////////////////
// Process the command-line options as to audio or video,
// duration, and file to save to.
//////////////////////////////////////////////////////
for (int i=0;i<args.length;i++) {
  if (args[i].equals("-d")) {
    try { duration = (new Double(args[++i])).doubleValue(); }
    catch(NumberFormatException e) { }
  }
  else if (args[i].equals("-w")) {
    try { waitFor = Integer.parseInt(args[++i]); }
          catch(NumberFormatException e) { }
  }
  else if (args[i].equals("-a")) {
    selected = AUDIO;
  }
  else if (args[i].equals("-v")) {
    selected = VIDEO;
  }
  else if (args[i].equals("-b")) {
    selected = BOTH;
  }
  else if (args[i].equals("-f")) {
    destinationName = args[++i];
  }
  else if (args[i].equals("-k")) {
    useKnownDevices = true;
  }
  else if (args[i].equals("-h")) {
    System.out.println("Call as java SimpleRecorder [-a | -v | -b]"
       + " [-d duration] [-f file] [-k] [-w wait]");
    System.out.println("\t-a\tAudio\n\t-v\tVideo\n\t-b\tBoth "
       + "audio and video (system dependent)");
    System.out.println("\t-d\trecording Duration (seconds)");
    System.out.println("\t-f\tFile to save to\n\t-k\tuse Known"
       + " device names (don't search for devices)");
    System.out.println("\t-w\tWait the specified time (seconds)"
       + " before abandoning capture");
    System.out.println("Defaults: 10 seconds, audio, and "
      +"captured.wav or captured.avi, 4x recording duration wait");
    System.exit(0);
  }
}


//////////////////////////////////////////////////////////////
// Perform setup for audio capture. Includes finding a suitable
// device, obatining its MediaLocator and setting the content
// type.
```

```
////////////////////////////////////////////////////////////
if (selected.equals(AUDIO)) {
  devices = CaptureDeviceManager.getDeviceList(AUDIO_FORMAT);
  if (devices.size()>0 && !useKnownDevices)
    audioDevice = (CaptureDeviceInfo)devices.elementAt(0);
  else
    audioDevice =
        CaptureDeviceManager.getDevice(AUDIO_DEVICE_NAME);
  if (audioDevice==null) {
    System.out.println("Can't find suitable audio " +
       "device. Exiting");
    System.exit(1);
  }
  captureLocation = audioDevice.getLocator();
  formats[0] = AUDIO_FORMAT;
  if (destinationName==null)
    destinationName = DEFAULT_AUDIO_NAME;
  container = audioContainer;
}
////////////////////////////////////////////////////////////
// Perform setup for video capture. Includes finding a suitable
// device, obatining its MediaLocator and setting the content
// type.
////////////////////////////////////////////////////////////
else if (selected.equals(VIDEO)) {
  devices = CaptureDeviceManager.getDeviceList(VIDEO_FORMAT);
  if (devices.size()>0 && !useKnownDevices)
    videoDevice = (CaptureDeviceInfo)devices.elementAt(0);
  else
    videoDevice =
        CaptureDeviceManager.getDevice(VIDEO_DEVICE_NAME);
  if (videoDevice==null) {
    System.out.println("Can't find suitable video "
       + "device. Exiting");
    System.exit(1);
  }
  captureLocation = videoDevice.getLocator();
  formats[0] = VIDEO_FORMAT;
  if (destinationName==null)
    destinationName = DEFAULT_VIDEO_NAME;
  container = videoContainer;
}
else if (selected.equals(BOTH)) {
  captureLocation = null;
  formats = new Format[2];
  formats[0] = AUDIO_FORMAT;
  formats[1] = VIDEO_FORMAT;
  container = videoContainer;
  if (destinationName==null)
    destinationName = DEFAULT_VIDEO_NAME;
}

////////////////////////////////////////////////////////////
// Perform all the necessary Processor and DataSink preparation via
// the Location2Location class.
////////////////////////////////////////////////////////////
destinationLocation = new MediaLocator(destinationName);
System.out.println("Configuring for capture. Please wait.");
capture = new Location2Location(captureLocation,
```

```
                    destinationLocation,formats,container,1.0);

  ////////////////////////////////////////////////////////////////
  // Start the recording and tell the user. Specify the length of the
  // recording. Then wait around for up to 4-times the duration of
  // recording (can take longer to sink/write the data so should wait
  // a bit incase).
  ////////////////////////////////////////////////////////////////
  System.out.println("Started recording " + duration +
          " seconds of " + selected + " ...");
  capture.setStopTime(new Time(duration));
  if (waitFor==0)
    waitFor = (int)(4000*duration);
  else
    waitFor *= 1000;
  int waited = capture.transfer(waitFor);

  ////////////////////////////////////////////////////
  // Report on the success (or otherwise) of the recording.
  ////////////////////////////////////////////////////
  int state = capture.getState();
  if (state==Location2Location.FINISHED)
    System.out.println(selected + " capture successful " +
      "in approximately " + ((int)((waited+500)/1000)) +
      " seconds. Data written to " + destinationName);
  else if (state==Location2Location.FAILED)
    System.out.println(selected + " capture failed " +
      "after approximately " + ((int)((waited+500)/1000)) +
      " seconds");
  else {
    System.out.println(selected + " capture still ongoing " +
      "after approximately " + ((int)((waited+500)/1000)) +
      " seconds");
    System.out.println("Process likely to have failed");
  }

  System.exit(0);
}
}
```

The application allows the user to record audio or video (or simultaneous audio and video if the system supports it) from devices attached to the machine. Via command-line arguments, the user can specify audio (-a) or video (-v) capture, duration, and other settings. Listing 8.17 shows the help output of the program. Thus, for instance, to capture 20 seconds of video and save the output to a file 20seconds.avi, the program would be invoked as java SimpleRecorder –v –d 20 –f file://20seconds.avi. The application is restricted as to media output format—linear for audio and Cinepak for video—and content type—Wave for audio and AVI for video— simply to keep the example small. A more thorough and useful application would provide the user with the means of specifying the formats.

Listing 8.17 The Help Output of the `SimpleRecorder` Application Showing Its Various Options

```
D:\JMF\Book\Code>java SimpleRecorder -h
Call as java SimpleRecorder [-a | -v | -b] [-d duration] [-f file]
```

```
  [-k] [-w wait]
        -a      Audio
        -v      Video
        -b      Both audio and video (system dependent)
        -d      recording Duration (seconds)
        -f      File to save to
        -k      use Known device names (don't search for devices)
        -w      Wait the specified time (seconds) before abandoning capture
Defaults: 10 seconds, audio, and captured.wav or captured.avi,
  4x recording duration wait
```

The application builds on the earlier `Location2Location` utility class example that took media in one location, transcoded it, and output it to another. Thus the `SimpleRecorder` class consists chiefly of the capture device related elements of the process (as well as user-interface), whereas all `Processor` and `DataSink` related work is dealt with by the `Location2Location` class.

Capture device setup is performed (regardless of audio or video) by querying the `CaptureDeviceManager` to obtain a list of devices that support the format in question. As a fallback position, hard-coded device names are also provided and can be used directly. You might want to use the earlier `ListCaptureDevices` application to ascertain what devices are available upon your machine and alter the source of `SimpleRecorder` (at the top of the class) where the device names are hard-coded. From the resulting `CaptureDeviceInfo` object, a `MediaLocator` can be obtained. Along with the necessary format information, this is sufficient to construct a `Location2Location` object that will perform the data transcoding and sinking.

Recording for a specified duration only is achieved through the `Location2Location` object's `setStopTime()` method. That sets the media stop time on the `Processor` object. This is necessary because microphone and video camera devices are push data sources—they will continue to supply data indefinitely. By setting the `Processor`'s stop time, it will automatically stop when the media time reaches that specified. That is detected by the `DataSink` of `Location2Location`, which subsequently closes itself.

As a final point, it is worth noting that capturing simultaneous audio and video is supported by `SimpleRecorder`, but it is system dependent on whether it will function correctly. (Does the system have a capture device that can provide audio and video?) One of the `ProcessorModel` constructors accepts specification of the output (an array of `Formats` and a `ContentDescriptor`) but no specification of the input `DataSource`. In this case, the JMF searches for a capture device(s) that can support those required formats. The `Location2Location` class is written in such a way that if it receives a null `MediaLocator` as a specification of the media source (to its constructor), it uses this second (no source information) constructor for `ProcessorModel`.

Summary

This chapter is the second of three that cover the control and processing of time-based media using the JMF. It is the core chapter on the JMF, covering the paradigms of control and processing of media both through illustration and discussion of the classes involved in the process. Chapter 7 serves as a general

introduction to media and the JMF, whereas Chapter 9 covers the more specialized topics of streaming media and extending the JMF.

This chapter falls into three broad modules or meta-sections. The first module serves as the building block or framework on which understanding of the JMF is built as well as on which the processing and media capture approaches sit. This module covers key concepts of the JMF including the asynchronous `Controller` model of time and state, the central registry role of the manager classes, and the means of sourcing and sinking (outputting) media through the `DataSource` and `DataSink` classes.

The second module concerns the play and processing of media via the `Player` and `Processor` classes. Creation of these two classes is discussed as well as the very important concept of programming a `Processor` in order for it to achieve the desired task. `PlugIns` and their role in the processing chain are also discussed, and the module is illustrated with several examples.

The final module concerns capture of media via devices attached to the computer such as cameras or microphones. The relevant classes representing capture devices are discussed and where they fit within the processing framework already discussed. The module concludes with an illustrative example that shows how capture can be performed, and which utilizes one of the earlier examples.

Chapter 9. RTP and Advanced Time-Based Media Topics

IN THIS CHAPTER

- What's RTP?
- RTP with the JMF
- Extending the JMF
- `JMFCustomizer`
- Synchronization
- The JMF in Conjunction with other APIs
- Java Sound
- Future Directions for the JMF

This is the third of three chapters concerning time-based media (typically sound and video) and Java (chiefly the JMF: Java Media Framework). Although the previous two chapters present an introduction to, and then details of, the JMF, this chapter covers more advanced topics in the field of time-based media processing in Java.

The greater portion of the chapter concerns the JMF in two areas. The first area is that of real-time streaming of media in the JMF via RTP (Real-Time Transport Protocol). This enables applications such as video-conferencing or Web broadcasts to be written using the JMF. The section discusses the basics of RTP before covering the JMF classes that provide the necessary support. The second area concerns extending the JMF by implementing one or more of the various interfaces that are the true core of the JMF. Finally, indications of how the JMF can be connected to other Java APIs or platform features and classes are given.

However, a portion of the chapter doesn't concern the JMF at all, but other Java APIs concerned with time-based media. In particular, Java Sound—a core (as of Java 1.3) platform API dealing with sampled and MIDI sound—is briefly addressed. Integrating the JMF with other Java APIs also is covered before the chapter concludes by examining the future of the JMF.

What's RTP?

RTP is the Real-Time Transport Protocol, an Internet standard for the transport of real-time data (such as audio or video). RTP is defined by the Audio-Video Transport Working Group (AVT Working Group) of the Internet Engineering Task Force (IETF). The IETF (http://www.ietf.org) is an open community concerned with the evolution of the Internet and part of the larger Internet Society (ISOC), a professional membership society that oversees the issues that affect the Internet.

Given RTP's pedigree, which is designed by the Audio-Video Transport Working Group of arguably the Internet's chief standards body, it shouldn't be surprising at all that Sun has adopted RTP as the mechanism for streaming media within the JMF. Thus, to write applications such as video conferencing or even a player of broadcast media in the JMF requires the use of the RTP. But what is RTP, and where does it fit in the scheme of things? Those readers wanting to skip the details and simply write streaming media applications without knowledge of the JMF can do so for a time. As with much of the JMF, the user is provided with a very abstract model that shields him from much of the detail. In that

case, readers should move through to the next section concerning RTP and the JMF. However, for those doing anything significant in the area of streaming media, it is likely that the material in this section will need to be visited at some time in the future.

RTP is described by an RFC (Request for Comments) of the IETF: RFC1889 (http://www.ietf.org/rfc/rfc1889.txt). Despite the innocuous name, RTP as described by the RFC is a standard that has been in its current form since early 1996 and is thus stable. The abstract of the RFC describes RTP as follows:

RTP provides end-to-end network transport functions suitable for applications transmitting real-time data, such as audio, video, or simulation data, over multicast or unicast network services.

The introduction section of the document states:

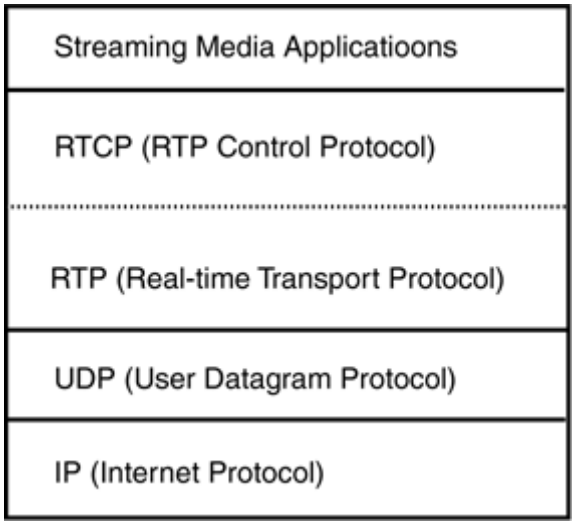
Applications typically run RTP on top of UDP to make use of its multiplexing and checksum services; both protocols contribute parts of the transport protocol functionality.

The services provided by the RTP include the identification of content type (that is, type and format of media) within a data packet, packet numbering, packet time stamping, and the ability to synchronize media streams from different sources. These are a minimal set of services that might be expected from a protocol providing media transport. Given that data might be delayed for different intervals, be corrupted, or be lost, it is possible for data packets to arrive out of order, not arrive, or that streams synchronized at the source site (for example, captured audio and video in a video-conference) arrive out of synch at the destination. Numbering, time stamps, and identification of content type provide the means for the detection of these problems and the ability for them to be rectified. However higher-level services such as connection negotiation or quality-of-service guarantees aren't part of RTP. RTP was designed to be lean and make the minimal demands on the bandwidth over which the media is being transported. This means that such services aren't the domain of the RTP.

IP and UDP

Figure 9.1 shows the typical case involving streaming media via the RTP, and the lower-level level protocols upon which it sits. Although RTP doesn't require UDP (User Datagram Protocol), it is by far the most common protocol atop which RTP is implemented.

Figure 9.1. Most common layering of RTP atop UDP/IP to provide media streaming capabilities.



IP (Internet Protocol), which is more commonly heard as part of TCP/IP (Transmission Control Protocol atop Internet Protocol), is a low-level protocol by which most hosts on the Internet communicate with one another. It is a means by which hosts and routers ensure that data packets travel from source to destination host while hiding the details of the transmission medium. A number of protocols are built atop IP.

UDP is a lightweight communication protocol for the transportation of data packets. Inherently packet oriented, UDP is a low overhead protocol (as opposed to say TCP) because of the restricted services it provides. No guarantee is made of packet delivery; UDP provides effectively blind transmission of data. This means that packets can be lost, corrupted, or out-of-order, and one or both ends of the communication channel could be unaware of the fact. Hence, it isn't uncommon for higher-level protocols to be built atop UDP in order to capitalize on its efficiency while building in the possibility of error checking and recovery. RTP is such a protocol.

RTP and RTCP

RTP is augmented by a control protocol—RTCP (RTP Control Protocol). The purpose of RTCP is to provide information about the quality of service of an RTP connection by identifying the participants and relevant information about each. Such information is sent by each participant and includes the number of packets received (if receiving) and sent (if sending), and other timing (clock) and synchronization information. The same RFC (http://www.ietf.org/rfc/rfc1889.txt) that describes RTP also describes RTCP.

All RTP packets are composed of two parts: a fixed header and the associated payload. The header includes a payload type (type of media), sequence number (packet number within the media sequence), time stamp, synchronization source, and contributing source (where the media originated from). The header can range in size from 12 bytes (the most common case of media originating from a single source) to 72 bytes (media originating from 16 different sources). The payload is the media data itself.

RTCP packets are compound (consisting of at least two, one of which is always a Source Description), but fall into one of five different types:

Sender Report— Produced by those who have been sending packets recently. A Sender Report includes the total number of packets and bytes sent as well as synchronization (timing) information.

Receiver's Report— Produced by those who have been receiving packets recently. Participants send a Receiver's Report packet for each participant they are receiving data from. Information includes number of packets lost, highest (packet) sequence number received, and a timestamp that can be used by the sender to estimate the lag/latency between sender and receiver.

Source Description— Description of the source of the report in canonical name; also possibly other information such as e-mail addresses or physical locations.

Bye— Sent by a participant who is leaving the session. Might include the reason for leaving.

Application Specific— A means for applications to define their own messaging across RTCP.

RTP Applications

RTP applications can be divided into clients, those that passively receive, and servers, those that actively transmit. Some, such as video-conferencing software, are both clients and servers—transmitting and receiving data.

The following terminology describes RTP as used by RTP applications:

RTP Session— An association between a group of applications, all communicating via RTP. A session is identified by a network address and a pair of ports—one for the RTP packets and one for the RTCP packets. Each media type has its own session. Hence for any number of applications participating in the stereotypical video conference (involving both audio and video), it will consist of two sessions—one for audio and one for video.

RTP Participant— An application taking part in an RTP Session.

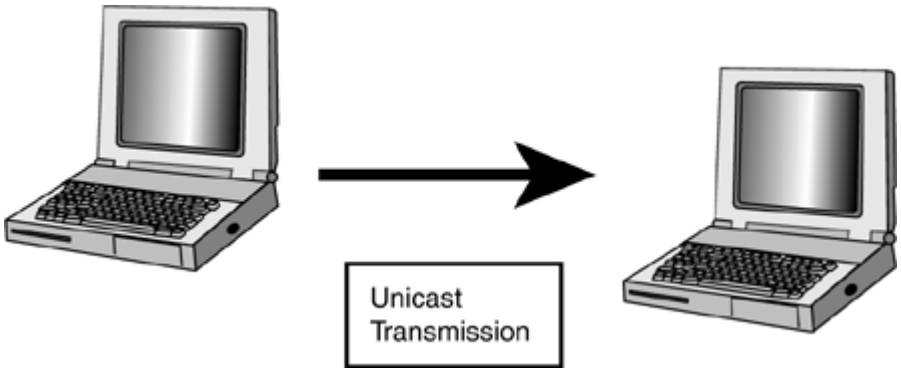
RTP Port— An integer number used to differentiate between different applications on the same machine. Many common network services have a port associated with them.

Unicast, Multi-Unicast, Broadcast, and Multicast

IP supports a number of addressing schemes: unicast, broadcast, and multicast. The type of addressing scheme is indicated by the IP address of a packet. The three modes can be used in conjunction with RTP (and the JMF).

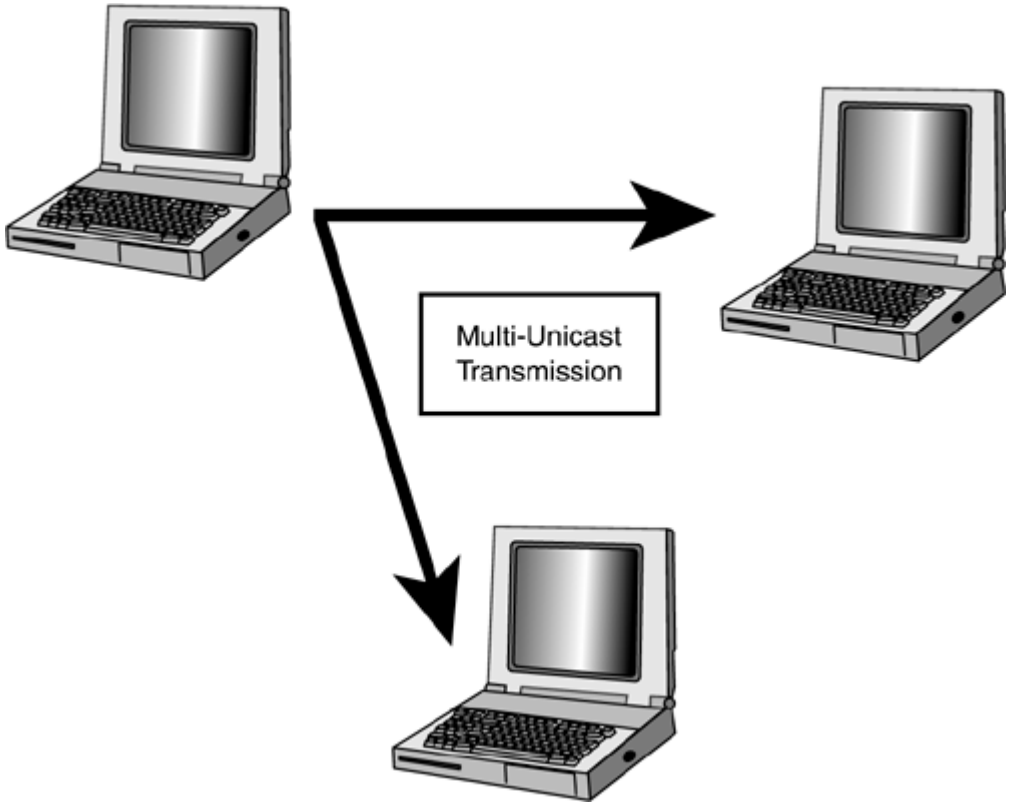
Unicast, also known as point-to-point, is by far the most common addressing scheme in use on the Internet today, and it describes the transmission of a packet (from a source) to a single address. Figure 9.2 is a schematic of this addressing scheme. In a time-based media context, this approach would be the most sensible for a simple two-person Internet phone scenario—two people transmitting directly to each other.

Figure 9.2. Typical unicast scenario—point-to-point.



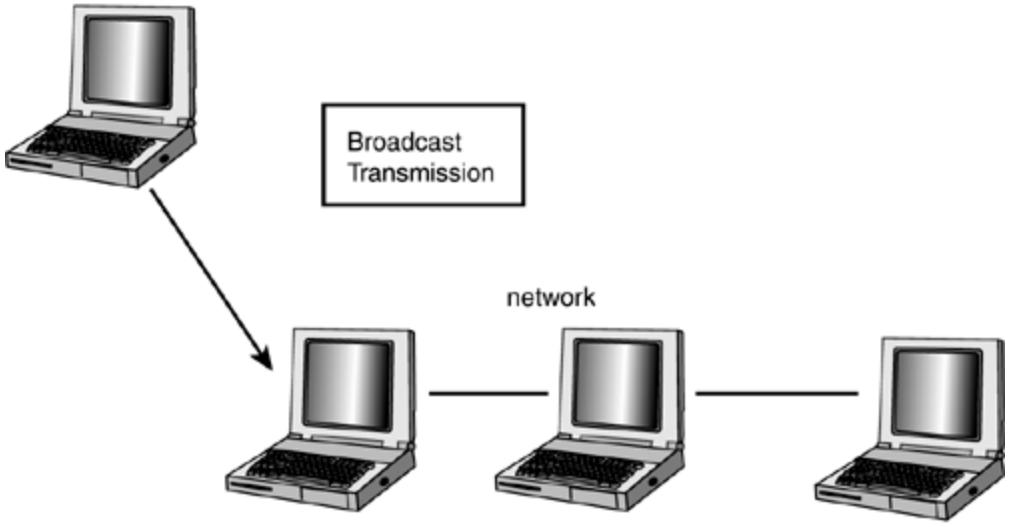
Multi-unicast is a simple expansion of unicast in that the transmitter sends duplicates of packets to a number of hosts, not just one. In multi-unicast, the packets are duplicated, so it has none of the bandwidth advantages of the multicast approach. Figure 9.3 is a schematic of this scheme. In Figure 9.3's scenario, the transmitter's data is duplicated and sent as two separate streams to the two recipients. A video-conferencing application between three or four participants might use multi-unicast: Each participant would know the address of the other members involved and transmit (audio and video streams) directly to each of those addresses.

Figure 9.3. A multi-unicast scenario with two recipients.



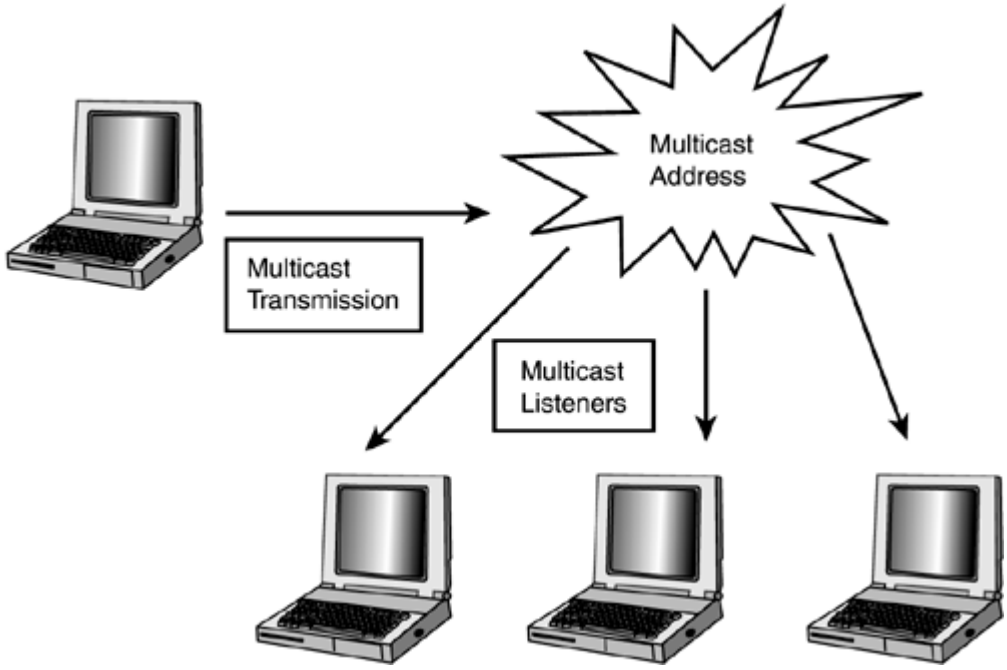
Broadcast describes the transmission of packets to all hosts on a particular subnet. Although it offers bandwidth savings (packets are not duplicated till necessary), it is limited by the constraint of a single subnet. Figure 9.4 is a schematic of this approach to addressing. As an example, broadcast might be used within an organization to send a video to all machines.

Figure 9.4. Typical Broadcast transmission—the data is sent to all machines on a particular subnet.



Multicast describes the most sophisticated and versatile means of addressing: one that is also of particular significance for many time-based media applications. Multicast is a receiver-centric scheme. The transmitter sends to a single address—that of a multicast session. Receivers join a session by indicating they want to listen to the address associated with a session. The network infrastructure (the routers) is then responsible for delivering data to all receivers (listeners). Figure 9.5 is a schematic of the approach. In this scenario, the transmitter sends to a multicast address. Receivers indicate that they want to listen to that address, and the network infrastructure (routers) is responsible for delivering the data.

Figure 9.5. Typical multicast transmission scenario.



Multicasting is of particular significance to applications such as multi-participant video-conferencing for at least two reasons. First, each participant isn't required to maintain an up-to-date list of other participants—a difficult task because participants come and go for various reasons. Each participant simply transmits and listens to the session address. Second, a multicast scheme means that data packets aren't duplicated until necessary, implying considerable potential bandwidth savings. Only when the route to listeners to a session diverges are the packets duplicated. This is all supported by the network.

Certain network addresses, namely those in the range 224.0.0.0 to 239.255.255.255, are assigned by IANA (Internet Assigned Numbers Authority) for multicast applications. Addresses within that range are further subdivided into various assigned purposes. For instance, the addresses from 224.2.0.0 to 224.2.127.253 (inclusive) are currently assigned for multimedia conference calls. The complete list of multicast assigned numbers can be found at http://www.iana.org/assignments/multicast-addresses.

Multicasting is a complex topic, particularly in terms of how the routing is achieved. That is further complicated by the fact that not all older routers are capable of supporting multicast packets. To this end, MBONE (the Internet Multicast Backbone) was created as a group of networks and routers that supported multicast.

RTP with the JMF

Three packages within the JMF are concerned with RTP. They are

`javax.media.rtp`— The top-level of the three packages dealing with RTP. It comprises 26 classes (most interfaces) dealing with streaming content with RTP.

`javax.media.rtp.event`— A package of 23 events that might result when using RTP.

`javax.media.rtp.rtcp`— A package of five classes (four of which are interfaces) defining usage of RTCP within the JMF.

Those applications employing the RTP directly will likely need to import classes from all three packages.

It is worth mentioning that as for `PlugIns` (discussed in Chapter 8), it isn't required that a JMF implementation support or provide the classes found in the preceding three packages. All current implementations of 2.1.1 (reference, Windows, Solaris, Linux) do so. However, it is possible that some future implementation of the JMF—possibly intended for a low-powered embedded system—won't support the RTP related classes.

One key aspect to understand about the RTP related classes of the JMF are that they are extensions to the JMF. They don't replace or supplant the core functionality of the JMF as found in the Player, Processor, `DataSource`, `DataSink`, and `Manager` (among others) classes. These remain unchanged and still lie at the heart of media handling. They fulfil the same role regardless of whether the media is streamed over the Internet or from the local filesystem.

In other words, a JMF program that plays RTP media will still employ a `Player` object obtained through the `Manager` class, as discussed in Chapter 8. Similarly, a JMF program transcoding RTP data it received (from a remote participant in an RTP session) to another format for saving to a local file will still use a `Processor`, `DataSource`, and `DataSink` object. A video-conferencing application will still use `CaptureDeviceInfo`, `Processor`, and `DataSource` objects (amongst others).

Indeed, as discussed in a following subsection, it is possible to play, process, and in general handle media originating from or destined for transport via RTP without employing a single class from the above three packages.

RTP Content Types and Formats

The full gamut of content types (media containers) and formats offered by the JMF aren't available for RTP. In the case of RTP, the choices are far more limited. Those users wanting to use RTP (in particular to transmit over RTP) must be aware of their choices and use only an RTP supported format and content type.

In the area of content type, although there are more than a dozen different `ContentDescriptors` (such as Wave, AVI, GSM, and QuickTime) within the JMF, there is only one for RTP media:

`ContentDescriptor.RAW_RTP`. Thus the creation of a `ContentDescriptor` object for RTP always has the following form:

```
ContentDescriptor rtpContainer = new
          ContentDescriptor(ContentDescriptor.RAW_RTP);
```

The JMF support for the format of RTP data is also limited. Although it is possible for the user to extend the JMF by implementing the appropriate interfaces and thus adding further RTP-conversant codecs, the JMF currently provides four standard RTP-specific audio formats and three standard RTP-specific video formats.

The audio formats are known as

- `ULAW_RTP`
- `GSM_RTP`
- `DVI_RTP`
- `G723_RTP`

Whereas the video formats are known as

- `JPEG_RTP`
- `H261_RTP`
- `H263_RTP`

As their names imply, these formats use exactly the same compression schemes as their non-RTP versions. Thus a `JPEG_RTP` stream has been compressed with a JPEG codec. Construction of RTP-specific `Format` objects follows the same form as that for non-streaming media. For instance, use the following code to construct a `Format` object for RTP video data compressed with the H263 codec:

```
Format streamedVideoFormat = new Format(Format.H263_RTP);
```

Handling RTP Data Without RTP Classes

It is completely possible to play or process RTP originating or destined data without employing any of the classes found in `javax.media.rtp` or its two sub-packages. This is attributed to the versatility of the `Manager` and `MediaLocator` classes. Certain restrictions are inherent in this approach: In particular, only the first media stream in a session is available for processing or playing, and there is no means of monitoring the session itself.

Undoubtedly the simplest means of handling RTP data is through Sun's demonstration `JMStudio`. Although it doesn't provide a means for monitoring an RTP session (that requires coding as discussed in the following subsections), it is very simple to both play streaming media and to transmit it. The Open RTP Session option of the File menu allows the play of RTP transmitted data. The user simply enters the IP address and port to which the data is being sent. Similarly, the Transmit option of the File menu provides the user with a mechanism for transmitting either captured (from devices attached to the machine) audio, video, or media in a file over RTP. Thus, it is possible to carry out a video conference using the JMF, but without writing a line of code. Each user would run several instances of `JMStudio`

simultaneously on his machine: one instance to capture and transmit his audio and video and another two instances for playing the other participant's media—one for audio and one for video.

Alternatively, it is possible to write code for handling RTP data, but without using the RTP-related classes of the JMF. It is quite possible to create a `MediaLocator` object for an RTP stream. This can then be used with `Manager`'s various create methods, namely `createProcessor()`, `createPlayer()`, `createDataSource()`, and `createDataSink()` in order to obtain the appropriate object for handling the RTP data.

In these cases, the `MediaLocator` constructor is passed a `String` of the form:

```
"rtp://address:port[:ssrc]/content-type/[ttl]"
```

`address` is an IP address, `port` is an integer port number, and `content-type` is a string such as video or audio. The SSRC (Synchronizing Source) and TTL (Time to Live) fields are optional. By default, SSRC is the originator of the media, and TTL, being the maximum number of router hops the packets can experience before they are not propagated, is 1.

The resulting `MediaLocator` object (assuming that it is non-`null`) can then be used in the appropriate `create()` method of `Manager`. For instance, the following code fragment is part of the creation of a `Player` to handle video data being broadcast to a multicast session with the address `224.123.111.101` and using port `4044`:

```
try {
  MediaLocator rtpLocation = new
            MediaLocator("rtp://224.123.111.101:4044/video/");
  Player player = Manager.createPlayer(rtpLocation);
  Player.realize();
  :
```

Indeed, several of the example utility classes from the previous chapter can be used without alteration to transmit or play RTP data. The `PlayerOfMedia` GUI application is capable of playing streaming media just as it is capable of playing media from the local file system. In the dialog box provided, the user simply enters a suitable RTP locator string, such as the one found in the preceding example, and a player will be created for the media.

The `MediaStatistics` utility that reports on the format of a specified media can equally report on an RTP stream.

The `Location2Location` utility that takes media from one specified location, performs any prescribed transcoding, and then sinks the media to another specified location can be used to handle RTP data in a number of ways. The input location might specify an RTP stream; in which case, that received stream could be transcoded and then saved to a file, for instance. If the output location specifies an RTP stream, `Location2Location` acts as a transmitter, streaming media out using the specified address. If both input and output locations describe RTP streams, `Location2Location` acts as a kind of re-transmitter: receiving a stream, possibly performing some transcoding, and retransmitting that transcoded stream.

Even the `SimpleRecorder` application, which captures audio or video from devices (that is, microphones, Webcams, and so on) attached to the machine, could be modified in a quite straightforward manner so that it transmits the captured data as an RTP stream. Half of the capability already exists in that `SimpleRecorder` allows the user to specify the destination of the captured media with the `-f` flag. However, the program is currently hard-coded as to content type (Wave and AVI) and formats that it uses for audio (Linear) and video (Cinepak). If this was altered so that it supported the RTP content type and formats, `SimpleRecorder` could transmit its captured media in a format that it could be played by another application (for example, `PlayerOfMedia`).

Using the RTP Classes of JMF

Given the previous section's discussion of handling streaming RTP media without the RTP classes of the JMF, it might appear that the RTP classes are superfluous at best. As with other matters concerning the JMF, it is a matter of the level of control and sophistication of the required application. Playing, sending, and transcoding a stream are all possible without recourse to the RTP-related classes of the JMF. However, user control is limited to functionality within those spheres.

Among the abilities provided by the RTP-specific classes are the following:

- Managing sessions, participants, and media streams (for example, starting, finishing, adding, and so on)
- Monitoring (through listener interfaces) of RTP events (for example, new participants joining the session)
- Gathering and generating statistics (for example, quality of connection)

In particular, the first and second set of capabilities are highly desirable for applications such as a multiway video-conferencing application in which participants can arrive and leave at different times, employ different coding schemes, and have different qualities of connection to the other participants. Automating control for these scenarios almost requires an automated system that can monitor and respond to the dynamic events across the lifetime of the session.

The central class in an implementation handling RTP data, and explicitly acknowledging that fact in order to maximize control is the `RTPManager`. As shown in Figures 9.6 (playback of media stream) and 9.7 (transmission of captured video), an `RTPManager` object effectively acts as an intermediary or shield between the JMF objects handling the media (for example, `Player`, or `Processor`) and the protocol and session details.

Figure 9.6. `RTPManager` as intermediary when receiving streaming media.

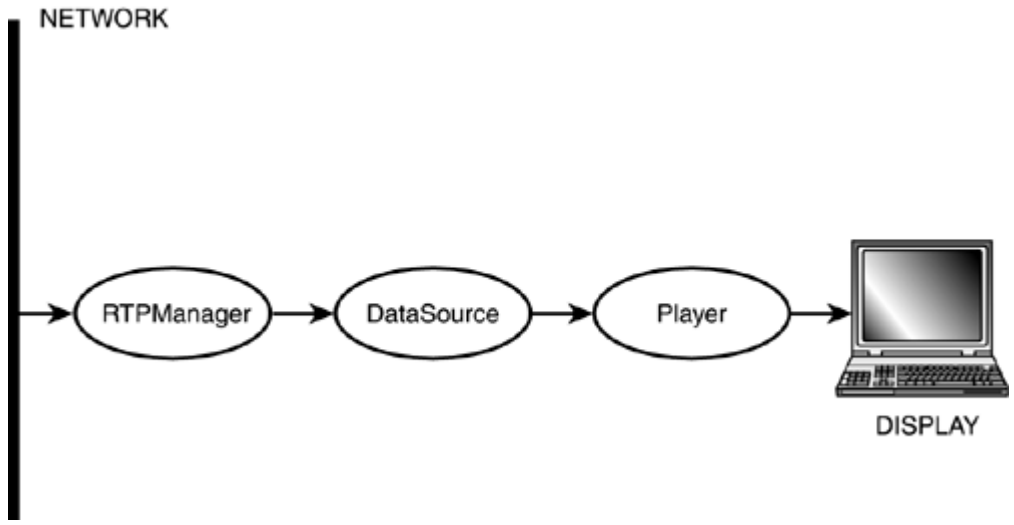
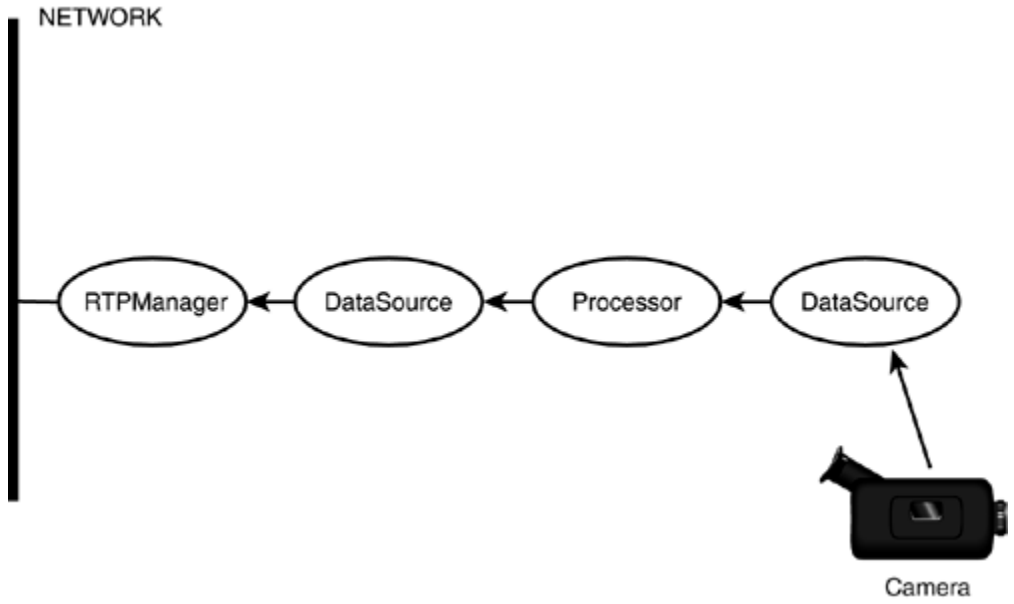


Figure 9.7. `RTPmanager` as intermediary in the transmission of captured media.



`RTPManager` is a new class as of JMF 2.1.1 and supersedes the depreciated `SessionManager` interface. Older examples can still be found that use `SessionManager` rather than `RTPManager`. `RTPManager` provides a uniform interface and method set regardless of whether a unicast, multi-unicast, or multicast session is being managed—something that `SessionManager` didn't provide.

However, `RTPManager` isn't the only class of relevance to handle RTP sessions. The following list summarizes the most important:

`InetAddress`— Java's representation of an internet address. Part of the java.net package. Needed to create session addresses.

`LocalParticipant`— The participant (listed next) that is also local to the machine. There is only one `LocalParticipant`: The rest are all remote.

`Participant`— An application sending or receiving streams within the session.

`ReceiveStream`— An interface representing a received stream of data within an RTP session. Each received stream in a session is represented by a separate `ReceiveStream` object.

`ReceiveStreamListener`— An interface that a class can implement in order to receive events associated with a `ReceiveStream` (such as timeouts, byes, a new stream, and so on).

`RemoteListener`— An interface that a class can implement in order to be informed about events generated by remote participants in a session (for example, the availability of sender or receiver reports or a name space collision).

`RemoteParticipant`— A participant in an RTP session that isn't on the same host (not local) as the `RTPManager`.

`RTPControl`— An interface allowing the control of an RTP `DataSource` object as well as a means of obtaining statistics about said `DataSource`.

`RTPManager`— A central class in managing an RTP session: an `RTPManager` object exists for each RTP session. Capable of creating sessions, streams, adding new participants, and so on.

`RTPStream`— Superclass of `SendStream` and `ReceiveStream`.

`SendStream`— An interface representing a sent (sending) stream of data within an RTP session. Each sending stream in a session is represented by a separate `SendStream` object.

`SendStreamListener`— An interface that a class can implement in order to receive events associated with a `SendStream` (such as timeouts, payload changes, byes, a new stream, and so on).

`SessionAddress`— The encapsulation of an RTP session's address as an `InetAddress` and associated port(s). These objects represent the unicast or multicast addresses associated with participants in the session.

`SessionListener`— An interface that a class can implement in order to be informed of session wide (that is, not specific to a particular stream) events such as name collisions or the addition of a new participant.

`RTPManager`

As previously stated, the `RTPManager` class plays the central role in the management of an RTP session. Unlike the key management classes of the core JMF—such as `Manager` or `CaptureDeviceManager`,

which are static classes—`RTPManager` is an abstract class. Instances of the class are created with the static `newInstance()` method, and each RTP session has its own associated `RTPManager` object. The following line of code shows the creation of an `RTPManager` object:

```
RTPManager sessionController = RTPManager.newInstance();
```

Managing an RTP session in which transmission is involved via an `RTPManager` object typically proceeds as follows:

1. Create an `RTPManager` object.
2. Initialize the `RTPManager` with the local host's address (or the multicast address if it is a multicast session).
3. For all targets to be sent to
   a. Create the target address as a `SessionAddress`.
   b. Add that `SessionAddress` as a target of the `RTPManager`.
4. Create a `SendStream` through the `RTPManager` for the `DataSource` to be output. `DataSource` must have appropriate format and content type—one supported for RTP.
5. Set up any listeners (for example, `SessionListener` or `SendStreamListener`).
6. Start the `SendStream`.
7. When the session is finished, perform the following:
   a. For each of the targets that were being sent to, remove them as targets of the `RTPManager` object.
   b. Dispose of the `RTPManager`.

Listing 9.1 and Listing 9.2 show the use of `RTPManager` to control an RTP session. In Listing 9.1, the video image captured from an appropriate device is multicast to two targets with the specified IP and port addresses. In Listing 9.2, a movie, stored as a file, has its video track multicast to a specified address. Neither are complete applications or classes, but they show the major steps in configuring the manager and starting the session.

Both listings share a number of common features that illustrate the basics of RTP session management. The listings differ chiefly in the configuration of the processor—one transmits H263 formatted video captured from a device, whereas the other transmits JPEG video transcoded from a file—and the setting of target addresses for the `RTPManager` object. For the unicast session, the `RTPManager` is initialized with the address of the host machine and has targets specified as the addresses of the machines (plus ports) to be transmitted to. On the other hand, for a multicast session, the `RTPManager` is both initialized with the multicast address and has its single target specified as that address.

Listing 9.1 An `RTPManager` Object Handles a Multi-Unicast Session—Broadcasting Video Using H263 Format

```
////////////////////////////////////////////////////////////////////////
// Need to capture video and output it as H263 RTP stream. Thus need a
// Processor, which needs a ProcessorModel that specifies ContentDescriptor,
// formats and datasource (in this case implicit to be a capture device that
// can supply the format).
////////////////////////////////////////////////////////////////////////
ContentDescriptor rtpContainer = new
                      ContentDescriptor(ContentDescriptor.RAW_RTP);
VideoFormat rtpH263 = new VideoFormat(VideoFormat.H263_RTP);
Format[] formats = {rtpH263};
ProcessorModel captureNTranscodeModel = new
                            ProcessorModel(formats, rtpContainer);
Processor captureNTranscodeProcessor =
                      Manager.createRealizedProcessor(captureNTranscodeModel);

////////////////////////////////////////////////////////////////////////
// Listen to the Processor and also obtain its output DataSource so it
// can be used to create a SendStream.
////////////////////////////////////////////////////////////////////////
captureNTranscodeProcessor.addControllerListener(this);
DataSource source = captureNTranscodeProcessor.getDataOutput();

////////////////////////////////////////////////////////////////////////
// Create the RTPManager to handle the session, then initialise it by
// providing the address of the local machine (with whatever port
// is available.
////////////////////////////////////////////////////////////////////////
RTPManager managerOfSession = RTPManager.newInstance();
SessionAddress hostAddress = new SessionAddress();
managerOfSession.initialise(hostAddress);

////////////////////////////////////////////////////////////////////////
// Create addresses for the two recipients and add them as targets for
//  the RTPManager. Note that these are arbitrary addresses and a user
// would substitute the known (IP) address of their recipient(s).
////////////////////////////////////////////////////////////////////////
InetAddress firstTargetIP = InetAddress.getByName("175.216.12.3");
SessionAddress firstTargetAddress = new SessionAddress(firstTargetIP,3000);
managerOfSession.addTarget(firstTargetAddress);

InetAddress secondTargetIP = InetAddress.getByName("131.236.21.177");
SessionAddress secondTargetAddress = new SessionAddress(secondTargetIP,3220);
managerOfSession.addTarget(secondTargetAddress);

////////////////////////////////////////////////////////////////////////
// Listen for all types of events that might occur in relation to this
// session. This would require that the class possess the
// appropriate listener methods (not found in this code fragment).
////////////////////////////////////////////////////////////////////////
managerOfSession.addSessionListener(this);
managerOfSession.addRemoteListener(this);
managerOfSession.addSendStreamListener(this);

////////////////////////////////////////////////////////////////////////
// Create and start the stream of video data. The stream is created from the
```

```
// Processor's output DataSource, with the 1 argument to createSendStream
// specifying that the first track (there should only be 1 for the DataSource
// anyway) be used as the stream.
//////////////////////////////////////////////////////////////////////
SendStream videoStream2Send = managerOfSession.createSendStream(source,1);
videoStream2Send.start();

//////////////////////////////////////////////////////////////////////
// When the transmission is over the targets should be removed (informed)
// and the resources acquired by the RTPManager released (via the
// dispose() call. Hence this fragment of code would be found in another
// portion of the class, such as in response to the user
// pressing a "Stop Transmission" button.
//////////////////////////////////////////////////////////////////////
managerOfSession.removeSessionListener(this);
managerOfSession.removeRemoteListener(this);
managerOfSession.removeSendStreamListener(this);
managerOfSession.removeTargets("Transmission Finished");
managerOfSession.dispose();
captureNTranscodeProcessor.stop();
captureNTranscodeProcessor.close();
```

Listing 9.2 An `RTPmanager` Object Handles a Multicast Session—Multicasting a Movie Track to a Particular Address

```
//////////////////////////////////////////////////////////////////////
// Need to transcode a file and transmit as JPEG RTP stream. Thus need a
//  Processor, which needs a ProcessorModel that specifies ContentDescriptor,
// formats and also the location of the media (in a file) to transmit.
//////////////////////////////////////////////////////////////////////
MediaLocator fileLocation = new
                     MediaLocator(file://D:\\jmf\\book\\media\\ex1.mov");
ContentDescriptor rtpContainer = new
                     ContentDescriptor(ContentDescriptor.RAW_RTP);
VideoFormat rtpJPEG = new VideoFormat(VideoFormat.JPEG_RTP);
Format[] formats = {rtpJPEG};
ProcessorModel transcodeModel =  new
                           ProcessorModel(fileLocation, formats, rtpContainer);
Processor transcodeProcessor =  Manager.createRealizedProcessor(transcodeModel);

//////////////////////////////////////////////////////////////////////
// Listen to the Processor and also obtain its output DataSource so it
// can be used to create a SendStream.
//////////////////////////////////////////////////////////////////////
transcodeProcessor.addControllerListener(this);
DataSource source = transcodeProcessor.getDataOutput();

//////////////////////////////////////////////////////////////////////////
// Create the RTPManager to handle the session. As it is a multicast session the
// multicast session address is used both as the target and to initialise the
// manager. In this case the IP address 224.123.109.101 with ports 4056 and 4057
// has arbitrarily been selected as the multicast session address.
//////////////////////////////////////////////////////////////////////////
RTPManager managerOfSession = RTPManager.newInstance();

InetAddress multicastIP = InetAddress.getByName("224.123.109.101");
```

```
SessionAddress multicastAddress = new SessionAddress(multicastIP,4056);
managerOfSession.initialize(multicastAddress);
managerOfSession.addTarget(multicastAddress);

////////////////////////////////////////////////////////////////////////
// Listen for all types of events that might occur in relation to this
// session. This would require that the class possess the appropriate listener
// methods (not found in this code fragment).
////////////////////////////////////////////////////////////////////////
managerOfSession.addSessionListener(this);
managerOfSession.addRemoteListener(this);
managerOfSession.addSendStreamListener(this);

////////////////////////////////////////////////////////////////////////
// Create and start the stream of video data. The stream is created from the
// Processor's output DataSource, with the 1 argument to createSendStream
// specifying that the first track (there should only be 1 for the DataSource
// anyway) be used as the stream.
////////////////////////////////////////////////////////////////////////
SendStream videoStream2Send = managerOfSession.createSendStream(source,1);
videoStream2Send.start();

////////////////////////////////////////////////////////////////////////
// When the transmission is over the target should be removed (informed) and the
// resources acquired by the RTPManager released (via the dispose() call. Hence
// this fragment of code would be found in another portion of the class, such as
// in response to a StopEvent from the Processor
////////////////////////////////////////////////////////////////////////
managerOfSession.removeSessionListener(this);
managerOfSession.removeRemoteListener(this);
managerOfSession.removeSendStreamListener(this);
managerOfSession.removeTarget(multicastAddress, "File Finished");
managerOfSession.dispose();
transcodeProcessor.stop();
transcodeProcessor.close();
```

Figure 9.8 shows the methods of `RTPManager`. As has already been stated and shown in Listings 9.1 and 9.2, `RTPManager` objects are created via the static `newInstance()` method.

Figure 9.8. The `RTPManager` class.

After an `RTPManager` object has been obtained and the appropriate pre-configuration performed (such as obtaining a multicast or unicast address as well as a `DataSource` object), the `RTPManager` should be initialized with the `initialize()` method. As its name implies, the method initializes the session. It can only be called once. It can throw either an `IOException` or an `InvalidSessionAddressException`. There are three versions of the method. The most commonly used version accepts a `SessionAddress`, which is the address of the local host and the associated data and control ports for the session. If a `null SessionAddress` is passed, a default local address will be chosen. If the `RTPManager` subsequently specifies a multicast session address as a target, the local address specified with `initialize()` is ignored. The multi-argument version of `initialize()` allows finer control in terms of the percentage of bandwidth consumed by RTCP traffic and even the encryption (if any) employed. The third version accepts an `RTPConnector` object, which is used when RTP isn't travelling over UDP.

The addTarget() method is used to specify the target of an RTP session. For transmission, this target is an IP address and port pair to be transmitted to. For receipt, this target is an IP address port pair to be listened to. The method is passed a SessionAddress object that specifies the IP address and port. The method can throw either an InvalidSessionAddressException or an IOException.

The addTarget() method effectively opens a session, causing RTCP reports to be generated as well as appropriate SessionEvents. The method should only be called after the associated RTPManager object has been initialized, and before the creation of any streams on a session.

Multi-unicast sessions— one host transmitting the same media to more than one recipient, where that transmission is specifically directed to each recipient—are supported by the mechanism of making multiple addTarget() calls. For instance, if there were four recipients, four addTarget() calls would be made; each one using a SessionAddress object that specified the receiving application's address.

Just as targets can be added to a session, they can be removed either individually with removeTarget() or en masse with removeTargets(). Typically, these methods are employed as an RTP session is being terminated. Although removeTarget() might also be used mid-session in a multi-unicast scenario to stop transmission to an address that is no longer participating or perhaps reachable. Both methods accept a String argument, which is the reason that the local participant has quit the session. This is transported via RTCP. The removeTarget() method has as its first argument a SessionAddress object that matches a current target of the session. The method might throw an InvalidSessionAddressException.

The dispose() method should be called at the end of all RTP sessions. It releases all resources that the RTPManager object has acquired during its existence and prepares the object for garbage collection.

RTPManager objects manage streams of data that fall into two categories: SendStreams for media transmission and ReceiveStreams for media receipt. SendStream objects are created with the createSendStream() method. ReceiveStream objects are created automatically as a new stream is received. They can be obtained through the NewReceiveStreamEvent (see the next subsection), or all current ReceiveStream objects can be obtained with the getReceiveStreams() method of RTPManager.

The createSendStream() method creates a new SendStream from an existing DataSource object (such as the output of a Processor). This is a necessary and vital step if data is to be sent in an RTP session. The method accepts two arguments—the DataSource and a track (or stream) index. The track index parameter specifies which track (stream) of the DataSource to use in creating the SendStream. The first track has an index of 1, the second track has an index of 2, and so on. Although an index of 0 that specifies an RTP mixer operation is desired, all tracks of the DataSource should be mixed as a single stream. The method can throw an UnsupportedFormatException or an IOException.

The getReceiveStreams() method returns a Vector, where each element of the Vector is a ReceiveStream that the RTPManager has created as the result of detecting a new source of RTP data. There is generally less call to use this method because the newly created ReceiveStream objects can be obtained through methods of the event that informs of their creation (see next subsection). Obtaining a ReceiveStream object allows its associated DataSource to be obtained and hence a Processor, Player, or DataSink created for that received media.

Listeners— `ReceiveStreamListener`, `RemoteListener`, `SendStreamListener`, and `SessionListener`—associated with the RTP session managed by the `RTPManager` object are added and removed through a set of add and remove methods of the `RTPManager` object. Listeners are vital in providing the monitoring and control capabilities of the RTP session. There are four methods for adding a listener: `addReceiveStreamListener()`, `addRemoteListener()`, `addSendStreamListener()`, and `addSessionListener()`. Listeners are usually added once an `RTPManager` object has been initialized. Correspondingly, there are four methods for removing listeners from an RTP session: `emoveReceiveStreamListener()`, `removeRemoteListener()`, `removeSendStreamListener()`, and `removeSessionListener()`. Listeners are usually removed at the end of an RTP session. RTP events and their associated listeners are discussed in greater detail in the following subsection.

The JMF represents participants in an RTP session by `Participant` objects—`LocalParticipant` and `RemoteParticipant`. An `RTPManager` object has several methods for determining the participants in the session it is managing. Those methods are

`getActiveParticipants()`— Those transmitting in the session

`getAllParticipants()`— All participants

`getLocalParticipant()`— The host participant who is also managing the session

`getPassiveParticipants()`— Those participating but not transmitting data

`getRemoteParticipants()`— All participants other than the local one

All methods except `getLocalParticipant()`, which returns a `LocalParticipant` object, return a `Vector` of `Participant` objects.

The final group of methods belonging to `RTPManager` pertain to session statistics. As their names indicate, the methods `getGlobalReceptionStats()` and `getGlobalTransmissionStats()` provide a means of obtaining transmission and reception statistics for the session. Session statistics are discussed further in a subsequent subsection.

RTP Events and Listeners

Four super classes of events, and their corresponding listeners, are defined with the JMF. They are

`SessionEvent`/`SessionListener`— Events that pertain to the session as a whole, such as a new Participant joining

`SendStreamEvent`/`SendStreamListener`— Changes in the streams that are being transmitted including a new stream, or a stream stopping

`ReceiveStreamEvent`/`ReceiveStreamListener`— Changes in the streams that are being received including a new stream, or a stream timing out

`RemoteEvent`/`RemoteListener`— Events that pertain to RTCP messages such as a new receiver or sender report being received

Each of the four events have subclasses that specialize in the information provided. For instance `SessionEvent` has two subclasses: `NewParticipantEvent`, and `LocalCollisionEvent`. All four events share the same parent class, `RTPEvent`, which is a subclass of `MediaEvent`. All events are found in the `javax.media.rtp.event` package.

RTP listeners— `SessionListener`, `SendStreamListener`, `ReceiveStreamListener`, and `RemoteListener`—are associated with an RTP session by means of the `RTPManager` object that is managing the session. The `RTPManager` class possesses four methods for adding and four methods for removing listeners—one for each type of listener. For instance, to add a `ReceiveStreamListener` for the session that the `RTPManager` object is managing, that object's `addReceiveStreamListener()` method is called.

All four listener interfaces— `SessionListener`, `SendStreamListener`, `ReceiveStreamListener`, and `RemoteListener`—define a single method `update()` that accepts an event of the type associated with the listener. That is, the `SessionListener` interface defines a single method `update(SessionEvent e)`, and so on for the other three with their events.

`SessionListener` objects receive two classes of events through their `update()` methods. The first is a `NewParticipantEvent`, indicating that a new participant has joined the session. The second one is a `LocalCollisionEvent`, indicating that the local host's SSRC has collided (is the same as) with that of another participant.

`SendStreamListener` objects receive five classes of events through their `update()` methods. A `NewSendStreamEvent` indicates that the local participant has just created a new `SendStream`. An `ActiveSendStreamEvent` indicates that data transfer has begun from the `DataSource` used to create the `SendStream`. An `InactiveSendStreamEvent` indicates that data transfer from the `DataSource` used to create the `SendStream` has stopped. A `LocalPayloadChangeEvent` indicates that the format of the `SendStream` has changed. A `StreamClosedEvent` indicates that the `SendStream` has closed. Listing 9.3 illustrates how an anonymous `SendStreamListener` class might terminate an RTP session when it detects that the stream being transmitted in the session is exhausted.

Listing 9.3 An Anonymous `SendStreamListener` Terminates an RTP Session After Transmitted Data Is Exhausted

```
DataSource source = processor.getDataOutput();
RTPManager managerOfSession = RTPManager.newInstance();
SessionAddress hostAddress = new SessionAddress();
managerOfSession.initialise(hostAddress);

managerOfSession.addTarget(target1);
SendStream stream2Send =  managerOfSession.createSendStream(source,1);
managerOfSession.addSendStreamListener(new SendStreamListener() {
    public void update(SendStreamEvent e) {
        if (e instanceof InactiveSendStreamEvent) {
           managerOfSession.removeSendStreamListener(this);
           managerOfSession.removeTarget(target1,"Data Source exhausted");
```

```
            processor.close();
            managerOfSession.dispose();
        }
    }});
stream2Send.start();
```

`ReceiveStreamListener` objects receive seven classes of events through their `update()` methods. A `NewReceiveStreamEvent` indicates that the `RTPManager` object has just created a new `ReceiveStream` object for a new data source. An `ActiveReceiveStreamEvent` indicates that data transfer has begun. An `InactiveReceiveStreamEvent` indicates that data transfer has stopped. A `TimeoutEvent` indicates that data transfer has timed out. A `RemotePlayloadChangeEvent` indicates that the format of a stream has changed. In a `StreamMappedEvent`, the originating participant is discovered for an existing stream with a previously unknown origin. An `ApplicationEvent` indicates that a special RTCP application specific packet has been received.

The `NewReceiveStreamEvent` is particularly important because this is the means by which new streams, transmitted by a remote participant, are discovered. The newly received `SendStream` can then be obtained with the `getReceiveStream()` method (inherited from `ReceiveStreamEvent`). `RTPStream`'s (the superclass of `ReceiveStream`) `getDataSource()` could then be used to obtain a `DataSource` object for the stream. With that `DataSource`, a `Player`, `Processor`, or `DataSink` object could then be created to handle the stream as desired. Listing 9.4 shows one way in which a newly received media stream might be handled through the creation of a `Player` object. The listing shows the use of an anonymous `ReceiveStreamListener` class that reacts to `NewReceiveStreamsEvents` by creating a realized `Player` object.

Listing 9.4 A `ReceiveStreamListener` Creates a New `Player` Object in Response to a New Stream Being Received

```
Player player;
SessionAddress destination = new SessionAddress(...);
RTPManager managerOfSession = RTPManager.newInstance();
managerOfSession.addReceiveStreamListener(new ReceiveStreamListener() {
  public void update(ReceiveStreamEvent e) {
    if (e instanceof NewReceiveStreamEvent) {
      ReceiveStream received = e.getReceiveStream();
      if (received==null)
        return;
      DataSource source = received.getDataSource();
      if (source==null)
        return;
      player = Manager.createRealizedPlayer(source);
     // etc. such as listening to the Player, getting itws Components, etc.
    }
}});
SessionAddress hostAddress = new SessionAddress();
managerOfSession.initialise(hostAddress);
managerOfSeesion.setTarget(destination);
```

`RemoteListener` objects receive three classes of events through their `update()` method. A `ReceiverReportEvent` indicates that a `ReceiverReport` RTCP packet has been received. A

SenderReportEvent indicates that a SenderReport RTCP packet has been received. A RemoteCollisionEvent indicates that two particpants' SSRC have collided (are the same).

RTP Streams

The JMF represents streams of data within an RTP session as RTPStream objects. RTPStream is an interface extended by two further interfaces—ReceiveStream for a stream of data being transmitted by and received from another participant, and SendStream for a stream that the current application (participant) is sending. Both types of RTPStream objects are associated with an RTP session and managed by an RTPManager object.

SendStream objects are created by an RTPManager object's createSendStream() method from a DataSource object. On the other hand, ReceiveStream objects are created automatically by an RTPManager object when a new stream of data is received by the manager.

SendStream objects are created by a broadcasting participant in an RTP session. After a SendStream has been created, it can be started with the start() method of the object. Starting s SendStream means that data will be transmitted over the network. Typically, transmitting programs then ignore the SendStream object until the transmission is completed—at which time, the object's close() method is called. Closing a SendStream frees all resources associated with that stream; hence, the method should always be called after the stream is no longer required. It is also possible to temporarily pause transmission of a stream by calling stop() on the SendStream object associated with that stream. This will also result in the DataSource that feeds the stream being stopped (via its stop() method). Hence data isn't lost simply because stop() was called.

ReceiveStream objects are created automatically by the RTPManager object managing a particular RTP session when a new stream of data is detected. User programs typically obtain ReceiveStream by calling the getReceiveStream() method of the NewReceiveStreamEvent that was posted to all ReceiveStreamListeners for the current session. Alternatively, RTPManager provides a method getReceiveStreams() that supplies all ReceiveStream objects being managed.

However, the ReceiveStream object is really an intermediary step in handling the received media. JMF media handlers such as Processors and Players, as well as other important objects such as DataSinks, all require a DataSource for their creation via the key Manager class. RTPStream has a method getDataSource() for obtaining the DataSource associated with a stream (whether ReceiveStream or SendStream). Hence, the standard approach when a new ReceiveStream is detected is to first obtain the ReceiveStream itself from the event and then use the stream to obtain the associated DataSource. That DataSource can then be employed to create the appropriate media handler or class. For instance, if the media was to be recorded, a DataSink would be created for that DataSource.

SessionAddress and InetAddress

RTP sessions are associated with one or, in the case of a multi-unicast session, multiple addresses. Those address(es) represent the participants within a session or the multicast address used for the session. For an RTP session, an address must consist of both an IP address and a port number. In fact, two are needed—one for data and one for control—but the control port defaults to be one greater than

the data port if it isn't supplied. The JMF employs the `SessionAddress` class to represent an RTP address, whereas the `InetAddress` class of java.net (core platform) represents an IP address.

An `InetAddress` object is Java's standard means of representing an IP address (that is, usually a machine on the Internet). Hence an `InetAddress` object is used as a stepping stone in the construction of a `SessionAddress` object.

The `InetAddress` class doesn't possess a constructor but rather three static methods from which an `InetAddress` object can be obtained. Those are `getLocalHost()`, which returns the `InetAddress` object for the machine on which the program is running; `getByName()`, which passes a `String` representing a machine, (for instance `"131.236.20.1"` returns an `InetAddress` object for that named); and `getAllByName()`, which also accepts a `String` as an address name and returns an array of valid `InetAddress` objects for that name. Because RTP sessions are initialized with the local machine's address via `RTPManager`'s `initialize()`, `InetAddress.getLocalHost()` is found commonly in JMF programs using RTP. Similarly, RTP broadcasting requires a specification of the address to broadcast to, so `InetAddress.getByName()` is also found commonly in JMF programs using RTP. Listing 9.5 shows both these methods being used.

`SessionAddress` objects are the JMF's means of specifying the addresses within an RTP session. In particular, `SessionAddress` objects are required to initialize an `RTPManager`, as well as set the (transmission) targets of that manager.

`SessionAddress` objects represent both an IP address and ports for data and control transmission. There are a number of constructors including no arguments (no functionality): an `InetAddress` and `int` data port (the most commonly used), two `InetAddress` and `int` port pairs (one for data and one for control), and one constructor that accepts an `InetAddress`, `int` data port, and `int` Time to Live. The most commonly used constructor for a `SessionAddress` object accepts an `InetAddress`, the data address, and an `int` port number—the port on which data is transmitted. The control port number defaults to one higher than the data port.

Listing 9.5 shows the initialization phase of an RTP session in which a stream of data will be sent from the local machine to that with the IP address `"145.201.33.9"` on port `3000`. It shows the construction of two `SessionAddress` objects—one to initialize the session and one as the target of the session.

Listing 9.5 Initialization Phase of an RTP Session

```
try {
  RTPManager managerOfSession = RTPManager.newInstance();
  managerOfSession.initialize(new
              SessionAddress(InetAddress.getLocalHost(),3000));
  managerOfSession.setTarget(new
           SessionAddress(InetAddress.getByName("145.201.33.9"),3000));
}
```

Participants

Participants within an RTP session—those receiving or transmitting streams of data—are represented by `Participant` objects within the JMF. The `RTPManager` object for a session keeps track of the participants within a session and provides methods for obtaining them:

Method	Participant Session Type
`getActiveParticipants()`	Transmitting
`getPassiveParticipants()`	Receiving only
`getLocalParticipant()`	Local
`getRemoteParticipants()`	Remote
`getAllParticipants()`	All participants

The `Participant` interface is extended by both `LocalParticipant` and `RemoteParticipant`. These subinterfaces are really placeholders though. They don't add any significant methods and, by their names, simply identify the participant types.

Knowing a participant in a session having a `Participant` object, it is possible to obtain `RTPStream` objects that represent all streams that the participant is transmitting. It is also possible to obtain all the most recent RTCP reports for that participant. These are the two most common uses of `Participant` objects—as a means to obtain the streams they are transmitting or as a means to obtain their most recent reports.

`Participant`'s `getStreams()` method returns a `Vector` of all streams that the participant is sending. If none are being sent by that participant, an empty `Vector` is returned. `Participant`'s `getReports()` method returns a `Vector` of `RTCPReport` objects. Those objects represent the most recent report for each stream the participant is sending or receiving.

Statistics

An important task when managing an RTP session is keeping track of the quality of the connection being experienced by all participants. Particularly clever management might, for instance, adjust the payload in response to changes in the network. If it becomes burdened, the frame rate or resolution might be dropped temporarily.

The RTCP provides a basic mechanism for this kind of monitoring through participants issuing periodic reports. In the JMF, RTCP reports are represented by the `Report` interface and its associated `Feedback` interface. `Report` objects are associated with a single participant and a stream they are transmitting or receiving. Session wide statistics are available as `GlobalReceptionStats` and `GlobalTransmissionStats` objects from the `RTPManager` object responsible for a session.

`Report` objects can be obtained for a `Participant` with the `getReports()` method, as well as from an `RTPStream` object with `getSenderReport()`. `Report` objects possess a `getFeedbackReports()`

method that returns a `Vector` of `Feedback` objects. `Feedback` objects provide methods for determining the number of packets lost, inter-arrival jitter, and other properties of the stream (see the JMF API for full details). The `Report` interface is subclassed into `SenderReport` and `ReceiverReport`. `ReceiverReport` is an empty interface, but `SenderReport` provides a number of methods for determining sender specific properties such as timestamps and byte counts.

Less specific but generally more useful are the global statistics provided by an `RTPManager` object for the session it is managing. These global statistics come in the form of `GlobalReceptionStats` and `GlobalTransmissionStats` objects and are obtained with the `getGlobalReceptionStats()` and `getGlobalTransmissionStats()` methods, respectively.

A `GlobalReceptionStats` object provides a number of methods for determining the reception quality for the entire session. These include the number of packets that failed to be transmitted, the number of bad packets received, and the number of local collisions. The JMF API provides a complete listing. A `GlobalTransmissionStats` object provides six methods for determining the transmission quality of an entire session. These include knowing the total number of bytes sent, number of failed transmissions, and number of local and remote collisions.

The manager of an RTP session can use these objects to maintain a profile of the session it is managing. Report objects are generated at regular intervals and have associated events so that it is easy to keep track of their arrival. However, the choice of what to do with the available statistics is still in the jurisdiction of the user's code. Simply present them to the local participant (for example, as part of their GUI) so that they are aware of the session state, or carry out dynamic adjustment of the session in response to the statistics.

Receiving and Transmitting Streams with `RTPManager`

As detailed in an earlier subsection, `RTPManager` plays the key central role in controlling an RTP session. It oversees the session in a number of ways: specifying the session address(es), creating streams for transmission, and providing a means for specifying listeners to the various events the session generates.

The earlier subsection on `RTPManager` provides the rough outlines of an algorithm when `RTPManager` is being used to transmit data. The corresponding generic and minimum algorithm for the reception (only) case is as follows:

1. Create an `RTPManager` object.

2. Initialize the `RTPManager` with the local host's address (or the multicast address if it is a multicast session).

3. For the target address on which transmissions will be received:

   a. Create the target address as a `SessionAddress`.

   b. Add that `SessionAddress` as a target of the `RTPManager`.

4. Set up any listeners (for example, `SessionListener` or `SendStreamListener`). You must have a `ReceiveStreamListener` to detect new streams created by other participants.

5. While the session isn't finished (however finished is defined), react to any receive stream events. For a new receive stream, perform the following:

   a. Get the `ReceiveStream`

   b. Get its `DataSource`

   c. Handle the stream (such as creating a `Player`, `Processor`, or `DataSink`, setting up GUI controls for it, adding appropriate listeners, and so on)

6. Dispose of the `RTPManager` object.

For the often used example of an audio-video conference in which two or more parties participate using a single multicast session address, the approach is a combination of both the receive and transmit cases:

1. Create an `RTPManager` object.

2. Create a `SessionAddress` to represent the multi-cast session address.

3. Initialize the `RTPManager` with the multicast `SessionAddress`.

4. Set the target of the `RTPManager` to be the multicast address also.

5. Set up any listeners (for example, `SessionListener` or `SendStreamListener`). You must have a `ReceiveStreamListener` to detect new streams created by other participants.

6. For all streams to send (for example, an audio and a video stream):

   a. Obtain the `DataSource` (with appropriate format and content type: one supported for RTP).

   b. Create a `SendStream` object (through the `RTPManager`) for the `DataSource`.

   c. Start that `SendStream`.

   d. Add a `SendStreamListener` (if appropriate).

7. While the session isn't finished (however finished is defined):

   a. React to any receive stream events. For a new receive stream, perform the following:

   Get the `ReceiveStream`.

   Get its `DataSource`.

Handle the stream (such as creating a `Player`, `Processor`, or `DataSink`, setting up GUI controls for it, adding appropriate listeners, and so on).

b. React to any send stream events. For an inactive send stream, close and dispose of the `SendStream`.

8. While the session isn't finished (however finished is defined), react to any receive stream events. For a new receive stream, perform the following:

   a. Get the `ReceiveStream`.

   b. Get its `DataSource`.

   c. Handle the stream (such as creating a `Player`, `Processor`, or `DataSink`, setting up GUI controls for it, adding appropriate listeners, and so on).

9. Dispose of the `RTPManager` object.

For both these algorithms, the `while` loops are a linear approximation of an event-driven program. The code doesn't poll for receive-stream or send-stream events, but simply adds itself as a listener for those events.

It is also feasible for a single program to employ several RTP sessions, and hence `RTPManager` objects, simultaneously. This isn't an uncommon technique used for managing multiple transmissions. For instance, video could be on one session, and audio could be on a separate session.

Sun's excellent `AVTransmit2` and `AVReceive2`, found on the JMF solutions site currently at http://java.sun.com/products/java-media/jmf/2.1.1/solutions follow such an approach. `AVTransmit2` is intended for transmitting on one or more RTP sessions, whereas `AVReceive2` is intended for receiving (where receipt entails the playing) of one or more streams on one or more sessions. Potentially, both employ multiple RTP sessions and hence multiple `RTPManager` objects. They are a good next step for those wanting to further understand RTP-based streaming. Indeed, not only can they be used out of the bag for AV conference, but they also serve as a good starting point for the reader wanting to implement his own specialized RTP-based application.

Cloning and Merging for Transmission

In two broadcast RTP scenarios, it is necessary to manipulate the `DataSource` that forms the basis of a `SendStream` before the `SendStream` object is created with the `RTPManager` object responsible for the RTP session.

In some multi-RTP session scenarios, the same media is being sent directly to different recipients as separate streams. For instance, a three-way AV transmission might be structured so that each pair of participants uses a different session—each participant is then transmitting the same data in two different sessions. In the JMF, a `DataSource` is single use. For instance, the same `DataSource` cannot be processed with a `Processor` and then played with a `Player`—although the output `DataSource` of the `Processor` could be played. This is true for `SendStream` creation also: A single `DataSource` can only be used to create a single `SendStream`. If multiple sessions are to employ the same `DataSource` to

create SendStream objects, the original DataSource must be transformed into a cloneable DataSource and clones created for each SendStream to be created. The following listing is an example of when a DataSource is being used in two different sessions that are managed by two RTPManagers: manager1 and manager2.

```
RTPManager manager1 = RTPManager.newInstance();
RTPManager manager2 = RTPManager.newInstance();
DataSource source = Manager.createDataSource("file://example.wav");
:        :        :
DataSource cloneable = Manager.createCloneableDataSource(source);
DataSource firstClone = cloneable.createClone();
SendStream firstStream = manager1.createSendStream(firstClone,1);
firstStream.start();
DataSource secondClone = cloneable.createClone();
SendStream secondStream = manager2.createSendStream(secondClone,1);
secondStream.start();
```

On the other hand, if data from separate sources (DataSource objects) is to be transmitted as a single stream, it is necessary to merge those DataSources into a single source before the associated SendStream is created. Also, when the SendStream object is created from the merged DataSource, the second parameter to createSendStream() should be the value of 0, indicating that all tracks (streams) of the DataSource should be mixed.

Buffering, Packet Size, and Jitter

Jitter is the phenomenon sometimes experienced when playing streaming media that isn't consistent in transmission rate (network delays) and reliability (packet loss). Jitter manifests as momentary pauses, drop outs, or jumps in the received (rendered) media.

The most useful technique in a receiver's arsenal is to use a buffer—a pool of data into which arriving media is added and from which media is taken to be rendered. The buffer acts to improve transmission inconsistencies, smoothing out differences in transmission rate. The larger a buffer, the greater the jitter that can be smoothed but also the longer the lag between receipt and rendering. For instance, a buffer size of 5 seconds implies that data currently being rendered was actually received 5 seconds ago. Large buffer sizes tend to adversely affect interactivity (for example, carrying out a conversation in which everything is delayed by a few extra seconds). Clearly there is a payoff or balance between smoothing jitter and loss of timely rendering of the data.

Another factor in the equation is the size of the packets transmitted. Larger packets use bandwidth more efficiently because less bandwidth is dedicated to packet headers. However the loss or damage of a large packet is more costly (and perhaps more likely) than that of a small packet in terms of the perceived quality of the media delivered. Losing a few milliseconds of speech might not even be noticed; on the other hand, one-half second lost could even affect comprehension of what was being said.

Both buffer size and packet size can be altered by JMF programs using RTP. There are few solid guidelines as to ideal values for each—they are both highly reliant on the particular scenario (bandwidth availability, network reliability, type of media being sent, and so on).

Receiving programs can alter the size of their buffer through the use of a `BufferControl` object. `BufferControl` is a `Control` interface and can be obtained for a `ReceiveStream` by first obtaining that `ReceiveStream` object's `DataSource` (using the `getDataSource()` method). The `DataSource`, which implements the `Controls` interface, can then be used to obtain the `BufferControl` object (using the `getControl()` method). `BufferControl` consists of six methods—the most important of which is `setBufferLength()`. This method accepts a single parameter of type long, being the length of the buffer in milliseconds.

Transmitting programs can alter the size of the packets they are transmitting if they can obtain a `PacketSizeControl` object for the transmission. `PacketSizeControl` is a `Control` object, but not all `Codecs` (`Processors`) expose `PacketSizeControl` objects through their `getControl()`/`getControls()` methods. The `PacketSizeControl` interface consists of two methods: `getPacketSize()` and `setPacketSize()`. The packet size is expressed as an `int` and is the maximum packet size output by the encoder.

Extending the JMF

Although the JMF provides support for an impressive number of formats (codecs), content types, and protocols, its coverage isn't complete. Although Sun's JMF team has pledged to support new open standard codecs and other similar advances in the area of time-based media, there will continue to be gaps between the coverage of the distributed JMF and the totality of time-based media. Several reasons for this difference are as follows:

Lag— Newer standards (for example, MPEG-4) can take some time to be implemented efficiency, tested fully, and brought into the JMF stable.

Proprietary— JMF is a free, open-standard with several complete implementations. The JMF couldn't be free and truly platform independent if it were encumbered by proprietary formats that have restrictions imposed on their use (for example, a fee per use of a particular codec).

Specialization— The JMF team at Sun has only limited resources, whereas the time-based media area is very large and continually advancing. Implementing more commonly used formats will clearly have higher priority than specialist niche formats, implying that specialist formats can take considerable time to appear.

Fortunately, the JMF is purposely built to be extended by users. Users can write their own codecs, multiplexers, demultiplexers, players, data sources, effects, and so on. These can then be seamlessly incorporated into the infrastructure that the JMF provides. They are then automatically (that is, through the central manager classes) available for subsequent usage.

Hence, for instance, an online community employing a particular form of time-based media might (over time) implement a `Codec`, `Multiplexer`, `Demultiplexer`, `Renderer`, and `DataSource` in support of that new format. These could then be seamlessly integrated into the JMF (and indeed distributed to the wider JMF-using community so that users also have automatic support for the new type of media), allowing full playing and processing of the media. Alternatively, a company might sell an AV

conferencing system that incorporates custom hardware for compression and decompression but who's software side is implemented in the JMF. By writing custom `Player` and `Processor` (or `PlugIn`) classes, the company can use the JMF infrastructure and paradigm of `DataSources`, `Players`, and `Processors`. However those custom classes would ensure that the system employs the hardware component—reaping the benefits of speed offered by the hardware. Yet another example is that a library of digital effects for both audio and video might be maintained on some community site. These effects would be implementations of the `Effect PlugIn`. Users wanting to employ an effect could simply download it from the repository and add it to their JMF installation on their machine. Those wanting to add a new effect (for example, motion blur) could implement a suitable `Effect` and place it on the site for all to use.

Conceptually, the task of extending the JMF is relatively simple, although the programming task might not be trivial (for example, adding support for a new video codec). The process consists of three steps:

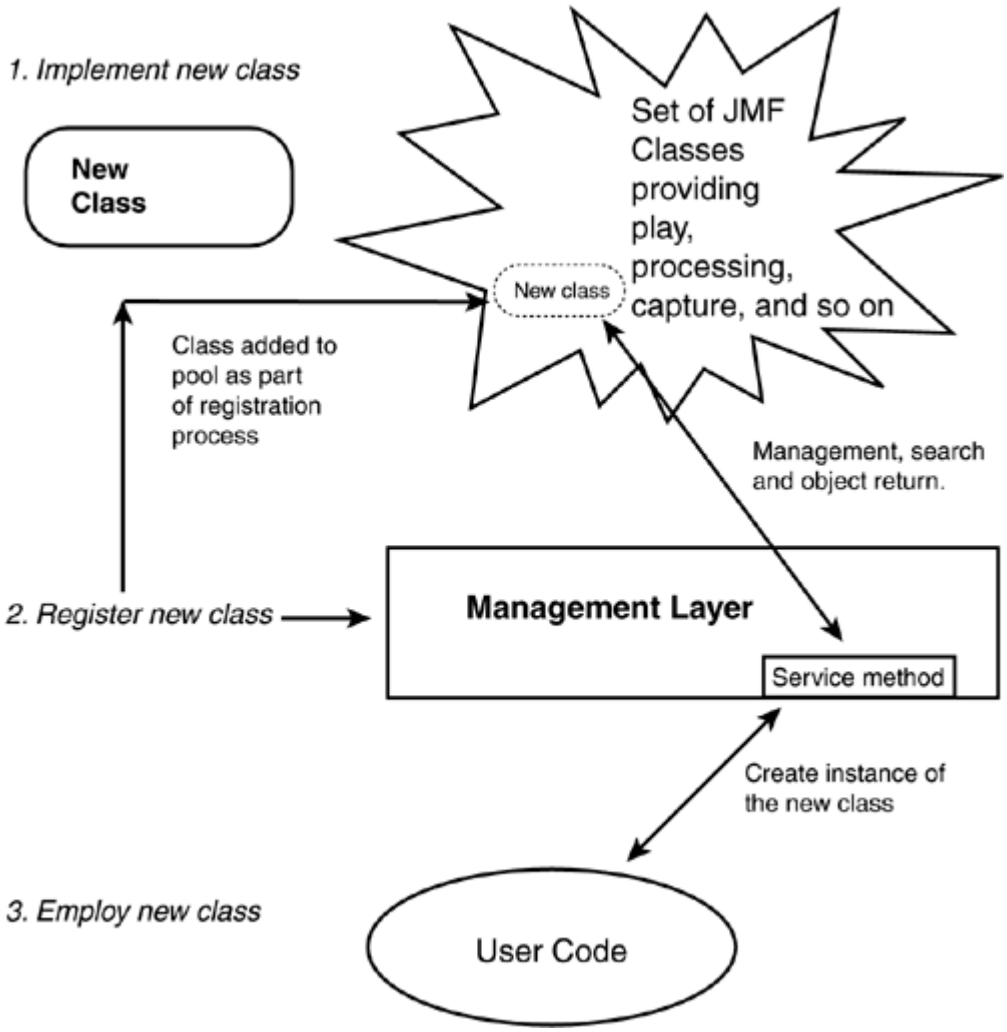
1. Implement the appropriate interface (for example, `DataSink` interface if writing a new `DataSink`).

2. Register the new class with the JMF.

3. Employ the new feature or support in the same manner as that for the prepackaged features (for example, create an instance of the new `DataSink` through the central `Manager` class).

By registering the new classes with the JMF, they effectively become added to the pool of resources that the manager classes (chiefly `Manager`) oversee. The manager is then aware of the new class (and the features it provides) and can create an instance of that class as needed. Figure 9.9 shows the three-step process. A correctly registered class is subsequently available through the `Manager` class for any application.

Figure 9.9. Registration of a new class that extends the JMF.



Before specifics are discussed, it is worth mentioning that extending the JMF isn't a trivial undertaking. The complexity does vary depending on the task at hand. However, a good understanding of the lower-level structures of the JMF (for example, `Buffer`, `Track`, `Format` and so on) is a virtual prerequisite for any such task. Those without such an understanding will no doubt acquire it if their extensions are successful.

Role of `Interfaces`

With the exception of `DataSource`, all the key infrastructure functional classes of the JMF employed repeatedly in user programs—`Player`, `Processor`, `DataSink`, and all the `PlugIns` (`Multiplexer`, `Demultiplexer`, `Codec`, `Effect`, and `Renderer`)—are all interfaces. Extending the JMF by adding another type of `Player` (for instance) is then a case of writing a class that implements the `Player` interface.

Implementing an interface is a case of writing a class that possesses all the methods that the interface lists. All methods must be present and must possess exactly the same signature as that in the interface. The signature of a method is its name, return type, visibility, argument numbers, type, and ordering.

No inheritance is involved in implementing an interface, so all code must be written by the user—there are no default versions for each method. On the other hand, the user class is free to extend another class (class, not interface) and thus inherit any methods and attributes it provides.

For instance an implementation of the `Codec` interface would have to possess 11 methods—five directly from the `Codec` interface along with four from the `PlugIn` interface that `Codec` extends, and two from the `Controls` interface that `Codec` also extends. Listing 9.6 shows an empty class that implements the `Codec` interface. It possesses all the necessary methods, but they are all stubs (empty methods that perform no useful task). As such, it is the bare skeleton of a real `Codec` implementation—it currently does nothing.

Listing 9.6 An Empty Class that Implements the `Codec` Interface

```
import javax.media.*;

/**************************************************
* A stub or empty class with no functionality but which
* shows the bare minimum methods a class must possess in
* order to extend the Codec interface (to be a Codec).
*
* @author Spike Barlow
**************************************************/
public class EmptyCodec implements Codec {

////////////////////////////
// 5 methods of Codec itself
////////////////////////////
public Format[] getSupportedInputFormats() { return null; }

public Format[] getSupportedOutputFormats(Format in) { return null; }

public int process(Buffer in, Buffer out) {
  return PlugIn.INPUT_BUFFER_NOT_CONSUMED;
}

public Format setInputFormat(Format in) { return null; }

public Format setOutputFormat(Format out) { return null; }

//////////////////////////////////////////
// The Controls interface that Codec extends
//////////////////////////////////////////
public Object[] getControls() { return null; }

public Object getControl(String name) { return null; }

//////////////////////////////////////
// The PlugIn interface that Codec extends
//////////////////////////////////////
public void close() { }
```

```
public String getName() { return "EmptyCodec"; }

public void open() { }

public void reset() { }
}
```

Two minor catches exist for the unwary in implementing an interface. The first is that many interfaces are tiered, extending one or more other interfaces. If such an interface is being implemented, not only must its own methods be provided (written), but also all methods of interfaces that the interface in question extends. For instance, the `Codec` interface extends both `Controls` and `PlugIn` interfaces. Thus any class that implements `Codec` must possess at least 11 methods: five of `Codec` itself, four of `PlugIn`, and two of `Controls`. Second, the visibility of methods within the current version of the JMF API documentation often appear as though they are a package (that is, the method has no visibility modifier), whereas they are in fact public methods. Fixing these cases are simple however because the compiler will raise an error indicating that the interface specifies public visibility.

Registering the New Classes

For a new class such as a new `Player` to be automatically available for subsequent use, it must be registered with the JMF. In effect this registration process can be thought of as adding the new class to the database of classes that the JMF maintains. After that addition is made, the JMF is aware of the class. From then on, JMF will create instances of the class as appropriate (in response to user requirements).

Several of the manager classes have a role in registration of new classes:

`PlugInManager`— New `PlugIns` (`Codecs`, `Renderers`, and so on) are added (registered) via the `PlugInManager`. They can also be removed via this class.

`PackageManager`— New `MediaHandlers`—`Players`, `Processors`, `DataSource`, and `DataSink` classes—are registered via this class.

`CaptureDeviceManager`— Newly attached capture devices are registered with the JMF via this class.

Registering a new class is a somewhat finicky process involving precise naming rules for the class (if a `MediaHandler`). It is also an uncommon process. Rather than writing code to register a new class, which is possible by making calls to the appropriate manager class, a new class can be registered interactively with the `JMFRegistry` program.

`JMFRegistry` is a GUI application that comes as part of the standard JMF distribution. Using menus, buttons, and text fields, it allows the user to interactively add or delete `PlugIns`, `CaptureDevices`, and packages to or from the database that the local implementation of the JMF maintains. Using `JMFRegistry` is simple (provided the user comprehends the JMF rules for naming) and is strongly recommended as an easier means for registering a new class. Sun maintains a short help document for `JMFRegistry`. It can be found at http://java.sun.com/products/java-media/jmf/2.1.1/jmfregistry/jmfregistry.html. Running `JMFRegistry` is as simple as `java JMFRegistry`.

The JMF employs naming rules as its means of keeping track of `MediaHandler` (`Player`, `Processor`, and `DataSink`) and `DataSource` classes, as well as knowing what they do. Classes are organized into packages, and the package name together with the classname indicates what the class does. The JMF employs these rules for all `MediaHandler` and `DataSink` classes that come as part of the JMF. If the same naming rules aren't followed for new `MediaHandler` classes, the `Manager` class will be unable to find the new class. Hence it will remain unavailable to the user.

These naming rules don't apply for `PlugIns`. Each `PlugIn` is added separately as a fully qualified classname.

However, for all `DataHandler` and `DataSource` classes, strict naming rules apply. All `DataHandler` classes (whether `Player`, `Processor`, or `DataSink`) must be called (has a classname of) `Handler`, whereas all `DataSources` must be called (have a classname of) `DataSource`. This apparent confusion stemming from a profusion of `Handler` and `DataSource` classes is resolved by each of them existing in their own package. Hence the fully qualified name of the class uniquely differentiates it from all other classes.

Package names are split into several portions—an initial user assigned name followed by a fixed portion that reflects the type of class it is (`Processor`, `Player`, `DataSink`, and so on), followed by a name that reflects the protocol or content-type that the class supports. That is followed by the classname.

In particular,

- `Players` are named as `<content package-prefix>.media.content.<content-type>.Handler`
- `Processors` are named as `<content package-prefix>.media.processor.<content-type>.Handler`
- `DataSinks` are named as `<content package-prefix>.media.datasink.<protocol>.Handler`
- `DataSources` are named as `<protocol package-prefix>.media.protocol.<protocol>.DataSource`

The names content package prefix and protocol package prefix are Sun's terms for the user assigned prefix. The JMF provides for two categories of prefixes: one for `MediaHandlers` (content package prefix) and one for `DataSources` (protocol package prefix). For instance the prefixes that are part of the JMF 2.1.1 (Windows Performance) distribution are `javax`, `com.sun`, and `com.ibm`.

So, for instance, `com.sun.media.protocol.rtp.DataSource` is the name of the DataSource class provided by Sun, as part of the JMF, for RTP DataSources. A user writing her own processor for handling a new content type known as, for example, `cs9` and having selected a package prefix of `au.edu.adfa`, for instance, would name the class `au.edu.adfa.media.processor.cs9.Handler.java`.

Implementing `PlugIns`

Implementing a `PlugIn` is generally easier than writing a `DataSource` or `DataHandler`. Not only is the task generally smaller (which isn't always so), it doesn't require the strict class and package naming needed for `DataSources` and `DataHandlers`.

The discussion under the previous section concerning interfaces and Listing 9.6 provides an example of how implementing a `Codec` might be started. Similar approaches apply for `Demultiplexers`, `Effects`, `Multiplexers`, and `Renderers`.

Registering a new `PlugIn` with the JMF (either using `JMFRegistry` or `PlugInManager` directly) means that the `PlugIn` will henceforth be available to default `Processors` or those created with a `ProcessorModel` object.

It is possible to use a nonregistered `PlugIn` by creating an instance of the class directly and using the `Processor` object's `TrackControl` objects (one per track composing the media) to specify that the `PlugIn` be employed. This might, for instance, be used as a means of testing a `PlugIn` under development. However, in general, registering a `PlugIn` is far preferable.

Implementing `DataHandlers`

Implementing a new `DataHandler`—a `Player`, `Processor`, or `DataSink`—requires writing a class named `Handler` that is part of a larger package and which implements the particular interface (`Player`, `Processor`, or `DataSink`).

As for `PlugIn` implementation, implementing a `DataHandler` requires writing a class that possesses all the methods listed for that interface, as well as for all the interfaces it extends. This is a nontrivial task—a `DataSink` must possess at least 12 methods, a `Player` must have 32 methods, and a `Processor` must have 38. The complexity of `Player` stems from the fact that it extends `Controller`, which extends `Clock`. `Processor` extends `Player` and hence has an additional six methods.

As discussed in the previous section on registering new classes, a new `MediaHandler` class must be called `Handler` and exist in a package with a particular name structure. The exact naming rules are found previously, whereas the following code fragment shows the start of a `Player` class that handles a hypothetical content-type known as `4XXXX`.

```
package com.samspublishing.mediaaips.media.content.4XXXX
import javax.media.*;
public class Handler implements Player {
:        :        :
}
```

Note that the package to which the class belongs begins with the top-level name. In this case, the hypothetical `com.samspublishing.mediaapis`, has the mandatory `media.content` as the mid-portion of the name, indicating it is a player, and `4XXXX` as the suffix, indicating the particular content-type that it handles.

It is worth noting that individual `Handlers` aren't registered with the JMF, but simply the top-level, user package name. After that is registered, all subsequently implemented `MediaHandlers` in the same top-level package will be automatically found by the `Manager` class when it is asked to create a new object.

Extending `DataSource`

The `DataSource` class is an exception in terms of extending the JMF. Whereas `PlugIns` and `DataHandlers` are interfaces and require one pattern for implementation, `DataSource` is a class and requires a different approach.

Writing a new `DataSource` means extending the existing `DataSource` class or one of its subclasses. When contrasted with implementing an interface, this can be an advantage because the default inherited behavior for some methods might not need to be altered. The following code fragment shows the start of a `DataSource` class that deals with a new protocol known as `sdtb`, which is a type of pull data source.

```
package com.samspublishing.mediaapis.media.protocol.sdtb
import javax.media.*;
import javax.media.protocol.*;
public class DataSource extends PullBufferDataSource {
:        :        :
}
```

As for `MediaHandlers`, the top-level, user-supplied, package name must be registered with the JMF for the new `DataSource` to be accessible to the `Manager` class. This is most easily done through the `JMFRegistry` application. The JMF differentiates user-package names into those for `DataHandlers`, which it names content package prefix, and those for `DataSources`, which it calls protocol package prefix. Registration of each package prefix is separate even, as is the usual case, if the names are the same.

Sun's Examples

As mentioned previously, extending the JMF is a non-trivial task and often involves writing a relatively large class. This section doesn't possess a complete example because of space and complexity restrictions.

However, Sun has provided a number of excellent examples of writing `PlugIns`, `MediaHandlers`, and `DataSources` in both its JMF guide http://java.sun.com/products/java-media/jmf/2.1.1/guide/JMFTOC.html and on its solutions Web page http://java.sun.com/products/java-media/jmf/2.1.1/solutions/index.html. These are very good starting points for those wanting to extend the JMF and go beyond the information provided here. They also provide a good gauge for the complexity of the task.

In particular, the current version of the JMF guide contains a `Demultiplexer PlugIn` implementation for GSM, a gain control, a `Renderer` for RGB (employing `AWT's Image` class), an ftp `DataSource`, and a `Controller` for a hypothetical type of data known as TimeLine.

The solutions page contains a particularly clever implementation of a `DataSource` that provides screen capture facilities. By defining a new protocol called `screen`, the `DataSource` can be used to capture a particular region of the computer screen as a video. The implementation is highly recommended not only as an example of what can be done with the JMF, but also is a particularly useful utility—one that can be installed within the JMF in minutes.

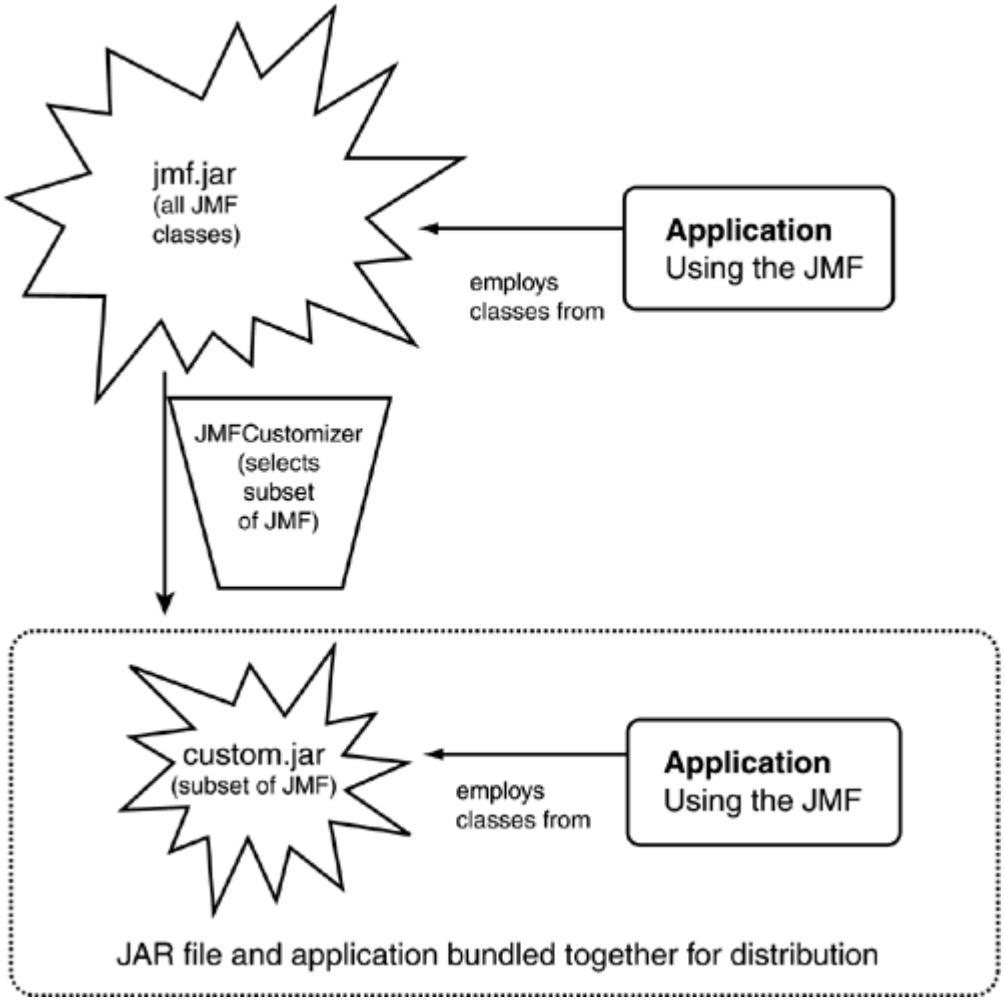
`JMFCustomizer`

The JMF is a large, optional API that extends the functionality of the Java platform. For users to run JMF-based programs, they must download and install the JMF API on their own machines. That can be problematic for some users who don't feel confident to complete such tasks. This can limit both the distribution and appeal of JMF-based programs—many users are happy to run a prepackaged application (or applet), but will balk at the prospect of having to download and install some software first.

The `JMFCustomizer` application has been provided by Sun to help JMF developers address this issue. `JMFCustomizer` is a simple application that allows the user to select a subset of the classes that compose the JMF. Those classes are then encapsulated as a JAR file, which can be distributed with the application (or applet).

The intention of this approach is that the user doesn't need to have the JMF installed on his machine. The developer of the application selects the subset of the JMF that is required by the application and distributes that (as a JAR file) along with the application itself. The user then has all that he requires in a single distribution. Figure 9.10 shows such a usage of `JMFCustomizer`.

Figure 9.10. Use of `JMFCustomizer` to create a JAR file containing the subset of the JMF necessary for a particular application.



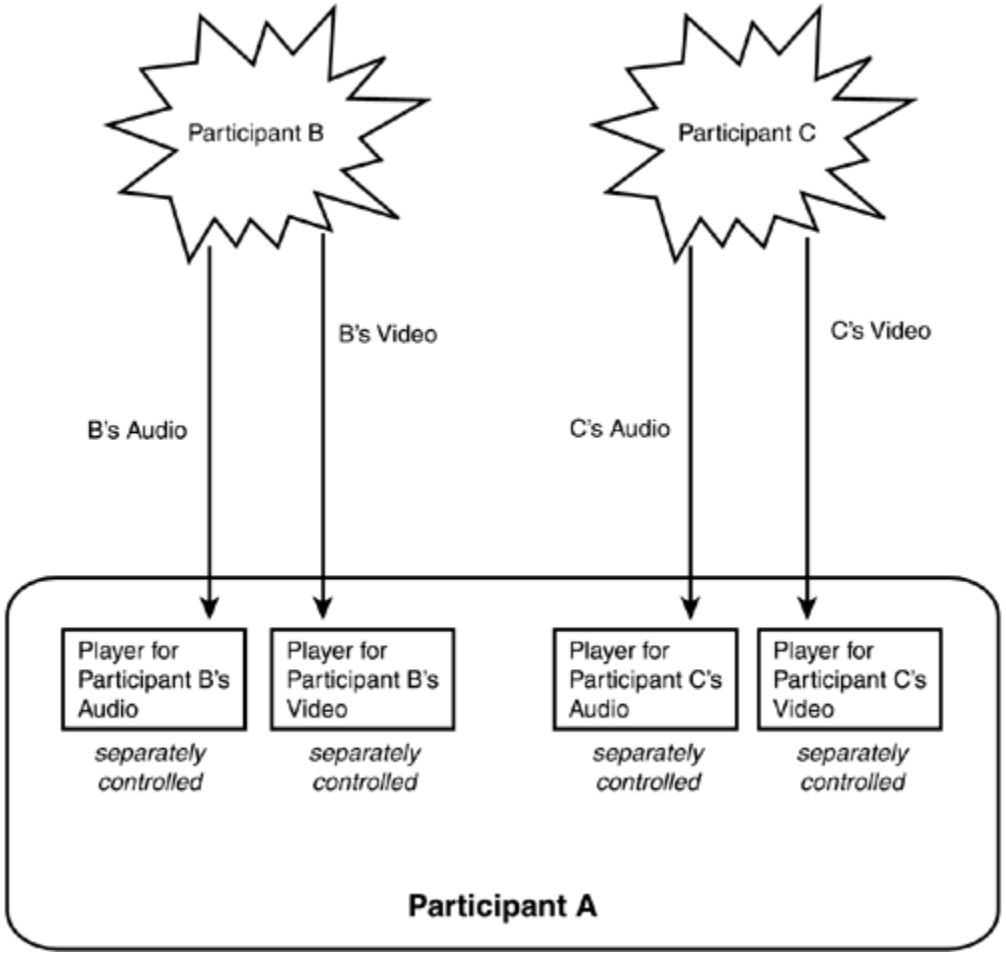
Running `JMFCustomizer` is as simple as `java JMFCustomizer`. After that GUI components such as check boxes and buttons lead the user through the selection of appropriate classes and the creation of the JAR file.

However, although the `JMFCustomizer` application is part of the standard JMF 2.1.1 distribution, it isn't (by default) in the classpath. As such the developer wanting to employ `JMFCustomizer` must modify the classpath to include the customizer.jar file (that is found in the same folder and directory as jmf.jar). The particulars of setting the classpath variable depend on the operating system in question. Sun's "Setting Up and Running JMF on a Java Client" at http://java.sun.com/products/java-media/jmf/2.1.1/setup-java.html describes the simple steps for all operating systems.

Synchronization

In some circumstances, it is necessary to have a number of individual `Players` (or perhaps `Processors`) as part of a single application. An example of such a situation might be an AV conference over RTP, particularly if there are multiple participants. Each participant is likely to send separate audio and video streams. Hence, recipients require a number of `Players`: one for each stream. Imagine such a three-way conference; each participant requires four `Players`, one audio player, and one video player for each of the two participants from which they are receiving. Figure 9.11 shows such a situation. Each player is controlled individually.

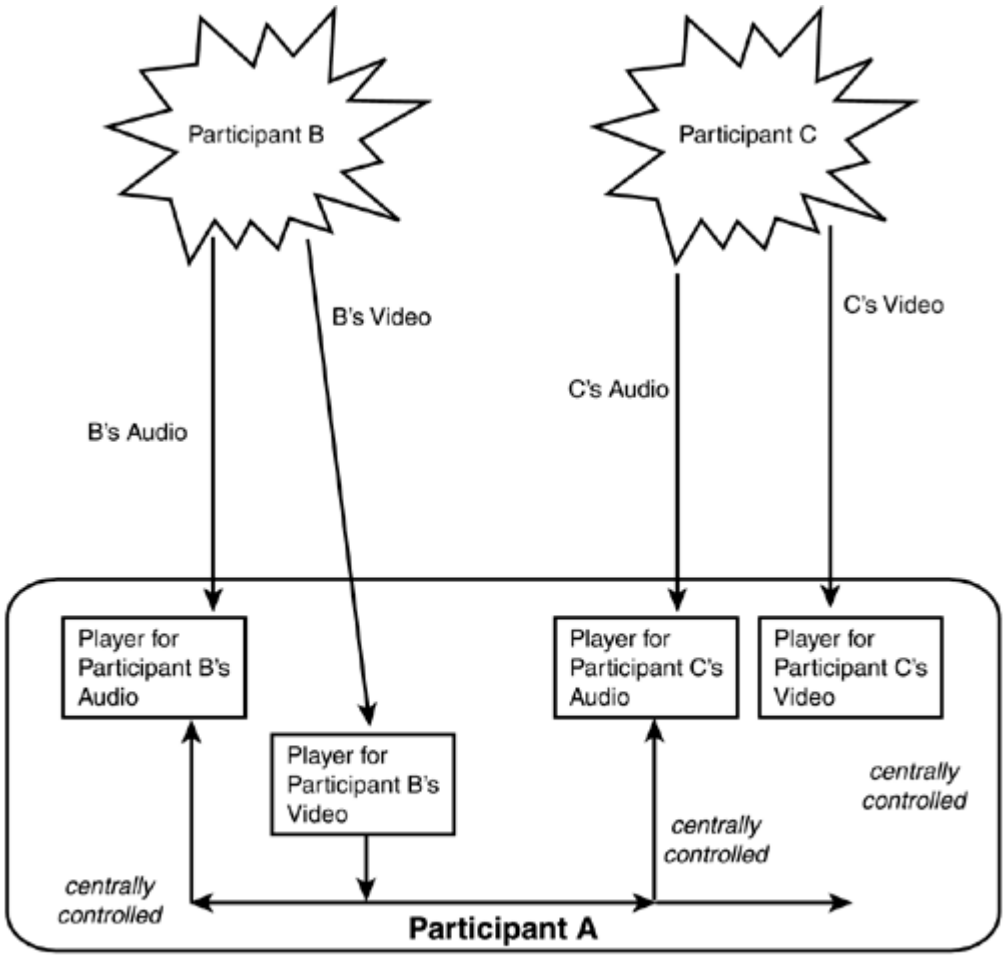
Figure 9.11. A three-way AV conference from the perspective of one of the participants showing the players needed in order to render the media being streamed to it.



Controlling all those players individually not only is tedious, but also often doesn't match the desired solution. Generally it is easier (both as a user and programmer) to centralize the control. Actions on the one centralized controller (`Player`) should then automatically propagate to all those it oversees. For instance, selecting Stop should stop all players. The JMF provides such a feature through the `Player` class.

A `Player` (`Processors` are also a type of `Player`) object is capable of controlling one or more additional `Controllers` (`Players` or `Processors`). Actions (methods) on the central `Player` are also propagated to all `Controllers` that the `Player` controls. For instance, invoking `prefetch()` on the central `Player` would cause `prefetch()` to be called on all `Controllers` that the `Player` controls. Figure 9.12 shows the previous scenario of a three-way AV conference from the perspective of one of the participants. However in this case, rather than controlling each `Player` separately, the `Player` for participant B's video is also the central control for all `Players`. Any actions carried out on that `Player` (such as stopping it) also affect the other players.

Figure 9.12. In this case, control of the players is centralized through one of their numbers.



A `Player` object is given control over another `Controller` with `Player's addController()` method. Similarly you can remove a `Controller` object from a `Player` with the `removeController()` method. The following code fragment shows two `Players` being constructed, and `player1` is given control over `player2`. They are both brought to the realized state by invoking `realize()` on the central Player, which is `player1`.

```
Player player1 = Manager.createPlayer(...);
Player player2 = Manager.createPlayer(...);
player1.addController(player2);
player1.realize();
```

Several features of the synchronized control are worth noting:

- The added controller assumes the centralized `Player` object's `TimeBase`.
- The `Player` object's duration is the maximum of its own duration and that of all `Controller` objects under its direction.
- The start latency of the `Player` is the maximum of the `Player's` own start latency and that of all `Controller` objects under its direction. This ensures that all `Controllers` will start simultaneously using the `syncStart()` method.
- The `Player` object only posts completion events (for example, `PrefetchCompleteEvent`) when all managed `Controllers` have also posted the event.
- All `Controller` methods invoked on the central `Player` object are propagated to all `Controllers` under its direction.
- Although `Controllers` are under the central direction of a `Player`, they shouldn't have their methods invoked individually. All method invocation should occur by way of the central `Player` object.

The JMF in Conjunction with Other APIs

The JMF is a powerful API, supporting a high-level and uniform approach to controlling, playing, and processing time-based media. An equally important JMF strength is that it is part of the larger Java platform. The implication of this is that the JMF approach to time-based media can be combined with other features of Java; either from the core platform, or one or more of the other specialist APIs that extend the functionality of Java. This leads to programs that can combine time-based media with other media (for example, 3D graphics) in either traditional multimedia paradigms, or in new and innovative approaches that are only possible because of the common platform-independent glue that is Java.

Consider an educational application dealing with Australian Aboriginal culture and civilization in the region of the Olgas (central Australian geographic formation) across the last 50,000 years. The application might combine a library of still images of the region, maps, hypertext about the languages and customs of the region, 3D interactive models of reconstructed camp sites, and audio and video interviews with tribal elders of today, perhaps telling some of the Dreamtime legends unique to the area. Such an application could be written entirely in Java, using features of the core platform, the JMF, and Java 3D (perhaps with other APIs also), and would run on any platform.

What about an application of tomorrow? A virtual or immersive audio-video conference with colleagues overseas who are speaking a different language unknown to the recipient. Such an application would use the JMF in conjunction with features of Java 3D, the JAI, and other APIs such as Java Speech. Not only would the audio and video be streamed between participants and projected into a virtual environment (for example, a model of a new building that is being jointly designed), but the audio stream would be extracted, processed by a recognizer (for the speaker's language); then translated

into the hearer's language and reintegrated with the video as subtitles or as a separate synthesized audio track. Again, Java provides the framework to enable such a future application.

Applications that involve the JMF in conjunction with other APIs fall into two broad categories. The simpler form, such as the preceding archaeological/cultural multimedia application, uses the JMF as a plug-in component of the entire application. The JMF portion (playing the interviews) is a logical, self-contained component of the entire application. Such applications, although often large, are relatively modular and don't hinge on low-level interfacing of different APIs. The more difficult applications, such as the virtual AV conference involving language translation, aren't so modular in their composition. They tend to rely on the fusing of APIs at a lower level of detail. In the multi-language AV conference application, the media streams must be demultiplexed and passed off to other APIs (the speech recognition and translation) that don't directly support the JMF data models. This requires a knowledge of the deeper data structures in the communicating APIs and the means of converting between them. For the JMF side, that might involve strong familiarity with the `Buffer`, `Format`, `Stream`, `Clock`, and related classes and interfaces.

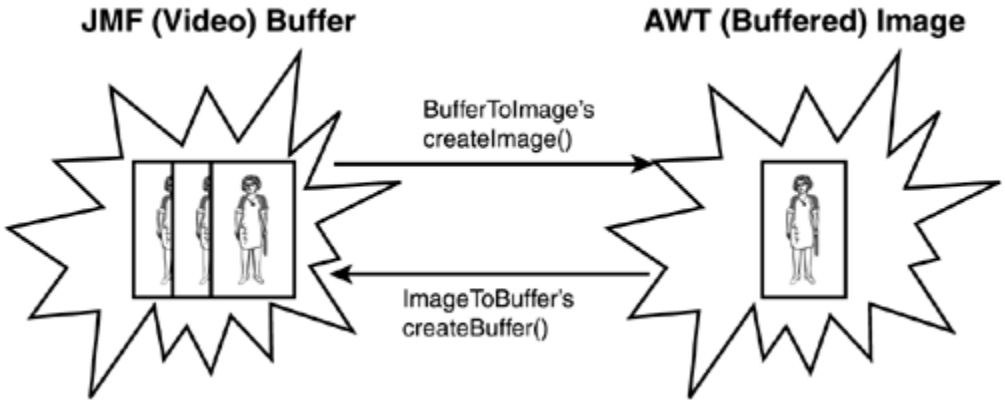
Chapter 14, "Integrating Across the Java Media API," is explicitly concerned with combining the media APIs discussed in this book. As a prelude to that chapter, the following subsection discusses the key JMF classes that allow a video stream to be treated as a sequence of images by other APIs.

`ImageToBuffer` and `BufferToImage`

Two classes, `ImageToBuffer` and `BufferToImage`, play central roles as the lynchpins between the video component of the JMF and 2D or 3D graphics. For instance, with these classes, it is possible to grab individual images from a video, insert frames into a video, or even extract, modify, and reinsert them. Indeed, it is possible to write a video `Renderer` by using `BufferToImage`—simply pull out each frame and render that image (using Java's Graphics class). `BufferToImage` is one mechanism by which JMF originated video can be imported into other contexts. For instance, a JMF video could be applied as a texture in a Java 3D world by texturing the individual images that compose the video. Conversely, a video can be constructed as a sequence of (AWT) images.

`ImageToBuffer` and `BufferToImage` are the sole members of the `javax.media.util` package. As shown in Figure 9.13, the `BufferToImage` class provides the ability to convert from a JMF (video) `Buffer` to an `AWT Image` (`BufferedImage`), while `ImageToBuffer` provides the reverse functionality.

Figure 9.13. The roles of `BufferToImage` and `ImageToBuffer` classes in moving between a JMF video frame and an AWT Image.



The `BufferToImage` class possesses a single constructor, accepting a `Format` object. That `Format` object specifies the format of the video buffer that must be converted. Hence, `BufferToImage` objects are specific to a particular `Format`, but can convert any `Buffer` in that `Format` into its equivalent `Image`. The class has a single method `createImage()` that accepts a `Buffer` (of the `Format` specified in the constructor) and returns an `AWT Image`. `Null` is returned if the conversion cannot be done.

Obtaining an `Image` from a video sequence devolves into obtaining a `Buffer` object that corresponds to the video frame in question. `BufferToImage`'s `createImage()` can then be applied to the `Buffer` to generate the `Image`. Obtaining a `Buffer` for a particular video frame can be achieved through use the `FrameGrabbingControl` interface. A `FrameGrabbingControl` can be exported by a `Player` or `Renderer` through the `getControl()` method. The interface possesses a single method `grabFrame()` that returns a `Buffer` object that corresponds to the current frame from the video stream. In addition to `FrameGrabbingControl`, there is also the useful `FramePositioningControl` that can be employed to precisely position a stream at a particular frame number. As shown in Listing 9.7, these can be used in conjunction so that a particular (known) frame number can be grabbed and turned into an image. In this case, the 500th frame is used.

Listing 9.7 A Particular Video Frame Is Grabbed and Transformed into an `AWT Image`

[View full width]

```
int    desiredFrame = 500;
Player    player = Manager.createRealizedPlayer(new MediaLocator(...));
FramePositioningControl positioner =
           player.getControl("javax.media.control.FramePositioningControl");
if (positioner==null) return;
FrameGrabbingControl grabber =
           Player.getControl("javax.media.control.FrameGrabbingControl");
if (grabber==null)  return;
player.prefetch();
// Some sort of pause/polling/event listening to ensure its prefetched

positioner.seek(desiredFrame);

// Assumption here is that Player object doesn't "regress" into an earlier
```

```
// "less prepared" state. More generally should wait till Player returns to // //
prefetched.
Buffer inTheBuff = grabber.grabFrame();
Format videoFormat = inTheBuff.getFormat();
BufferToImage converter = new BufferToImage(videoFormat);
Image captured = converter.createImage(inTheBuff);

// Now do something useful with captured, the image that corresponds to the
// desired (500th) frame.
```

Going in the opposite direction—from an `AWT Image` to a JMF `Buffer`—employs the `ImageToBuffer` class. The class possesses a single static method that accepts an `AWT Image` and desired frame rate and returns a JMF `Buffer` with an RGB `Format`. Although that operation is simple, construction of a stream from the `Buffers` is somewhat more difficult. Sun's example Screen Grabber [`DataSource`] linked from their JMF solutions page, http://java.sun.com/products/java-media/jmf/2.1.1/solutions/index.html, is a good example of building a stream from individual images.

Java Sound

Java Sound (`javax.sound`) is a core API of the Java platform, meaning that it is part of all Java runtime environments that support Java 1.3 and later. The Java Sound API is low-level, concerning itself with the input and output plus processing of both sampled audio and MIDI (Musical Instrument Digital Interface) data.

The Java Sounds API provides the lowest-level of sound support on the Java platform, which incorporates explicit control over the resources concerned with sound input and output. Among the abilities provided by the API are direct access to system resources such as MIDI synthesizers, audio mixers, audio and MIDI devices, converters between sound formats, and file-based I/O.

The capabilities provided by Java Sound partially overlap with those of the JMF, although their target audiences are different. The JMF is larger, encompassing video and sound, and also high-level, providing a unified architecture for handling, processing, and transporting time-based media. On the other hand, Java Sound is both more specialized and lower level. It is concerned only with audio, but provides far finer control over the audio parameters of a system. In particular, Java Sound's MIDI functionality is considerably more sophisticated than that of the JMF (which has limited playback only as of 2.1.1), as well as providing the ability to control aspects such as buffering and mixing.

Java Sound is a large API. The programmer's guide from Sun can currently be found at http://java.sun.com/products/java-media/sound/index.html, and it runs more than 150 pages. If it has moved, do a search on the main Java Web page http://java.sun.com. This short section serves to inform you of the existence of the API. A number of audio-only applications are better suited to Java Sound than the JMF, and those developing a time-based application that is audio only should contrast the features of both APIs as to which is most suitable for the application.

Java Sound divides its MIDI and sampled audio support into two separate packages, with an additional two packages for service providers:

`javax.sound.sampled`— Classes for playing, capturing, and processing (mixing) of digital (sampled) audio

`javax.sound.midi`— Classes for MIDI synthesis, sequencing, and event transport

`javax.sound.sampled.spi`— Service provider's package for sampled audio

`javax.sound.midi.spi`— Service provider's package for MIDI sound

Implementations of the Java Sound API provide a basic set of audio services. The Service Provider Interface (SPI) packages are provided as a means for third-party software developers to develop new services. These new services then become another aspect of Java Sound.

The Java Sound API employs a similar approach to management of the services it provides to that of the JMF. Two central management classes act as registries or database managers for the audio components and audio resources on the system. Those management classes are `AudioSystem` for sampled audio and `MidiSystem` for MIDI resources. These classes act as access points for obtaining the services provided by Java Sound. For instance, the `AudioSystem` class provides a means of obtaining mixers, lines, format converters, and I/O functionality for sampled audio data.

The reader wanting to know more about Java Sound should consult Sun's "Java Sound API Programmer's Guide," as well as the API documentation. Both can be found at the Java Sound documentation page: http://java.sun.com/j2se/1.3/docs/guide/sound/.

Future Directions for the JMF

The JMF is a powerful and rapidly maturing API that provides a high-level and uniform structure for the handling of time-based media. All key structures and classes for capturing, playing, processing, receiving, and transmitting time-based media already exist in the API. Currently the API is undergoing a solidification phase—developers are exploring and adopting the API as suitable to a range of potential applications. Simultaneously, Sun is continuing its strong support for the API with the addition of further features (for example, new codecs and formats) and enhancements in the short term with specific goals for the longer term.

The next significant release of the JMF, 2.2, is expected shortly (perhaps around the time this book is released). JMF 2.2 is expected to incorporate a new RTP implementation, together with significant optimizations and fixes.

The JMF team at Sun has set supporting MPEG-2 and MPEG-4, the two most frequently requested codecs, as its highest priority. This support will hopefully be appearing in the next version of the JMF, but licensing and patent issues are likely to be the real decider of exactly when the support appears. In this regard, Sun has promised not only to continue to optimize and update the JMF, but also to add support for open-standard, industry-leading codecs.

The maturity, stability, and size of the JMF mean that it is unlikely to ever be incorporated into the core Java platform. However a closer integration with other optional packages (for example, Java 3D) is

likely. In that sense, standards such as MPEG-4 might well act as drivers in that direction, stressing the coding of audio-visual objects and composite media that incorporates interactivity—for instance, combining 2D imagery and 3D synthetic objects with audio and video streams. Indeed Sun has stated that it believes MPEG-4 to be an important standard—one for which it will provide increased functionality in the future.

As an example of the directions that the JMF and related developments are headed, Sun is working closely with Nokia and other international telecommunication companies (including Motorola, Mitsubishi, Siemens, and NTT) on a multimedia API for J2ME (Java 2 Micro Edition). J2ME is the small footprint (less demanding of memory and processor power) version of the Java platform suitable for the newer generation of mobile devices including phones, pagers, digital set-top boxes, car navigation systems, and personal digital assistants.

The J2ME Multimedia API is being designed under the Java Community Process as JSR (Java Specification Request) 135. The publicly available documentation on the JSR can be found at http://www.jcp.org/jsr/detail/135.prt. The intention of the API, as described in the JSR documentation, is to provide a high-level interface to sound and multimedia capabilities on a device running J2ME, which would thus enable versatile and scalable multimedia applications on these devices. The package's proposed name is `javax.microedition.media`. Although its primary focus is sound, it also is intended to incorporate the control of other time-based multimedia formats. Both the JMF and Java Sound are listed as starting points for the new API. This development, and the future of platform-independent handling of media through the JMF and its derivatives, is put in context by the fact that Nokia (the filers of the JRS) plans to ship 50 million Java-enabled phones by the end of 2002 and 100 million by the end of 2003.

Summary

This chapter is the last of three covering the handling of time-based media with the JMF. It has covered more advanced topics in time-based media handling, including the streaming of media and extending the JMF. Chapter 8 served to cover the core functionality of the JMF, whereas Chapter 7 introduced time-based media and the JMF.

More than half of the chapter is dedicated to RTP—the Real-time Transport Protocol and its integration into the JMF to support the streaming of audio and video. The fundamentals of RTP and streaming data are introduced before the particular classes involved in managing an RTP session are discussed.

The other major topic of the chapter is extending the JMF. Details of writing a new `DataSource`, `Player`, `Processor`, or `DataSink` are covered as well as the means of registering the new class so that it is available for subsequent use in any JMF-based program.

The chapter concludes with a miscellany of topics including synchronization of multiple players, interfacing the JMF to other APIs, the Java Sound API, and finally a glimpse at some of the future paths for the JMF.