

# 29

## MULTIMEDIA SUPPORT IN JAVA

Joseph Celi<sup>1</sup> and Borko Furht<sup>2</sup>

<sup>1</sup>*Internet Development Group  
IBM, Boca Raton, Florida*

<sup>2</sup>*Department of Computer Science and Engineering  
Florida Atlantic University, Boca Raton, Florida 33431*

1.	<b>INTRODUCTION</b> .....	606
2.	<b>ANIMATION SUPPORT IN JAVA</b> .....	607
2.1	FRAME-BASED ANIMATION .....	607
2.2	SPRITE-BASED ANIMATION .....	614
3.	<b>AUDIO SUPPORT IN JAVA</b> .....	620
3.1	PLAYING AN AUDIO CLIP FROM A JAVA APPLLET .....	620
3.2	PLAYING AN AUDIO CLIP USING A MORE SOPHISTICATED APPROACH.....	620
3.3	PLAYING AND STOPPING AN AUDIO CLIP .....	622
4.	<b>VIDEO SUPPORT IN JAVA</b> .....	622
4.1	STANDARD VIDEO .....	622
4.2	STREAMING VIDEO .....	623
5.	<b>JAVA MEDIA API</b> .....	624
6.	<b>CONCLUSION</b> .....	625
	<b>REFERENCES</b> .....	626

**Abstract.** In this chapter, we discuss multimedia support in Java. The object-oriented paradigm, applied in Java, allows software developers to build upon the available class libraries in order to create new custom class libraries, thus extending the language. We present Java support for animation and audio, and we also create working Java applets, which can accomplish various animation and audio tasks. The current Java class libraries do not provide support for MIDI, streaming audio, or any form of video. However, a new Java Media Application Programming Interface (API), which is briefly introduced in the chapter, defines a set of multimedia classes to support interactive media, including audio and video.

## 1. INTRODUCTION

Java is an exciting new technology that has hit the computer industry by storm. This technology, developed by Sun Microsystems, is gaining widespread acceptance at an unprecedented pace. What makes the Java programming language stand out from the others?

To answer this and other questions about Java, one must first understand exactly what Java is. Java is an object oriented programming language that allows developers to create platform independent applications [1]. The language borrows its syntax and style from C++ but does not carry the extra baggage of having to be backward compatible with a predecessor language. Java is a clear, concise language that adheres closely to the object-oriented design principles of information hiding, encapsulation, data abstraction, inheritance, and polymorphism. These features promote code reuse, which is a highly desirable feature for most software development shops. Java provides support for exceptions, which aid programs to become more reliable since they are better equipped to handle error conditions.

Java is an interpreted language requiring a Java interpreter to be present on the machine running the Java program. The Java interpreter is often called the Java Virtual Machine or JVM for short. The Java compiler compiles a Java program into byte codes that reside in a .CLASS file. The .CLASS file is fed to the JVM for execution. Figure 1 depicts the areas described in this section.

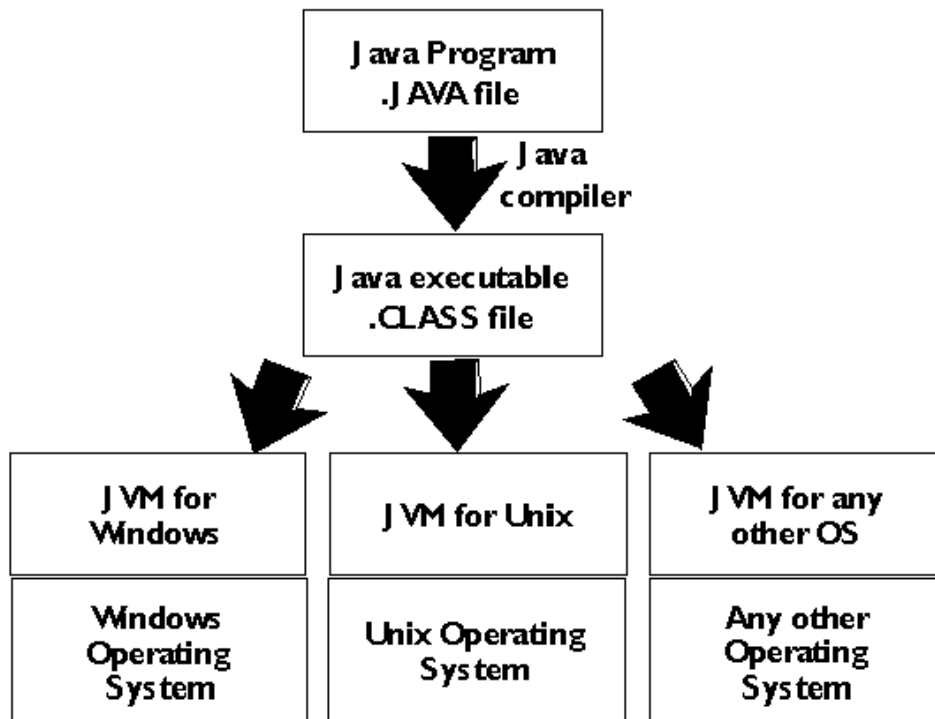


Figure 1. Java environment.

The Java program is machine independent. The machine dependencies are contained within the JVM [2]. Currently there are JVMs written for all of the popular operating systems. Java applications can be run under any operating system that supplies a JVM.

Java applets are similar to Java applications in that they define an executable entity. Java applets are typically located on a Web server and run remotely on a client machine. For security reasons, Java applets have limited access to the API at run time. The built-in security features of Java make it a very attractive solution for distributed programming. Most Web browsers available today support Java applets.

The portable nature of the Java language is the key element driving the industry toward this new technology [3]. The “write once, run many” philosophy that Java presents has attracted many programming organizations to this new and exciting language.

## 2. ANIMATION SUPPORT IN JAVA

Displaying a sequence of pre-drawn images one after another creates an animation. Each image or frame in the animation is carefully drawn with respect to a previous frame to simulate movement. If the system rendering the animation is capable of displaying the images at a rate of approximately 15 frames per second, the transitions from frame to frame will seem fluid, hence simulating real movement.

The Java application programming interface (API) provides support for animation by allowing users to load and manipulate images using classes and methods are included in the Java Development Kit (JDK). Java distributes the task of retrieving, decoding, and rendering images over several classes and packages. The Applet, Image, Graphics, and Toolkit classes all play a part in producing the finished product.

Other classes provide additional support. Network support is provided by the URL and URL Connection classes. The MediaTracker class and the ImageObserver interface provide a means to monitor the image loading progress. These capabilities are integrated into the Java runtime system and perform their respective roles in the background. The following sections concentrate on the direct support classes and the methods that can be used to build image animations.

### 2.1 FRAME-BASED ANIMATION

Frame-based animation, as described above, is an animation sequence, which is produced by displaying a set of frames in some predefined sequence. In the simplest case, the images are all of the same size so that each frame simply overlays the previous one. The key issues to be tackled here are performance and picture clarity. The following example demonstrates the techniques used to ensure the best possible frame-based animations.

The *FrameAnimation* example loads three images and displays them in sequence to produce a simple animation. The animation frames are displayed in Figure 2.

The frames are displayed in the following order {1, 2, 3, 2, 1}. This frame sequence, when played over in a continuous manner gives the illusion of the famous “let your fingers do the walking” animation that is often used in commercials used to advertise the yellow pages telephone book.

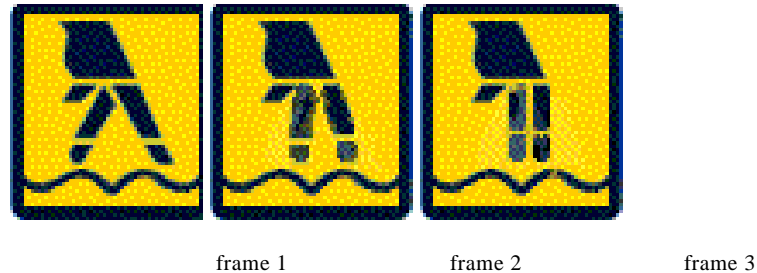


Figure 2. Frames uses for animation sequence.

The *FrameAnimation* example is in the form of a Java applet, which executes within a Java enabled browser or within the applet viewer utility. The two most popular Internet browsers in use today, from Netscape and Microsoft, both support Java applets. The applet viewer utility comes as part of the JDK from Sun. The code for this applet is shown in Figure 3.

The *FrameAnimation* applet loads the three images from its *init* method. The browser or applet viewer informs the applet that it has been loaded into the system and calls the *init* method of an applet. It is always called before the first time the *start* method is called.

The *getImage* method returns an *Image* object that can then be painted on the screen. This method requires the caller to specify a URL pointing to the location of the image. The image name is also required to be provided as a parameter. This method will always return immediately even if the image cannot be located. For this reason one should always check for a valid return from the *getImage* method. The Java libraries will load the actual image data, asynchronously, under the execution of another thread. If an attempt is made to draw an image before all of the data has been transmitted, only the portion of the image that has been loaded at that point in time will be displayed.

In order to reduce the initial image flickering caused by partially loaded images, the *FrameAnimation* applet prepares each image that it uses for the animation. The *prepareImage* method, contained in the *Component* class, will return *false* if the image is not fully prepared for rendering. Prepared, in this context, means that all of the bits of the image have been transmitted across the Internet and the image has been scaled to the height and width specified. In this example, no dimension parameters were provided to the *prepareImage* method so the image will not be scaled.

Once all of the images in the animation sequence have been prepared, the program is ready to start displaying the animation. To create animation, a series of images or frames are displayed in some logical order. A pause or sleep between the rendering of consecutive images is necessary to provide the illusion of movement. The amount of time to sleep between frames must be carefully chosen. If the time interval is too short, the motion will be too fast.

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

public class FrameAnimation extends Applet implements Runnable
{
    Thread    runner;
    Image     img1, img2, img3, curImg;
    boolean   fOkToPaint = false;

    public void init()
    {
        System.out.println("In init method");

        img1 = getImage(getDocumentBase(), "fingers1.gif");
        if(img1 == null)
            System.out.println("getImage failed to obtain fingers1.gif");

        img2 = getImage(getDocumentBase(), "fingers2.gif");
        if(img2 == null)
            System.out.println("getImage failed to obtain fingers2.gif");

        img3 = getImage(getDocumentBase(), "fingers3.gif");
        if(img3 == null)
            System.out.println("getImage failed to obtain fingers3.gif");

        while(!prepareImage(img1, this))
        {
            // System.out.println("Sleeping while preparing img1");
            mySleep(10);
        }
        // System.out.println("Finished preparing img1");
        // System.out.println("img1 = " + img1);
        curImg = img1;

        while(!prepareImage(img2, this))
        {
            // System.out.println("Sleeping while preparing img2");
            mySleep(10);
        }

        // System.out.println("Finished preparing img2");
        // System.out.println("img2 = " + img2);

        while(!prepareImage(img3, this))
        {
            // System.out.println("Sleeping while preparing img3");
            mySleep(10);
        }
        // System.out.println("Finished preparing img3");
        // System.out.println("img3 = " + img3);
        fOkToPaint = true;
    }

    public void start()
    {
        System.out.println("In start method");
        if(runner == null)
```

```

    {
        runner = new Thread(this);
        runner.start();
    }
}

```

Figure 3. Java applet for frame animation.

```

public void stop()
{
    System.out.println("In stop method");
    if(runner != null)
    {
        runner.stop();
        runner = null;
    }
}

public void run()
{
    System.out.println("In run method");
    while(true)
    {
        if(curImg == img1)
            curImg = img2;
        else if(curImg == img2)
            curImg = img3;
        else
            curImg = img1;
        repaint();
        mySleep(100);
    }
}

//
// We will override the update method in order to
// get rid of the flicker. The default update
// method will always fill in the area with the
// background color and then call the paint
// method.
//
public void update(Graphics g)
{
    paint(g);
}
public void paint(Graphics g)
{
    // System.out.println("In paint method");
    if(fOkToPaint)
        g.drawImage(curImg, 0, 0, this);
}

public void mySleep(long millis)
{
    try
    {
        Thread.sleep(millis);
    }
}

```

```

        catch(InterruptedException e) {}
    }
}

```

Figure 3 (cont.) Java applet for frame animation.

The motion could also appear jerky if there is not sufficient time for the eye to keep up with the frames. If the time interval chosen is too long, the output will look more like a slide show than animation. The sample animation applet uses a 100-millisecond sleep time between frames.

**2.1.1 Using Multiple Threads of Execution**

The FrameAnimation applet utilizes the multi-threading capabilities of Java to display a continuous animation. If Java did not include support for multiple threads then applets like the FrameAnimation example would have to build their own multi-threading logic to allow other tasks to run while the continuous animation was being played. The multiple thread support in Java allows programmers the ability to build sophisticated programs and games that can accept user input while performing some other complex task [4].

Overall, using multiple threads often reduces program execution time. This is depicted in Figure 4. Referring to the figure, three systems are displayed: a single threaded system, a multi-threaded system running under a single CPU, and a multi-threaded system utilizing multiple CPUs. On the single threaded system, a task runs to completion before the next task is executed. If a task performs an expensive I/O operation, the CPU will remain idle until the operation has completed.

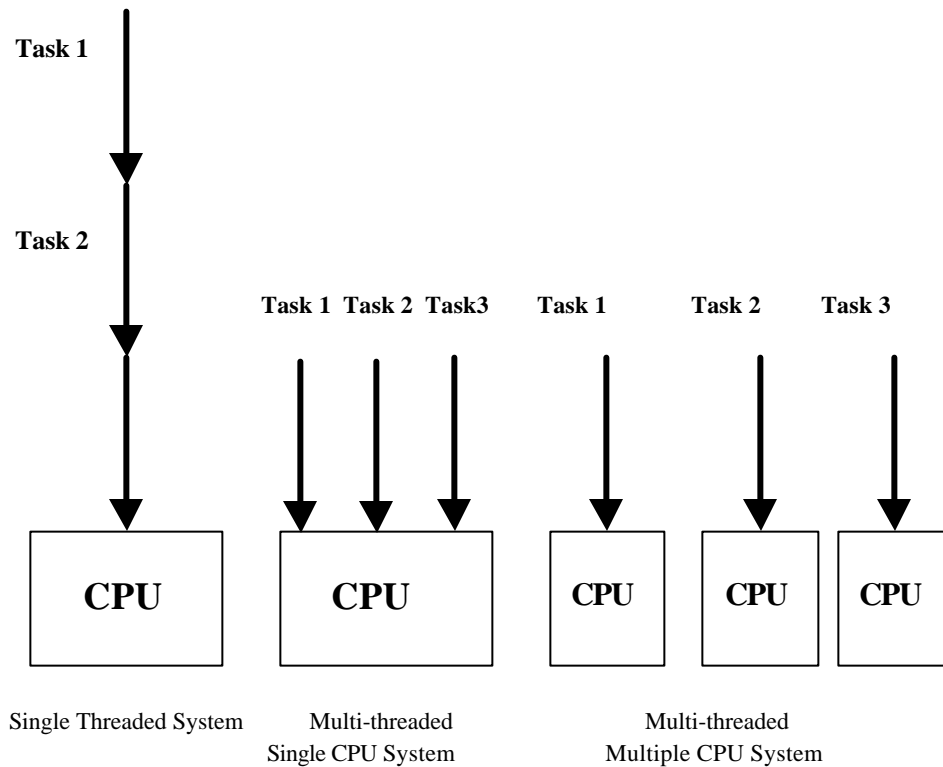


Figure 4. Single vs. multiple threading environments.

Of the three, the single threaded system will often yield the worst performance. The second system shows a multi-threaded system running under a single CPU. In this system the tasks are allowed to run for an interval of time called a time slice. When the time slice expires, the next task is given the CPU and the state of the previous task is saved. This system often yields better overall CPU utilization than the first, since the CPU does not have to sit idle during I/O requests. In this system, since there is only one CPU, each task may take a bit longer to run, due to time-slicing, but the overall time to execute all tasks is almost always shorter than the single threaded counterpart. The third system shows a multi-threaded system running under multiple CPUs. This system will almost always yield the best overall performance for the obvious fact that adding more CPUs will boost system performance.

It is important to understand the execution process of a Java applet when writing applets, which require multiple thread support. Figure 5 shows the execution cycle of a Java applet. The arrows show the events that take place in the browser; the circles represent the corresponding functions in the Java applet class called to handle those events.

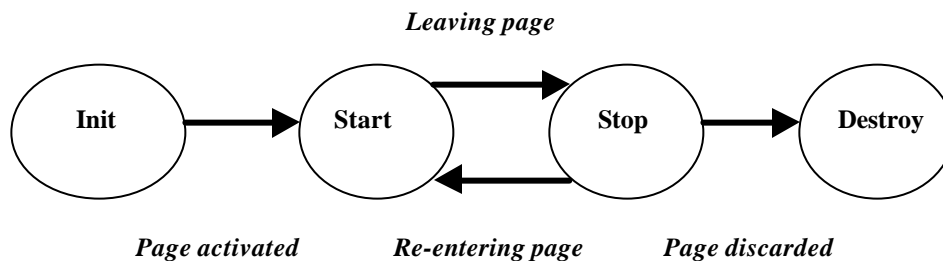


Figure 5. Life cycle of a Java applet.

A Java applets' *init* and *start* method are automatically called by the JVM when the applet is started. In order to keep the Java environment responsive, a Java applet must return from the invocation of these methods in a timely manner.

The animation sequence in this example continues to play until the applet is destroyed. The *FrameAnimation* applet implements the *Runnable* interface. Any class whose instances are intended to be executed by a thread can implement the *Runnable* interface.

During the *start* method, the *FrameAnimation* applet will create a new *Thread* class and execute its *start* method. The invocation of a thread *start* method will cause an objects *run* method to be invoked. The class, provided to the *Thread*'s constructor, determines the owning object of the *run* method. In this example, the “*this*” keyword was used when the *Thread* was constructed. The actual line of code is displayed below:

```
runner = new Thread(this);
```

The “*this*” keyword is used to represent the current object, which in this case is the applet itself. Therefore, the *run* method of this applet will gain control. In that method, the images are cycled through and repainted in a loop. The *mySleep* method is used to sleep between the rendering of consecutive frames.



The animation thread is halted when the *stop* method is invoked. The *stop* method of an applet is invoked when user leaves the page that contains the applet. Although the applet is still running, the *stop* method provides the applet writer with a means to perform some action when the applet is no longer in view. In this case, the CPU intensive animation sequence is halted. An applet will receive a *destroy* method before it is actually terminated by the JVM.

### 2.1.2 Eliminating Flicker

The FrameAnimation applet employed a technique to eliminate flicker. Flicker, in this context, is an annoying visual side effect caused by the unnecessary insertion of blank frames in the animation sequence. Figure 6 graphically displays an animation sequence that produces screen flicker.



Figure 6. Animation frame sequence when default *update* method is being used.

A good understanding of the Java painting protocol allows developers the ability to produce flicker-free animation sequences [5]. Figure 7 graphically displays the final frame sequence of a flicker free animation. In order to produce this sequence, the FrameAnimation applet subclasses the *update* method.



Figure 7. Animation frame sequence when default *update* method is subclassed.

In this example, the *repaint* method is called from within the animation thread to signal the painting of the next frame in the animation sequence. When the *repaint* method is invoked, the JVM will call the corresponding *update* method for that object. The default *update* method, as contained in the Java 1.0 release is displayed below:

```
public void update(Graphics g)
{
    g.setColor(getBackground());
    g.fillRect(0, 0, width, height);
    g.setColor(getForeground());
    paint(g);
}
```

The *update* method clears the graphics area with the background color before calling the *paint* method. In this example the *update* method is subclassed and simply calls the *paint* method. The code for this method is displayed below:

```
public void update(Graphics g)
{
    paint(g);
}
```

Referring to Figure 5, the default *update* method is causing a black rectangle to be displayed between each of the consecutive animation frames. Although this black frame is only displayed for a few milliseconds, it will cause the animation to flicker. By subclassing the *update* method we eliminate the flicker since the black frames will no longer be displayed between each of the consecutive animation frames. This technique works fine for this application since it is drawing the images at the same location over and over. If the images were moving or other graphics were being rendered onto the screen, then a more sophisticated version of the *update* method would need to be employed.

## 2.2 SPRITE-BASED ANIMATION

In the first animation example the animation frames were simply overlaid on top of the previous one in the exact same location. The applet was able to optimize things based on the characteristics of that type of animation. Although simple frame-based animations are frequently used, a more powerful type of animation is required for applications such as computer games. A sprite is an image that has a transparent background. In computer games the sprites are moved about the game while the background scenery is preserved. How is this accomplished in Java?

There are other issues involved in sprite-based animation. For example, how is the background restored when the sprite is moved across the screen? The next example will demonstrate techniques that allow programmers to build programs using sprite-based animations. The *SpriteAnimation* example, presented in Figure 8, draws a red bouncing ball on a blue background. The ball bounces off the edges of the applet as a real ball bounces off a wall. The source code for the *SpriteAnimation* applet is given in Figure 9.

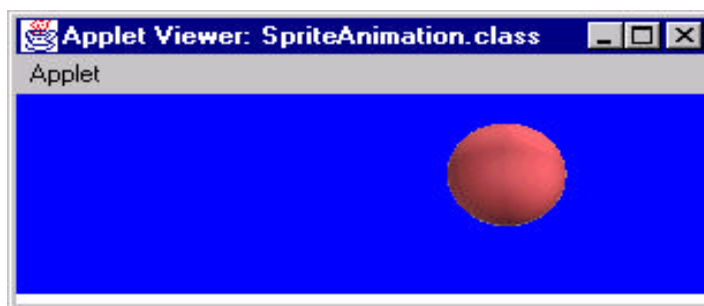


Figure 8. Output of the *SpriteAnimation* Java applet.

### 2.2.1 Transparency

Java supports the GIF89a image file format. In this format an RGB color value is assigned within the GIF file to represent the transparent color [6]. When the GIF is rendered to the screen, all pixels in the GIF matching this assigned RGB color value are not drawn, hence yielding the transparent effect. Java programmers can create images with the proper transparency settings using any image editor, which supports the GIF89a image file format. The Java programmer simply uses these special images to create transparent effects. This is an improvement upon previous windowing systems, such as Windows, where the programmer must perform special instructions to draw transparent images onto the screen.

### 2.2.2 Double Buffering

The SpriteAnimation applet uses a technique called “double buffering” to render the sprite across the screen in a fast manner. Double buffering tackles two problems inherent in sprite-based animation. These problems are avoiding flicker and preserving the background.

In double buffering, an application draws to memory before updating the screen. By first drawing into memory, all flicker can be eliminated since only the finished frame is rendered to the screen. This is demonstrated in Figure 10.

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

public class SpriteAnimation extends Applet implements Runnable
{
    Thread    runner;
    Image     ball, offImg;
    int       x = 0;
    int       xInc = 1;
    int       yInc = 1;
    int       y = 0;
    int       imgX, imgY;
    Graphics  offG;

    public void init()
    {
        System.out.println("In init method");

        ball = getImage(getDocumentBase(), "Ball.gif");
        if(ball == null)
            System.out.println("getImage failed to obtain Ball.gif");

        while(!prepareImage(ball, this))
        {
            // System.out.println("Sleeping while preparing ball");
            mySleep(10);
        }

        imgX = ball.getWidth(this);
        System.out.println("imgX = " + imgX);

        imgY = ball.getHeight(this);
        System.out.println("imgY = " + imgY);
    }
}
```

```

public void start()
{
    System.out.println("In start method");
    if(runner == null)
    {
        runner = new Thread(this);
        runner.start();
    }
}
public void stop()
{
    System.out.println("In stop method");
    if(runner != null)
    {
        runner.stop();
        runner = null;
    }
}
public void run()
{
    System.out.println("In run method");
    while(true)
    {
        repaint();
        mySleep(32);
    }
}

```

Figure 9. The Java applet for sprite animation.

```

public void update(Graphics g)
{
    Dimension d = size();

    if(offG == null)
    {
        offImg = createImage(d.width, d.height);
        offG = offImg.getGraphics();
    }

    offG.setColor(Color.blue);
    offG.fillRect(0, 0, d.width, d.height);
    offG.drawImage(ball, x, y, this);

    //
    // Now test to see if the ball hit the right or left wall.
    //
    if((x + imgX) >= d.width)
    {
        xInc = -1;
    }
    else if(x <= 0)
    {
        xInc = 1;
    }
}

```

```
    }
    x += xInc;

    //
    // Now test to see if the ball hit the bottom or top wall.
    //
    if((y + imgY) >= d.height)
    {
        yInc = -1;
    }
    else if(y <= 0)
    {
        yInc = 1;
    }
    y += yInc;

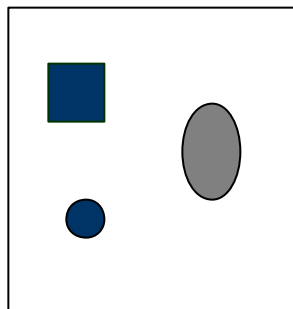
    g.drawImage(offImg, 0, 0, this);
}

public void paint(Graphics g)
{
}

public void mySleep(long millis)
{
    try
    {
        Thread.sleep(millis);
    }
    catch(InterruptedException e) {}
}
}
```

Figure 9 (cont.) The Java applet for sprite animation.

*Double buffer in system memory*



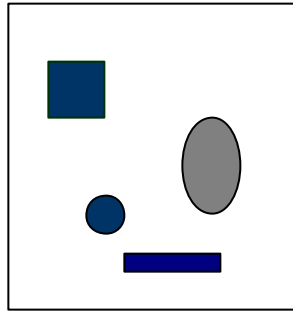
*Video memory*



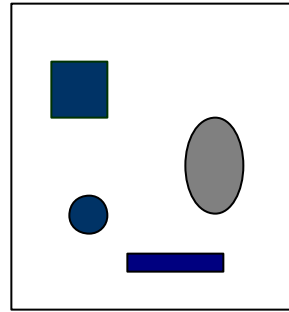


(a) Instead of drawing to video memory, a double buffer is used.

*Double buffer in system memory*



*Video memory*



(b) When finished drawing, the double buffer is copied to video memory.

Figure 10. Double buffering concept.

Double buffering is handled in the *update* method of the *SpriteAnimation* applet. The *update* method is shown in Figure 11.

An off-screen memory image is created using the *createImage* method. The *createImage* method takes the width and height of the image as parameters. In this example, the size of the applets client area is used. Once a memory image is created, a corresponding *Graphics* instance is obtained using the *getGraphics* method.

In the *SpriteAnimation* example, the background is always blue. This makes it easy for the applet to preserve the background pixels. In applications with sophisticated background scenery, the applet would typically keep a copy of the area under the sprite so that it can be restored when the sprite is moved.

```
public void update(Graphics g)
{
    Dimension d = size();

    if(offG == null)
    {
        offImg = createImage(d.width, d.height);
        offG = offImg.getGraphics();
    }

    offG.setColor(Color.blue);
    offG.fillRect(0, 0, d.width, d.height);
    offG.drawImage(ball, x, y, this);
}
```

```
//  
// Now test to see if the ball hit the right or left wall.  
//  
if((x + imgX) >= d.width)  
{  
    xInc = -1;  
}  
else if(x <= 0)  
{  
    xInc = 1;  
}  
x += xInc;  
  
//  
// Now test to see if the ball hit the bottom or top wall.  
//  
if((y + imgY) >= d.height)  
{  
    yInc = -1;  
}  
else if(y <= 0)  
{  
    yInc = 1;  
}  
y += yInc;  
  
g.drawImage(offImg, 0, 0, this);  
}
```

Figure 11. The “update” method for sprite animation.

### 2.2.3 Collision Areas

Most computer games that use sprite animation also maintain “collision areas.” The collision areas are regions on the screen that cause an event when the sprite intersects it. The `SpriteAnimation` example uses trivial collision detection, which is graphically depicted in Figure 12.

When the sprite hits one of the boundaries of the applet’s viewable area, the direction of the sprite is changed. This gives the effect of a ball bouncing against a wall. The code for the collision detection is shown in Figure 13.



Figure 12. Collision area detection diagram.

```

offG.drawImage(ball, x, y, this);

//
// Now test to see if the ball hit the right or left wall.
//
if((x + imgX) >= d.width)
{
    xInc = -1;
}
else if(x <= 0)
{
    xInc = 1;
}
x += xInc;

//
// Now test to see if the ball hit the bottom or top wall.
//
if((y + imgY) >= d.height)
{
    yInc = -1;
}
else if(y <= 0)
{
    yInc = 1;
}
y += yInc;

```

Figure 13. The Java code for collision detection.

If the ball hits the right boundary, then the x incrementor value (xInc) is set to -1. This will start the ball moving to the left on the next pass. Similarly, if the ball hits the left boundary, then xInc is set to 1, causing the ball to go toward the right on the next pass. The tests are made for the Y-axis. The origin of the ball is with respect to the upper left-hand corner of the screen, so the size of the image must be used when testing for a right or bottom wall hit test.



### 3. AUDIO SUPPORT IN JAVA

An essential element to any multimedia experience is the inclusion of sounds. The Java Applet class provides the necessary methods, which allow programmers to load and play sound files [7]. The current level of the JDK supports only the Sun ".AU" audio file format. This limitation can be evaded by using utilities to convert audio files from some other format to the AU format.

In most cases, this evasion seems to be sufficient except for the MIDI (Musical Instrument Digital Interface) file format. A MIDI audio file is sufficiently different than the others in that it is not an actual recording as AU or WAV files are. MIDI files are computer programs, which instruct a synthesizer to produce sounds. Therefore, one cannot rely on utility programs to convert MIDI files to AU files. JavaSoft has addressed MIDI in its recently released Java Media API. The Java Media API defines a new and exciting set of multimedia classes. The Java Media API is covered in more detail later in this chapter.

#### 3.1 PLAYING AN AUDIO CLIP FROM A JAVA

The first audio-enabled Java applet presented here will play an audio clip. The source code for this applet is presented below. It takes only one line of Java code to create a simple Java applet, which plays a predefined audio clip. Although this is an oversimplified example, it provides a compelling reason for programmers to investigate Java for multimedia programming.

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

public class Sound1 extends Applet
{
    public void init()
    {
        play(getCodeBase(),"spacemusic.au");
    }
}
```

#### 3.2 PLAYING AN AUDIO CLIP USING A MORE SOPHISTICATED APPROACH

In the first Java audio example, the program loaded and played the audio clip using the *play* method of the applet class. Although that applet was trivial to write, it is not very useful for a practical solution since it does not provide a means to replay the audio clip. It simply plays the audio clip once and does nothing after that.

Most applications that utilize audio provide the user with a mechanism to play the audio clip when they are ready to hear it. They also allow the user to replay the audio clip if they want to hear it again. The second Java audio example adds a "play" button to the applet. The output screen for the *Sound2* applet is displayed in Figure 14. The audio clip is played whenever the "play" button is clicked on. The source code for the *Sound2* Java applet is shown in Figure 15.

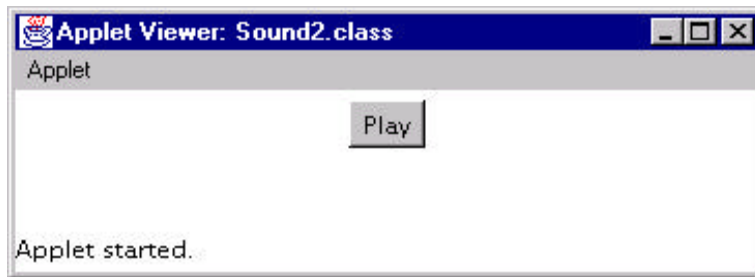


Figure 14. Output of the Sound2 Java applet.

```

import java.applet.*;
import java.awt.*;

public class Sound2 extends Applet
{
    Button    play;
    AudioClip aclip;

    public void init( )
    {
        aclip = getAudioClip(getCodeBase( ), "spacemusic.au");

        play = new Button("Play");
        add(play);
    }

    public boolean action(Event evt, Object arg)
    {
        if(evt.target instanceof Button)
        {
            if(arg.equals("Play"))
            {
                if(aclip == null)
                {
                    System.out.println("Sound2: Err audio clip not loaded.");
                }
                if(aclip != null)
                {
                    aclip.play();
                }
                return true;
            }
        }
        return false;
    }
} // end of action
}

```

Figure 15. The source code for Sound2 Java applet.

The audio clip is loaded at initialization time using the following Java statement:

```
aclip = getAudioClip(getCodeBase( ), "spacemusic.au");
```

When the “play” pushbutton is pressed, the Sound2 applet will gain control in its *action* method. The audio clip will be played using the *play* method of the AudioClip class. If the audio clip has not been successfully loaded, the *action* method will print a message out accordingly.

### 3.3 PLAYING AND STOPPING AN AUDIO CLIP

In the next audio example, Sound3, the program allows the user to start and stop playing the audio clip via the user interface. The output screen from the Sound3 Java applet is displayed in Figure 16.

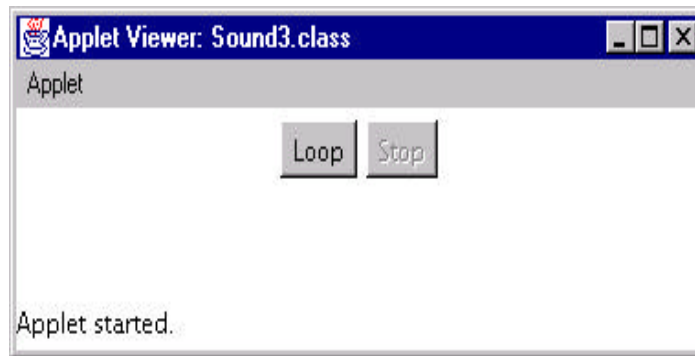


Figure 16. Output of the Sound3 Java applet.

Two pushbuttons are provided. When the user presses the “Loop” pushbutton, the audio clip is played using the *loop* method of the AudioClip class. This method differs from the *play* method in that it will continuously play the audio clip in a loop. The *stop* method of the AudioClip class is used to stop playing the clip. The source code for this final Java audio example is given in Figure 17.

## 4. VIDEO SUPPORT IN JAVA

Similar to MIDI support, the base Java classes in the latest level of the JDK do not support video playback. This is most probably due to the fact that there are numerous non-Java video solutions available for the Internet today.

There are two primary approaches to delivering video on the Internet: downloading stored files vs. “streaming” video (and audio). Each presents special challenges and opportunities for bandwidth allocation.

### 4.1 STANDARD VIDEO

Downloading and playing stored video files over the Internet is the older of the two available video playback approaches. Here, digital video files are stored on a server (typically in MPEG1

or QuickTime Movie format) and, when someone wants to access the video, they must download it in full before viewing it. The time it takes to download the file largely depends on two things: first, how large the file is (i.e., the larger the file, the longer it takes to download) and, second, how much bandwidth is available, both on the server and client sides of the network.

```

import java.applet.*;
import java.awt.*;

public class Sound3 extends Applet
{

    Button    loop;        // Starts playing loop
    Button    stop;        // Stops playing loop
    AudioClip loopClip;    // Audio clip

    public void init( )
    {
        loopClip = getAudioClip(getCodeBase( ),"spacemusic.au");

        loop = new Button("Loop");
        loop.enable();
        add(loop);

        stop = new Button("Stop");
        stop.disable();
        add(stop);
    }

    public boolean action(Event evt, Object arg)
    {
        if(evt.target instanceof Button)
        {
            if(arg.equals("Loop"))
            {
                if(loopClip != null)
                {
                    loopClip.loop();
                    stop.enable();
                    loop.disable();
                }
                return true;
            }
            else if(arg.equals("Stop"))
            {
                loopClip.stop();
                loop.enable();
                stop.disable();
                return true;
            }
        }
        return false;
    }
} // end of action
}

```

Figure 17. Java applet for playing and stopping an audio clip.

#### **4.2 STREAMING VIDEO**

Streaming video (and audio) represents a second and more complex area of video on the Internet. Streaming involves the delivery of video or audio "in real time" through a number of techniques, including some that place several frames of video into a buffer on the client's hard drive, and then begin playing the video, as more files are placed into the buffer. To the viewer, the video plays in approximately real time, without having to wait for an entire large video file to download.

Each streaming video solution defines its own proprietary movie file format with a corresponding encoder utility that converts a standard movie file such as AVI or MPEG to their own native format. These streaming video providers usually give away the streaming player and sell the encoder.

Providing support for video in Java requires the selection of some standard movie file format such as the selection of AU files for audio. Since most users are accessing the Internet via modem, one would assume that a streaming video solution would be provided.

In trying to obtain all available information on this subject, it appears that JavaSoft's future support for video and all multimedia support will be defined by the Java Media API. JavaSoft has been working with a group of industry-leading companies to establish the standards for Java Media: Adobe®, Apple, Intel, Macromedia, Netscape, SGI, and Sun Microsystems. The Java Media API is briefly described in the next section.

### **5. JAVA MEDIA API**

The Java Media API defines a set of multimedia classes which support a wide range of rich, interactive media on and off the Web, including audio, video, 2D, 3D, animation, telephony, and collaboration [8]. An extensible Media Framework provides common control and synchronization of all time-based media (audio, video, animation, video conferencing) as well as filters and processors.

The Java Media API is composed of several distinct components, each associated with either a specific type of media (audio, video, 2D, 3D), or a media-related activity (animation, collaboration, telephony). Collectively, these interfaces provide Java Language programmers with the ability to handle a wide variety of different media types within their applications and applets.

The Java Media API is highly extensible. The API accommodates today's large and ever-changing suite of media transports, containers, and encoding formats, and allows the addition of new media-related functionality as they become available.

The components of the Java Media APIs are as follows:

- Java 2D API - Provides graphics and imaging capabilities beyond those available with the Java Applet API. The 2D API allows the creation of high quality, platform-independent graphics including line art, text, and images in a single model that uniformly addresses color, spatial transforms and compositing. It also provides an extension mechanism to support a wide array of different presentation devices (such as displays and printers), image formats, image encoding, color spaces, and compositors.

- Java Media Framework API - Handles traditional time-critical media, such as audio, video, and MIDI. The framework provides a common model for timing, synchronization, and composition, which can be applied to media components to allow them to interoperate. It is designed to handle streaming data, live or stored, compressed or raw, as well as from sampled audio and video streams.
- Video API - Accommodates both streaming and stored video sources. It defines basic data formats and control interfaces.
- Audio API - Supports sampled and synthesized audio. It includes a specification for 3D spatial audio, and accommodates both streaming and stored audio sources.
- MIDI API - Provides support for timed-event streams. It uses the Media Framework for synchronization with other activities, and for an extensibility mechanism for new synthesizers and effects.
- Java Animation API - Supports traditional 2D animation of sprites, with stacking order control. It makes use of 2D interfaces for compositing and the Media Framework for synchronization, composition, and timing.
- Java Share API - Provides the basic abstraction for live, two-way, multi-party communication between objects over a variety of networks and transport protocols. The API enables synchronization and session management, and allows sharing of both "collaboration-aware" and "collaboration-unaware" applets.
- Java Telephony API - Unifies computer/telephony integration. It provides basic functionality for control of phone calls: 1st-party call control (simple desktop phone), 3rd-party call control (phone call distribution center), teleconferencing, call transfer, caller ID, and DTMF decode/encode.
- Java 3D API - Provides high-performance, interactive, 3D graphics support. It supports VRML, and has a high-level specification of behavior and control of 3D objects. The 3D API simplifies 3D-application programming and provides access to lower level interfaces for performance. The 3D API is closely integrated with Audio, Video, MIDI, and Animation areas.

## 6. CONCLUSION

Programming in any object-oriented language requires a good understanding of the tools and class libraries. The class libraries define the set of functions that are available to the Java programmer. The more robust the class libraries are, the less work will be required to be completed by the Java programmer. The object-oriented paradigm allows software developers to build upon the available class libraries to create new custom class libraries, thus extending the language.

The Java class libraries available today provide enough support to allow developers to write Java programs that do simple animations and play audio clips. This chapter presented working Java applets that accomplished these tasks. The current Java class libraries did not provide support for MIDI, streaming audio or any form of video playback.

Language extensions via custom classes can be used to add additional multimedia support to Java [9]. The sheer size and complexity of the code required to extend Java's multimedia support are the main reasons why these extensions are not available today. This problem is magnified by the fact that playing audio or video will eventually require low level programming, which is, at the least, operating system dependent.

Java provides an interface for this type of low level coding called the "native method" interface. Native methods are code fragments that can be written in some other programming language. In most cases, the native methods are written in "C" and interface directly with the operating system. Native methods are platform dependent and must be re-written for every operating system that one chooses to support. Any custom multimedia classes would require native methods.

The shortcomings of Java in the multi-media arena appear to be temporary. The Java Media Framework (JMF) API is on the horizon. This API offers a robust set of multi-media services that provide support for almost any current audio or video format by building upon an established media playback framework. Current research in multimedia, based on object models [10], appears to be a direct match for the flexible object-oriented Java language.

#### REFERENCES

- [1] J. Lam, "Java and Java Tools: Waking Up the Web," *PC Magazine*, June 11, 1996.
- [2] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko, "Compiling Java Just In Time," *IEEE Micro*, Vol. 17, No. 3, pp. 36-43, May - June 1997.
- [3] S. Ritchie, "Systems Programming in Java," *IEEE Micro*, Vol. 17, No. 2, pp. 30-35, May - June 1997.
- [4] B. Lewis and D. J. Berg, "How to Program with Threads (An Introduction to Multi-threaded Programming)," *Sun World*, Vol. 10, No. 2, February 1996.
- [5] P. Buchheit, "Flicker-free Animation Using Java," *Linux Journal*, October 1996.
- [6] Graphics Interchange Format (sm) - Version 89a (c) 1987,1988,1989,1990. Copyright CompuServe Incorporated Columbus, Ohio, 31 July 1990.
- [7] C. S. Wah and J. D. Mitchell, "How to Play Audio in Applications," *Java World*, February 1997.
- [8] B. M. Day, Jr., "Java Media Framework Player API," *Java World*, April 1997.
- [9] J. Begole, C. A. Strumble, and C. A. Shaffer, "Leveraging Java Applets: Toward Collaboration Transparency in Java," *IEEE Micro*, Vol. 17, No. 2, March - April 1997.
- [10] V.M. Bove, Jr., "Multimedia Based on Object Models: Some Whys and Hows," *IBM Systems Journal*, Vol. 35, Nos. 3&4, pp. 337-348, 1996.
- [11] B.R. Montague, "JN: OS for an Embedded Java Network Computer," *IEEE Micro*, Vol. 17, No. 2, pp. 54-61, May-June 1997.
- [12] E. Yourdan, "Java, the Web, and Software Development," *IEEE Computer*, Vol. 29, No. 8, pp. 25-30, August 1996.
- [13] E. Evans and D. Rogers, "Using Java Applets and CORBA for Multi-User Distributed Applications," *IEEE Internet Computing*, Vol. 1, No. 3, pp. 43-55, May/June 1997.