# Fun with Java: Sprite Animation, Part 2

*Baldwin shows you how to instantiate and use a MediaTracker object to track the loading status of Image objects. He shows you how to use the getImage method of the Toolkit class to create Image objects from GIF files, how to register those Image objects on a MediaTracker object, and how to use the waitForID method of the MediaTracker object to force the program to block and wait until the images in a group are successfully loaded, or until a loading error occurs.*

**Published:** October 8, 2001
**By Richard G. Baldwin**

Java Programming, Lecture Notes # 1452

---

## Preface

Programming in Java doesn't have to be dull and boring. In fact, it's possible to have a lot of fun while programming in Java. This is one of the lessons in a miniseries that will concentrate on having fun while programming in Java.

This miniseries will include a variety of Java programming topics that fall in the category of *fun programming*. This particular lesson is the second in of a group of lessons that will teach you how to write animation programs in Java. The first *(and previous)* lesson in the group was entitled Fun with Java: Sprite Animation, Part 1.

### Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different figures and listings while you are reading about them.

### Supplementary material

I recommend that you also study the other lessons in my extensive collection of online Java tutorials. You will find those lessons published at Gamelan.com. However, as of the date of this writing, Gamelan doesn't maintain a consolidated index of my Java tutorial lessons, and sometimes they are difficult to locate there. You will find a consolidated index at Baldwin's Java Programming Tutorials.

## Preview

This is one of a group of lessons that will teach you how to write animation programs in Java. These lessons will teach you how to write *sprite* animation, **frame** animation, and a combination of the two.

### Colored spherical sea creatures

The first program, which is the program discussed in this lesson, will show you how to use sprite animation to cause a group of colored spherical sea creatures to swim around in a fish tank. A screen shot of the output produced by this program is shown in Figure 1.

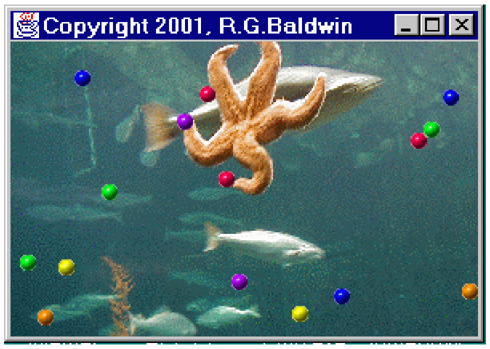

Figure 1. Animated spherical sea creatures in a fish tank.

### Many sea creatures can change color

Many sea creatures have the ability to change their color in very impressive ways. The second program that I will discuss in subsequent lessons will simulate that process using a combination of sprite and frame animation.

### Animated multi-colored sea worms

The third program, also to be discussed in a subsequent lesson, will use a combination of sprite animation, frame animation, and some other techniques to cause a group of multi-colored sea worms to slither around in the fish tank. In addition to slithering, the sea worms will also change the color of different parts of their body, much like real sea creatures.

A screen shot of the output from the third program is shown in Figure 2.

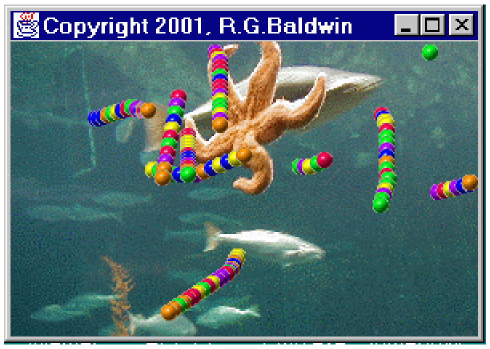

Figure 2.  Animated sea worms in a fish tank.

## Get your GIF images here

Figure 3 shows the GIF image files that you will need to run these three programs.

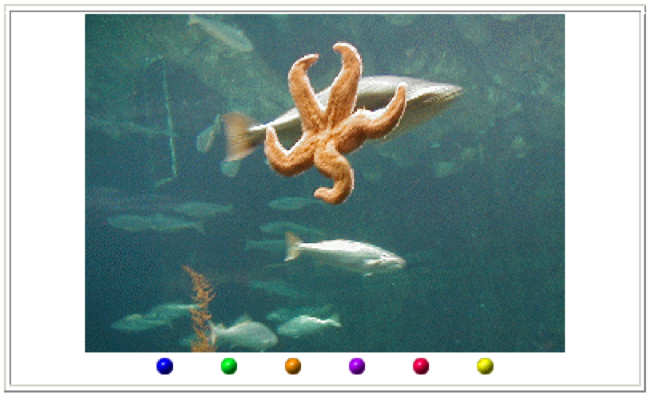

Figure 3.  GIF image files that you will need.

You should be able to capture the images by right-clicking on them individually, and then saving them into files on your local disk.  Having done that, you will need to rename the files to match the names that are hard-coded into the programs.

## Classes and concepts discussed earlier

In the previous lesson, I introduced you to several classes and concepts that you must understand in order to understand animation in Java.  Included in that discussion were the following classes, which are particularly important to sprite animation:

- Image
- Toolkit
- Graphics
- MediaTracker
- Random

I also discussed a number of concepts, including the following, which are particularly important to sprite animation:

- offscreen graphics contexts
- coordinates in Java graphics
- translation origins
- the drawImage method
- animation repetition rates
- pseudo-random numbers

### Discussed beginning portions of controlling class

Also, in the previous lesson, I discussed the beginning portions of the controlling class for the program named **Animate01**. *(A complete listing of the program is provided near the end of this lesson.)*

### The plan for this lesson

In this lesson, I will teach you how to instantiate and use a **MediaTracker** object to track the loading status of **Image** objects.

I will teach you how to use the **getImage** method of the **Toolkit** class to create **Image** objects from GIF files, and how to register those **Image** objects on a **MediaTracker** object.

Finally, I will teach you how to use the **waitForID** method of the **MediaTracker** object to force the program to block and wait until the images in a group are either successfully loaded, or until a loading error occurs.

## Discussion and Sample Program

This program is so long that several lessons will be required to discuss it fully. Rather than to make you wait until I complete all of those lessons to get your hands on the program, I have provided a copy of the entire program in Listing 7 near the end of the lesson. That way, you can copy it into a source file on your local disk, compile it, run it, and start seeing the results.

### Will discuss in fragments

As usual, I will discuss the program in fragments. In addition to the controlling class named **Animate01**, the program contains several other important classes. I began the discussion of the controlling class in the previous lesson. I will continue discussing the controlling class in this lesson picking up with the constructor for the controlling class.

### A MediaTracker object

Listing 1 shows the beginning of the constructor for the controlling class. The first statement in Listing 1 instantiates a new object of the **MediaTracker** class.

```
  Animate01() {//constructor
    mediaTracker =
                 new
MediaTracker(this);

Listing 1
```

## What is a MediaTracker object?

As explained in some detail in the previous lesson, the **MediaTracker** class is a utility class, objects of which can be used to track the status of a number of media objects. An object of the **MediaTracker** class can help you to cope with the delays involved in loading image files into the computer's memory.

## The MediaTracker constructor

As of JDK 1.3, the **MediaTracker** class has only one constructor whose signature is shown below:

```
public MediaTracker(Component comp)
```

This constructor creates a media tracker object to track images for a given component. The single parameter of type **Component** is the component on which the images will eventually be drawn.

## Will draw images on *this* object

In this program, the images will be drawn on an object of the controlling class, which extends the **Frame** class. Therefore, in the code in Listing 1, a reference to the object being constructed by the controlling class' constructor **(this)** is passed as a parameter to the constructor for the **MediaTracker** class. The resulting **MediaTracker** object can be used to track images that will eventually be drawn on an object instantiated from the controlling class.

## The background image

As shown in Figure 1, the animated spherical sea creatures will be drawn against a background image representing the inside of a fish tank.

The code in Listing 2 creates an object of type **Image** containing the picture described by the physical file named **background02.gif**.

```
    backGroundImage =
        Toolkit.getDefaultToolkit().
          getImage("background02.gif");

Listing 2
```

*(Instructions were provided earlier in this lesson for capturing a GIF file containing the background image from this HTML document. Once you have captured the GIF file, in order to make it compatible with the code in Listing 2, you will need to rename the file background02.gif, and store it in the directory containing the class file produced by compiling the controlling class for this program.)*

**The getImage method of the Toolkit class**

The **Image** object is created in Listing 2 by invoking the **getImage** method of the **Toolkit** class, passing the name of the GIF file as a parameter.

As of JDK 1.3, the **Toolkit** class provides two overloaded versions of the **getImage** method. One takes a URL as a parameter, while the other *(the one used in this program)* takes a file name represented by a **String** object as a parameter.

**The getDefaultToolkit method**

Before getting into the details of the **getImage** method, it is appropriate to make a few comments about the **getDefaultToolkit** method.

The **getDefaultToolkit** method creates and returns an object of some class that extends the **Toolkit** class. This class defines the methods declared in the **Toolkit** class in such a way as to handle the peculiarities of the platform on which the Java program is running.

Many such methods are declared as *abstract* methods in the **Toolkit** class. In this particular case, we are interested in the overridden version of the method named **getImage**.

**Back to the getImage method**

The signature of the **getImage** method, as declared in the **Toolkit** class, is as shown below *(note that the method is abstract and must be fully defined in the object returned by the getDefaultToolkit method).*

```
public abstract Image getImage(
                     String filename)
```

**Creates Image object from an image file**

As of JDK 1.3, the **getImage** method returns a reference to an object of type **Image**, which contains pixel data from the specified file. The file format can be either GIF, JPEG or PNG.

The parameter to this overloaded version of the **getImage** method is a **String** object representing the name of a file containing pixel data in a recognized file format. For the code in Listing 2, this is the file named **background02.gif**, stored in the same directory as the program file.

## Coping with the loading delay

Even when the file is stored on a local hard drive, some finite amount of time is required to find the file, read it, and create the **Image** object. Any attempt to use the **Image** object before that process is completed is very likely to result in problems. That is where the **MediaTracker** object instantiated earlier comes into play.

## Registering the Image object on the MediaTracker

The code in Listing 3 invokes the **addImage** method of the **MediaTracker** class to register the **Image** object referred to by the reference variable named **backGroundImage** on the **MediaTracker** object with an ID value of 0.

```
    mediaTracker.addImage(
                    backGroundImage,
0);
```

**Listing 3**

*(Note that there is another version of the addImage method that allows for scaling the image as it is being loaded. That version is not used in this program.)*

According to the Sun documentation, the version of the **addImage** method used in Listing 3

> *"Adds an image to the list of images being tracked by this media tracker. The image will eventually be rendered at its default (unscaled) size."*

## Getting Image object status

Having accomplished this, the **MediaTracker** object can then be queried to determine the loading status of the image. A variety of methods are provided by the **MediaTracker** class to support such queries.

When an image is registered for tracking with a **MediaTracker** object, the second parameter of the **addImage** method provides an integer identifier value. Many of the methods that can be used to query the **MediaTracker** object take the identifier as a parameter. Those methods provide information about the image(s) having that identifier value to the exclusion of images having other identifier values.

## Tracking images as a group

More than one image can be registered with the same identifier value. In this case, all of the images having the same identifier value will be tracked as a group. A query based on the common identifier value will return information about the group as a whole rather than returning information about the individual members of the group.

## Seven different Image objects

This program uses seven different **Image** objects based on seven different GIF files. One of the **Image** objects is used to create the background shown in Figure 1. The other six **Image** objects are used to create the spherical sea creature shown in Figure 1.

## An Image object for sea creatures

The code to create and register the **Image** object used for the background was shown in Listings 2 and 3. The code to create and register the first of the **Image** objects used for sea creatures is shown in Listing 4.

```
    gifImages[0] =
           Toolkit.getDefaultToolkit().
               getImage("redball.gif");
    mediaTracker.addImage(
                      gifImages[0], 0);

Listing 4
```

This is essentially the same as the code in Listings 2 and 3 with the one exception that the reference to the **Image** object is stored in an element of an array of type **Image[]** instead of being stored in an ordinary reference variable.

## The GIF file name

The code in Listing 4 also shows the name to be applied to one of the GIF files that you capture from this HTML document: **redball.gif**.

## The remaining Image objects

The code for creating and registering the remaining five **Image** objects is shown in Listing 5.

```
    gifImages[1] =
           Toolkit.getDefaultToolkit().
             getImage("greenball.gif");
    mediaTracker.addImage(
                      gifImages[1], 0);
    gifImages[2] =
           Toolkit.getDefaultToolkit().
              getImage("blueball.gif");
    mediaTracker.addImage(
                      gifImages[2], 0);
    gifImages[3] =
           Toolkit.getDefaultToolkit().
            getImage("yellowball.gif");
    mediaTracker.addImage(
                      gifImages[3], 0);
    gifImages[4] =
           Toolkit.getDefaultToolkit().
            getImage("purpleball.gif");
    mediaTracker.addImage(
                      gifImages[4], 0);
    gifImages[5] =
           Toolkit.getDefaultToolkit().
            getImage("orangeball.gif");
    mediaTracker.addImage(
                      gifImages[5], 0);


Listing 5
```

## The registration identifier

Note that the identifier value used to register all seven of the **Image** objects on the **MediaTracker** object is the same (0).  As a result, the **MediaTracker** object will track the loading of all seven **Image** objects as a group.

Listing 5 also shows the names to be applied to the five remaining GIF files that you capture from this HTML file.

## Using the MediaTracker object

Listing 6 shows the use of the **MediaTracker** object to delay execution of the program until all seven of the images have been loaded and are ready for use.

```
    try {
      mediaTracker.waitForID(0);
    }catch (InterruptedException e) {
      System.out.println(e);
    }//end catch

Listing 6
```

## The waitForID method

The **waitForID** method call in Listing 6 causes the program to block and wait for all of the images to complete loading *(or for a loading error to occur).*  Here is part of what Sun has to say about this method:

*"Starts loading all images tracked by this media tracker with the specified identifier. This method waits until all the images with the specified identifier have finished loading.*

*If there is an error while loading or scaling an image, then that image is considered to have finished loading. Use the isErrorAny and isErrorID methods to check for errors."*

### No guarantee of successful loading

The fact that this method returns doesn't guarantee that all of the images identified by the ID value of 0 have successfully loaded. As indicated by Sun, the occurrence of a loading error will also cause this method to return.

Once the method returns, in order to be completely safe, the program should use one of the error checking methods listed in the quotation from Sun to confirm that an error did not occur.

### Another version of the waitForID method

Note that in theory, this method could block and wait forever. Another version of the **waitForID** method allows for the specification of a maximum length of time in milliseconds to block before returning, with or without a successful load.

## Summary

In this lesson, I have explained how to instantiate and use a **MediaTracker** object to track the loading status of **Image** objects.

I have explained how to use the **getImage** method of the **Toolkit** class to create **Image** objects from GIF files, and how to register those **Image** objects on a **MediaTracker** object.

Finally, I explained how to use the **waitForID** method of the **MediaTracker** object to force the program to block and wait until the images in a group are either successfully loaded, or until a loading error occurs.

## What's Next?

In the next lesson, I will continue explaining the constructor for the controlling class. The first topic in that lesson will deal with the methodology used to cause the size of the **Frame** object to be based on the size of the **Image** object used as a background.

## Complete Program Listing

A complete listing of the program is provided in Listing 7.

```
/*File Animate01.java
Copyright 2001, R.G.Baldwin

This program displays several animated
colored spherical creatures swimming
around in an aquarium.  Each creature
maintains generally the same course
with until it collides with another
creature or with a wall.  However,
each creature has the ability to
occasionally make random changes in
its course.

**************************************/
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Animate01 extends Frame
                  implements Runnable {
  private Image offScreenImage;
  private Image backGroundImage;
  private Image[] gifImages =
                          new Image[6];
  //offscreen graphics context
  private Graphics
                  offScreenGraphicsCtx;
  private Thread animationThread;
  private MediaTracker mediaTracker;
  private SpriteManager spriteManager;
  //Animation display rate, 12fps
  private int animationDelay = 83;
  private Random rand =
                new Random(System.
                  currentTimeMillis());

  public static void main(
                        String[] args){
    new Animate01();
  }//end main
  //---------------------------------//

  Animate01() {//constructor
    // Load and track the images
    mediaTracker =
                new MediaTracker(this);
    //Get and track the background
    // image
    backGroundImage =
        Toolkit.getDefaultToolkit().
          getImage("background02.gif");
    mediaTracker.addImage(
                   backGroundImage, 0);

    //Get and track 6 images to use
    // for sprites
    gifImages[0] =
           Toolkit.getDefaultToolkit().
               getImage("redball.gif");
    mediaTracker.addImage(
                      gifImages[0], 0);
    gifImages[1] =
           Toolkit.getDefaultToolkit().
             getImage("greenball.gif");
    mediaTracker.addImage(
                      gifImages[1], 0);
    gifImages[2] =
           Toolkit.getDefaultToolkit().
              getImage("blueball.gif");
    mediaTracker.addImage(
                      gifImages[2], 0);
    gifImages[3] =
           Toolkit.getDefaultToolkit().
            getImage("yellowball.gif");
    mediaTracker.addImage(
                      gifImages[3], 0);
    gifImages[4] =
           Toolkit.getDefaultToolkit().
            getImage("purpleball.gif");
    mediaTracker.addImage(
                      gifImages[4], 0);
    gifImages[5] =
           Toolkit.getDefaultToolkit().
            getImage("orangeball.gif");
    mediaTracker.addImage(
                      gifImages[5], 0);

    //Block and wait for all images to
    // be loaded
    try {
      mediaTracker.waitForID(0);
    }catch (InterruptedException e) {
      System.out.println(e);
    }//end catch

    //Base the Frame size on the size
    // of the background image.
    //These getter methods return -1 if
    // the size is not yet known.
    //Insets will be used later to
    // limit the graphics area to the
    // client area of the Frame.
    int width =
        backGroundImage.getWidth(this);
    int height =
       backGroundImage.getHeight(this);

    //While not likely, it may be
    // possible that the size isn't
    // known yet.  Do the following
    // just in case.
    //Wait until size is known
    while(width == -1 || height == -1){
      System.out.println(
                  "Waiting for image");
      width = backGroundImage.
                        getWidth(this);
      height = backGroundImage.
                       getHeight(this);
    }//end while loop

    //Display the frame
    setSize(width,height);
    setVisible(true);
    setTitle(
        "Copyright 2001, R.G.Baldwin");

    //Create and start animation thread
    animationThread = new Thread(this);
    animationThread.start();

    //Anonymous inner class window
    // listener to terminate the
    // program.
    this.addWindowListener(
                   new WindowAdapter(){
      public void windowClosing(
                        WindowEvent e){
        System.exit(0);}});

  }//end constructor
  //-------------------------------------//

  public void run() {
    //Create and add sprites to the
    // sprite manager
    spriteManager = new SpriteManager(
             new BackgroundImage(
               this, backGroundImage));
    //Create 15 sprites from 6 gif
    // files.
    for (int cnt = 0; cnt < 15; cnt++){
      Point position = spriteManager.
        getEmptyPosition(new Dimension(
           gifImages[0].getWidth(this),
           gifImages[0].
                     getHeight(this)));
      spriteManager.addSprite(
        makeSprite(position, cnt % 6));
    }//end for loop

    //Loop, sleep, and update sprite
    // positions once each 83
    // milliseconds
    long time =
            System.currentTimeMillis();
    while (true) {//infinite loop
      spriteManager.update();
      repaint();
      try {
        time += animationDelay;
        Thread.sleep(Math.max(0,time -
          System.currentTimeMillis()));
      }catch (InterruptedException e) {
        System.out.println(e);
      }//end catch
    }//end while loop
  }//end run method
  //---------------------------------//

  private Sprite makeSprite(
      Point position, int imageIndex) {
    return new Sprite(
          this,
          gifImages[imageIndex],
          position,
          new Point(rand.nextInt() % 5,
                  rand.nextInt() % 5));
  }//end makeSprite()
  //---------------------------------//

  //Overridden graphics update method
  // on the Frame

  public void update(Graphics g) {
    //Create the offscreen graphics
    // context
    if (offScreenGraphicsCtx == null) {
      offScreenImage =
         createImage(getSize().width,
                     getSize().height);
      offScreenGraphicsCtx =
          offScreenImage.getGraphics();
    }//end if

    // Draw the sprites offscreen
    spriteManager.drawScene(
                 offScreenGraphicsCtx);

    // Draw the scene onto the screen
    if(offScreenImage != null){
         g.drawImage(
           offScreenImage, 0, 0, this);
    }//end if
  }//end overridden update method
  //---------------------------------//

  //Overridden paint method on the
  // Frame
  public void paint(Graphics g) {
    //Nothing required here.  All
    // drawing is done in the update
    // method above.
  }//end overridden paint method
```

```
}//end class Animate01
//===================================//

class BackgroundImage{
  private Image image;
  private Component component;
  private Dimension size;

  public BackgroundImage(
                  Component component,
                  Image image) {
    this.component = component;
    size = component.getSize();
    this.image = image;
  }//end construtor

  public Dimension getSize(){
    return size;
  }//end getSize()

  public Image getImage(){
    return image;
  }//end getImage()

  public void setImage(Image image){
    this.image = image;
  }//end setImage()

  public void drawBackgroundImage(
                          Graphics g) {
    g.drawImage(
               image, 0, 0, component);
  }//end drawBackgroundImage()
}//end class BackgroundImage
//===========================

class SpriteManager extends Vector {
  private BackgroundImage
                       backgroundImage;

  public SpriteManager(
     BackgroundImage backgroundImage) {
    this.backgroundImage =
                       backgroundImage;
  }//end constructor
  //-----------------------------------//

  public Point getEmptyPosition(
                 Dimension spriteSize){
    Rectangle trialSpaceOccupied =
      new Rectangle(0, 0,
                    spriteSize.width,
                    spriteSize.height);
    Random rand =
         new Random(
           System.currentTimeMillis());
    boolean empty = false;
    int numTries = 0;

    // Search for an empty position
    while (!empty && numTries++ < 100){
      // Get a trial position
      trialSpaceOccupied.x =
        Math.abs(rand.nextInt() %
                      backgroundImage.
                      getSize().width);
      trialSpaceOccupied.y =
        Math.abs(rand.nextInt() %
                     backgroundImage.
                     getSize().height);

      // Iterate through existing
      // sprites, checking if position
      // is empty
      boolean collision = false;
      for(int cnt = 0;cnt < size();
                                cnt++){
        Rectangle testSpaceOccupied =
              ((Sprite)elementAt(cnt)).
                    getSpaceOccupied();
        if (trialSpaceOccupied.
                 intersects(
                   testSpaceOccupied)){
          collision = true;
        }//end if
      }//end for loop
      empty = !collision;
    }//end while loop
    return new Point(
                 trialSpaceOccupied.x,
                 trialSpaceOccupied.y);
  }//end getEmptyPosition()
  //-----------------------------------//

  public void update() {
    Sprite sprite;

    //Iterate through sprite list
    for (int cnt = 0;cnt < size();
                                cnt++){
      sprite = (Sprite)elementAt(cnt);
      //Update a sprite's position
      sprite.updatePosition();

      //Test for collision. Positive
      // result indicates a collision
      int hitIndex =
              testForCollision(sprite);
      if (hitIndex >= 0){
        //a collision has occurred
        bounceOffSprite(cnt,hitIndex);
      }//end if
    }//end for loop
  }//end update
  //----------------------------------//

  private int testForCollision(
                  Sprite testSprite) {
    //Check for collision with other
    // sprites
    Sprite  sprite;
    for (int cnt = 0;cnt < size();
                              cnt++){
      sprite = (Sprite)elementAt(cnt);
      if (sprite == testSprite)
        //don't check self
        continue;
      //Invoke testCollision method
      // of Sprite class to perform
      // the actual test.
      if (testSprite.testCollision(
                             sprite))
        //Return index of colliding
        // sprite
        return cnt;
    }//end for loop
    return -1;//No collision detected
  }//end testForCollision()
  //----------------------------------//

  private void bounceOffSprite(
                   int oneHitIndex,
                   int otherHitIndex){
    //Swap motion vectors for
    // bounce algorithm
    Sprite oneSprite =
        (Sprite)elementAt(oneHitIndex);
    Sprite otherSprite =
      (Sprite)elementAt(otherHitIndex);
    Point swap =
           oneSprite.getMotionVector();
    oneSprite.setMotionVector(
        otherSprite.getMotionVector());
    otherSprite.setMotionVector(swap);
  }//end bounceOffSprite()
  //----------------------------------//

  public void drawScene(Graphics g){
    //Draw the background and erase
    // sprites from graphics area
    //Disable the following statement
    // for an interesting effect.
    backgroundImage.
              drawBackgroundImage(g);

    //Iterate through sprites, drawing
    // each sprite
    for (int cnt = 0;cnt < size();
                                 cnt++)
      ((Sprite)elementAt(cnt)).
                     drawSpriteImage(g);
  }//end drawScene()
  //-----------------------------------//

  public void addSprite(Sprite sprite){
    add(sprite);
  }//end addSprite()

}//end class SpriteManager
//=====================================//

class Sprite {
  private Component component;
  private Image image;
  private Rectangle spaceOccupied;
  private Point motionVector;
  private Rectangle bounds;
  private Random rand;

  public Sprite(Component component,
                Image image,
                Point position,
                Point motionVector){
    //Seed a random number generator
    // for this sprite with the sprite
    // position.
    rand = new Random(position.x);
    this.component = component;
    this.image = image;
    setSpaceOccupied(new Rectangle(
          position.x,
          position.y,
          image.getWidth(component),
          image.getHeight(component)));
    this.motionVector = motionVector;
    //Compute edges of usable graphics
    // area in the Frame.
    int topBanner = (
                 (Container)component).
                       getInsets().top;
    int bottomBorder =
                ((Container)component).
                    getInsets().bottom;
    int leftBorder = (
                (Container)component).
                     getInsets().left;
    int rightBorder = (
                (Container)component).
                    getInsets().right;
    bounds = new Rectangle(
         0 + leftBorder,
         0 + topBanner,
         component.getSize().width -
```

```
          (leftBorder + rightBorder),
       component.getSize().height -
         (topBanner + bottomBorder));
}//end constructor
//----------------------------------//

public Rectangle getSpaceOccupied(){
  return spaceOccupied;
}//end getSpaceOccupied()
//----------------------------------//

void setSpaceOccupied(
            Rectangle spaceOccupied){
  this.spaceOccupied = spaceOccupied;
}//setSpaceOccupied()
//----------------------------------//

public void setSpaceOccupied(
                     Point position){
  spaceOccupied.setLocation(
             position.x, position.y);
}//setSpaceOccupied()
//----------------------------------//

public Point getMotionVector(){
  return motionVector;
}//end getMotionVector()
//----------------------------------//

public void setMotionVector(
                 Point motionVector){
  this.motionVector = motionVector;
}//end setMotionVector()
//----------------------------------//

public void setBounds(
                   Rectangle bounds){
  this.bounds = bounds;
}//end setBounds()
//----------------------------------//

public void updatePosition() {
  Point position = new Point(
   spaceOccupied.x, spaceOccupied.y);

  //Insert random behavior.  During
  // each update, a sprite has about
  // one chance in 10 of making a
  // random change to its
  // motionVector.  When a change
  // occurs, the motionVector
  // coordinate values are forced to
  // fall between -7 and 7.  This
  // puts a cap on the maximum speed
  // for a sprite.
  if(rand.nextInt() % 10 == 0){
    Point randomOffset =
       new Point(rand.nextInt() % 3,
                 rand.nextInt() % 3);
    motionVector.x += randomOffset.x;
    if(motionVector.x >= 7)
                 motionVector.x -= 7;
    if(motionVector.x <= -7)
                 motionVector.x += 7;
    motionVector.y += randomOffset.y;
    if(motionVector.y >= 7)
                 motionVector.y -= 7;
    if(motionVector.y <= -7)
                 motionVector.y += 7;
  }//end if

  //Move the sprite on the screen
  position.translate(
     motionVector.x, motionVector.y);

  //Bounce off the walls
  boolean bounceRequired = false;
  Point tempMotionVector = new Point(
                     motionVector.x,
                     motionVector.y);


  //Handle walls in x-dimension
  if (position.x < bounds.x) {
    bounceRequired = true;
    position.x = bounds.x;
    //reverse direction in x
    tempMotionVector.x =
                 -tempMotionVector.x;
  }else if ((
    position.x + spaceOccupied.width)
        > (bounds.x + bounds.width)){
    bounceRequired = true;
    position.x = bounds.x +
                bounds.width -
                 spaceOccupied.width;
    //reverse direction in x
    tempMotionVector.x =
                 -tempMotionVector.x;
  }//end else if

  //Handle walls in y-dimension
  if (position.y < bounds.y){
    bounceRequired = true;
    position.y = bounds.y;
    tempMotionVector.y =
                 -tempMotionVector.y;
  }else if ((position.y +
                spaceOccupied.height)
       > (bounds.y + bounds.height)){
    bounceRequired = true;
    position.y = bounds.y +
               bounds.height -
                spaceOccupied.height;
    tempMotionVector.y =
                 -tempMotionVector.y;
  }//end else if

  if(bounceRequired)
    //save new motionVector
                 setMotionVector(
                   tempMotionVector);
  //update spaceOccupied
  setSpaceOccupied(position);
}//end updatePosition()
//------------------------------------//

public void drawSpriteImage(
                         Graphics g){
  g.drawImage(image,
            spaceOccupied.x,
            spaceOccupied.y,
            component);
}//end drawSpriteImage()
//------------------------------------//

public boolean testCollision(
                   Sprite testSprite){
  //Check for collision with
  // another sprite
  if (testSprite != this){
    return spaceOccupied.intersects(
      testSprite.getSpaceOccupied());
  }//end if
  return false;
}//end testCollision
}//end Sprite class
//======================================//
```

**Listing 7**

---

**About the author**

**Richard Baldwin** *is a college professor and private consultant whose primary focus is a combination of Java and XML. In addition to the many platform-independent benefits of Java applications, he believes that a combination of Java and XML will become the primary driving force in the delivery of structured information on the Web.*

*Richard has participated in numerous consulting projects involving Java, XML, or a combination of the two.  He frequently provides onsite Java and/or XML training at the high-tech companies located in and around Austin, Texas.  He is the author of Baldwin's Java*

*Programming Tutorials, which has gained a worldwide following among experienced and aspiring Java programmers. He has also published articles on Java Programming in Java Pro magazine.*

*Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.*

*baldwin.richard@iname.com*

-end-