Fun with Java: Sprite Animation, Part 1

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 the first lesson in a miniseries that will concentrate on having fun while programming in Java.

Published: October 1, 2001
By Richard G. Baldwin

Java Programming, Lecture Notes # 1450

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 the first lesson in a miniseries that will concentrate on having fun while programming in Java.

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

Animation is fun

When it comes to having fun while programming, it's hard to beat a good old fashioned program that provides visual feedback and stimulation. And in that category, it's hard to beat an animation program.

This is the first of several 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. Once you know how to do animation, there are lots of ways to put that knowledge to use. For example, you could use that newfound knowledge to write some neat game programs. Or, you could take your newfound knowledge and use it to explore the world of *Artificial Life*.

Descriptions of upcoming programs

The first program that I will discuss in this and the next few lessons will show you how to write a program in which you animate a group of colored spherical sea creatures swimming around in a fish tank. A screen shot of the output produced by this program is shown in Figure 1.

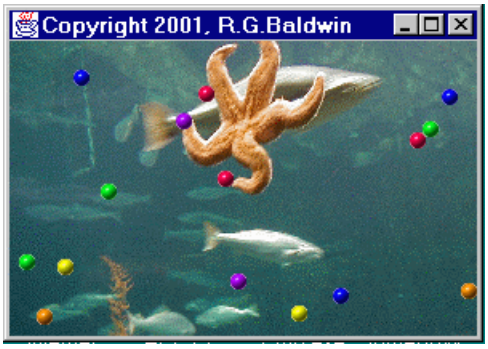

Figure 1. Animated spherical sea creatures in a fish tank.

Uses sprite animation

This program will use sprite animation to cause the spherical creatures to swim around. Of course, the screen shot doesn't do justice to the effect that you will see when you run the program in its animated form.

Using frame animation to change colors

If you watch The Discovery Channel or The Learning Channel very much, you will already know that many sea creatures have the ability to change their color in very impressive ways. The second program that I will discuss will simulate that process. It will use sprite animation to cause the spherical creatures to swim, and will also use frame animation to cause them to change their color at the same time. Since a screen shot can't show the creatures changing colors, a screen shot of the second program would look very similar to the screen shot in Figure 1 above. Therefore, I didn't provide a screen shot of the second program.

How about some sea worms?

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

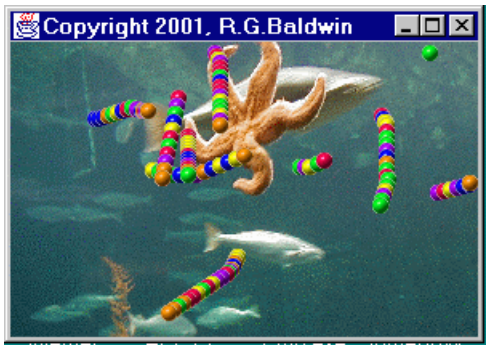

Figure 2. Animated sea worms in a fish tank.

This program 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 the real sea creatures that have this amazing ability to change the colors on their bodies do.

The required GIF files

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

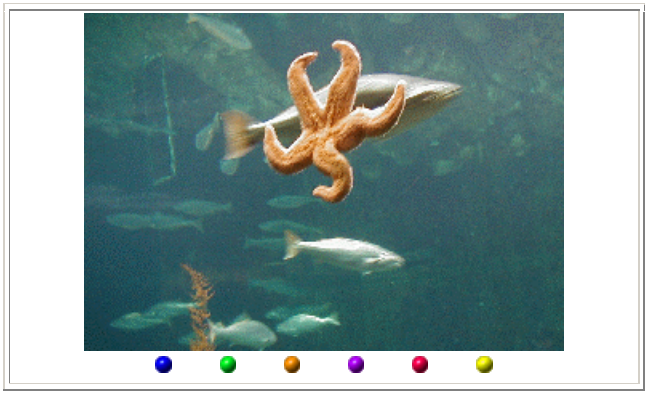

Figure 3. GIF image files that you will need.

You should be able to capture the various images from Figure 3 by right-clicking on them individually, and then saving them into files on your local disk.

Rename the captured files

Having done that, you will need to rename the files to match the names that are hard-coded into the programs *(or change the names in the programs to match the names of your files).*

Important classes

In this lesson, I will introduce you to several classes and concepts that you must understand in order to understand animation in Java.

Included in the classes that I will discuss will be the following, which are particularly important to sprite animation:

- Image
- Toolkit
- Graphics
- MediaTracker
- Random

Important concepts

I will also discuss 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

Preview of control structure

Here is a preview of the control structure that I will use for this animation program.

The controlling class extends the **Frame** class and implements the **Runnable** interface. Thus, an object of the controlling class is used to provide the visual manifestation of the program as a visual **Frame** object. An object of the controlling class is also suitable for using as an animation thread, which controls the overall behavior of the animation process. In other words, an object of the controlling class acts both as the director of the play, and the stage upon which the play is performed.

The **main** method of the controlling class instantiates an object of the controlling class, thus causing the constructor for the controlling class to be executed.

Objects of type Image

The constructor for the controlling class causes seven **Image** objects to be created. Each **Image** object is based on the pixel contents of a GIF file.

One of the **Image** objects is used to produce the background scenery against which the animation is played out. The other six **Image** objects are used to provide the visual manifestation of the sprites.

Each **Image** object provides the visual manifestation for more than one sprite. Therefore, some of the sprites look alike *(twins in some cases and triplets in others).*

Set the Frame size

After the **Image** objects have been created, the size of the **Image** object used for the background scenery is used by the constructor to set the size of the **Frame**. Then the **Frame** is made visible.

Start the animation thread

Finally, the constructor creates the animation thread and starts it running. From this point forward, the **run** method of the controlling class controls the animation behavior of the program.

The run method

The **run** method begins by creating and populating a **SpriteManager** object. An object of the **SpriteManager** class is capable of managing a collection of sprites, causing them to update their positions on demand, and dealing with collisions between the sprites.

The SpriteManager object

The **SpriteManager** object is populated with fifteen separate **Sprite** objects. Each sprite has a visual manifestation based on one of the six **Image** objects. Each sprite also has an initial position based on a random number and has a *motion vector* whose components are also based on random numbers. The motion vector is used to determine the next position of the sprite when the sprite is told by the **SpriteManager** to change its position.

The animation loop

Then the **run** method enters an infinite loop, iterating approximately twelve times per second. At the beginning of each iteration, the **SpriteManager** is told to update the positions of all of the sprites in its collection. It does so, dealing with collisions in the process.

The **run** method sends a message to the operating system asking it to repaint the **Frame** object on the screen.

The upDate method

When the operating system honors the request to repaint, it invokes the **upDate** method on the **Frame** object, *(which normally does some initialization and then invokes the paint method).*

The **update** method is overridden in this program to cause the new scene to be drawn in its entirety, showing each of the sprites in its new position superimposed upon the background image. Note that in this case, the **update** method does not invoke the **paint** method, because there is nothing for the **paint** method to do.

An offscreen image

When drawing the scene, the **update** method first draws the scene on an offscreen graphics context, and then causes the scene to be transferred from that context to the screen context. This is done to improve the animation quality of the program.

Discussion and Sample Program

That's enough of the preliminaries. It's time to get down to business and start discussing code.

A fairly long program

This is a fairly long program. It is so long, in fact, that several lessons will be required to discuss it fully. However, 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 6 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 immediately.

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 will discuss the controlling class in this lesson and defer my discussion of the other classes until future lessons. In fact, the controlling class itself is quite long, so I will partition the discussion of the controlling class into several consecutive lessons as well.

Acknowledgment

Before getting into the details, I want to acknowledge that some of the techniques used in this program, such as the animation timer and the collision detector, were taken from the book entitled Teach Yourself Internet Game Programming with Java in 21 days, by Michael Morrison.

The copy of the book that I have is the first edition *(I don't know if there are later editions)* and is somewhat dated by now *(for example, it uses the original JDK 1.0 event model)*. However, even though Java has been updated significantly since the publication of the book, some techniques discussed in the book are still appropriate for use.

In addition, the book provides a good discussion of the benefits of Object-Oriented Programming. That information is beneficial to anyone embarking on a career as a Java programmer.

The controlling class

The beginning of the class definition for the controlling class named **Animate01** is shown in Listing 1.

```
public class Animate01 extends Frame
                    implements
Runnable{
  private Image offScreenImage;
  private Image backGroundImage;
  private Image[] gifImages =
                           new
Image[6];
```

Listing 1

Extends the Frame class

As you can see, the controlling class extends the **Frame** class *(extending JFrame would work just as well provided that you take the Swing content pane into account).* This causes an object instantiated from the controlling class to be suitable as a drawing surface for the animation. Thus, the animation images are presented directly on the surface of the **Frame** as shown in Figure 1 and Figure 2.

Implements the Runnable interface

The controlling class also implements the **Runnable** interface. This makes it suitable for use as a **Thread** object. We will see later that the animation loop is actually implemented inside the **run** method of the controlling class.

The Image Class

The code in Listing 1 declares three reference variables. The first two are reference variables of the type **Image**. The third is a reference variable that refers to an array object containing six references to objects of type **Image**. From this, you might surmise that an understanding of the **Image** class is important to this type of animation, and if so, you are correct.

What does Sun have to say about the Image class?

Here is part of what Sun has to say about the **Image** class:

> *"The abstract class Image is the superclass of all classes that represent graphical images. The image must be obtained in a platform-specific manner."*

Because **Image** is an abstract class, we can't directly instantiate objects of the class. We will see later that we obtain our objects of type **Image** using a roundabout approach involving the **Toolkit** class. *(I will have more to say about that later.)*

The Toolkit class

For the time being, suffice it to say the **Toolkit** class makes it possible to gain access to *system-dependent* resources using *system-independent* code.

(Other examples of the use of the Toolkit class have to do with the system event queue, and access to system printers, which I discuss at length in other lessons.)

Getting Image objects

We will get our **Image** objects by invoking one of the overloaded **getImage** methods of the **Toolkit** class. Once we get an **Image** object, we really won't know the name of the class from which it was instantiated. Furthermore, we won't care about the name of the class from which it was instantiated. We will know simply that we can treat it as type **Image** and let polymorphic behavior take care of us.

(Hopefully, you already know all about polymorphic behavior. If not, I discuss it in detail in several other lessons, including the lessons on the Collections Framework.)

Using Image objects

The **Image** class *(and the classes that extend it)* define *(or override)* a number of useful methods that we will use throughout the program. This will include the methods named **getGraphics**, **getWidth**, and **getHeight**.

The Graphics class

The code in Listing 2 declares two more reference variables. Of particular interest at this point is the reference variable of type **Graphics**. This particular variable will be used to refer to an object that will serve as offscreen graphics context.

```
  private Graphics
                  offScreenGraphicsCtx;
  private Thread animationThread;

Listing 2
```

What is an offscreen graphics context?

Put simply, in this program, an offscreen graphics context is an area of memory that serves as a stand-in for the computer screen.

We use the methods of the **Graphics** class to draw pictures in that memory without disturbing the pictures currently showing on the computer screen.

Why use an offscreen graphics context?

Then we can blast the pictures from the offscreen graphics context to the actual computer screen very rapidly.

This is an important capability for animation. A noticeable amount of time is often required to create a picture. Because this approach doesn't disturb the visible image during the time required to create the picture, it usually results in smoother animation than can be achieved by creating and drawing the pictures directly on the computer screen. It eliminates the flashing and other distractions that can occur when the material is being displayed as it is being created.

What does Sun have to say about the Graphics class?

The sun documentation has quite a lot to say about the **Graphics** class in general. Here is a brief sampling:

> *"The Graphics class is the abstract base class for all graphics contexts that allow an application to draw onto components that are realized on various devices, as well as onto offscreen images."*

For example, printing in Java involves the use of methods of the **Graphics** class to draw pictures on the paper in the printer. It doesn't matter whether those pictures represent landscapes or letters; they are pictures nonetheless. In that sense, the printer paper can be thought of as a *graphics context*.

Our graphics contexts

In this program, we will be particularly interested in two graphics contexts:

- The computer screen.
- An offscreen image.

More info from Sun

Here is more of what Sun has to say about the **Graphics** class:

> *"A Graphics object encapsulates state information needed for the basic rendering operations that Java supports. This state information includes the following properties ..."*

Sun goes on to list several properties, which won't be too important to us in this lesson.

Location, width, and height

In this lesson, we will frequently be working with the location, width, and height of images. This requires some knowledge of how coordinate positions are treated. In this regard, Sun says:

"All coordinates that appear as arguments to the methods of this Graphics object are considered relative to the translation origin of this Graphics object prior to the invocation of the method."

What is a translation origin?

By default, the plotting origin of a graphics surface is the upper left-hand corner of the surface on which the plotting is being performed. That origin can be translated to a different spot *(the translation origin),* but none of the code in this lesson does that.

Positive horizontal coordinates progress from left to right across the graphics surface *(relative to the origin).* Positive vertical coordinates progress from top to bottom down the surface *(relative to the origin).*

(The translation origin for the images produced by this program is the upper-left corner of the Frame object.)

The drawImage methods

The **Graphics** class, *(and its subclass named Graphics2D)* provide dozens of methods that can be used to draw pictures on a graphics context. However, most of those methods have to do with drawing lines, circles, polygons, etc.

Only about eight methods are provided for drawing images, and most of those methods are overloaded versions of the method named **drawImage**. The **drawImage** method will surely become our friend in this and the next few lessons.

The Thread class

The other reference variable declared in the code in Listing 2 is of type **Thread**. Hopefully you already know all about Java threads. If not, I have published several lessons explaining the use of threads on my web site, and you should probably refer to them before getting too far into this program.

The MediaTracker class

The variable declaration in Listing 3 exposes one of the more abstract issues involved in this program, the **MediaTracker** class.

The primary purpose of the **MediaTracker** class is to help you deal with time delays that may occur when loading image data into memory. If the images are being loaded via the Internet, those time delays can be quite long. Even if the images are being loaded from a local hard drive, the delays can be long enough to be troublesome.

```
  private MediaTracker mediaTracker;
```

```
Listing 3
```

In other words, when you are using images, you need to know the load status of each image before you try to use it. If it hasn't finished loading, you must be careful what you try to do with it.

What does Sun have to say about MediaTracker?

Here is part of what the Sun documentation for JDK 1.3 has to say about the **MediaTracker** class:

> *"The MediaTracker class is a utility class to track the status of a number of media objects. Media objects could include audio clips as well as images, though currently only images are supported.*
>
> *To use a media tracker, create an instance of MediaTracker and call its addImage method for each image to be tracked.*
>
> *In addition, each image can be assigned a unique identifier. This identifier controls the priority order in which the images are fetched. It can also be used to identify unique subsets of the images that can be waited on independently. Images with a lower ID are loaded in preference to those with a higher ID number."*

How do you use a MediaTracker object?

Once you have registered an image with a **MediaTracker** object *(using the addImage method and identifying the image with a specific ID value)*, there are several methods that you can invoke on the **MediaTracker** object to learn the current status of the image.

Some of the methods allow you to manipulate the images in other ways, such as *unregistering* an image using the **removeImage** method.

The MediaTracker methods

Here is a partial list of the available methods *(note that, as usual, some of the methods have several overloaded versions).*

- checkAll
- checkID
- getErrorsAny
- getErrorsID
- isErrorAny
- isErrorID
- removeImage
- statusAll
- statusID

- waitForAll
- waitForID

The names of these methods are fairly descriptive, so you should be able to surmise what most of them do.

I will use some of these methods in this program to track the loading of GIF images that are used for the background graphic and the sprites.

The SpriteManager class

Listing 4 shows the declaration of three additional instance variables.

```
  private SpriteManager spriteManager;
  //Animation display rate, 12fps
  private int animationDelay = 83;
  private Random rand =
                new Random(System.
                  currentTimeMillis());

Listing 4
```

The **SpriteManager** class is defined in this program. As the name implies, an object of this class is used to manage the sprites involved in the animation process. This class will be discussed in detail in a subsequent lesson.

Animation repetition rate

The variable named **animationDelay** is used to control the repetition rate of the animation process.

As in the movies, or on TV, animation is achieved by presenting a series of pictures on the screen. Each picture represents a slightly different version of an object being animated.

(When I was a child, I used to create stick-man movies by drawing different versions of a stick-man doing acrobatics on the edges of the pages in a book. By rapidly flipping through the pages with my thumb and forefinger, I could animate the stick-man and cause him to do his acrobatics.)

What is the required repetition rate?

The pictures need to be presented at a sufficiently fast rate to fool the brain and give the illusion of continuous motion. On the other hand, presenting the pictures too rapidly simply wastes computer resources because the animation quality is not significantly improved.

Is twelve repetitions per second adequate?

The **animationDelay** variable in Listing 4 is initialized to a value of 83 milliseconds. This is used by the program to insert 83 milliseconds between repetitions of the animated sprites. This works out to approximately 12 repetitions per second. Many authors agree that this rate is a good compromise between too slow and too fast. However, only you can be the final judge of that.

Changing the repetition rate

To the extent that you computer can handle it, it isn't difficult to increase the repetition rate. Decrease the initialization value for the **animationDelay** variable to increase the repetition rate, or increase the value to decrease the repetition rate.

Divide the animationDelay value into 1 to get the repetition rate. Note, however, that if you make the animationDelay value too small, you computer won't be able to achieve the repetition rate specified by your new value for animationDelay. In that case, the computer will simply be displaying new pictures as fast as it can create them.

Pseudo-random numbers

As we go through the program, you will see a number of instances where a random number is needed for some purpose. The third reference variable in Listing 4 contains a reference to an object of the class **Rand**. Here is part of what the Sun documentation has to say about the **Rand** class:

> *"An instance of this class is used to generate a stream of pseudo-random numbers. The class uses a 48-bit seed, ...*
>
> *If two instances of Random are created with the same seed, and the same sequence of method calls is made for each, they will generate and return identical sequences of numbers."*

The converse is also true

Although it isn't explicitly stated in the Sun documentation, the converse of the second paragraph above is also true. In particular, if two instances of **Random** are created with different seeds, and the same sequence of method calls is made for each, they will generate and return different sequences of numbers.

In this program, I didn't want identical sequences of numbers. Therefore, in the code shown in Listing 4, the **Random** object was constructed using the current time in milliseconds *(relative to midnight on January 1, 1970)* as the seed. Using this approach, unless two **Random** objects are created within the same millisecond, they will produce different sequences of numbers.

In some cases, using time as a seed is inadequate. Other instances of **Random** are created at other places in the program using seed values based on something other than time.

What can you do with a Random object?

Once you have an object of the **Random** class, a number of methods are available that allow you to extract random numbers from the object.

For example, the method named **nextInt r**eturns the next pseudo random, uniformly distributed **int** value from a random number generator's sequence. This method will be used frequently, in conjunction with the modulus operator **(%)** to obtain random numbers that are uniformly distributed between the positive and negative values of a particular whole number *(between -8 and +8, for example).*

The main method

The code shown in Listing 5 is the **main** method for this application. This code simply creates a new instance of the controlling class.

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

Listing 5
```

This code, working in conjunction with the constructor and the **run** method of the animation thread starts the program running.

Summary

In this lesson, I have introduced you to several classes and concepts that you must understand in order to understand animation in Java.

I have introduced and discussed a number of classes used by the program. Included were the following, which are particularly important to sprite animation:

- Image
- Toolkit
- Graphics
- MediaTracker
- Random

I have 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

What's Next?

The next lesson in this series will pick up with a discussion of the constructor for the **Animate01** class.

Complete Program Listing

A complete listing of the program is provided in Listing 6.

```
/*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 6

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-