Fun with Java: Sprite Animation, Part 3

In this lesson, Baldwin discusses the unusual nature of the getter methods for the width and height properties of an Image object. He introduces and briefly discusses the concept of an ImageObserver object in conjunction with the getWidth and getHeight methods of an Image object. He shows you how to set the size of a Frame to be the same as the size of a background image in the Frame. He discusses the use of an object of the controlling class as an animation thread.

Published: October 15, 2001
By Richard G. Baldwin

Java Programming, Lecture Notes # 1454

- Preface
- Preview
- Discussion and Sample Programs
- Summary
- What's Next
- Complete Program Listing

Preface

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 third in of a group of lessons that will teach you how to write animation programs in Java. The first lesson in the group was entitled Fun with Java: Sprite Animation, Part 1. The previous lesson was entitled Fun with Java: Sprite Animation, Part 2.

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.

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.

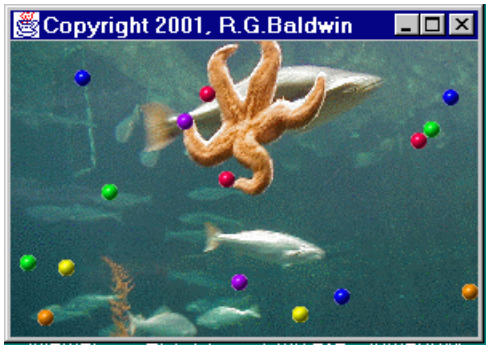

Figure 1. Animated spherical sea creatures in a fish tank.

Changing color using frame animation

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.

Multicolored 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.

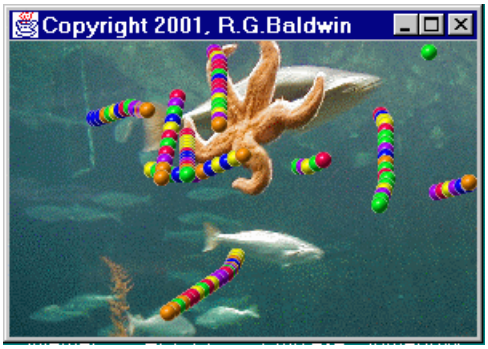

Figure 2. Animated sea worms in a fish tank.

Getting the GIF files

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

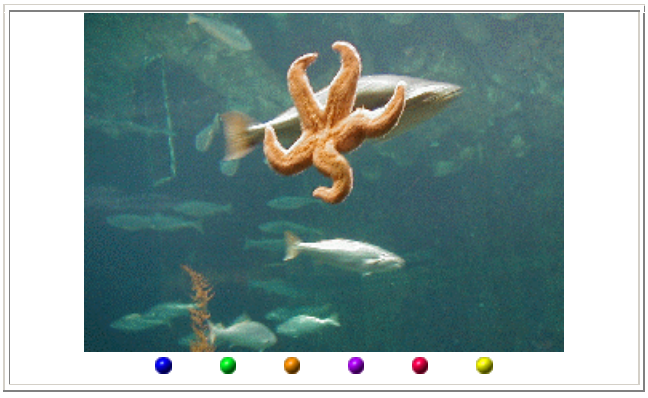

Figure 3. GIF image files that you will need.

You should be able to capture the 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.

Review of previous lesson

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

I also taught 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 taught you how to use the **waitForID** method of the **MediaTracker** class 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 in this lesson?

In this lesson, I will discuss the unusual nature of the *getter* methods for the **width** and **height** properties of an **Image** object.

I will introduce and briefly discuss the concept of an **ImageObserver** object in conjunction with the **getWidth** and **getHeight** methods of an **Image** object.

I will show you how to set the size of the **Frame** to be the same as the size of the background image.

I will discuss the use of an object of the controlling class as an animation thread.

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 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.

Will discuss in fragments

As usual, I will discuss the program in fragments. At the close of the previous lesson, I was discussing the constructor for the controlling class. In this lesson, I will pick up where I left off before, and will complete my discussion of the constructor for the controlling class.

The background image

If you scroll back and take a look at Figure 1, you will see that the background image completely fills the **Frame** object used as a drawing surface for this program. That was no accident. It was planned that way. If you were to substitute a different background image, the size of that image would be used to control the size of the **Frame**.

As it turns out, this is not a trivial operation, at least from an understanding viewpoint.

Sizing the Frame

The code in Listing 1 begins the process of using the size of the background image to set the size of the **Frame**. This code gets the width and height of the background image.

```
    int width =
        backGroundImage.getWidth(this);
    int height =
       backGroundImage.getHeight(this);
```

```
Listing 1
```

You will recall that the code in the constructor at the end of the previous lesson used the **getImage** method of the **Toolkit** class to read seven GIF files from the disk. *(One of those GIF files, named background02.gif, contains the data used to construct the background image.)*

The **getImage** method uses the pixel contents of a GIF file to create an **Image** object containing those pixels. One of the **Image** objects is used to draw the background with the fish and the starfish shown in Figure 1.

This code is strange

The code in Listing 1 is not completely straightforward. Can you spot anything in code that looks a little strange, or at least a little unusual? For example, how do the methods in Listing 1 compare with typical *getter* methods that are commonly used to obtain the values of properties?

The getter method requires a parameter

The thing that is unusual about this code is the fact that a reference to the object being constructed **(this)** is passed as a parameter to the **getWidth** and **getHeight** methods of the **Image** object.

This is not your typical *getter* method for a property. A typical *getter* method for a single-valued property doesn't usually require any parameters.

The timing can be critical

The reason for this unusual syntax results from the fact that it is possible to invoke the **getWidth** and **getHeight** methods before the image is completely loaded. In that case the value of the width and height properties may not yet be known.

Two different return values are possible

The **getWidth** and **getHeight** methods of the **Image** class can each return two different values, depending on when they are called relative to the loading of the image.

If they are called before the width and height information becomes available during the load process, one or both methods will return -1. If they are called after the width and height information becomes available, they will return the actual values of those properties.

An ImageObserver object

The signature of the **getWidth** method is shown below:

```
public abstract int getWidth(
                ImageObserver observer)
```

If you examine this signature carefully, you will see something that I haven't previously discussed. The method requires a parameter of type **ImageObserver**. I will have more to say about that shortly.

What does Sun have to say?

Right now, let's see what Sun has to say about the **getWidth** method of the **Image** class.

> *"Determines the width of the image. If the width is not yet known, this method returns -1 and the specified ImageObserver object is notified later."*

An examination of the documentation for the **getHeight** method reveals a similar description. This confirms what I said earlier about the possibility that the method can return either -1 or the actual property value.

So, what is an ImageObserver?

I was afraid you were going to ask that. To begin with, **ImageObserver** is an interface and not a class. Therefore, an **ImageObserver** object is any object instantiated from any class that implements the **ImageObserver** interface.

Frame implements ImageObserver

The object being constructed by this constructor is a **Frame** object. **Frame** extends **Component**, and **Component** implements **ImageObserver**. Therefore, an object of the **Frame** class implements **ImageObserver** through inheritance. An object of the **Frame** class is qualified to be passed as a parameter to any method that requires an incoming parameter of type **ImageObserver**.

Where am I going with this discussion?

On my first attempt to write this lesson, I tried to explain how **ImageObserver** fits into the grand scheme of things so that you would understand the underlying processes. However, the more I wrote, the longer my text became. Although I understand how it all fits together, I don't know how to explain how it fits together in a small amount of text. *(Therefore, I plan to dedicate an entire lesson to the concept of an ImageObserver later on.)*

Take it on faith

Although I hate to do so, this is one of those cases where I am simply going to say, *"take it on faith."* If the **getWidth** method is invoked on an **Image** object, *(passing an ImageObserver as a parameter)*, the method will either return the width of the image or will return -1.

If the image is loaded to the point that the width information is available, the value of the width will be returned. If the width information is not yet available, a value of -1 will be returned.

Therefore, after the code in Listing 1 is executed, the variables named **width** and **height** will either contain the width and height values, or will contain -1.

Previously invoked waitForID

In the code that I explained in the previous lesson, I invoked the **waitForID** method on the **MediaTracker** object to force the constructor to block and wait for all seven of the images to be loaded. As a result, assuming that the proper GIF files are available and no loading errors occur, I don't believe that it is possible for these calls to **getWidth** and **getHeight** to return -1.

waitForID will return on a loading error

However, as I explained in the previous lesson, the **waitForID** method will also return when there is an error in loading the images. It is possible for the **getWidth** and **getHeight** methods to return -1 in that case in this program.

Dealing with the problem

The code in Listing 2 is a crude attempt to deal with that possibility. If **waitForID** returns due to a loading error, the width or height values will not be available. The code in Listing 2 will simply become an infinite loop displaying **Waiting for image** on the standard output device until the program is terminated.

```
    while(width == -1 || height == -1){
      System.out.println(
                    "Waiting for image");
      width = backGroundImage.
                          getWidth(this);
      height = backGroundImage.
                         getHeight(this);
    }//end while loop

Listing 2
```

Not very robust

This is not a very robust approach to dealing with the problem. A more robust approach would be to invoke some of the error testing methods of the **MediaTracker** class to identify an error condition before entering this loop, and to avoid entering the loop if a loading error has occurred. *(If you use this animation code in any serious program, you should give some thought to improving it and making it more robust in this area.)*

Continuing on

Assuming that control doesn't get trapped in the above loop as a result of an image loading error, the code in Listing 3 uses the resulting width and height property values of the background image to set the width and height of the **Frame** object. The code in Listing 3 also puts a title on

the **Frame** and makes it visible.

```
    setSize(width,height);
    setVisible(true);
    setTitle(
        "Copyright 2001,
R.G.Baldwin");

Listing 3
```

Some background image is hidden

The code to this point has caused the size of the entire **Frame**, including its banner and its borders, to be the same size as the background image. Consequently, the edges of the background image won't be visible because they will be hidden by the **Frame's** borders and banner.

(This topic will come up again later. I will use the getInsets method of the Frame object to cause the spherical sea creatures to bounce off the inside edges of the borders and the banner. Without that correction, they would appear to slide under the borders and the banner and to bounce off the outside edges of the Frame.)

We're just about finished with our discussion of the constructor for the controlling class.

A Thread object

The controlling class implements the **Runnable** interface, meaning that the class must provide a definition for the method named **run**. As such, an object of the controlling class is capable of running as a thread. *(I have written other lessons explaining the use of multi-threaded programming in Java, so I won't go into a lot of detail here.)*

In this case, the **run** method of the controlling class actually causes the animation to take place, so I refer to it as the animation thread.

The code in Listing 4 instantiates a new **Thread** object and invokes the **start** method on that object. Invoking the **start** method on a **Thread** object causes a few housekeeping chores to be taken care of first, and then causes the **run** method of the **Thread** object to be invoked.

```
    animationThread = new
Thread(this);
    animationThread.start();

Listing 4
```

What does the run method do?

The behavior of the **run** method of the controlling class will be the topic for one or more future lessons.

An anonymous WindowListener object

Finally, the code in Listing 5 instantiates an anonymous **WindowListener** object of an anonymous inner class. This listener object is registered on the **Frame** to cause the program to terminate when the user clicks the *close* button on the **Frame**. *(I also discuss event driven programming extensively in other tutorial lessons, so I won't go into any further details here.)*

```
    this.addWindowListener(
                    new
WindowAdapter(){
      public void windowClosing(
                          WindowEvent
e){
        System.exit(0);}});

  }//end constructor

Listing 5
```

The closing curly brace in Listing 5 signals the end of the constructor, and it also signals the end of this lesson.

Summary

In this lesson, I discussed the unusual nature of the *getter* methods for the **width** and **height** properties of an **Image** object.

I introduced and briefly discussed the concept of an **ImageObserver** object in conjunction with the **getWidth** and **getHeight** methods of an **Image** object. This is a topic that will come up again in a subsequent lesson in conjunction with the **drawImage** method of the **Graphics** class.

I showed you how to set the size of the **Frame** to be the same as the size of the background image.

I discussed the use of an object of the controlling class as an animation thread.

What's Next?

The next lesson will begin the explanation of the **run** method of the controlling class, which controls the animation behavior of the program.

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-