Creating Motion in a Java GUI Program

Previously, we discussed drawing various objects onto a JComponent and attaching that JComponent onto a JFrame. Unfortunately, video games with no motion aren't very fun. In order to achieve motion, we need the ability to draw many pictures onto a JComponent, over and over again, with each subsequent picture containing some small changes as compared to the previous picture.

In Java, to do this, we must have our JComponent use a thread. Runnable is a Java interface that implements a thread. Here is our set-up from bouncingball.java:

```
class MyComponent extends JComponent implements Runnable {

	private ball soccer;

	public MyComponent() {
		soccer = new ball(new Random());
		soccer.initialize();
		Thread t = new Thread(this);
		t.start();
	}
}
```

In this example, the ball class will store all the information about our moving ball. We will create this object, initialize it and then start our thread. When we implement Runnable, we are required to write a run method. For our run method, we'll continuously loop while updating our ball object and then repaint our screen. Here is the remainder of the MyComponent class:

```
	public void run() {
		while (true) {
			soccer.update();
			repaint();
			try {
				Thread.sleep(5); // Slows it down.
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}

	public void paintComponent(Graphics g) {
		soccer.draw(g);
	}
```

In our public class, we will have to do the same exact thing we did when statically drawing: create a JFrame and then add our JComponent to it. Here is the whole main in bouncingball that accomplishes this:

```
public class bouncingball {
	public static void main(String[] args) {
        JFrame frame = new JFrame("Bouncing Ball");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(ball.MAXX, ball.MAXY);
        MyComponent view = new MyComponent();
        frame.add(view);
        frame.setResizable(false);
        frame.setVisible(true);
    }
}
```

Once we have these two classes set up, we just have to define our own code in the ball class. The ball class handles creating our ball object, defining the draw method that we call from the MyComponent class in the paintComponent method and executing the update method that we call inside the run() method of the MyComponent class. As you might imagine, draw's job is to draw, and update's job is to change the instance variables to get ready for the next frame.

<u>Creating Motion</u>
To create motion, what we need to do is make sure that each time the draw method is getting called, it's drawing something slightly different than when it was called before. To understand one way to accomplish this, let's first look at all the instance variables in the ball class related to motion:

```
int positionX, positionY, velocityX, velocityY;
```

The first two represent the current position of the ball (top left corner) while the last two represent how much the ball will move in between "frames". (In between two different drawings.

All our draw method cares about is where the ball is now. It's just is just to draw it there. In addition to knowing its position, this method has access to the size of the ball. Here is the draw method from the ball class:

```
public void draw(Graphics g) {
    g.setColor(this.backgroundColor);
    g.fillRect(0, 0, MAXX, MAXY);
    g.setColor(this.ballColor);
    g.fillOval(positionX, positionY, sizeX, sizeY);
}
```

All we do here is first draw the background (after setting its color) and then draw the ball after setting its color. (Both backgroundColor and BallColor are instance variables in the ball class as well.)

Thus, the key is that the update method will ***<u>change</u>*** the position of the object. Here is the first part of the update function that accomplishes this task:

```
public void update() {

    positionX = positionX + velocityX;
    positionY = positionY + velocityY;
```

If we just used this code, eventually, the ball would just go off the screen, since it would keep on heading in the same direction. But, to make things interesting, we'll make it bounce off all four walls of the screen. To do this, we simply notice that any time the ball bounces off the left or right walls, we just want to redirect its dx to be opposite of what it was previously. Similarly, any time the ball bounces off the top or bottom walls, we just want to redirect its dy to be opposite of what it was previously. Here is the full update method with this logic included:

```
public void update() {

      positionX = positionX + velocityX;
      positionY = positionY + velocityY;

      if (positionX >= MAXX-sizeX || positionX < 0)
            velocityX = -velocityX;
      if (positionY >= MAXY-sizeY-BORDER || positionY < 0)
            velocityY = -velocityY;
}
```

Basically, if you're x is at either edge (0 or the right end), then we just negate your velocity in X. We do the same for Y. The check on the right end and bottom end of the screen seems a bit strange because positionX, positionY represents the ***top left*** corner of the ball. If we didn't subtract an offset, it would look like the ball was bouncing somewhere ***below*** the bottom and right side of the window.

The last piece of our puzzle is the constructor of the ball class that sets up a default ball object. This just needs to initialize all of the instance variables. Here is a list of the instance variables and the constructor:

```
class ball {

      final public static int MAXX = 500;
      final public static int MAXY = 500;
      final public static int BORDER = 25;

      int positionX, positionY, velocityX, velocityY;
      int sizeX, sizeY;
      Color ballColor, backgroundColor;

      public ball(Random r) {
            positionX = r.nextInt(400);
            positionY = r.nextInt(400);
            velocityX = r.nextInt(5)-2;
            velocityY = r.nextInt(5)-2;
            if (velocityX == 0) velocityX++;
            if (velocityY == 0) velocityY++;
            sizeX = 50;
            sizeY = 50;
            ballColor = Color.black;
            backgroundColor = Color.white;
      }
}
```

We just assign each instance variable as we would like for the ball to be. I make sure the ball is moving in both X and Y and set the ball color to black and the background to white.

Preventing Moving off of the Screen

Another simple way of preventing an object from leaving the screen instead of having it bounce back is to have it get "stuck" at a boundary. Here is some sample code (in x) that does this:

```
if(c.x > this.getWidth()) c.x = this.getWidth();
```

This way, if we ever go out of bounds, we can just place the character right back at the boundary. This ends up looking nice when painted in the application, because we never paint the object out of bounds.