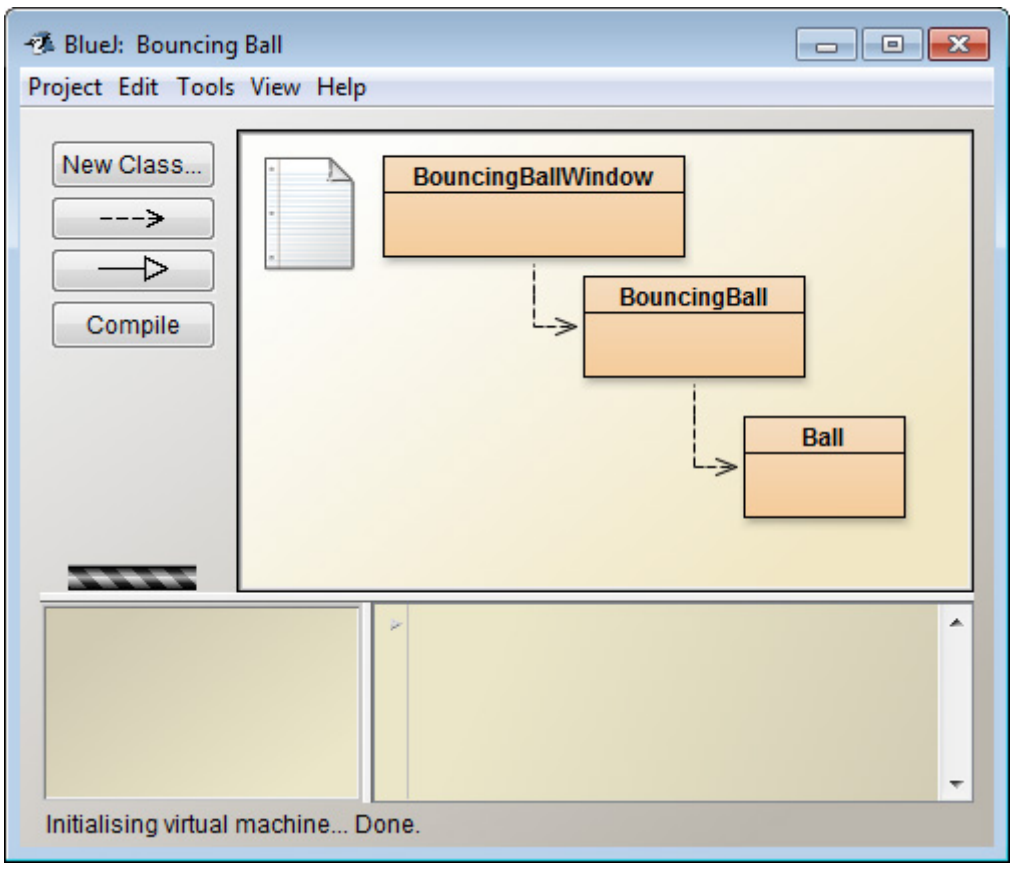
Chapter 2: Animating One Ball

Animating One Ball

We'll start by animating one ball. I'll say up front that like most projects, there are many ways to do this. I've chosen the one that I like best. So if you see other implementations of animations like this one, please understand that you're not locked into my way of doing things.

I'll start by showing you my BlueJ project window because I've added a third class to the drawing process:



Bouncing ball project window

The top two classes in the window pretty much fill the same purpose as the two classes we used in our drawings in the last lesson. There's only one change to the top-level class, BouncingBallWindow, other than class names, and I'll mention it in just a minute.

The middle class, BouncingBall, is the one that provides our JPanel object to draw in, just like in the examples from Lesson 11. But in this case there are several changes from the last lesson. All the code that controls the animation is in this class, and we'll talk about it in detail soon.

The third class, Ball, is new. It represents a single ball that can bounce around a window. It contains all the information on the ball's size, color, speed, and position. It also includes code to draw itself into the panel that BouncingBall manages. We'll see how all that works after we discuss the first two classes.

BouncingBallWindow Changes

Here's the code for BouncingBallWindow:

```
import javax.swing.*;
/**
 * BouncingBallWindow is the top level of three that produce a window with
 * an animated bouncing ball in it.
 *
 * @author Merrill Hall
 * @version 1
 */
public class BouncingBallWindow extends JFrame {
    public BouncingBallWindow() {  // Create window
        setTitle("One Bouncing Ball");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        add(new BouncingBall());  // Add bouncing ball panel

        pack();  // Set the window size
        setLocationRelativeTo(null);  // Center the window in the display
    }

    public static void main(String[] args) {
        // Run GUI in the Event Dispatcher Thread (EDT)
        SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    // Set up main window (using Swing's Jframe)
                    BouncingBallWindow bbw = new BouncingBallWindow();
                    bbw.setVisible(true);
                }
            });
    }
}
```

The only change other than class names and the window title is the next-to-last line in the constructor, which calls the pack() method. I used this method for a couple of reasons. One is to show you how it works. It sets the size of the window as small as it can be and still display all its contents. So pack() sets the window size based on what's in the window. The other reason is—as we'll see when we get to the next class—that I need to know the size of the panel for drawing before I can get the size from the frame, just as we did in Lesson 11.

And that's the total of the changes to our frame code. That was simple, wasn't it?

BouncingBall Class

Let's move on to the second class. It'll be a bit more involved, I promise. Take a look at my code for it, and then we'll talk about it:

```
import java.awt.*;
import javax.swing.*;
/**
 * BouncingBall produces a panel with one ball bouncing off the edges.
 *
 * It uses a single instance of the Ball class to provide the bouncer.
 */
public class BouncingBall extends JPanel {
    private final Dimension SIZE = new Dimension(600,450); // Panel size
    private final int UPDATE_RATE = 24; // Number of refreshes per second
    private Ball ball; // The beautiful bouncing ball
    private Thread ballThread; // Animation thread

    /**
     * Constructor sets up the panel and starts its animation thread
     */
    public BouncingBall() {
        setPreferredSize(SIZE); // Set panel size
        ball = new Ball(SIZE); // Create a Ball object
        // Start the ball bouncing (in its own animation thread)
        ballThread = new Thread() {
                public void run() {
                    while (true) { // Execute one update step
                        // Calculate the balls' new positions
                        ball.move();
                        // Refresh the display
                        repaint(); // Callback paintComponent()
                        // Delay so the animation's not too fast
                        try {
                            Thread.sleep(1000 / UPDATE_RATE);  // milliseconds
                        } catch (InterruptedException ex) { }
                    }
                }
            };
        ballThread.start();  // Start the animation
    }

    /**
     * Override the JPanel paintComponent method with our drawing of a ball
     */
    public void paintComponent(Graphics g) {
        super.paintComponent(g);    // Paint background
```

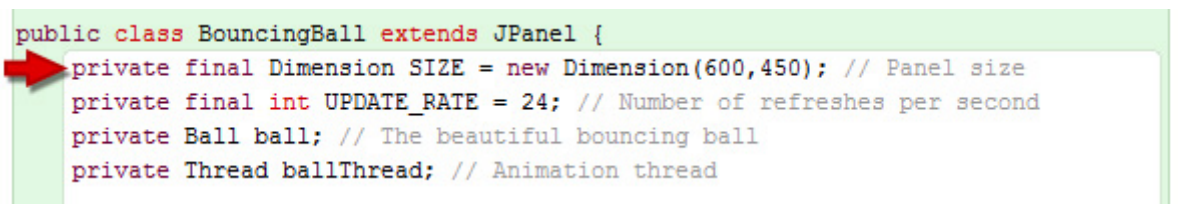
```
        Graphics2D g2d= (Graphics2D) g;
        // Draw the box's background
        g2d.setColor(Color.BLACK);
        g2d.fillRect(0, 0, getWidth(), getHeight());

        // Draw the ball
        ball.paint(g2d);
    }
}
```

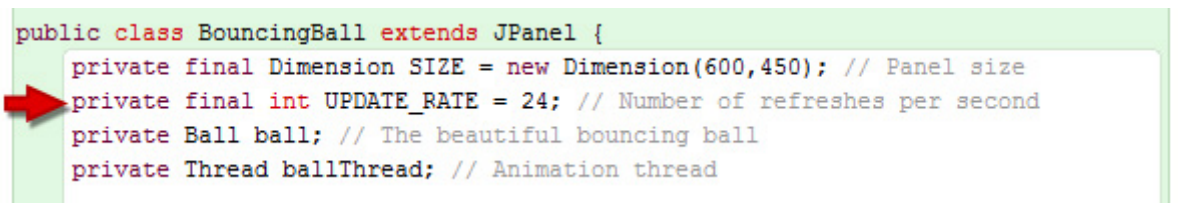
Let's take it from the top!

The first four lines in our class declare two constants and two variables that we'll use in the animation.

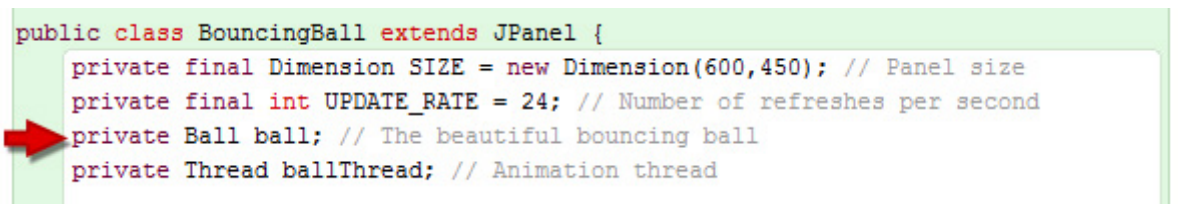
The first constant, SIZE, is a Dimension object that we'll use to set the size of our panel. Note that the name is in all uppercase letters. If you remember our discussion of Java names back in Lesson 2, you'll know that constants are named that way.



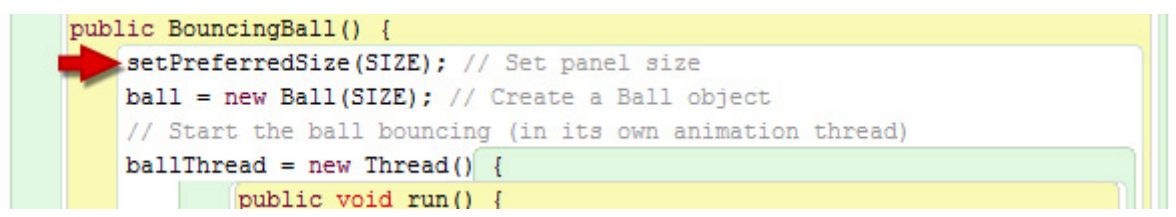
The second constant, UPDATE_RATE, is an integer that will control how many times per second we refresh the screen. If we do it too often, our ball will move too fast. If we don't do it often enough, and our ball will move too slowly. I set it at 24, the same refresh rate that most films use. It's fast enough that your eyes won't notice the pauses between refreshes.



The first variable, ball, refers to a Ball variable that will control the movement and appearance of our ball. The second variable, ballThread, refers to the Thread object that will run our animation. If you remember from Lesson 11, Java recommends that you do GUI work in a separate thread. It's also common practice to put animation into its own thread, so that's what we'll do here.



Following the instance variable declarations is our constructor. Its first line calls a method we haven't seen before, setPreferredSize(). Just as its name suggests, it's the method that sets our panel's size, which the frame will use to determine its size.



In order to use the panel's size in the constructor, I needed to know its size before it's available from the outer frame. So I needed to set it in this class rather than in BouncingBallWindow. That's why I used the pack() method there instead of setSize().

```
import javax.swing.*;
/**
 * BouncingBallWindow is the top level of three that produce a window with
 * an animated bouncing ball in it.
 *
 * @author Merrill Hall
 * @version 1
 */
public class BouncingBallWindow extends JFrame {
    public BouncingBallWindow() {  // Create window
        setTitle("One Bouncing Ball");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        add(new BouncingBall());  // Add bouncing ball panel

        pack();  // Set the window size
        setLocationRelativeTo(null);  // Center the window in the display
    }

    public static void main(String[] args) {
        // Run GUI in the Event Dispatcher Thread (EDT)
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                // Set up main window (using Swing's Jframe)
                BouncingBallWindow bbw = new BouncingBallWindow();
                bbw.setVisible(true);
            }
        });
    }
}
```

The second line of the method creates our Ball object. It passes it the size of the screen so the ball will "know" how much room it has to bounce around in. We'll see how Ball uses that information when we look at the Ball class in the next chapter.

The rest of the constructor creates and starts the animation thread ballThread. It's similar to the Runnable object we used in the main() method of the first class. It also has a run() method that controls the actions the thread will take. In our run() method this time, we want to repeatedly draw our window as long as the window is open, so we set up a simple while(true) loop that will repeat until the user shuts down the program.

An Animation Needs Its Own Thread

Rule

Why do we want to run our animation in its own thread? Here's a general rule in Java: Anything you do in a Java program that's independent, that runs continually, and that takes up a lot of processing time should run in its own thread.

Our animation is a loop that runs constantly and can run on its own without depending on other code. That meets two out of the three criteria. Our animation doesn't take up a lot of processing time, since it's sleeping most of the time (as we'll see in a minute). But more complex animation can take a *lot* of processing, so it's a good idea to get used to running animations in their own threads.

Inside our animation loop we do three things. First we call the ball's move() method to move the ball slightly. The ball knows how fast it's supposed to move and in what direction, so we let it take care of that.

Then we call JPanel's repaint() method to redraw the window. The repaint() method will call paintComponent() at the right time to do our drawing.

After that the loop pauses for some number of milliseconds (thousandths of a second) to control the animation speed. If we didn't include this pause, the ball would move too fast to appear to be smooth movement. To pause, we call the thread's sleep() method and tell it how many milliseconds to wait before waking up and running again. Since that method can throw an exception if something interrupts it, we need to enclose it in a try block. But since there's nothing going on in our program to cause an interruption, the catch block is just empty.

The last statement in our constructor is a call to the thread's start() method to start the animation running.

The last part of our BouncingBall class is our paintComponent() method, which works the way it did in our previous lesson to paint our image. First it paints a black background in the window. Then comes the only difference I want to point out. Instead of painting the whole image in this method, we call the ball's paint() method, which knows where to paint the ball in the window. In order for it to do that, though, it needs the panel's Graphics2D object, so we pass it to the method as an argument.

That covers our middle class. The only complicated new piece is the thread code in the middle of the constructor. Please let me know in the Discussion Area if you have any questions about it.

Now let's look at the Ball class and see how it does what it does.