Chapter 9: Simulation and animation

A. Introduction

This chapter will present a program that implements a discrete-time simulation and animation of a ball being dropped and bouncing. Perhaps that sounds daunting, but it is relatively simple once you know how. The only new programming construct required for simulation and animation is threads. The only new concept is discrete time simulation, which may turn out to be more familiar than you expect.

B. An introduction to Threads

As discussed throughout, information processing is accomplished by sending messages to objects. A message invokes a method; first the parameter linkage is performed, then the method body is executed. When a method body is executed, first the first statement is executed, then the second, and so on, until the last statement finishes execution; then the method returns -- returns control to wherever the message that invoked it was sent.

Imagine the path of execution through a program over time. It includes the sequential execution of statements in method bodies and the transfer of control from one method to another by sending messages.

At any one time, one particular statement is being executed. If it is a message statement, it invokes a method, which executes until it reaches the end of its block statement. For example, in the Eye Applet, when the user pushes the shrinkButton, the run-time system invokes its `ActionPerformed()` method (see Code Example 9.1), which sends a

Code Example 9.1

```
1 private void shrinkButtonActionPerformed(java.awt.event.ActionEvent evt) {
2     leftEye.shrinkPupil();
3     rightEye.shrinkPupil();
4     repaint();
5 }
```

`shrinkButtonActionPerformed()` from EyeApplet.

`shrinkPupil()` message to first the leftEye, and then the rightEye, and then finally sends a `repaint()` message to itself (since `repaint()` is changed to `this.repaint()` by the compiler.

The `shrinkPupil()` method, in the Eye class (see Code Example 9.2) sends a `setRadius()`

Code Example 9.2

```
1    public void shrinkPupil() {
2        pupil.setRadius(pupil.getRadius() - 2);
3    }
```

`shrinkPupil()` from the Eye class

message to the pupil object, which is a FilledCircle, but to do the parameter linkage, first it must evaluate the parameter, `pupil.getRadius() - 2`. To do that, it first sends the `getRadius()` message to the pupil object, and then subtracts 2 from whatever value that returns. Both `getRadius()` and `setRadius()`, although sent to a FilledCircle, end up in Circle (as FilledCircle inherits them from Circle).

So, the sequence of messages sent (and methods executed) is:

```
1.EyeApplet:actionPerformed()
2.         Eye:shrinkPupil()
3.                        Circle:getRadius()
4.                        Circle:setRadius()
5.         Eye:shrinkPupil()
6.                        Circle:getRadius()
7.                        Circle:setRadius()
8.         EyeApplet:repaint()
```

This sequence of statements is executed in order; each statement has control while it is executing. If you were to print the whole program, tape all the sheets together, and draw a line from the first statement executed to the second, on to the third, through the 8th; you would have a record of the path of execution when the shrinkButton is pushed. It would lead from one method to another across various classes, and would look a bit like a thread running through fabric. By following this path you can see which statements in the program are executing in which order. This imaginary line, this path through the program, is referred to as the ***thread of control,*** or thread, for short.

Each program you have seen thus far has had a single explicit thread, but to implement animation a second thread is needed. Fortunately Java has a built-in Thread class you can extend. Here's how.

i) Simplest threaded animation

The simplest threaded animation is an animated counter. It has two classes, an Applet and a Thread. The Applet creates and kicks off a Thread that repeatedly increments a counter (in the Applet) and has the Applet display its current value. To begin implementation of this example, create a GUI Applet named ThreadedApplet. Next, create a Controller class and copy the code from Code Example 9.3. The Controller class has a single instance

Code Example 9.3

```
1  public class Controller extends Thread {
2     private ThreadedApplet theApplet;
3
4      /** Creates a new instance of Controller */
5     public Controller(ThreadedApplet theApplet) {
6          this.theApplet = theApplet;
7      }
8
9     public void run() {
10         for(;;) {
11             step();
12             pause();
13         }
14     }
15
16     private void step() {
17         theApplet.incCounter();
18         theApplet.repaint();
19     }
20
21     private void pause() {
22         try {
23             Thread.sleep(50);
24         } catch (Exception e) {}
25     }
26 }
```

Threaded Controller class

variable named theApplet, which is used to both increment the counter in the Applet and to send the `repaint()` message to it (in the `step()` method).

a) Controller(ThreadedApplet)

The initializing constructor takes a single parameter, of type ThreadedApplet, and simply stores it in the theApplet instance variable. This is a standard technique to establish a two-way linkage between objects; the Applet object has a Controller variable which points to the Controller object and the Controller object has a ThreadedApplet variable which points back to the Applet. It is common to need a reference back to the object that created another object and that is how it's done.

b) run()

When the `start()` message is sent to the Controller from the Applet, because Controller extends Thread, it first ***spawns*** a new thread of control, and then sends the Controller in that Thread the `run()` message. This new Thread exists until the `run()` method returns. The run method here is an infinite loop (see "An infinite for loop" on page 205). It runs forever (or until the user closes the Applet), and it does just two things, `step()` and `pause()`, over and over.

c) step()

The `step()` method sends first `incCounter()` and then `repaint()` to the Applet. The pause method does only one thing, it sleeps for 50 milliseconds, but it looks rather complicated because the `sleep()` method, which is sent to the Thread class throws an exception, which must be caught (see ???). The simplest thing to do right now, is anytime you need a program to pause, simply copy and invoke this method. The `Thread:sleep()` method takes an int parameter and sleeps for that many milliseconds.

Add the bold lines in Code Example 9.4 to your Applet. That's all it takes! Run it, and if

Code Example 9.4

```
1 public class ThreadedApplet extends java.applet.Applet {
2     private Controller theController;
3     private int counter;
4     public void incCounter() {counter++;}
5
6     /** Initializes the applet BallApplet */
7     public void init() {
8         initComponents();
9
10        theController = new Controller(this);
11        theController.start();
12    }
13
14    public void paint(java.awt.Graphics g) {
15        g.drawString(""+counter, 100,100);
16    }
```

Changes to the ThreadedApplet class

Line 2: A variable of type Controller; this is the new Thread.
Line 3: A counter; will start, by default as 0.
Line 4: A method to increment (add 1 to) the counter; the Controller will use this.
Line 10: Create and store the Controller. Notice `this` as a parameter; `this` is the Applet.
Line 11: Spawn a new Controller thread and send the `run()` message to it.
Lines 14-16: The `paint()` method; draws the counter value in the Applet.

you've made no mistakes you should see a counter counting... forever. Close the Applet to make it stop.

To make it seem more animated modify `paint()` as in Code Example 9.5. Or, if you'd

Code Example 9.5

```
1    public void paint(java.awt.Graphics g) {
2        g.drawString(""+counter, 100,100);
3        drawOval(counter%200, counter%200, counter%177, counter%177);
4    }
```

Addition to the ThreadedApplet:paint() to draw a circle

like the circle to stay the same size, use a constant (like 20) for the 3rd and 4th parameters. Do you see why this does what it does? Recall that the % operator gives the remainder after integer division (see "Arithmetic operators" on page 119), so the first two parameters range from 0-199. When counter is 200, or 400, or any even multiple of 200, `counter%200` is zero, that's why the circle goes back to the upper left periodically.

C. The programming task

Your task for this chapter is to implement and animate a user controlled simulation of a ball falling under the influence of gravity and bouncing until it stops. Let the user start and stop the simulation by pressing a button. Make the radius of the ball 20 pixels and its color red. Assume the elasticity of the collision with the floor is 90%; thus if the downward velocity of the ball were 100 when it hit the floor, its subsequent upward velocity would be 90.

i) Design

As always, before starting to write code, you should do enough design work to avoid wasting many frustrating hours going down blind alleys. Never start programming before you have a clear idea what you are trying to accomplish and how you will approach the problem. To design this program, in addition to sketching the GUI and deciding which classes to use, you must also have some idea what discrete time simulation is. These three will be taken up in the next three subsections.

a) GUI

The problem description says there must be a Button to start and stop the simulation; so, obviously you will need a Button (or two). It also says the ball should bounce when it hits the floor; thus drawing a line for the floor, or perhaps a box around the ball would make it less mysterious for the user when the ball changes direction.

How big should the Applet be? The bigger it is, the farther the ball will be able to fall before it hits the floor; but the particular size is not critical.

b) Discrete time simulation

Models are built to learn about systems of interest. A model allows you to practice with and/or experiment with a system in a safe and inexpensive manner. A simulation of a jetliner allows pilots to train without risking lives and expensive equipment. A simulation of an economy allows planners to try out different measures without disrupting the actual economy. A simulation of the interaction of greenhouse gasses, temperature and glaciation allows scientists to make predictions about the likely effect of various levels of greenhouse gas emissions. A model is always simpler than what it models (and so, no model is ever perfect). Simulation is the implementation of a model in software. Thus, the essence of simulation is simplification.

A discrete time simulation is named that because time moves in discrete steps. The size of the step is up to the modeler; a simulation of a computer might have a time step of a nanosecond, whereas one of continental drift might use a time step of a century or a millennium. In the real world, time is entirely beyond our understanding or control. In a simulation the modeler has complete control over time; it is whatever time the time variable says it is. Welcome to cyberspace!

A simulation has some set of simulated objects. The state of each simulated object is completely specified by its state variables. Similarly, the state of a simulation is completely specified by the values of all its state variables. To move from one time step to the next, the modeler provides a transition function. Given the values of all the state variables at one time step, it calculates their values at the next time step. This transition function may be simple or extremely complex depending on the application.

In the Snowman program, each time the user pushed the button the Snowman melted some and the puddle under it grew some. If you think of that as a simulation of a snowman melting, the state variables for a Snowman were x, y, size, and the size of the puddle. The transition function decreased the size by 10% and increased the size of the puddle by 10 pixels. The location of the snowmen never changed (unless you made them move!). There was no interaction between the snowmen.

In a more complicated simulation, for instance one implementing a model of planetary motion around the sun, the various elements of the simulation affect each other. For a famous old simulation of a cellular automaton, Conway's Life see ???.

A dropped ball bouncing straight up and down under the effect of gravity will need two state variables; one for its height from the floor and one for its velocity (up or down). Given a position and velocity and a gravitational constant, its position and velocity at the next time step can be computed using equations from elementary physics. Using standard physics notation; s stands for position, v for velocity, and a for acceleration. As you might guess v_t is the velocity at time t, v_{t+1} is the velocity at the next time step. Thus the transition function may be written as two equations:

$$s_{t+1} = s_t + v_t$$
$$v_{t+1} = v_t + a_t$$

So the ball's height at the next time step is just its current height plus its current velocity; its velocity at the next time step is just its current velocity plus the acceleration due to gravity.

If you have not studied physics, those equations may seem a bit mysterious; but they are quite simple once you understand them. If a ball is travelling at 10 pixels/step and is currently at location 100, after the next step it will be at 110. In symbols, $s_t = 100$, $v_t = 10$, $s_{t+1} = s_t + v_{t=} 100+10 = 110$. Velocity is updated similarly.

c) Classes

The only thing in this program is the bouncing ball, so an obvious choice for a class is Ball. If Ball extends FilledCircle, the position, color and `paint()` code is already written. The only additional variable needed is velocity in the y direction. The only additional method needed is `step()`, which implement the transition function from one time step to the next.

ii) Implementation

Given your experience with the counter animation, you have seen code that does almost everything the code for this program must do. You only need to create three classes; the Applet, Controller and Ball (although you will need to copy FilledCircle and Circle from your previous project).

a) The Applet

The Applet needed for this program is almost identical with Code Example 9.4. Create a GUI Applet, name it BallApplet, and copy the code from Code Example 9.6. Or, since

Code Example 9.6

```
1 import java.awt.*;
2
3 public class BallApplet extends java.applet.Applet {
4     Controller theController;
5
6     /** Initializes the applet BallApplet */
7     public void init() {
8         initComponents();
9         theController = new Controller(this);
10         theController.start();
11     }
12
13    public void paint(Graphics g) {
14        g.drawRect(1,1,300,750);
15        theController.paint(g);
16    }
17 }
```

Initial BallApplet

As you can verify, this is identical with Code Example 9.4 with the counter removed, and the addition of line 14.

Line 14: Draw a rectangle for the Ball to bounce in. The height is 750 so that with an Applet height of 800 the bottom of the rectangle still shows.

you just typed most of this code in doing the previous program, simply copy it from there, delete the counter code and add the `drawRect()` message (line 14). Notice that this draws a rectangle 750 pixels high; you could chose another size if you wanted.

b) The Controller class

The Controller class is very much like the Controller in Code Example 9.5. Copying and pasting it, then editing it would be the most efficient, but feel free to retype it if that

would help you remember it better. The code you need is in Code Example 9.7 with the

Code Example 9.7

```
1 import java.awt.*;
2 import java.applet.*;
3
4 public class Controller extends Thread {
5     private Applet theApplet;
6     private Ball theBall;
7
8     /** Creates a new instance of Controller */
9     public Controller(Applet theApplet) {
10        this.theApplet = theApplet;
11        theBall = new Ball(100,100,20,Color.RED);
12    }
13
14    public void run() {
15        for(;;) {
16            step();
17            pause();
18       }
19    }
20
21    public void paint(Graphics g) {
22        theBall.paint(g);
23    }
24
25    private void step() {
26        theBall.step();
27        theApplet.repaint();
28    }
29
30    private void pause() {
31        try {
32            Thread.sleep(50);
33        } catch (Exception e) {}
34    }
35}
```

Controller for the BallApplet

As you can see, this Controller is very similar to the one in Code Example 9.5 with the addition of the Ball and the `paint()` method.

Line 11: The Ball will initially be centered at (100,100).

changes from the previous Controller indicated in **bold**. As you can see, you must declare a Ball variable (line 6), instantiate it in the constructor (line 11) and send it the `step()` method when the Controller steps (line 26). Plus, you must `paint()` it when the Controller is painted. That's all. All that remains is writing the Ball class.

Be sure you are familiar with the Controller code. Read each method, line by line. Think about how they interact (i.e. which invokes which, when). After doing that, read the descriptions below. Pay special attention if there are any surprises -- surprises are clues for what to focus on.

- **Controller(Applet)**

The constructor stores the reference to the Applet in the variable named theApplet. Then it instantiates a red Ball of radius 20, centered at (100,100), and stores it in the variable named theBall.

- **run()**

An infinite for-loop with two statements. Each ***iteration*** it steps and then pauses (for 50 msecs).

- **paint(Graphics)**

Sends paint(Graphics) to theBall. The only thing in the simulation is the Ball, so that's all that needs to be painted.

- **step()**

Sends `step()` to theBall, then repaints the Applet so the Ball's new position will be displayed.

c) The Ball class

The Ball class extends FilledCircle. It needs a variable for the y velocity (say, vy), an initializing constructor, and a `step()` method. The constructor can just use FilledCircle's initializing constructor, i.e. it is one line: `super(x,y,r,c);`. The `step()` method has just two lines; one to update the position (`y = y + vy;`) and one to update the velocity (`vy = vy + GRAVITY;`). Recall that constants by convention are all uppercase (see "Case conventions" on page 128), and gravity is definitely constant!

Gravity is constant, but what value should it have? That decision is entirely arbitrary unless you assign some correspondence between pixels and distances in the world. That would be possible, but not necessary in this case. Let the acceleration due to gravity be one pixel per time step. By the way, the units of velocity is also pixels/time step.

Given that description, you can write the Ball class. If some part of that still seems mysterious feel free to look at Code Example 9.8. But, if you actually want to learn to

Code Example 9.8

```
1 import java.awt.*;
2
3 public class Ball extends FilledCircle {
4     public int GRAVITY=1;
5     private int vy;
6
7     public Ball(int x, int y, int r, Color c) {
8         super(x,y,r,c);
9    }
10
11    public void step() {
12        y += vy;
13        vy += GRAVITY;
14
15        System.out.println("v=" + vy);
16    }
17 }
```

The Ball class

Line 4: Define acceleration due to gravity as one pixel per time step.
Line 12: Update the y-coordinate.
Line 13: Update the y-velocity.
Line 15: Print the velocity so you can get some idea how it changes.

program, instead of copying that example, keystroke for keystroke, take the time/spend the effort to study and understand it; then go to the screen and type it in without peeking (if you have to peek once or twice, that's okay). Or, just copy it and waste your time; your choice -- it doesn't really matter, you're going to live forever, right?

d) Testing

With those three classes written, there's enough code to test. Make the Applet taller, so you can see the Ball hit the bottom of the rectangle (see "Changing the size of an Applet" on page 342).

Once you fix all the syntax errors, you should see the Ball falling faster and faster and disappearing off the bottom of the screen. If you look at the output you will see that it is going one unit faster each time step. Now it's time to add the bouncing off the floor code.

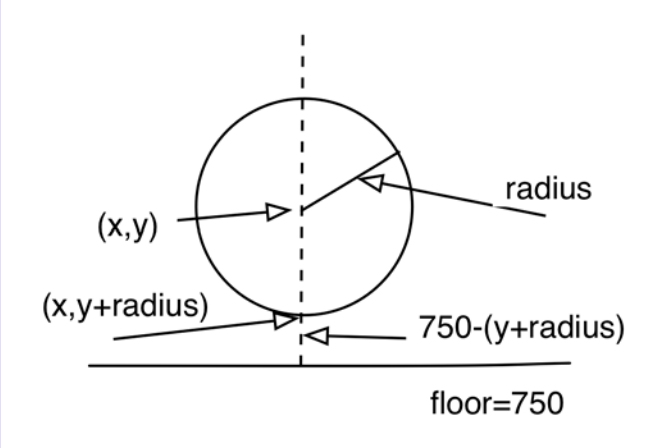
e) Making the Ball bounce - design

There are two steps to making the Ball bounce when it hits the floor: 1) detecting that it hits the floor, and 2) reversing its direction. The latter is simple, just set the y-velocity to its opposite (i.e. `vy = -vy;`). Note that the minus sign is unary minus, it only has one operand. The former is a bit more complicated.

Logically, when the ball hits the floor it should reverse direction. In the world, physics takes care of that; in the simulation, the programmer must decide how to simulate the physics. Here's an idea. On each time step, after updating the position, if the Ball has hit

the floor, reverse its direction. Looking at Figure 9.1 you can see that the distance from

Figure 9.1



the bottom of the Ball to the floor is 750-(y+radius), since y is the y-coordinate of the center of the Ball. Thus the Ball has hit the floor if 750-(y+radius)<0, or equivalently if (y+radius) > 750. In that case you wish to reverse its direction.

f) Making the Ball bounce - implementation

The `step()` method from Code Example 9.8 can be modified to reverse directions when the Ball hits the wall as shown in Code Example 9.9. Add that if statement and run the

Code Example 9.9

```
1     public void step() {
2        y += vy;
3        vy += GRAVITY;
4
5        if (y+radius > 750) // if it hit the wall
6            vy = -vy;       // reverse direction
7     }
```

The Ball:step() method with bounce code

Line 5-6: If the bottom of the Ball is through the floor, reverse the direction.

simulation again. If the rectangle you drew was some other height than 750, use that height instead of 750!

If your machine works like mine (and it might not, exactly), you saw the Ball bounce, but it goes part way into the floor and gradually bounces higher and higher. This is a bit surprising, and not at all like a real ball! What's gone wrong? There are actually two different, interacting causes. One of them stems from using ints which can only represent whole numbers; the other from a flaw in our simulation methodology.

First we will confront a shortcoming of discrete time simulations. Assume the Ball is travelling 10 pixels/step and before the time step it is 3 pixels from the floor. After the time step, if it moves 10 pixels down, it will be 7 pixels into the floor before we check. But, let's ignore that for now until we get the bouncing higher problem solved.

Consider carefully what happens when the `step()` method in Code Example 9.9 is executed with vy=17, and y=721. The first line changes y to 738. The second changes vy to 18. The if compares (738+30) to 750; since 758>750 it evaluates to true and so changes vy to -18. So, not only has the Ball gone 8 pixels into the floor, but also the velocity has *increased*! Perhaps you should only increase the velocity if the Ball does not

change directions as in Code Example 9.10. Try out this version. If your machine works

Code Example 9.10

```
1    public void step() {
2        x += vx;
3        y += vy;
4
5        if (y+radius > 750) { // if it hit the wall
6            vy = -vy;       // reverse direction
7            System.out.println("vy=" + vy + " y=" + y);
8        }
9        else vy += GRAVITY;  // no bounce? accelerate
10   }
```

The Ball:step() only accelerate if no bounce

Line 5-9: If the bottom of the Ball is through the floor, reverse the direction otherwise add the influence of gravity to the velocity.

like mine, the ball always bounces the same height (although it continues to go into the floor a bit).

The problem description said the elasticity of the ball/floor collision was 90%, so when it changes direction you really should reduce the speed by 10% as in Code Example 9.11.

Code Example 9.11

```
1    public void step() {
2        x += vx;
3        y += vy;
4
5        if (y+radius > 750) { // if it hit the wall
6            vy = -vy*9/10;       // reverse direction and reduce 10%
7            System.out.println("vy=" + vy + " y=" + y);
8        }
9        else vy += GRAVITY;  // no bounce? accelerate
10   }
```

The Ball:step() 90% elasticity

Line 6: The collision between the ball and the floor has 90% elasticity, so reduce magnitude by 10% when changing direction.

Try this out. It comes to a stop for me, but into the floor by 8 pixels. Here is some code to fix it is shown in Code Example 9.12. This code is a bit complicated and if you don't

Code Example 9.12

```
1 public void step() {
2     x += vx;
3
4     int bottomY = 750-(radius+y);
5     if (vy >= bottomY) {
6         //System.out.println("vy=" + vy + " y=" + y);
7          int bounceHt = vy - bottomY;    // how high it bounces this step
8          y = 750 - (radius + bounceHt);  // y-coord after this step
9          vy = -(vy-1)*9/10;
10    }
11    else {
12        y += vy;
13        vy += GRAVITY;
14    }
15}
```

The Ball:step() improved?

Line 7: Calculate how far it will travel up after bouncing
Line 8: From that calculate the new y-coord of the center
Line 9: Reverse vy and subtract 1 to avoid endless small bounces.

understand it, that's okay. If you want to figure it out, draw a picture. This code causes the Ball to eventually come to rest exactly touching the floor. Without the -1 in vy-1 on line 9 it got caught in a loop and never stopped bouncing due to the way int arithmetic works. This is by no means a perfect simulation, but its close enough for now.

g) Starting and stopping with two Buttons

The last task remaining from the project description is to allow the user to start and stop the simulation by pressing a button. This will be done first with two Buttons, then with one. The former is easier to understand, the latter is less cluttered and illustrates a useful technique.

Assume the two Buttons are named stopButton (which sends a `stop()` message to the Controller) and goButton (which sends a `go()` message). The question is how to implement `stop()` and `go()`? Recall that the `Conroller:run()` method (shown in Code

Example 9.13) is running in a separate Thread. It is an infinite loop that does `step()`, and

Code Example 9.13

```
16     public void run() {
17         for(;;) {
18             step();
19             pause();
20         }
21     }
```

Controller:run() from Code Example 9.3

then `pause()`, over and over. How can you arrange that `stop()` will stop the simulation and `go()` will resume it? What is needed is if `stop()` has been executed since `go()`, the loop should just `pause()` and not `step()`; otherwise it should do both. Do you know what programming construct to use?

Whenever you want to either do something or not, you use an if statement. You would like to modify `run()` as shown in Code Example 9.14 so the `step()` message is only sent

Code Example 9.14

```
1     public void run() {
2         for(;;) {
3             if (last button pushed was go)
4                 step();
5             pause();
6         }
7     }
```

An if statement to make step() conditional

Lines 3-4: step() will only happen if the last button pushed was the goButton

if the last button pushed was go. But how to write that in Java? The answer is to use a boolean variable, called perhaps, running. The type boolean is used when you only need to represent two values, true and false (see "types, values, operators" on page 118). This situation is perfect for a boolean variable, the simulation is either running or paused. When the Controller gets the `stop()` message, it should set running to false, when it gets the `go()` message it should set it to true, and the expression guarding the `step()` message should just be the boolean variable named running. This is illustrated in Code Example

9.15. Notice that the stop() method has been renamed to `userStop()`. This was because

Code Example 9.15

```
1     private boolean running=true;
2
3     public void go() {
4         running = true;
5     }
6
7     public void userStop() {
8         running = false;
9     }
10
11    public void run() {
12        for(;;) {
13            if (running)
14                step();
15            pause();
16       }
17    }
```

Modifications to make the run method stoppable

Line 1: Declares a boolean variable named running whose initial value is true (so the simulation will start when `run()` starts).
Lines 3-5: The `go()` method sets running to true.
Lines 7-9: the `userStop()` method sets running to false. Java would not allow a method named `stop()` in a Thread (since there already was one declared final).
Line 13: Guards the `step()` message; it is only sent if running is true.

when it was named stop() the compiler complained that:

```
Controller.java [29:1] stop() in Controller cannot override stop() in
java.lang.Thread; overridden method is final
```

What this means is that the Thread class already has a method with that signature (i.e. `public void stop()`), and it is declared `final`, so it *cannot* be overridden. You will never have to declare any methods final (in the context of introductory programming), so you may safely ignore this, but you do need to learn how to cope when you bump into this kind of error message. The rule is, when an error message says "...cannot override...", or "...access type...", then you have stumbled on a method declared in a superclass. The simplest solution is to rename your method.

That's all you need to know to make it start and stop. Do that now. Then return here to learn how to accomplish it with only one Button.

h) Starting and stopping with only one Button

Using one Button to stop the simulation and another to restart it works, but it is not very elegant. You can unclutter your GUI by writing code to make your Button a ***toggle*** switch. Light switches are sometimes push button toggles; if the light is on, pushing the switch turns it off, if it is off, pushing the switch turns it on. Here's how to do that in Java.

The Button must do one of two things; make the Controller stop if it is going, or go if it is stopped.. The construct in Java that does either one thing or the other, is if-else. So, logically the `actionPerformed()` method might be as in Code Example 9.16; if the

Code Example 9.16

```
1 private void toggleButtonActionPerformed(java.awt.event.ActionEvent evt) {
2     if (the simulation is running)
3         theController.userStop();
4     else theController.go();
5
6     change the label on the Button to say what it does now
7}
```

Logic of actionPerformed() for a toggle Button

simulation is currently running, send the Controller the `userStop()` message, otherwise send `go()`.

One's first idea might be to add a boolean variable in the Applet to keep track of whether the Controller is currently running, and set it to the same value as the running variable in the Controller. This is a natural mistake. The problem is that having two different variable keeping track of the same information creates an unnecessary situation where bugs can occur. There will be plenty of bugs no matter what; not reason to encourage them!

The Controller already knows whether the simulation is running. The job of the Applet is to manage the GUI and pass along information to the Controller. When it needs to know if the Controller is running, it should ask it (and then set the toggleButton's label appropriately). The logic of changing the state of the Controller belongs in the Controller.

Thus, when the toggleButton is pushed, it should send a `toggle()` message to the Controller it change the label on the toggle Button to reflect what action will be performed if it is pushed. See Code Example 9.17 for how to do these two things. Notice

Code Example 9.17

```
1 private void toggleButtonActionPerformed(java.awt.event.ActionEvent evt) {
2     theController.toggleRunning();
3
4     if (theController.getRunning())
5         toggleButton.setLabel("stop");
6     else toggleButton.setLabel("go");
7 }
```

Making a toggle Button

Line 2: Toggle the running variable in the Controller.
Lines 4-6: An if-else. Notice that both the if and else parts have only one statements so they do not need to be enclosed in {}s.
Lines 5: Sets the Button label to "stop"; since theController is running.
Lines 6: Sets the Button label to "go".

that there are no {}s around the statements after the if and else parts. If you wanted to do two (or more) things in either the if or else parts you must put {}'s around them to make the two statements into one, syntactically.

The Controller class must be modified to add these two new methods, but you can eliminate two methods at the same time. See Code Example 9.18 for the changes.

Code Example 9.18

```
1     private boolean running=true;
2
3     public boolean getRunning() {return running;}
4
5     public void toggleRunning() {
6        running = !running;
7     }
8
9     public void run() {
10        for(;;) {
11           if (running)
12               step();
13           pause();
14       }
15    }
```

Controller modifications to support the toggle Button

Line 3: The accessor for running.
Lines 5-7: The `toggleRunning()` method replaces `go()` and `userStop()`.
Lines 6: Set the value of running to the opposite of what it was before. The ! operator is called, not. Not true is false, not false is true.
Compare to Code Example 9.15 on page 233, which it replaces.

D. Recapitulation

To do animation in Java you must spawn a separate Thread. To do this you must create a class that extends Thread, create an instance of it and send it the `start()` message. The `Thread:start()` method spawns the new Thread and then sends `run()` to your class; the new Thread exists until `run()` returns.

Discrete time simulation advances time by some fixed amount each step. Each modelled object has a set of state variables with completely determine its state in the context of the simulation. A transition function calculates the next state of each thing in the simulation from its previous state variables (and possible those of other things in the simulation).

Boolean variables are used to store information when the only possible values are true and false. Boolean expressions are used to determine whether to execute the if part of if statements.

E. Conclusion

This chapter introduced two powerful techniques, simulation and animation. It also introduced Java Threads. Simple animations can greatly improve your GUIs. Simple simulations can provide interesting demonstrations. It also presented a java toggle switch to allow the user to control the simulation.

F. End of chapter material

i) New terms in this chapter

iteration - another word for repetition. See the next chapter for iterative constructs. 219
spawned - technical term for created; used only for Threads. When a new thread of control is initiated, i.e. begins execution, it is said to be spawned. 212
thread of control - the sequence of statements executed when a program executes. A temporal map of where control resides during execution. 210
toggle - a two state switch which changes state each time you activate it 228

ii) Review questions

9.1 What are boolean variables used for?
9.2 Where do boolean expressions appear in Java?
9.3 What is Thread short for?
9.4 If your program has two Threads and is executing on a single CPU machine, since only one instruction can be executed at a time, how can both Threads be running at the same time?
9.5 What is the job of the transition function in a discrete time simulation.
9.6 What are the two unary operators?

iii) Programming exercises

9.7 Experiment with Buttons; `setLabel()` looks exactly like an accessor; try out `getLabel()`.
9.8 A Button is a Component (like an Applet, or a Frame). You saw `setBounds()` sent to a Frame... try sending it to a Button. What happens?

9.9 Modify your code so that the Ball can be throw to start the simulation. All that is needed is an additional initializing constructor that takes two additional parameters, the x-velocity and y-velocity.

```
public Ball(int x, int y, int r, Color c, int vx, int vy) {
    this(x,y,r,c);
    this.vx = vx;
    this.vy = vy;
}
```

Then you can initialize the ball in the Controller with:

```
theBall = new Ball(10,100,20,Color.RED, 2, -20);
```

For this to work, you must update the x-coordinate in Ball:step()

9.10 Add code to keep the Ball in the box. It's just like the bounce code except you must check for hitting the other sides of the box.

9.11 Add another Ball (or two) going in a different initial direction. Add 5!

9.12 Modify your code so that it create a pattern like on the back cover of this text. All this requires is adding an `update(Graphics g)` method to the Applet that simply sends `paint(g)`, as shown in Code Example 9.19. An explanation can be found in

Code Example 9.19

```
1 public void update(Graphics g) {
2       paint(g);
3 }
```

BallApplet:update(Graphics)

Add this method to prevent `update()` from filling a rectangle the size of the drawable area in the background color before sending `paint()`.

"repaint(), paint() and update()" on page 351.