Chapter 15

Java Swing, UML, and Concurrency

15.1 Introduction

In this chapter, we'll first look how we can use Java packages to write programs that user with a graphical interface and respond to the user's manipulation of interface elements, such as clicking on buttons. In particular, we'll look at "Java Swing," which supports writing such programs.

Finally, we'll use these programs as the setting in which to introduce the more conceptually significant material, namely concurrency. A concurrent system is one in which multiple activities go on at once. We'll show how to develop programs that can divide their attention between multiple activities. Most importantly, we'll show how the concept of representation invariant, which we've emphasized in prior chapters, gains renewed importance as the key to preventing unwanted interactions between concurrent activities. The chapter concludes with an opportunity for you to apply concurrent programming techniques to a simulation program.

15.2 Event-driven graphical user interface apps

Many programs interact with their users in a very old-fashioned style, characterized by two primary features:

- All input and output is textual: the user and the program both type lines of text, instead of the user pointing at visual information that

the program shows.

- The program is in charge of the interaction. The user is reduced to answering the questions the program asks, rather than taking charge and directly manipulating the program.

This textual, program-directed style of interaction made sense in its original historical context, roughly the early 1960s. Among other things, the typical computer user didn't have access to any hardware that supported graphical interaction: sending the computer a line of text and receiving a line of text in response was the best that most could hope for. (Many users had to settle for *batch processing*, which involved sending enough textual input for the entire running of the program and then receiving back a printout of the entire output, rather than being able to incrementally give and take.)

However, times have changed, and today users typically have computers that allow for a tightly-coupled graphical interaction, in which the user takes charge and directly manipulates the program's interface by pushing buttons, sliding sliders, checking checkboxes, typing into fields, etc. The role of the program is no longer to step through a fixed sequence of questions and answers, but rather to perform two basic functions:

- Present a collection of objects to the user.
- Respond to whatever manipulations of those objects the user performs.

In this section, we'll see how such programs are written. They are called event-driven graphical user interface (GUI—pronounced gooey) programs, because they not only present a GUI, but moreover are driven by outside events, such as the user clicking on a button. In particular, we'll look at *Java Swing* applications, which are event-driven GUI programs (apps) built using a java package called `javax.swing` which provides many GUI classes the can be used "off the shelf."

Java Swing is built on top of the *Abstract Window Toolkit (AWT)*, which provides a platform-independent framework for graphical and window-based programming. In particular, the AWT was used for programming *applets*, which are event-driven GUI programs that are designed to be components of documents on the World Wide Web, rather than standing alone. Rather than running lots of separate programs, the user just runs a single web browser program, which lets them view and interact with lots of different kinds of multimedia hypertext documents. Those documents contain all the usual kinds of content, like words, tables, and pictures, but also interactive

content in the form of applets. We will not be considering applets, but instead will concentrate on Swing apps.

Object-oriented programming plays a critical role in event-driven GUI programs. From our own perspective, we will be able to write our programs reasonably simply and easily because there is a large "library" of pre-written classes for such interaction components as buttons and checkboxes. Thus we can just create appropriate instances of these classes, without worrying about the details of how they work inside.

From the perspective of GUI programmers, the key fact is that all these individual component classes, like `JButton` and `JCheckbox`, are actually subclasses of a general `Component` class. Any `Component` knows how to draw itself. Any `Component` knows how to respond to the fact that a mouse button has been pressed while the mouse was pointing into that `Component`'s area. Thus the programmer can more easily deal with the wide variety of different kinds of interaction mechanisms by simply telling the programs components how to deal with the particular user events they need to deal with. The programmer can then just treat the whole program as nothing but a bunch of `Component`s, asking each `Component` to draw itself on the screen, without knowing or caring that they do so in varying ways. When a mouse button is pressed, it notifies the appropriate `Component`, without caring that a `JTextField` might treat this entirely differently from a `JButton`—they are both still `Component`s.

Java packages

Java provides a library of thousands of classes for the programmer's use. Additionally, other organizations and vendors provide other special-purpose classes which programmers choose to use. Given the organizational complexity of maintaining and using such a large number of usable classes, and in particular the strong probability that the same names might be used in different contexts, Java has specified a *package* system to organize these classes in smaller groupings (packages). For example, one very common and useful class is `ArrayList`, which is part of the `java.util` package, which (as you might imagine) contains a host of utility classes. You can use this class in a program by referring to it with its *fully qualified name* which is `java.util.ArrayList`. Since referring to it in that manner is cumbersome, Java allows you to shorten your references to `ArrayList` by including the following *import* statement statement towards the top of your program:

```
import java.util.ArrayList;
```

Having done this, your can then reference the class `java.util.ArrayList` (for example, when declaring a variable) with the shortened form `ArrayList`.

Packaging is done in a hierarchical or tree-structured manner. Thus, the `java.util` package is contained inside the `java` package (which happens to be empty), and in turn contains the `java.util.concurrent` subpackage (which is not empty).

In addition to allowing Java to organize its classes, the same packaging mechanism is used by programmers to organize their own classes. Certain naming conventions have evolved for these naming packages. For example, suppose you work in the *Simulation Games* division of a game-writing company called *Outrageous Games*, which maintains a website at `www.outrageousgames.com`. If the programmer is asked to write a program that plays the Game of Life, then the programming for this project might be done in the `com.outrageousgames.simulation.life` package. Notice that this name goes from most general (the company) to more specific (the division, then the particular project).

Packaging is implemented in Java by using directories (folders). For example, if you were to write a Java class `Life.java` in the `com.outrageousgames.simulation.life` package, it would be located in the following sub-directory of the base source folder:

```
com/outrageousgames/simulation/life
```

Since your program might involve several packages, it can become very tedious remembering where the various packages are and doing the appropriate compile statements, since the compiled files must also be stored according to the same directory structure. Fortunately, most *Integrated Development Environments (IDEs)* such as *Eclipse* simplify the creation, computation, execution, and even the documentation of programs written with packages.

Fifteen puzzle, version 1

Our first example GUI program is shown in figure 15.1. This is a simulation of the sliding 15-tile puzzle. The real puzzle has 15 numbered tiles that can slide around inside a frame that has room for 16 tiles, so that there is always one empty position. After sliding the tiles around for a while to scramble them, the goal is to get them back into their original arrangement, in numerical order with the blank space in the lower right corner. Our program simulates the puzzle with a grid of 16 buttons, of which 15 are labeled with the numbers 1 through 15, while the remaining one has a blank

Figure 15.1: The sliding 15-tile puzzle app

label, representing the empty position. If the user clicks on a numbered button that is in the same row or column as the blank one, that means they want to slide the tile they clicked on, pushing along with it any others that intervene between it and the empty position. We simulate this by copying the numeric labels over from each button to its neighbor. We also set the clicked-on button's label to an empty string, because it becomes the newly empty one.

In order to program this puzzle, we will adopt the object-oriented perspective and view the program as a collection of interacting objects. Some of the objects have visible representations on the screen when our program is running. Most of these objects are either in `java.awt` (the base package for the *Abstract Window Toolkit,* abbreviated AWT) or `javax.swing` (the base package for *Java Swing*). For example, most of these objects instances of the subclasses of the `Component` class, which is in the `java.awt` package. The `Components` in our program are as follows:

- There are 16 instances of the class `JButton`, which is a Swing class derived from `Component`. Each `JButton` has a label (empty in one case), and can respond to being pushed.
- There is another instance of the class `JButton`, labeled "Initialize," that will allow the user to initialize the numbers as in figure 15.1

- There is also one object representing the program as a whole, containing the grid of `Button`s. This object is an instance of a class we'll define, called `Puzzle`. The `Puzzle` class is a subclass of an Swing class called `JFrame`, used for all Swing programs that create their own windows on the screen. As the class hierarchy in figure 15.2 shows, the `JFrame` class is indirectly descended from `Component`, since a `JFrame` is visibly present on the screen. More specifically, since a `JFrame` can contain other `Component`s (like our `Button`s), the `JFrame` class is descended from a subclass of `Component` called `Container`, which is an AWT class providing the ability to contain subcomponents.

In addition to the objects mentioned above, there are some others that operate behind the scenes:

- Our `Puzzle`, like any `Container`, needs a *layout manager* to specify how the subcomponents should be laid out on the screen. Actually, there are two layout managers:
  - The default layout manager for a `JFrame` (of which `Puzzle` is a subclass) is the `BorderLayout`, which allows us to put a control panel at the top (`BorderLayout.NORTH`) which only contain the Initialize button now, and the main panel, which contains the tiles in the center (`BorderLayout.CENTER`).
  - We set the layout manager for the main panel to a `GridLayout`, since we want the tile to form a 4×4 grid.

We can describe the class structure of the classes that are involved in our program using a standard notation known as the *Unified Modeling Language*, or *UML*, as illustrated in figure 15.2. This notation also provides means for expressing many other aspects of object-oriented design, not just the class hierarchy. We'll gradually explain more and more of the notation, as the need arises. (Even so, we'll only see the tip of the iceberg.)

The portion of the class hierarchy used in this program is shown in figure 15.2. Note that the only class in the diagram that is not a library class is `Puzzle`, which is a direct subclass of `JFrame`. All the other classes (other than the fundamental `Object` class) are in `java.awt` or `javax.swing`. Most of the Swing classes start with the letter J.

Without further ado, let's go directly to the portion of the `Puzzle.java` file that deals with the initial construction and layout of the `Puzzle` object:

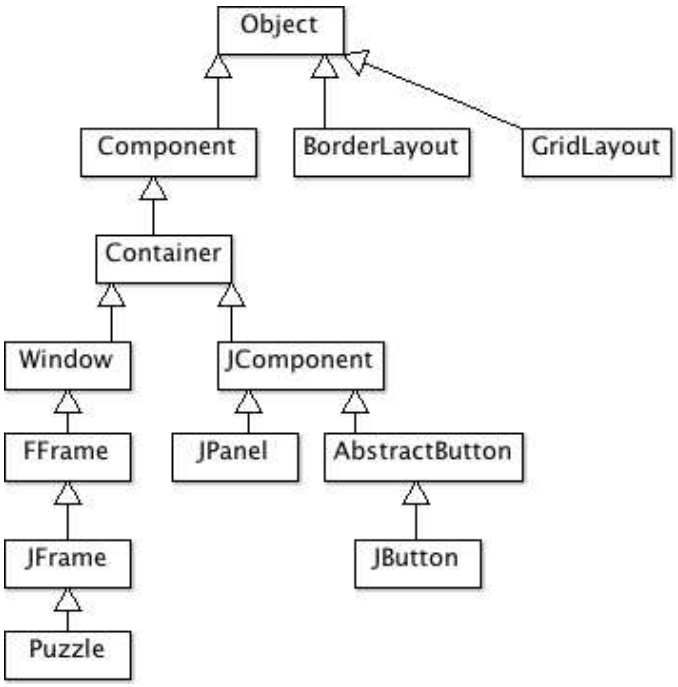

Figure 15.2: Class hierarchy for 15-tile puzzle app

```
package edu.gustavus.mcs178.npuzzle.v1;

import java.awt.BorderLayout;
import java.awt.EventQueue;
import java.awt.GridLayout;
import java.awt.HeadlessException;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Puzzle extends JFrame {

    // The tiles of the puzzle are a square grid of PUZZLE_SIZE by PUZZLE_SIZE
    public static final int PUZZLE_SIZE = 4;

    // Each tile is an approximate square of approximately this size in pixels
    public static final int BUTTON_WIDTH = 70;

    // Computed dimensions of the JFrame
    public static final int WINDOW_WIDTH = PUZZLE_SIZE * BUTTON_WIDTH;
    public static final int WINDOW_HEIGHT = (PUZZLE_SIZE + 1) * BUTTON_WIDTH;

    private JButton[][] tiles;
    private int blankRow;
    private int blankCol;

    public Puzzle() throws HeadlessException {

        int numButtons = PUZZLE_SIZE * PUZZLE_SIZE;
        setTitle((numButtons - 1) + " Puzzle");
        setSize(WINDOW_WIDTH, WINDOW_HEIGHT);

        // controlPanel contains high-level controls for the game
        JPanel controlPanel = new JPanel();
        add(controlPanel, BorderLayout.NORTH);

        JButton initializeButton = new JButton("Initialize");
        controlPanel.add(initializeButton);
```

```
        // mainPanel contains the grid of tiles (which are JButtons)
        JPanel mainPanel = new JPanel(new GridLayout(PUZZLE_SIZE, PUZZLE_SIZE));
        tiles = new JButton[PUZZLE_SIZE][PUZZLE_SIZE];
        add(mainPanel, BorderLayout.CENTER);

        // set up the grid of tiles in mainPanel
        for (int row = 0; row < PUZZLE_SIZE; row++) {
            for (int col = 0; col < PUZZLE_SIZE; col++) {
                JButton tile = new JButton();
                tiles[row][col] = tile;
                mainPanel.add(tile);
            }
        }

        int tileCount = 0;
        for(int row = 0; row < PUZZLE_SIZE; row++){
            for(int col = 0; col < PUZZLE_SIZE; col++){
                tileCount++;
                tiles[row][col].setText("" + tileCount);
            }
        }
        blankRow = PUZZLE_SIZE - 1;
        blankCol = PUZZLE_SIZE - 1;
        tiles[blankRow][blankCol].setText("");
    }

    public static void main(String[] args) {
        EventQueue.invokeLater(new Runnable()
        {
            public void run() {
                Puzzle frame = new Puzzle();
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}
```

Before explaining the code, we should note that the program, which can be compiled and will run, *does not yet do what we want!* It will result in the an application resembling the one in figure 15.1 (which is a screenshot from a Mac), but the buttons do nothing other than highlighting themselves when pressed. We will construct the program in stages (versions), where this first version simply arranges the basic geometry of our app.

This code includes a number of Java features that bear explaining, which we will explain from top to bottom:

- The `package` line at the top specifies the package within which `Puzzle` lives. Note that the naming of the package corresponds to the naming convention we described above, with two explanations: (a) `npuzzle` refers to the fact that the Fifteen Puzzle is just one example of such a puzzle: 15 corresponds to a 4×4 square ($N = 15$), 24 corresponds to a 5×5 square ($N = 24$), etc; (b) `v1` indicates that this is the first version of the puzzle.

- The import statements import all of the AWT and Swing classes use in the puzzle.

- The constants `PUZZLE_SIZE`, `BUTTON_WIDTH`, `WINDOW_WIDTH`, and `WINDOW_HEIGHT` determine the size of the tiles and windows.

- The instance variables for the consist of:

  - `tiles`, which is a 2-dimensional array of `JButton`s; and
  - `blankRow` and `blankCol`, which are `int` variable keeping track of the blank tile.

- The constructor `Puzzle()`, as a subclass of `JFrame`, throws a `HeadlessException`, which occurs "when code that is dependent on a keyboard, display, or mouse is called in an environment that does not support a keyboard, display, or mouse", quoting from the Java documentation. This is a `RuntimeException` that doesn't need to be caught and shouldn't even occur on a desktop computer.

- The remaining code in the constructor `Puzzle()` puts the puts a control panel (with an initialize button) in the top (NORTH) of the Puzzle's content, and the main panel (with the tile buttons) in the center (CENTER) of the Puzzle's content, puts a reference to each tile into the `tiles` grid, initializes the strings in all of the tiles, and records the position of the blank tile.

- Finally, the `main(String[] args)` procedure is called in the manner required by Swing. Without getting into the details, we simply say that it creates a runnable process that, when evoked, will respond as programmed to user input (mouse-clicks and keyboard input).

At this point, we have a nice looking app that unfortunately doesn't do anything interesting. We next consider how to make it react to user input.

Fifteen puzzle, version 2

To start this process, we will start by making the initialize button work properly. Specifically, we will alter the program so that when launched, it will present the initialize button and tiles, except that the tiles will have no text. The user can then click on the initialize button in order to put the appropriate text on the tiles (as well as recording where the blank tile is located).

Before getting into the details regarding how we can get Swing and AWT classes to react appropriately to user input events, let's first remove the initialization code from the constructor and put it into a method called `initializeTiles()` in `Puzzle`:

```
public void initializeTiles() {
    int tileCount = 0;
    for(int row = 0; row < PUZZLE_SIZE; row++){
        for(int col = 0; col < PUZZLE_SIZE; col++){
            tileCount++;
            tiles[row][col].setText("" + tileCount);
        }
    }
    blankRow = PUZZLE_SIZE - 1;
    blankCol = PUZZLE_SIZE - 1;
    tiles[blankRow][blankCol].setText("");
}
```

Having isolated this code, let's now consider how AWT and Swing deal with user events. AWT has class called `ActionEvent` that is used to specify, for example, the event of clicking on a button. In order to have the button respond as desired to that action event, it must implement the `ActionListener` interface, which is also contained in the AWT. The only method that the `ActionListener` interface requires to be implemented is

`actionPerformed(ActionEvent event)`, which defines how the button reacts to the event.

To carry out this process for the initialize button, we proceed by doing the following two things:

1. write a class called `InitializeActionListener` that implements `actionPerformed(ActionEvent event)`; and
2. connect up an `InitializeActionListener` object to the actual initialize button.

Following is the code for the `InitializeActionListener` class:

```
package edu.gustavus.mcs178.npuzzle.v2;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class InitializeActionListener implements ActionListener {

    private Puzzle puzzle;

    public InitializeActionListener(Puzzle puzzle) {
        this.puzzle = puzzle;
    }

    public void actionPerformed(ActionEvent event) {
        puzzle.initializeTiles();
    }
}
```

The constructor for `InitializeActionListener` must be passed an instance of the `Puzzle` object. Then, when it is asked to preform its action, it tells that `Puzzle` object to initialize its tiles, as was desired.

To connect up the `InitializeActionListener` with the initialize button and the puzzle, we simply add one line (the middle line below) to the two lines in `Puzzle` that deal with `initializeButton`:

```
    JButton initializeButton = new JButton("Initialize");
    initializeButton.addActionListener(new InitializeActionListener(this));
    controlPanel.add(initializeButton);
```

With these two additions, the initialize button will now function as desired.

Fifteen puzzle, version 3

To finish up the puzzle, we need to add action listeners to each of the tiles (which are `JButton`s) that the tiles to "move"' appropriately. Actually, they won't actually move; instead, there textual labels simply need to change appropriately in order to simulate the desired movement.

First, given that we have know the position of each tile within the grid by its position within the two-dimensional tile array, let's write a method called `pushTile(int row, int col)` in `Puzzle` that does this relabeling:

```
public void pushTile(int row, int col){
    if (row == blankRow) {
        for ( ; blankCol < col; blankCol++) {
            tiles[blankRow][blankCol].setText
            (tiles[blankRow][blankCol+1].getText());
        }
        for ( ; blankCol > col; blankCol--) {
            tiles[blankRow][blankCol].setText
            (tiles[blankRow][blankCol-1].getText());
        }
    } else if (col == blankCol) {
        for ( ; blankRow < row; blankRow++) {
            tiles[blankRow][blankCol].setText
            (tiles[blankRow+1][blankCol].getText());
        }
        for ( ; blankRow > row; blankRow--) {
            tiles[blankRow][blankCol].setText
            (tiles[blankRow-1][blankCol].getText());
        }
    }
    tiles[blankRow][blankCol].setText("");
}
```

The correctness of the above code critically depends on a feature of `for` loops that we haven't stressed previously. Namely, if the test condition isn't true to start with, the loop's body will be executed zero times. That is, the test is done before each iteration of the loop, even the first one.

Exercise 15.1
Explain in more detail how this assures correctness. In particular, answer the following questions:

a. Suppose both the row and the column are equal to the blank position. What will happen?

b. Suppose the row is equal, but the column number of the blank square is less than that which is clicked on. This means that the first `for` loop's body will be executed at least once. When that first loop finishes, how do you know that the second `for` loop's body won't also be executed?

To finish the program, we need to give each of the tiles an action listener that will call `pushTiles(int row, int col)` appropriately. We do this by creating a class called `TileActionListener`, analogous to `InitializeActionListener`, that will do exactly this. Clearly, we must also pass the constructor for `TileActionListener` the tiles row and column:

```
package edu.gustavus.mcs178.npuzzle.v3;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class TileActionListener implements ActionListener {

    private Puzzle puzzle;
    private int row, col;

    public TileActionListener(Puzzle puzzle, int row, int col){
        this.puzzle = puzzle;
        this.row = row;
        this.col = col;
    }

    public void actionPerformed(ActionEvent event) {
        puzzle.pushTile(row, col);
    }
}
```

Finally, we modify the code at the end of the `Puzzle` constructor as follows in order to connect up each tile with the an appropriate `TileActionListener`:

```
// set up the grid of tiles in mainPanel
for (int row = 0; row < PUZZLE_SIZE; row++) {
    for (int col = 0; col < PUZZLE_SIZE; col++) {
        JButton tile = new JButton();
        tiles[row][col] = tile;
        tile.addActionListener(new TileActionListener(this, row, col));
        mainPanel.add(tile);
    }
}
initializeTiles();
```

Note that we ended with a call to the `initializeTiles()`, so that the tiles have been appropriately initialized.

Fifteen puzzle, version 4

If we want to add additional features, we can just `add` more items to the `controlPanel`. For example, it would be nice if there was a "Randomize" button next to the "Initialize" one, for people who don't want to do their own scrambling of the tiles. We'll write a `randomizeTiles` method for the `Puzzle` class, and then leave you to add the appropriate `ActionListener` class and `JButton`. One aside: when you add a new `JButton` to the `controlPanel`, we said it would go next to the "Initialize" one. Why? Well, that has to do with the `controlPanel`'s layout. But if you look at the above `Puzzle` constructor, you'll see we didn't set a layout for the `controlPanel`, just for the program itself and the `mainPanel`. The solution to this mystery is that each `Panel` when constructed starts out with a default layout, a so-called `FlowLayout`. For the other two panels, we had to change to a different layout, but for the `controlPanel` the default was just what we wanted. It puts a little space between the constituents (for example, between the "Initialize" and "Randomize" buttons) and then centers the whole group.

There are two basic approaches to how the `randomizeTiles` method could work. One would be to literally randomize the sixteen labels, by selecting any of the sixteen to be on the first tile, then any of the remaining fifteen to go on the next tile, etc. A different approach would be to simply

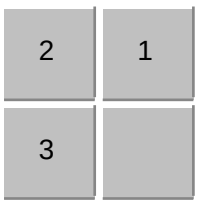

Figure 15.3: From this configuration, no amount of sliding the tiles will put the numbers in order with the blank in the lower right.

randomly shove the tiles around for a while, using `pushTile`, until we were satisfied that they were adequately scrambled. The big problem with the first approach is that you can get into configurations that can't be solved. To see this in a simpler setting, consider what might happen in a 2×2 sliding tile puzzle, which has three numbered tiles. It shouldn't take much playing around to convince you that the configuration shown in figure 15.3 can't be solved, since the tiles can only be cycled. Keep in mind that for the puzzle to be solved, the blank space needs to be in the lower right corner—it doesn't suffice for the numbers to be in order. A similar but more complicated argument can be made to show that half the configurations in the fifteen-tile puzzle are also unsolvable.

We therefore make the design decision to write `randomizeTiles` so that it randomly slides tiles around. This can be accomplished by the following code:

```
public void pushRandomTile(){
  int row = (int) (Math.random() * size);
  int col = (int) (Math.random() * size);
  pushTile(row, col);
}

public void randomizeTiles(){
  for(int i = 0; i < 100; i++){
    pushRandomTile();
  }
}
```

The only question you are likely to have about this code is, "why 100?" The answer is that it seemed like a reasonable number: big enough to do a pretty good job of scrambling, while not so large as to take a long time. It

would be fine to change it if you want to make a different trade-off. At any rate, to see how long it does take on your computer, you'll need to provide a button to activate it:

Exercise 15.2
Provide a "Randomize" button, by doing the following:

a. Write a `RandomizeActionListener` class, similar to `InitializeActionListener`, but which invokes the `randomizeTiles` method.

b. Add a "Randomize" `JButton` to the `controlPanel` with a `RandomizeActionListener`.

We'll see a rather different program in the application section at the end of the chapter. For now, let's stick close to home and do another puzzle that involves a square grid of `JButton`s. The physical version of this puzzle is played with a square grid of pennies, initially all head side up. At each move you can flip any penny over, but then also have to flip over the four neighboring ones (not counting the diagonal neighbors). If the penny you flipped was on an edge of the grid, so that it is missing a neighbor, or in a corner, where it misses two neighbors, you just flip the neighbors that it does have. As with the sliding tile puzzle, the goal is to do a bunch of moves to scramble the grid up, and then try to get back to the starting position. If you want to make the puzzle relatively easy, you might want to change `size` from 4 to 3; if you like challenge, you might up it to 6.

Exercise 15.3
Change the puzzle app to this new puzzle, by making the following changes:

a. Get rid of the `blankRow` and `blankCol` instance variables, and in their place add two new instance variables of type `String`, called `heads` and `tails`, each set equal to an appropriate string. The strings you choose needn't have any resemblance to coins, and it is best if the two are visually very distinct, for example `heads = "Flip!"` and `tails = ""`.

b. Change the `initializeTiles` method to set the label of all the buttons to `heads`.

c. Add a new method, `flip`, which takes a `JButton` as an argument and changes its label. If the current label is `heads`, it should change to `tails`, and vice versa.

d. Change the `pushTile` method to `flip` the `JButton` in the pushed position, and also `flip` each of its four neighbors, provided that they exist.

15.3 Concurrency

In the introduction to this chapter, we defined a concurrent system as one in which multiple activities go on at once, but we didn't say anything about why anyone would want to build such a system. You might think that the answer is to get a computation done faster, by doing multiple subproblems simultaneously. This can indeed be a motivation for concurrency, but it is not the most important one in practice. To start with, most "concurrent" computer programs don't truly carry out their multiple activities at the same time; instead, the computer switches its attention back and forth between the activities, so as to give the impression of doing them simultaneously. This is because a truly concurrent computation would require the computer hardware to have multiple processors; although some systems have this feature, many don't. On a single-processor computer, all the activities necessarily have to take turns being carried out by the single processor. At any rate, concurrency has far more fundamental importance than just as a way to (maybe) gain speed. Why? Because the world in which the computer is embedded is concurrent:

- The user is sitting in front of the computer thinking all the while the computer is computing. Maybe the user decides some other computation is more interesting before the computer is done with the one it is working on.
- Computers today communicate via networking with other computers. A *client* computer may well want to do other computations while waiting for a response from a *server* computer. A server, in turn, may well want to process requests it receives from clients, even if it is already busy doing work of its own, or handling other clients' earlier requests.

In other words, the primary motivation for concurrent programming is because a computer needs to interact with outside entities—humans and other computers—who are naturally operating concurrently with it.

In this section we'll see how to write a concurrent program, and some of the interesting issues that arise from interactions between the concurrently executing parts of the program. To illustrate our points, we'll use some

further variations on the sliding 15-tile puzzle program from the previous section. The basic premise is that the puzzle isn't challenging enough for some users, so we're going to add a new twist, which requires the user to stay alert. Namely, the computer will occasionally slide the tiles on its own, without the user doing anything. We call this the "poltergeist" feature, because it resembles having a mischievous invisible spirit (i.e., a poltergeist) who is playing with your puzzle, and thereby with your mind.

In broad outline, it seems relatively obvious how to program a poltergeist into the puzzle. We'll just add a third button to the control panel after the "Initialize" and "Randomize" buttons, with some appropriate label (maybe "Mess with me"), and an `ActionListener` that when the button is pushed goes into an infinite loop where each time around the loop it pushes a random tile.

The one big problem with this plan is that when the user pushes a button and the ActionListener is notified, the user interface goes dead until the ActionListener's `actionPerformed` method has finished responding. Depending on how fast your computer is, you may have noticed this with the "Randomize" button. If not, you could try the experiment of increasing how many random pushes it does from 100 to some larger number, say 500. You should be able to notice that no additional button pushes get responded to until all the random sliding around is done. Thus a button that didn't loop 100 or 500 times, but instead looped forever, would never let the user push any tiles of their own. That defeats the whole point of the poltergeist—the point is to have it sliding tiles *while* the user slides them too.

Thus, we need the program to be truly concurrent: one part of the program should loop forever, sliding tiles randomly, while another part of the program should continue to respond to user interaction. Rather than speaking of "parts" of the program, which is rather vague, we'll use the standard word: *threads*. One thread will do the random pushing, while the original main thread of the program continues handling the user's actions. Thus our program will now be *multi-threaded*.

Rather than attempting a precise definition of threads, let us instead suggest that you think of them as independently executing strands of computations which can be braided together to form a concurrent program. This description implies a different model of computation from the one presented in chapter 11, where the computer followed a single strand of execution determined by a program's SLIM instructions. In our new multi-threaded case, you can still follow linearly along any one strand and see the instructions one after another in their expected sequence. However, if you look not at

the one strand but at the entire braid, you'll see the instructions from the various strands mingled together. If you are wondering how the computer can mingle different instruction sequences this way, we congratulate you. You should indeed be wondering that. We'd like to answer the question, and for our own students we do—but in a later course. We unfortunately don't have the time or space here.

From our perspective, however, it is enough to note that the Java language requires a specific model of concurrency from its implementations. To be more specific, Java implementations must support the class `Thread`, which allows the creation and simultaneous execution of concurrent threads of computation. As you will soon see, even though the operations involving threads are designed and specified well in Java, the very nature of concurrency gives rise to new and interesting problems not encountered in single-threaded applications. The Java language specification gives the implementation considerable flexibility with regard to how it mingles the threads of execution—different implementations might take the same strands and braid them together in different ways. This will be one of the reasons why we'll need to marshal our intellectual tools so that we can keep things simple rather than succumbing to potential for complexity.

Fifteen puzzle, version 5

We create our poltergeist by defining a subclass of `Thread` called `PoltergeistThread`. The only `Thread` method we need to override is `run`, which tells what the thread does when it executes. Here then is our definition of the class `PoltergeistThread`:

```
package edu.gustavus.mcs178.npuzzle.v5;

public class PoltergeistThread extends Thread {

    private Puzzle puzzle;

    public PoltergeistThread(Puzzle puzzle) {
        this.puzzle = puzzle;
    }

    public void run() {
        try {
```

```
            while(true) {
                Thread.sleep(1000); // 1000 milliseconds = 1 second
                puzzle.pushRandomTile();
            }
        } catch (InterruptedException e) {
            // If the thread is forcibly interrupted while sleeping,
            // an exception gets thrown that is caught here. However,
            // we can't really do anything, except stop running.
        }
    }
}
```

The one part we hadn't warned you about in advance is that rather than just madly looping away at full speed, pushing random tiles as fast as it can, the poltergeist instead sleeps for one second between each random push. This is important: otherwise the user still wouldn't have any real chance to do anything. Therefore, we've programmed in a one-second delay using `sleep`, a static method in the `Thread` class. The only nuisance with using `sleep` is that it can throw an `InterruptedException`, so we have to be prepared to `catch` it. This exception gets thrown if some other thread invokes an operation that interrupts this thread's sleep. That never happens in our program, but we're still required to prepare for the eventuality. This requirement that we include a `catch` arises because the `run` method's declaration doesn't list any exceptions that might be thrown out of it, and the Java system interprets this as a claim that none will be. It therefore requires us to back this claim up by catching any exceptions that might be thrown by other procedures that `run` calls, such as the `InterruptedException` that `Thread.sleep` can throw.

Here is the `PoltergeistActionListener` class, which responds to a push of the poltergeist button by creating a new `PoltergeistThread` object and telling it to `start` running:

```
package edu.gustavus.mcs178.npuzzle.v5;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class PoltergeistActionListener implements ActionListener {

    private Puzzle puzzle;

    public PoltergeistActionListener(Puzzle puzzle) {
        this.puzzle = puzzle;
    }

    public void actionPerformed(ActionEvent e) {
        new PoltergeistThread(puzzle).start();
    }
}
```

Note that the `actionPerformed` method creates a new `PoltergeistThread` object and then calls the `start` method on the newly created object. This is where the concurrency happens: the `start` method immediately returns, so that the main thread can go on its way, processing other button presses from the user. However, although the `start` method has returned, the new `PoltergeistThread` is now off and running separately.

Assuming you add the appropriate `JButton` to the `controlPanel`, you now have an program that can (if the user chooses) go into poltergeist mode, where the tiles slide around on their own sporadically. The only problem is, the program is a bit buggy. We'll spend much of the rest of this section explaining the bug, and what can be done about it.

> **Exercise 15.4**
> Even a buggy program is worth trying out. Add a `JButton` to the `ControlPanel` for firing up a poltergeist, and try it out.

Recall that different Java implementations can braid the same strands of a multi-threaded program together in different ways. Therefore, we can't be sure what behavior you observed when you ran the program. The chances are good that it behaved fine, which may leaving you wondering why we called the program buggy. The problem is this: what happens if just as the poltergeist is sliding a tile, the user chooses to push a button too? Normally

one or the other will get there first, and be already done with the sliding before the other one starts. In this case, all is well. But if by an amazingly unlucky coincidence of timing one starts sliding a tile *while* the other is still doing so, then interesting things happen. Our main focus in this section will be on how you can design a program such that timing-related bugs like this one can't possibly occur, rather than merely being unlikely. However, because it is worthwhile to have some appreciation of the kind of misbehaviors we need to prevent, we'll first take some time to look at how we can provoke the program to misbehave.

There are two ways to experimentally find out some of the kinds of interesting behavior that can occur. One is to click the poltergeist button and then click away on the other buttons a lot of times until you get lucky and hit the timing just right. (Or maybe we should say until you get unlucky and hit the timing just wrong.) The problem with this approach is that you might get a repetitive strain injury of your mouse finger before you succeeded. So, we'll let you in on the other approach, which exploits a special feature of the program: you can have more than one poltergeist. If you think about it, clicking on the poltergeist button creates a new `PoltergeistThread` and `start`s it `run`ning. Nothing checks to see whether there already is one running. So, if you click the button again, you get a second `PoltergeistThread`, concurrent with the first one and the user. A few clicks later you can have half a dozen poltergeists, all sliding away at random. Now you just sit back, relax, and wait for something interesting to happen when one of the poltergeists happens to slide a tile while another is.

When we tried this experiment, the first interesting thing that happened was that the number of blank tiles gradually started going up. (Initially there was just one, of course.) Occasionally, though much less frequently, the same number appeared on more than one tile. After a while there were just a few numbered tiles left, and mostly blanks. The final interesting thing, which happened after most of the tiles were blank, was that we got error messages from the Java system telling us that some of the array references being done in `pushTile` were out of bounds. In other words, one of the array indices (row or column) was less than 0 or greater than 3.

Looking at the code, it appears at first that none of these problems should occur. For example, consider the following argument for why our array references should never be out of bounds: The row and column being pushed on are necessarily always in the range from 0 to 3. The blank row and blank column should also always be in this range. Why? Because they are initially, and the blank spot only ever moves from where it is one position at a

time towards the tile being pushed, until it reaches that position. Therefore, since it starts at a legal position, and moves one space at a time to another legal position, it will always be in a legal position, and all the array accesses will always be in bound—except that they aren't!

The flaw in our reasoning is where we said that the blank position only moved one space at a time, stopping at the destination position. Suppose two threads both push the tile that is immediately to the right of the blank spot. Both check and find that the blank column is less than the destination column. Then both increment the blank column. Now the blank column has increased by two—shooting right passed where it was supposed to go.

This kind of anomaly, where two threads interact in an undesirable fashion when the timing is just wrong, is known as a *race*. Such errors can occur when two independent threads are accessing the same data (in our case, the instance variables in the `Puzzle` object itself) and at least one of them is modifying it. We should point out that our explanation of how the array reference errors might occur is just one possible scenario. The Java language specification provides sufficient freedom in how the threads are intermingled that lots of other possibilities exist as well.

Exercise 15.5
Having given a plausible explanation for the out of bound array references, let's consider the other two bugs we found:

a. Explain how two threads could interact in a manner that would result in two blank tiles.

b. Explain how two threads could interact in a manner that would result in two tiles with the same number.

As you can see, even detecting a race can be difficult; trying to understand them can be downright perplexing. Therefore, one of our main goals in this section will be to show you a way to avoid having to reason about races, by ensuring that they can't occur. It is incredibly important to make sure that the races *can't* occur, because you can never rely on experimentally checking that they *don't* occur. Since a race by definition depends on the timing being just wrong, you could test your program any number of times and never observe any misbehavior, and still have a user run into the problem.

This is not just a theoretical possibility: real programs have race bugs, and real users have encountered them, sometimes with consequences that have literally been fatal. For example, there was a race bug in the software

used to control a medical radiation-therapy machine called the Therac 25. This machine had two modes of operation: one in which a low-energy beam directly shined on the patient, and one in which the beam energy was radically increased, but a metal target was put between the beam source and the patient, so that the patient received only the weaker secondary radiation thrown off by the metal when struck by the beam. The only problem was that if a very quick-typing therapist set the machine to one mode, and then went back very quickly and changed it to the other mode, the machine could wind up with the beam on its high power setting, but the metal not in the way. This caused horrifying, and sometimes fatal, damage to several patients; the descriptions are too gruesome to repeat. The problem causing this was a race condition between two concurrent threads; it only showed up for the very fastest typists, and only if they happened to carry out a particular action (rapidly changing the operating mode). Because of this, it not only wasn't found in initial testing, but also showed up so sporadically in actual use that the service personnel failed to track the problem down and allowed the machine to continue causing (occasional) harm.

Not every concurrent system has the potential to kill, but many can at least cause serious financial costs if they fail unexpectedly in service. Therefore, it is important to have some way to avoid race conditions, rather than just hoping for the best. Luckily we've already taught you the key to designing race-free concurrent systems: representation invariants.

Recall that a representation invariant of a class is some property which is established by the class's constructor and preserved by all of the class's mutators, so that all of the class's operations can count on the property being true (by induction). For example, if we ignore the concurrency muddle for the moment, the `Puzzle` class has the following representation invariant:

> **Puzzle representation invariant:** Any instance of the `Puzzle` class will obey the following constraints at the time each method is invoked:
>
> - $0 \leq$ `blankRow` $<$ `size`
> - $0 \leq$ `blankCol` $<$ `size`
> - The `JButton` stored in `buttons[blankRow][blankCol]` has the empty string as its label.
> - The remaining `size`$^2 - 1$ `JButtons` are labeled with the numerals from 1 to `size`$^2 - 1$ in some order.

The whole point of having such a representation invariant is that it frees us from having to reason about what specific mutations are done in what order, because we have an inductive guarantee that holds over *all* sequences of mutations.

This ability to know what is true over all sequences, so that we don't have to consider each individual sequence, is exactly what we need for dealing with concurrency. Consider, for example, a simple program with two threads, each of which performs two mutations. The first thread does mutations a and then b, while the second thread does A and then B. Then even in this very simple concurrent system, there are six possible interleaved sequences in which the mutations might occur: $abAB$, $aAbB$, $aABb$, $AabB$, $AaBb$, and $ABab$. Would you really want to check that each of these six orders leaves the program working? And if six hasn't reached your pain threshold, consider what happens as the number of threads or mutations per thread grows much beyond two. So clearly it is a big win to be able to show that the program is correct under any ordering, without considering each one individually.

However, having representation invariants that we can inductively rely on to be true after any sequence of mutator operations only helps us if we have some way of knowing that the program's execution *is* some sequence of mutator operations. In the case of the `Puzzle` program, the two mutator operations that are in charge of maintaining the invariants are `pushTile` and `initializeTiles`. Therefore, we need some way of knowing that the Java system will invoke those operations in some sequential fashion, rather than jumbling together parts of one invocation with parts of another. The reason why individual parts of the mutators can't be jumbled is that they don't preserve the invariant; for example, even if the invariant holds before executing `blankCol++`, it won't hold afterwards. So, what we need to do is identify for the Java system the invariant-preserving units that it needs to treat as indivisible, i.e, that it is not allowed to intermingle.

Java provides the ability to mark certain methods as indivisible in this sense, using the modifying keyword `synchronized`. Since `initializeTiles` and `pushTile` are the two `Puzzle` mutators that preserve the invariant (if left uninterrupted), we use the following code to mark them as `synchronized`:

```
public synchronized void initializeTiles(){
  // body same as before
}
```

```
// (continued)

public synchronized void pushTile(int row, int col){
  // body same as before
}
```

With these keywords in place, the Java system won't let any thread start into one of these methods if another thread is in the midst of one of them on the same `Puzzle`. Instead, it waits until the other thread has left its `synchronized` method. One way to envision this is that each object has a special room with a lock on the door. There is a rule that `synchronized` methods may only be performed in that room, with the door locked. This forces all threads that want to perform `synchronized` methods to take turns.

In the `Puzzle` class, the only methods that directly rely on the representation invariant are the two mutator operations that are also responsible for preserving the invariant. In some other programs, however, there are classes with methods that rely on the invariant but play no active role in preserving it, because they perform no mutation. (They just observe the object's state, but don't modify it.) These methods need to be `synchronized` too, in order to ensure that they only observe the object's state after some sequence of complete mutator operations has been performed, rather than in the middle of one of the mutator operations.

As you can see, freedom from races is the result of teamwork between the programmer and the Java system: the programmer uses a representation invariant to ensure that all is well so long as `synchronized` methods are never executing simultaneously in different threads, and the Java system plays its part by respecting those `synchronized` annotations.

Exercise 15.6
Add the keyword `synchronized` at the two places indicated above and verify experimentally that these race conditions no longer occur.

Nested calls to `synchronized` methods and deadlock

You might wonder what happens if one `synchronized` method invokes another one. In terms of our analogy, a thread that is currently inside a locked room is trying to do another operation that requires being inside a locked room.

In Java, if the second method is on the same object, there is no problem at all. The thread is already inside that object's locked room, and so can go ahead with the nested `synchronized` method. Moreover, when it is done with that inner method, it doesn't make the blunder of unlocking the door and leaving the room. Instead, it waits until the outer `synchronized` method is done before unlocking.

How about if the inner `synchronized` operation is on a different object? Our physical analogy of locked rooms starts to break down here. The thread manages to stay inside its current locked room while waiting for the other room to become available. Then without unlocking the room it is in, it locks the new room and is (somehow) simultaneously in two locked rooms.

There is a real pitfall here for unwary programmers. Suppose one thread is inside the locked room for object A, while another is inside the locked room for object B. Now the first thread tries to invoke a `synchronized` method on B while the second thread tries to invoke a `synchronized` method on A. Each thread waits for the other room to become available. But since each is waiting with its own room locked, neither room ever will become available—the two will simply wait for each other forever. This situation, in which threads cyclically wait for one another, is known as *deadlock*.

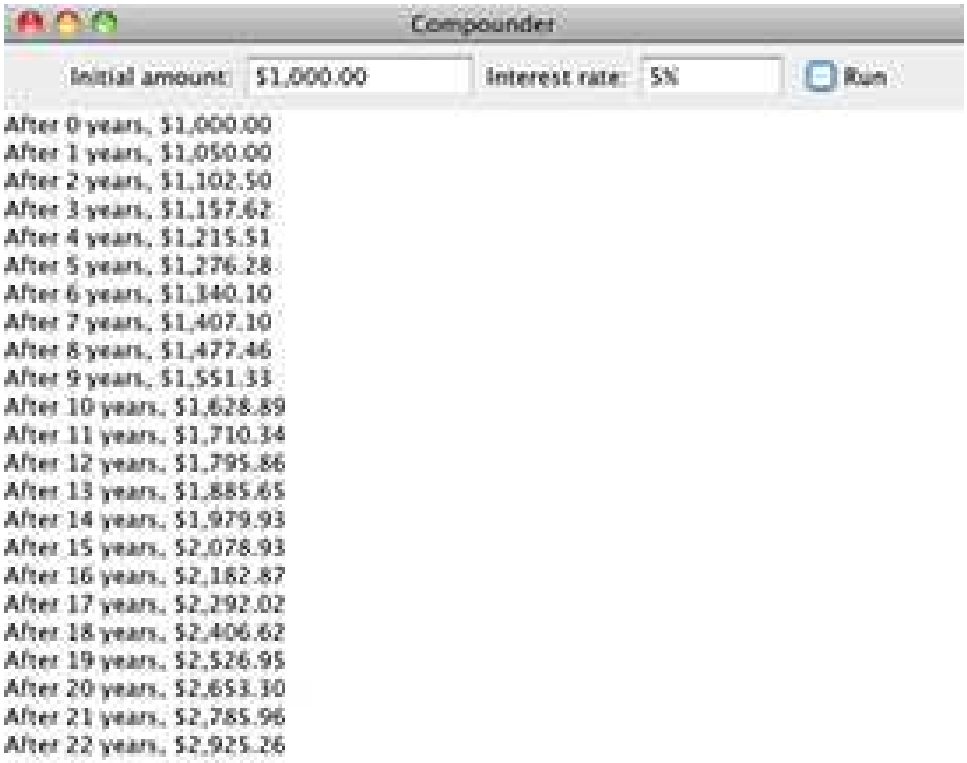

Figure 15.4: Compound interest simulation app

15.4 An application: simulating compound interest

Imagine that you have just started work for a small company that produces Java programs for use in education. One of the company's programs is used to illustrate how compound interest works; it is shown in figure 15.4. This program simulates the passage of years at a rate of one year per second, displaying information in the scrolling area that occupies the main portion of the program. The figure is just a snapshot, showing what it looked like after 22 simulated years had passed, but keep in mind that it keeps getting

updated. Meanwhile the top "control panel" portion of the program has three controls. One is a checkbox labeled "Run" that can be used to pause the simulation or resume it. (The program actually starts in the paused state; the box was clicked on 22 seconds prior to the snapshot in the figure.) The other two controls allow the initial amount of money and the interest rate to be changed. If the user changes either of these, the output area is cleared and the simulation is reset back to year 0. The program is included on the publisher's website, so you can try it out.

Like many junior programmers, you have been assigned to fix bugs in the company's existing programs, rather than writing a new program from scratch. Occasionally you may get to add a new feature.

The boss comes to you with an interesting problem concerning the compound-interest simulation program. Although it generally seems to work fine, a few customers have reported seeing it occasionally produce bizarre behavior, which they have never managed to replicate. The common thread is that after changing one of the values (initial amount or interest rate) while the simulation was running, the customers report seeing output on the screen that was clearly wrong, or was missing some years. Your boss normally wouldn't care that a few customers were claiming to occasionally see strange things, but it happens that some of them are very important clients that the company is trying to make a good impression on, and right now the reliability of the program is in question. The boss tells you your job is to get to the bottom of the matter and restore the company's reputation for rock-solid quality.

Since you have had the benefit of learning from a textbook that introduced concurrent programming, you immediately blurt out to the boss that you are sure—without even looking at the code—that you know what the problem is. Obviously the program must have two threads, one to simulate the passage of years and one to respond to the user interface (much like in the puzzle with the poltergeist). Clearly whatever boneheaded programmer preceded you at the company didn't bother to put "`synchronized`" where he should have, and so there is a race condition that causes problems when the user makes a change just at the instant a year is being simulated. You say that you can fix the problem in a few minutes by just sticking "`synchronized`" in front of some methods.

The boss is not thrilled. This may be partially an emotional response, given that you just called his teenaged son a bonehead. However, mostly it is just good, cautious business sense. Right now, the program appears to work when tested. When you add the `synchronized` keywords, it still

will appear to work when tested. How can the boss confidently tell the VIP clients that you definitely have gotten to the bottom of the matter and solved their mysterious problem? How can he know that your explanation accounts for their symptoms when the symptoms aren't even showing up in testing in the first place? How can he be sure the symptoms won't keep showing up for the client?

Therefore, you agree on a more careful plan of work:

1. You will examine the code and come up with a few specific race scenarios that would exhibit the kind of behavior the clients have mentioned. That is, you'll map out exactly what order things would have to happen in to make the symptoms show up.

2. Then you'll rig the program so that these race conditions can be made to repeatably happen, rather than just once in a blue moon, so as to show your boss that they are real. You'll do this by introducing extra time-delay sleeps at the critical points, so that rather than having to change one of the values at just exactly the wrong moment, you'll have a much bigger window of opportunity.

3. Then you'll put the `synchronized` keywords in that you are convinced will solve the problem.

4. Finally, you'll show that even with the extra time delays that you put in to make the races easy to trigger, the symptoms no longer show up in your fixed version.

If your theory is correct, then the problem is definitely localized within the `CompoundingThread` class, which provides the guts of the simulation; the other classes just provide the user interface, and seem quite innocent. From your perspective, all you need to know about them is how they relate to the `CompoundingThread` class:

- The main program class, `Compounder`, provides two methods for managing the scrolling output area: `outputLine` (for adding an additional line of output) and `clearOutput` (for clearing the area).

- The user interface calls two of the `CompoundingThread`'s methods, `setInitialAmount` and `setInterestRate`, to convey this information in, and also uses `enable` and `disable` methods, much like the poltergeist's.

Here is the code for the class in question:

```
package edu.gustavus.mcs178.compounder;

public class CompoundingThread extends Thread {

    private boolean enabled;
    private double initial, current, multiplier;
    private int year;
    private Compounder c;
    private java.text.NumberFormat fmt;

    // Invariant:
    //   (1) current = initial * (multiplier raised to the
    //       year power)
    //   (2) year also specifies how many lines of output c has
    //       gotten since it was last cleared, corresponding to
    //       years from 0 up through year-1.


    public CompoundingThread(Compounder comp) {
        c = comp;
        fmt = java.text.NumberFormat.getCurrencyInstance();
    }

    synchronized private void waitUntilEnabled() throws InterruptedException {
        while(!enabled){
            wait();
        }
    }

    synchronized public void disable() {
        enabled = false;
    }

    synchronized public void enable() {
        enabled = true;
        notify();
    }

  // (continued)
```

```
    public void run() {
        try{
            while(true) {
                Thread.sleep(1000);
                waitUntilEnabled();
                doYear();
            }
        } catch (InterruptedException e) {
            // ignore, but stop running
        }
    }

    private void doYear() {
        c.outputLine("After " + year + " years, " + fmt.format(current));
        year++;
        current *= multiplier;
    }

    public void setInitialAmount(double amount) {
        initial = amount;
        initialize();
    }

    public void setInterestRate(double rate) {
        // note that a rate of 5% (e.g.) would be .05, *not* 5
        multiplier = 1 + rate;
        initialize();
    }

    private void initialize() {
        current = initial;
        year = 0;
        c.clearOutput();
    }
}
```

The most important thing to note about this code is that we maintain an instance variable `enabled` in that keeps track of whether the thread is enabled, and we needed to synchronize the procedures `waitUntilEnabled()`,

and `disable()`, and `enable()` to avoid race conditions.

Although we don't describe the other class files here, they are included in the software included with the associated lab, and we encourage you to peruse those files. We note that we did add one other Swing class, a `JCheckBox` to the main class `Compunder`, which allows you to turn the thread off and on using `disable()`, and `enable()`. You should read the code in `Compunder` and look at the Java documentation for `JCheckBox` to see how it works.

Finally, `CompoundingThread` uses a fancy library class called `java.text.NumberFormat` to format the current amount of money for the output line. For example, if `current` is 19.5, then the expression `fmt.format(current)` would evaluate to the string `"$19.50"`. Not only does this take care of details like making sure there are two digits after the decimal point, it also has an additional big win: it automatically adjusts to other currencies that are used elsewhere in the world. (For more details, look up the documentation for this library class.)

Exercise 15.7
As an important preparation for figuring out the race conditions, you need to understand the class's invariant. Assume for the moment that there is no concurrency, and write out explanations of how the invariant is preserved by each of the three methods `doYear`, `setInitialAmount`, and `setInterestRate`.

Exercise 15.8
Now come up with at least three different specific misbehaviors that could result from a race between `doYear` and one of the other methods. Explain exactly what order the events would have to occur in. For example, you might say that right between the user interface thread setting the year to 0 and clearing the output, the simulation thread might slip in and do a complete invocation of `doYear`. Also, explain for each scenario what the user would see. Try to come up with at least three scenarios with different symptoms from one another.

Exercise 15.9
Now make each of your misbehaviors happen. Rather than developing the knack of getting the timing just perfect, you should

use `Thread.sleep` to open up a great big window of opportunity. For example, if you put a several second sleep between setting the year to 0 and clearing the output, it is a sure thing that at least one `doYear` will slip into that gap. (Provided, of course, that the simulation is enabled to run, rather than being paused in the disabled state.)

Exercise 15.10
Now add the `synchronized` keyword to the appropriate methods, and verify that the misbehaviors have all gone away, even when you use `Thread.sleep` to give them ample opportunity to show up.

Your boss is sufficiently impressed with your work to let you add a new feature customers have been requesting. Many people aren't as interested in answering questions like "if I invest $1000 now, how much will I have when I retire" as they are in questions like "if I invest $1000 each year from now until I retire, how much will I have?" So, you are to add a feature to the program so that it has *two* fields for monetary input: the initial amount, and the additional amount to add each year.

Of course, this gets you into the user-interface part of the program, which you've been able to ignore until now. The most relevant portions are the `InitialAmountField` class and the part of the `Compounder` class that creates the initial amount field. You'll be able to add the new field just by following that example, since it is another currency amount field.

The part of the `Compounder` class's constructor that creates the initial amount field and adds it to the control panel is as follows:

```
controlPanel.add(new Label("Initial amount:", Label.RIGHT));
controlPanel.add(new InitialAmountField(1000.00, compThread));
```

The first line adds a `Label`, which is just a fixed chunk of text. The argument `Label.RIGHT` indicates that it should be positioned to the far right end of the space it occupies, which looks correct given that it ends with a colon and is followed by the field in which the amount is entered. The `InitialAmountField` itself follows. The first argument to its constructor is the value the field should start out with (`1000.00`), pending any modification by the user, while the second argument, `compThread`, is the `CompoundingThread` that should receive `setInitialAmount` notifications.

Here's the `InitialAmountField` class:

```
public class InitialAmountField extends FormattedField {

  private CompoundingThread compThread;

  public InitialAmountField(double initialValue, CompoundingThread ct){
    super(10, java.text.NumberFormat.getCurrencyInstance());
    compThread = ct;
    Double value = new Double(initialValue);
    setValue(value);
    valueEntered(value);
  }

  public void valueEntered(Object value){
    compThread.setInitialAmount(((Number) value).doubleValue());
  }
}
```

There is some fairly tricky stuff in this little class, and although you don't really need to understand it to make another one just like it, we can't stand to pass up an opportunity for explanation.

The superclass, `FormattedField`, handles the general problem of being a text-entry field that has some specified special format—in this case, the format of a currency amount. Its constructor needs to be told how wide a field is desired and what format should be used; those are the two arguments in

```
super(10, java.text.NumberFormat.getCurrencyInstance());
```

The format object that is passed in as the second argument here formats numbers as currency amounts, but in general it can specify formats for all sorts of things—for example, dates as well as numbers. Therefore, the interface of the `FormattedField` class needs to be very general. In particular, its `setValue` method takes an arbitrary `Object` as an argument, so as not to be limited to numbers. The only problem is that the `initialValue`, which is a `double`, isn't an `Object`. Any instance of any class is an `Object`, since all classes are descended from `Object`. However, `double` isn't a class (nor are `int`, `boolean`, or the other basic numeric and character types). So, we need to make an `Object` that holds the `double` inside; this is what the `Double` class is for. We make a `Double` called `value` holding the `initialValue`, and we pass that `Double` object into `setValue`.

When the user types a new value into the field, the `FormattedField` class responds by invoking the `valueEntered` method to process this newly entered value. We also do this with the initial value, so that it gets handled the same way. Again, because `FormattedField` needs to work for all kinds of data, it passes an `Object` to `valueEntered`. Our `valueEntered` method needs to recover the actual `double` value from that `Object`. The first thing we do is to declare that we know the `Object` must be a `Number`. (The `Number` class is the superclass of `Double`. It has other subclasses that similarly hold other kinds of numbers.) We do this with the notation `(Number)`, which another *cast*, much like the `(int)` we saw earlier. Then we invoke the `Number`'s `doubleValue` method to retrieve the actual value as a `double`, and finally notify the `CompoundingThread` by invoking its `setInitialAmount` method.

Exercise 15.11
Add a new labeled field for the annual contribution, and arrange that it gets passed to the `CompoundingThread`. Modify the `CompoundingThread` so that it incorporates this additional amount each year.