Out of print; full text available for free at http://www.gustavus.edu/+max/concrete-abstractions.html

CHAPTER FIFTEEEN

Java, Applets, and Concurrency

15.1 Introduction

In this chapter, we'll turn our attention from Scheme to another programming language, Java. Scheme has served us well, providing a simple context for learning many important computing concepts. However, now it is advantageous to look at another language, for the following reasons:

- We want to emphasize the continuity of concepts between what we've done in the rest of the book and the programming you are likely to do after leaving us.
- Although the big concepts carry over, there are some variations at a more superficial level, and it will help you if we point those out.
- We have a couple new concepts we consider it important to introduce that would not be easy or natural to introduce in Scheme. In particular, we are going to move away from programs that sequentially do one thing at a time to programs that divide their attention between multiple activities.

In the next section, we'll provide a basic introduction to the Java programming language by taking a program you are familiar with—the compu-duds program from the previous chapter—and rewriting it in Java. We'll show how the same object-oriented programming concepts are realized in a different notation.

After completing our crash course in how object-oriented programming maps into Java, we'll turn our attention to a new class of programs: those that present the user with a graphical interface and respond to the user's manipulation of interface

Excerpted from Concrete Abstractions; copyright © 1999 by Max Hailperin, Barbara Kaiser, and Karl Knight

elements, such as clicking on buttons. In particular, we'll look at *applets*, which are programs with graphical user interfaces that are intended to be embedded into World Wide Web documents.

Finally, we'll use Java applets as the setting in which to introduce the most conceptually significant material, *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 applet.

15.2 Java

The Java programming language is heavily biased toward the object-oriented style of programming we introduced in the previous chapter. All programming is organized into classes: Every single procedure you write in a Java program must be associated with some class, unlike in Scheme, where in addition to classes' methods, we also had "normal" procedures that floated free from the class hierarchy. In Java, even a procedure that just squared a number would have to be part of a class.

Another big difference between Java and Scheme is that we'll need to explicitly declare what *type* of object or value each name is a name for. For example, in the compu-duds program, we'll have to explicitly state that the argument to the add method of the item-list class is an item, that the choose method returns an item, and that the item-vector is a vector that holds items (no big surprise). On the one hand, this is sort of a nuisance because you have to type in all sorts of *declarations* that don't do any of the program's real work but instead just state what type of objects are being manipulated. On the other hand, the presence of these declarations provides a number of advantages:

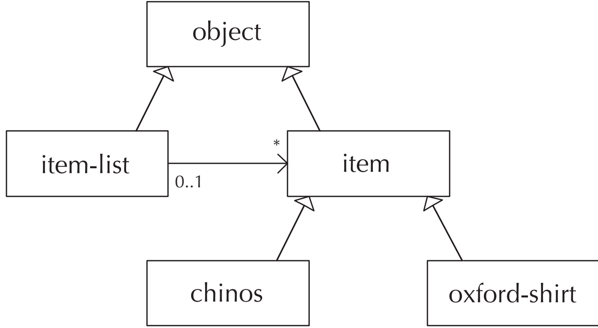
- The declarations allow the automatic detection of many common programming errors. By now you surely have applied a Scheme procedure to the wrong kind of argument. If the faulty argument gets passed around untouched for a while before any operation is performed on it that causes symptoms to arise, you can have quite a debugging puzzle. If the symptoms only arise in some infrequently executed part of your program, matters are worse. In Java, by contrast, you can get immediate feedback regarding any place in your program where a type mismatch occurs.
- The declarations provide useful information to readers of your program. For example, does the item-list class's delete method expect to be told which item to delete or which numerical position it should delete the item from? One look at the type declaration for the parameter answers this question.

- The declarations can actually contribute to the brevity of your program by allowing method names to be abbreviated. In Scheme, every time you operated on an item, you needed to specify that it was an item, by using method names like `item/price` or `item/display`. In Java, you declare once and for all that it is an item and then can operate on it with method names like `price` or `display`.

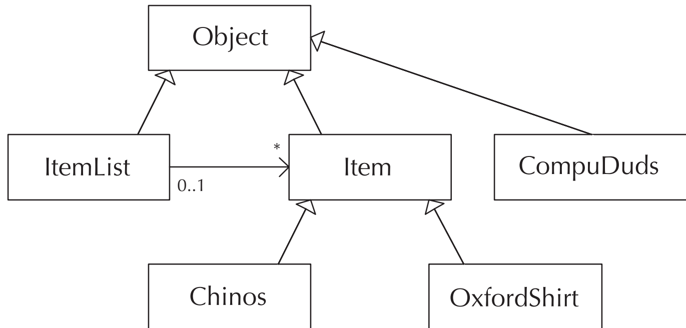
These two areas are broad differences between Java and Scheme: types must be declared, and the entire program must be organized into classes. However, smaller differences also exist, particularly at the superficial level of what the notation looks like. A few of these are purely arbitrary, reflecting the different background and taste of the languages' designers or intended audience. Most, however, can be accounted for in terms of a desire for succinctness: Java is a much more terse notation than Scheme and allows lots of optional shorthands that can make it even terser. For an example, consider what a method of the item-list class does to get the current value of the `num-items` instance variable stored in `this` item-list (i.e., the one the method is operating on). In Scheme we wrote `(item-list/get-num-items this)`. In Java, the "longhand" option would be `this.numItems`, and the shorthand that is normally used is just `numItems`, because if you don't specify what object an instance variable should be fetched from, it is assumed to be `this`.

We renamed the instance variable from `num-items` to `numItems` because arithmetic operators in Java are written between their operands, so `num-items` would mean `num` minus `items`. Because the hyphen means subtraction—and no spaces are required around it—we are forced to separate words some other way. The Java convention is to use capitalization of the first letter of each new word within the name. (For class names, even the first word is conventionally capitalized, for example, `ItemList`, whereas for other names the first word is left lowercase, as in `numItems` or `displayPrice`.)

Enough general comments about Java; the time has come for an example program. We'll do a straightforward translation of the compu-duds (or CompuDuds) program into Java, sticking close to the Scheme original so you have a basis for comparison. To refresh your memory, here is the class hierarchy from the Scheme version:



The variant of this class hierarchy we will implement in Java includes two changes:



The first change concerns the Java naming convention described in the previous paragraph (e.g., `ItemList` instead of `item-list`). The second change is the addition to our hierarchy of the class called `CompuDuds`. We need this class as a place to put the "normal" procedures that in Scheme floated free from the class hierarchy. One of those procedures was the main `compu-duds` procedure that was the program itself; in Java, this procedure is renamed to be the `main` method of the `CompuDuds` class, because the Java system assumes that the program to run is the `main` method of the specified class.

Because of its relative simplicity, we will start our description of the CompuDuds program with the `Item` class, followed in short order by its subclasses `OxfordShirt` and `Chinos`, after which we will look at the `ItemList` and `CompuDuds` classes. Because the code is so extensive, and there are so many new language features you will be learning, we will divide the remainder of this section into subsections corresponding to each of these classes.

Item Class

Recall that the `Item` class maintained the price of each item and otherwise existed mainly as a base for its subclasses. Here is the full Java implementation of `Item`, which must be in a file named `Item.java`:

```
public class Item extends Object {

  private int price;

  public Item(int initialPrice){
    price = initialPrice;
  }
```

```
  public int price(){
    return price;
  }

  public void display(){
    CompuDuds.displayPrice(price);
  }

  public void inputSpecifics(){
  }

  public void reviseSpecifics(){
    inputSpecifics();
  }
}
```

To understand the meaning of this notation, recall that in Scheme we first used `define-class` to provide some basic information about the class: its name, superclass, instance variable names, and method names. We then added the method implementations one by one, using `class/set-method!`. In Java, by contrast, we provide the method implementations in the same chunk of code with the rest of the information about the class. This notation is as though in Scheme all the calls to `class/set-method!` were stuck in as extra arguments to `define-class`. The overall form is therefore as follows:

```
public class Item extends Object {
  // all the instance variables and methods go in here
}
```

The first line states that the Item class has the `Object` class as its superclass. Actually, we can leave "`extends Object`" out because Java assumes the superclass is `Object` if you don't specify otherwise. The keyword `public` indicates that this class can be freely used from anywhere within the overall program. For simplicity, we'll stick exclusively to `public` classes, although in large programs it can be helpful to delimit the visibility of some classes, as a complexity control measure. Each `public` class needs to be defined in a file named after the class; for example, we stated earlier that the definition of the `Item` class must be in `Item.java`. Because all our classes will be `public`, each will need to be put in its own individual file.

We've taken this early opportunity to show what a comment looks like in Java; the line

```
  // all the instance variables and methods go in here
```

is a comment, because anything from // to the end of the line is ignored by the Java system, much like Scheme comments that extend from ; to the end of the line. Java also offers comments that extend from /* to */, even if the */ is on a subsequent line, such as this example:

```
/* all the instance variables and methods go
   in here  */
```

The instance variables and methods in a class's definition can be listed in any order between the outer curly braces. We could list all the instance variables and then all the methods or vice versa or mix them up. In the examples we give, we'll follow the convention of first listing the instance variables and then the methods.

The class `Item` has a single instance variable, named `price`, that is declared in the line

```
private int price;
```

Notice that `price` is flagged `private`. This means that no code outside the class can get or set it: The only access is through this class's methods. If you look back over the code in the previous chapter, you'll see that we've always used instance variables that way. Doing so makes the program as a whole much more resilient to changes in individual classes.

In addition to the instance variable's name, we also included one other piece of information, between the `private` keyword and the name, namely, what *type* of value the instance variable will hold. (We warned you that such declarations were coming.) In particular, we've declared that the `price` instance variable will always hold an integer (called an `int` in Java).

Consider next how the five methods for class `Item` are written in Java. The first of these, which is named `Item`, has a special role—it is used to initialize a new instance of the `Item` class when that instance is constructed, which is analogous to the Scheme version's `init` method. In Java, instead of being called `init`, it is given the same name as the class. Also, in Java terminology it isn't strictly speaking a method but rather a *constructor*. In Scheme, we used the same word, *constructor*, to instead refer to the `make-item` procedure. Recall that in Scheme we could construct an item with the following code:

```
(define example-item (make-item 1950))  ; 1950 cents = $19.50
```

The corresponding Java code would be

```
Item exampleItem = new Item(1950);
```

This statement declares `exampleItem` to be a variable of type `Item` and assigns it the value returned by the *class instance creation expression*, `new Item(1950)`. Because

of the parenthesized `1950`, the newly created `Item` is initialized by the constructor applied to 1950. We define the constructor `Item` by

```
public Item(int initialPrice){
  price = initialPrice;
}
```

This code introduces several features of Java. As you can see, this constructor is `public`, in other words, available for use from outside the class. In parentheses after the constructor name is the list of parameters; if there were more than one, they would be seperated by commas. The `this` parameter is not listed. Thus, `Item` takes a parameter named `initialPrice`, which is of type `int`. The reason Java omits the `this` parameter is not because it doesn't exist but rather because every constructor and method necessarily has it, so for conciseness it isn't explicitly mentioned. Next comes the body of the constructor, surrounded by braces. The single statement within the body illustrates the use of the equal sign as a setter: The instance variable to the left of the equal sign is being set to the value indicated on the right. Thus, the `price` instance variable is set to the desired initial price. This assignment statement ends with a semicolon.

Consider next the `price` method:

```
public int price(){
  return price;
}
```

The `int` between the `public` and `price()` says that `price` is a method that returns an integer value. Notice that we need to explicitly say that the value should be `return`ed, unlike in Scheme. The empty parentheses are because this method takes no arguments, other than the implicit `this`.

Before describing the remaining methods in the `Item` class, consider the following code, which illustrates how the `price` method could be invoked:

```
int itsPrice = exampleItem.price();
```

This statement (which assumes that `exampleItem` was declared as before) declares a new variable of type `int` and assigns it the value returned by invoking the `price` method on `exampleItem`, namely, `1950`. The pair of parentheses after the method name contains all the arguments other than which object the method is being invoked on; that special argument (`exampleItem`) appears before the method name, separated by a dot. Because it is so common for methods to invoke other methods on the same object (i.e., on `this`), Java allows the shorthand form in which you just write `price()` instead of `this.price()`.

Returning to the remaining methods in the Item class, consider next display, which illustrates how to call a method that is not associated with any particular object:

```
public void display(){
  CompuDuds.displayPrice(price);
}
```

First note that the word void, appearing before the method name, is in the position where normally we would specify what kind of value this method returns. What it means is that no value at all is returned: The method is invoked solely for its effect. This use of void contrasts with the way we declared the Item constructor. Constructors have no return type specifier at all, not even void, whereas methods must have a return type specifier, whether void or a real type like int.

The interesting feature of display's invocation of displayPrice is that CompuDuds isn't an object; it is a class. As descibed, we normally put an object before the dot to show which object the method should be invoked on. However, the displayPrice procedure isn't really an object's method at all; it is just a normal procedure. As we mentioned earlier, even normal procedures in Java need to be grouped into classes; in Java terminology, they are called *class methods* as opposed to *instance methods*. We've given the CompuDuds class the responsibility of holding all the utility procedures for our program, like this one for showing a price as dollars and cents. To access such a procedure, we just write the class name before the dot.

This leaves the final two methods in the Item class, which are fairly straightforward:

```
public void inputSpecifics(){
}

public void reviseSpecifics(){
  inputSpecifics();
}
```

We don't have much new to say about these two procedures, other than to point out that inputSpecifics() has an empty body and therefore does nothing. Otherwise, these two procedures are direct translations of their Scheme counterparts.

OxfordShirt Class

The OxfordShirt class extends a superclass other than Object, namely, Item. Following is the portion of its implementation dealing with its instance variables and constructor. Take a particularly close look at the line involving super that we've flagged with a comment; we'll have more to say about it presently:

```
public class OxfordShirt extends Item {

  private String color;
  private int neck;
  private int sleeve;
  private boolean specifiedYet;


  public OxfordShirt(){
    super(1950);  // <-- allow Item class to initialize the price
    specifiedYet = false;
  }

  // Methods display() and inputSpecifics() go here;
  //  we will show them later.
}
```

First note that two new Java types are used for the instance variables: `String` and `boolean`. These types have pretty much the same meaning as in Scheme.

The line we flagged invokes the constructor from the superclass (i.e., the `Item` class). In the Scheme version, the `init` method for the `oxford-shirt` class invoked the `init` from the `item` class as follows:

```
(item^init this 1950)
```

Here in Java, the analogue is `super(1950)`.

The `display()` method in `OxfordShirt` introduces a number of new features:

```
  public void display(){
    if(specifiedYet){
      System.out.print(color);
      System.out.print(" Oxford shirt, size ");
      System.out.print(neck);
      System.out.print("/");
      System.out.print(sleeve);
      System.out.print("; ");
    } else {
      System.out.print("Oxford shirt; ");
    }
    super.display();  // <-- now do displaying the Item way
  }
```

First note that `display()` uses the `if...else` construct to choose between two forms of output. Also, the code contains a number of instances of the expression `System. out.print(...)`. These produce output for the user to see, analogously to Scheme's `display` procedure. Similarly, `System.out.println(...)` will produce output and then start a new line. In particular, `System.out.println()` has the same effect as Scheme's `newline` procedure. Actually, `print` and `println` are methods that can be used to cause output to be produced elsewhere as well (like into a file), and `System.out` is the object we invoke those methods on when we want the output to go to the normal place, typically the user's display screen.

Regarding the statement flagged with a comment in the preceding code, recall that the Scheme version of the `oxford-shirt` class's `display` method ended by invoking the `item` class's `display` as follows:

```
(item^display this)
```

In Java, the `display` method from the superclass is invoked as `super.display()`.

Finally, consider the `inputSpecifics()` method in the `OxfordShirt` class:

```
public void inputSpecifics(){
  System.out.println("What color?");
  String[] colors = {"Ecru", "Pink", "Blue", "Maize", "White"};
  color = CompuDuds.inputSelection(colors);
  System.out.print("What neck size? ");
  neck = CompuDuds.inputIntegerInRange(15, 18);
  System.out.print("What sleeve length? ");
  sleeve = CompuDuds.inputIntegerInRange(32, 37);
  specifiedYet = true;
}
```

Note first that the actual input is offloaded to the two utility procedures, `inputSelection` and `inputIntegerInRange`, just like in the Scheme version. We'll provide their definitions when we get to the `CompuDuds` class.

We need to explain a number of new Java features that are implicit in the following line:

```
String[] colors = {"Ecru", "Pink", "Blue", "Maize", "White"};
```

This is Java shorthand for the following code, which is what we will actually explain:

```
String[] colors = new String[5];
colors[0] = "Ecru";
colors[1] = "Pink";
```

```
colors[2] = "Blue";
colors[3] = "Maize";
colors[4] = "White";
```

The first line declares `colors` to be a temporary local variable; that is, rather than being a permanent instance variable of the object, it is a name that can only be used from its declaration to the end of the curly braces. The variable `colors` is declared to have the type `String[]`, which means that `colors` is an *array* (Java's word for vector) that can hold `String`s. (The brackets indicates that it is an array, whereas the name before the brackets indicates what type of object the array holds.) The expression `new String[5]` is analogous to Scheme's `(make-vector 5)` except that the array can only hold `String`s. Because of the equal sign, the newly made array becomes the value of `colors`.

The remainder of the code illustrates how array elements are set in Java. For example, the Java code

```
colors[2] = "Blue";
```

is directly analogous to the Scheme code

```
(vector-set! colors 2 "Blue")
```

Just as in Scheme, array elements in Java are numbered from 0, not 1.

Finally, note that the shorthand form involving {`"Ecru", "Pink", "Blue", "Maize", "White"`} has the big advantage of leaving the counting to the computer.

Chinos Class

The `Chinos` class is so similar to the `OxfordShirt` class that we won't bother showing it all here; the full version is on the web site for this book. The only really novel issue arises in the `inputSpecifics` method, at the point where the Scheme version does the following:

```
(chinos/set-inseam! this (input-integer-in-range
                          29
                          (if (chinos/get-cuffed this)
                              34
                              36)))
```

This code sets the `inseam` instance variable to a value chosen from a range that always has 29 as its minimum value but can extend to a maximum of either 34 or 36 depending on whether the chinos are to be cuffed or not. We've seen that Java

also has an `if` construct; however, it can't be used in this fashion. The reason is that Java, unlike Scheme, distinguishes between *expressions*, which calculate values, and *statements*, which command the computer to carry out some action. An `if`, in Java, is a conditional statement that selects between two alternative actions to carry out, not a conditional expression that can select between alternative values. For example, we used `if` in the `OxfordShirt` class's `display` method to select which of two forms of output to produce, but we can't use it here to choose between 34 and 36. We'll see that there are two options for dealing with this in Java.

The most common approach taken by Java programmers is simply to reformulate the task in such a way that a conditional statement becomes appropriate. For example, they would write

```
if(cuffed){
  inseam = CompuDuds.inputIntegerInRange(29, 34);
} else {
  inseam = CompuDuds.inputIntegerInRange(29, 36);
}
```

The disadvantage, of course, is that the part that doesn't vary winds up repeated.

An alternative is to stick with a conditional expression, as we had in the Scheme version. It turns out that Java *does* have conditional expressions; they just aren't called `if`. Here is how the code would look written this way:

```
inseam = CompuDuds.inputIntegerInRange(29, cuffed ? 34 : 36);
```

As you can see, the conditional expression has a peculiar syntax; you state the controlling boolean expression (here `cuffed`), then a question mark, the alternative to use if the condition is true, a colon, and the other alternative. Because this syntax is rather idiosyncratic, many Java programmers avoid it. However, it does solve the problem of allowing control to be exerted only over that which needs to vary.

ItemList Class

Following is an abbreviated version of `ItemList`'s implementation, giving complete code only for its instance variables, its constructor, and the method `empty`:

```
public class ItemList extends Object {

  private Item[] itemVector;
  private int numItems;
```

```
  public ItemList(){
    itemVector = new Item[10];
    numItems = 0;
  }

  public boolean empty(){
    return numItems == 0;
  }

  /* Code will be given later for the following methods:
   *
   *   public void display()
   *   public int totalPrice()
   *   public void add(Item item)
   *   private void grow()
   *   public void delete(Item item)
   *   public Item choose()
   */
}
```

Note the double equal sign in `empty`, which checks for equality just like a single equal sign does in Scheme. We have to have a separate notation for equality checking because the single equal sign is used in Java for assigning new values to variables.

The `ItemList` class's `display` method needs to loop through all the individual items, displaying them one by one using the Item class's `display` method. In Scheme we used the `from-to-do` procedure to loop through the positions the items occupy in the item vector; in Java, the equivalent is the `for` loop:

```
  public void display(){
    for(int index = 0; index < numItems; index = index + 1){
      System.out.print(index + 1);
      System.out.print(") ");
      itemVector[index].display();
      System.out.println();
    }
    System.out.print("Total: ");
    CompuDuds.displayPrice(totalPrice());
    System.out.println();
  }
```

The `for` loop is controlled by the parenthesized code immediately following the keyword `for`, which consists of three parts separated by two semicolons. The first

part is an initializer, which is evaluated once before everything else is evaluated. In our case we declare and initialize the integer variable `index`, used for counting through the positions. It is a temporary local variable that can be used up to the curly brace ending the `for` loop's body. The second part of the loop control, `index < numItems`, specifies how long the loop should continue: so long as the index hasn't run past the part of the array that is actually in use. Finally, before the body of the loop we specify how `index` should be changed each time, by setting it to the old value plus 1. Note that this setting is actually done *after each execution* of the loop body, even though it is written before. (For example, the first time the body is executed, `index` is 0, because it hasn't yet been incremented to 1.)

The line

```
itemVector[index].display();
```

is the equivalent of the Scheme

```
(item/display (vector-ref item-vector index))
```

In other words, the brackets are being used here to indicate that an element is to be fetched from the array, like Scheme's `vector-ref` operation. We saw earlier that if the brackets are used to the left of an equal sign, they corresponded instead to `vector-set!`, and after `new`, they indicated the size of an array being created, analogous to `make-vector`. One of the prices of concision is that the same symbols get pressed into service for multiple roles. Once you become familiar with the language, you should have no problem telling from context what role a particular pair of brackets is serving.

The method for computing the total price of the items doesn't introduce any new Java features, but it does give another example of a `for` loop:

```
public int totalPrice(){
  int sum = 0;
  for(int i = 0; i < numItems; i = i + 1){
    sum = sum + itemVector[i].price();
  }
  return sum;
}
```

Because there is nothing new in this method, maybe it is time to introduce a couple extra Java shorthands that you'll frequently see used. You'll notice that there are two places where a variable is set to a new value found by adding the old value to something: `i` gets 1 added to it, and `sum` gets a price added to it. This pattern shows up frequently enough to have a shorthand; using it we can write

```
public int totalPrice(){
  int sum = 0;
  for(int i = 0; i < numItems; i += 1){
    sum += itemVector[i].price();
  }
  return sum;
}
```

Because incrementing a variable by 1 is so common, you can get one step more terse and instead of writing `i += 1`, write `i++`.

The `add` method is a fairly straightforward translation of the Scheme version into Java:

```
public void add(Item item){
  if(numItems == itemVector.length){
    grow();
    add(item);
  } else {
    itemVector[numItems] = item;
    numItems++;
  }
}
```

The only novel aspect concerns how one finds the length of a array in Java: Although the notation is different (the dot and then `length`), what it does is the same as Scheme's `vector-length`.

The `grow` method is also a fairly straightforward translation of the corresponding Scheme method:

```
private void grow(){
  Item[] newItemVector = new Item[itemVector.length * 2];
  for(int i = 0; i < itemVector.length; i++){
    newItemVector[i] = itemVector[i];
  }
  itemVector = newItemVector;
}
```

Note that `grow` is declared `private`, because it is only used internally within the `Item` class.

In the `delete` method, we'll use not only the `++` shorthand but also the analogous `--` shorthand, which decrements a variable by 1:

```
public void delete(Item item){
  for(int i = 0; i < numItems; i++){
    if(itemVector[i] == item){
      for(int j = i + 1; j < numItems; j++){
        itemVector[j - 1] = itemVector[j];
      }
      itemVector[numItems - 1] = null;
      numItems--;
      return;
    }
  }
  System.err.println
    ("Error: ItemList delete done with Item not in list.");
  System.exit(1);
}
```

When this method finds the item that is to be deleted, it slides all the items after it in the array down by one position. Then it puts a special `null` value into the newly vacated array position. The `null` value is used as a placeholder when an array element or variable does not refer to any object at all. Once the array has been updated in this manner, and the `numItems` is decremented, the method immediately `return`s, without continuing to search further for the item (because it has already found and deleted it). Notice that because this method isn't supposed to return any value, we have no expression between the keyword `return` and the semicolon. Rather than specifying a value to return, the `return` statement is serving only to terminate the method without further execution.

If the loop instead *does* run to completion, because the item is not found, an error message is printed out and the program exits. The error message is printed on `System.err`, which is just like `System.out`, except that even if the user asks for normal output to go somewhere other than the display screen (for example, to go into another program for further processing), the output produced on `System.err` will still be displayed on the screen. The `println` method is being used here both to print the message and to start a new line afterward. The call to `System.exit` causes the program to stop; by convention a value of 0 is passed to indicate a normal termination, whereas nonzero values (like 1) are used to indicate that something went wrong.

The remaining method from the `ItemList` class, `choose`, doesn't introduce any new features of Java:

```
public Item choose(){
  if(empty()){
    System.err.println("Error: choose done on empty ItemList.");
```

```
      System.exit(1);
    }
    System.out.println("Which item?");
    display();
    return
      itemVector[CompuDuds.inputIntegerInRange(1, numItems) - 1];
  }
```

CompuDuds Class

At this point, we've covered all of the classes that existed in the Scheme version of compu-duds, but the Java version still has one class left, the `CompuDuds` class that we are using to collect together all the utility procedures, like `displayPrice` and `inputIntegerInRange`. This class will also provide the main program itself, which repeatedly lets the user choose what to do next.

The `CompuDuds` class is a rather strange class in that there will never be any objects that are instances of it—not even by way of subclasses. So, there will be no instance variables, no constructor called `CompuDuds`, and no methods in the normal sense. Instead the class will contain nothing but class methods, which are really just normal procedures. Each class method is flagged as such with the keyword `static`. Here's how the class starts out:

```
public class CompuDuds {

  public static void displayPrice(int totalCents){
    int dollars = totalCents / 100;
    int remainingCents = totalCents % 100;
    System.out.print("$");
    System.out.print(dollars);
    if(remainingCents < 10){
      System.out.print(".0");
    } else {
      System.out.print(".");
    }
    System.out.print(remainingCents);
  }

  // more class methods go here
}
```

As you can see, the `static` keyword is used to signal that `displayPrice` is a class method; when it is invoked, it isn't going to be associated with any object—there

will be no implicit `this` argument. The other novelties occur in the first two lines of the body: the use of the slash and percent-sign characters to find the quotient and remainder when `totalCents` is divided by 100. In Java, the rule is that if a slash has an integer on both sides, it represents the quotient operation. (By contrast, in Scheme, a slash would produce a fractional answer, and so we use an explicit `quotient` procedure.) The percent sign indicates remainder; the rationale is that it visually resembles the slash, to remind you that they go together.

The `CompuDuds` class is the main class for the program—the one you tell the Java system you want to run. (How you tell the Java system that depends what sort of system you have.) When you say that you want to start up the program associated with the `CompuDuds` class, what the Java system does is run the procedure named `main` in that class; here it is:

```
public static void main(String[] commandLineArgs){
  ItemList itemList = new ItemList();
  while(true){ // infinite loop terminated by program exiting
    System.out.println();
    System.out.println("What would you like to do?");
    System.out.println(" 1) Exit this program.");
    System.out.println(" 2) Add an item to your selections.");
    System.out.println(" 3) List the items you have selected.");
    System.out.println
      (" 4) See the total price of the items you selected.");
    int option;
    if(itemList.empty()){
      option = inputIntegerInRange(1, 4);
    } else {
      System.out.println(" 5) Delete one of your selections.");
      System.out.println
        (" 6) Revise specifics of a selected item.");
      option = inputIntegerInRange(1, 6);
    }
    if(option == 1){
      System.exit(0);
    } else if(option == 2){
      Item item = inputItem();
      itemList.add(item);
      item.inputSpecifics();
    } else if(option == 3){
      itemList.display();
    } else if(option == 4){
      displayPrice(itemList.totalPrice());
```

```
        System.out.println();
      } else if(option == 5){
        itemList.delete(itemList.choose());
      } else if(option == 6){
        itemList.choose().reviseSpecifics();
      }
    }
  }
```

The most interesting parts of this procedure are actually in the first three lines:

```
  public static void main(String[] commandLineArgs){
    ItemList itemList = new ItemList();
    while(true){ // infinite loop terminated by program exiting
```

As you can see, the `main` procedure (which constitutes the actual program itself) needs to accept an argument that is an array of `String`s. This is used because in some systems you can start a program up with arguments provided to the program as a whole—with *command-line arguments*. We'll ignore these entirely but need to have the declaration there to keep the Java system happy. The second line shows how a new object is created as an instance of a class; the `new ItemList()` is the analog of Scheme's `(make-item-list)`. Finally, we have a new kind of loop, the `while` loop. This loop is just a stripped down version of the `for` loop that doesn't have a variable to initialize before the loop or to step from one iteration of the loop to the next. Instead, all that remains is the test condition that determines whether the loop should continue. (Of course, the loop body remains as well.) In this case, our test expression is the boolean constant `true`, the Java analog of Scheme's `#t`. That means the loop will always keep looping—at least until `System.exit` causes the whole program to stop.

By the way, notice that where the `main` procedure invokes other procedures contained within the `CompuDuds` class, it doesn't need to explicitly put the `CompuDuds` class name and a dot before the procedure name; for example, it can say `inputIntegerInRange(1, 4)` rather than `CompuDuds.inputIntegerInRange(1, 4)`. This is a shorthand, just like omitting the prefix `this` when one method invokes another on the same object.

To reinforce the notion of how objects are created, here is the `inputItem` procedure, which is used by `main`:

```
  private static Item inputItem(){
    System.out.println("What would you like?");
    System.out.println(" 1) Chinos");
    System.out.println(" 2) Oxford shirt");
```

```
    if(inputIntegerInRange(1, 2) == 1){
      return new Chinos();
    } else{
      return new OxfordShirt();
    }
  }
```

As you can see, this code creates either a new instance of the `Chinos` class or a new instance of the `OxfordShirt` class. Either one gets returned; notice that the procedure is declared as returning an `Item`, and sure enough, it does return an `Item`—of some kind. The empty parentheses after the class name occur because no arguments are being passed to the class's constructor. For an example where this case wouldn't be true, consider creating a plain `Item` with a price of $19.50, such as our earlier `exampleItem`; you would write `new Item(1950)`.

There are only two procedures left in the `CompuDuds` class: `inputIntegerInRange` and `inputSelection`. The latter is rather straightforwardly written in terms of the former, so we'll skip it here; you can get it from the web site for this book. The `inputIntegerInRange` procedure, on the other hand, is quite interesting. Not only does it show how input is done, it also shows the handling of *exceptions* (i.e., unusual circumstances). In this case, two kinds of exceptions could occur: Something could go wrong while reading the input (such as the computer noticing that the keyboard is unplugged) and something could go wrong while trying to convert the input into a number (such as the input not being composed of digits). The following procedure has handlers for both of these exceptions:

```
  public static int inputIntegerInRange(int min, int max){
    System.out.print("(enter ");
    System.out.print(min);
    System.out.print("-");
    System.out.print(max);
    System.out.println(")");
    String inputAsString = null;
    try{
      inputAsString = reader.readLine();
    } catch(java.io.IOException e){
      System.err.print("Problem reading input: ");
      System.err.println(e);
      System.exit(1);
    }
    if(inputAsString == null){ // this means end of file on input
      System.exit(0);  // handle as a normal program termination
    }
```

```
    int inputAsInt;
    try{
      inputAsInt = Integer.parseInt(inputAsString);
    } catch(NumberFormatException e){
      System.err.println("input must be an integer and wasn't");
      return inputIntegerInRange(min, max);
    }
    if(inputAsInt < min || inputAsInt > max){
      System.err.println("input out of range");
      return inputIntegerInRange(min, max);
    } else{
      return inputAsInt;
    }
  }
```

As we said, there are two places where we need to be prepared for exceptional things to go wrong (the input itself and the conversion of the input to an integer). Each of these is handled with a `try` and `catch` construct. The code between the `try` and the `catch` is where the exception might originate, and the code after the `catch` is what deals with it if it does occur. Each `catch` specifies what kind of exception it is prepared to catch; our first one catches `java.io.IOException`s, and the second one catches `NumberFormatException`s. Either way, we provide a name for the exception that is caught; that is what the `e` is. This `e` is like a parameter to a procedure; a particular exception object has been caught, and we give it a name so that we can refer to it in the code that handles it. In fact, it is literally an object; in our second example, it is an instance of the class `NumberFormatException`. We don't do anything with that exception object, but in the first exception handler, we print the `java.io.IOException` object itself out as part of our error message.

Another generally useful feature is the double vertical bar to indicate "or" in the following line:

```
    if(inputAsInt < min || inputAsInt > max){
```

If the input is either too small or too big, we want to deal with it as out of range. One fine point is that the second condition is only tested if the first one isn't true, which means you can safely write code like

```
    if(diplomacySucceeds() || goingToWarSucceeds()){
```

and know that you won't even try going to war if diplomacy succeeds. An analogous situation occurs with `&&`, which is used for "and." For example, you can write

```
    if(diplomacyFails() && goingToWarFails()){
```

The second condition is only tested if the first one is true—if the first is false (diplomacy didn't fail), we already know that they aren't both true (diplomacy and going to war didn't both fail).

One detail we glossed over in the `inputIntegerInRange` method was the `reader` that the `readLine` method is invoked on, to get a line of input as a `String`. What is this `reader`? It is defined by the following component of the `CompuDuds` class:

```
  private static java.io.BufferedReader reader =
   new java.io.BufferedReader
    (new java.io.InputStreamReader(System.in));
```

This example looks like a `private` instance variable, except that it has a `static` keyword, which means there is a single variable shared by the whole class rather than one per instance. (Remember, the `CompuDuds` class has no instances, so that wouldn't make much sense.) This variable called `reader` is declared to be of the class `java.io.BufferedReader` and is initialized by a big long mess that constructs a `java.io.BufferedReader`. To tell you the full story would get into grungy details of input that you can probably afford to ignore, because most Java programs don't interact this way; instead they use graphical user interfaces, such as we'll consider in the next section. If you ever need to know more about what a `java.io.BufferedReader` really is, plenty of good documentation is available—see the end-of-chapter notes for references.

Now that we've translated CompuDuds into Java, you can try it out and can try your hand at adding various enhancements (just like you did in Scheme in the prior chapter).

Exercise 15.1

Find out how to run Java programs on your computer system, and try out the CompuDuds program as it comes from the web site for this book. It is important to try out the program unmodified first so that you can work out any kinks in the mechanics of running a Java program.

Exercise 15.2

Change the `ItemList` class so that it keeps a running total price as `Item`s are `add`ed and `delete`d and then just returns that from the `totalPrice` method, rather than looping through adding up all the prices. Check that the program still works.

Exercise 15.3

Add some more subclasses of `Item`, reflecting your own taste in clothing. Change `inputItem` so that these additional choices are available to the user, and check that your changes work correctly.

Exercise 15.4

Assuming the CompuDuds program is being operated by sales personnel, rather than directly by the customer, it might be desirable to provide a means of applying a discount.

a. Add a method to the `Item` class that marks down the price by 10 percent. Be forewarned that you can't legally say

```
price = .9 * price;
```

because multiplying by .9 results in something that isn't an `int` and hence can't be stored into `price`. Instead you can multiply by 9 and then take the quotient upon division by 10.

b. Use your new `Item` method to add a method to the `ItemList` class that marks down the price of all the `Item`s by 10 percent.

If you've done Exercise 15.2, in which you changed the `ItemList` class to maintain a running total price, you'll have a problem with simply discounting each `Item` by 10 percent and discounting the accumulated total price by 10 percent. If you do so, under some circumstances you'll wind up with a total price that doesn't match the sum of the individual prices, due to roundoff. The solution is to recompute the total price whenever a discount is applied.

c. Finally, add a new option to the main CompuDuds user interface that discounts everything that has been selected by 10 percent.

15.3 Event-Driven Graphical User Interfaces in Applets

The CompuDuds program interacts with its user 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 applets, which are event-driven GUI programs that are designed to be components of documents on the World Wide Web rather than standing alone. Instead of running lots of separate programs, users run 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, and also interactive content in the form of applets.

Object-oriented programming turns out to play a critical role in event-driven GUI programs, both from our perspective as programmers of individual applets and also from the perspective of the programmers of the overall web browsers that our applets run inside of.

From our own perspective, we will be able to write our programs reasonably simply and easily because there is a large "library" of prewritten 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 the programmers of the main browser program, the key fact is that all these individual component classes, like `Button` and `Checkbox`, are actually subclasses of a general `Component` class. Any `Component` knows how to draw itself. Any `Component` knows how to respond to a mouse button being pressed

while the mouse is pointing into that `Component`'s area. Thus the browser doesn't have to concern itself with the many different kinds of interaction mechanisms. It can just uniformly treat the whole applet as a bunch of `Component`s. It asks 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 `TextField` might treat this action entirely differently from a `Button`—they are both still `Component`s.

Our first example applet is shown in Figure 15.1. It 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 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 applet simulates the puzzle with a grid of 16 buttons, of which 15 are labeled with the numbers 1 through 15, and the remaining one has a blank 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 he or she wants to slide the tile clicked on, pushing along with it any others that intervene between it and the empty position. We simulate this action 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.

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 applet is running. These are the objects that are instances of the subclasses of `Component`. The `Component` class, like many others that we'll use, is found in the *Abstract Window Toolkit* (AWT), a GUI-specific portion of Java's rich hierarchy of library classes. The `Component`s in our applet are as follows:

Figure 15.1 The sliding 15-tile puzzle applet

- There are 16 instances of the class `Button`, which is an AWT class derived from `Component`. Each `Button` has a label (empty in one case) and can respond to being pushed.
- One object represents the applet 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 AWT class called `Applet`, used for all applets. As the class hierarchy in Figure 15.2 shows, the `Applet` class is indirectly descended from `Component` because an `Applet` is visibly present on the screen. More specifically, because an `Applet` can contain other `Component`s (like our `Button`s), the `Applet` 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, our applet has 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. Because our subcomponents are the `Button`s, which we want to form a 4×4 grid, we'll use a `GridLayout` as

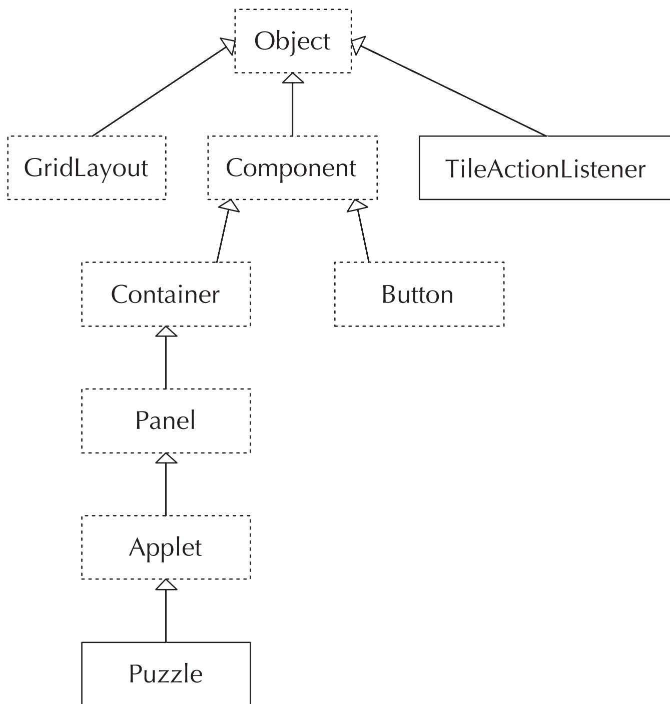


Figure 15.2 Class hierarchy for 15-tile puzzle applet (dashed boxes indicate library classes)

the layout manager. The `GridLayout` class is also provided for us as part of the AWT because this grid is a commonly desired layout.

- Each of our 16 `Button`s needs to cause some response when it is pushed, rather than just being displayed on the screen. The `Button` class lets us tailor the response by giving each `Button` a companion object to notify when the `Button` is pushed. That companion object is in charge of providing the appropriate response. We'll define a class, `TileActionListener`, for these companion objects and make 16 `TileActionListener`s, one per `Button`.

The portion of the class hierarchy used in this applet is shown in Figure 15.2. Note that we mark library classes with dashed boxes, and those we will write ourselves with solid boxes. Of the library classes, all except the fundamental `Object` class are from the AWT. (Actually, the `Applet` class is technically not in the AWT but rather is in a package of its own that builds on top of the AWT. For many purposes it is convenient to treat it as part of the AWT, as we do.)

Now, let's go directly to the portion of the `Puzzle.java` file that deals with the initial construction and layout of the `Puzzle` object:

```
import java.awt.*;

public class Puzzle extends java.applet.Applet {

  private int size = 4;          // how many buttons wide and high
  private Button[][] buttons;
  private int blankRow, blankCol;  // position of the blank space

  public void init(){
    setLayout(new GridLayout(size, size));
    buttons = new Button[size][size];
    int buttonCount = 0;
    for(int row = 0; row < size; row++){
      for(int col = 0; col < size; col++){
        Button b = new Button();
        buttons[row][col] = b;
        b.addActionListener(new TileActionListener
                              (this, row, col));
        buttonCount++;
        b.setLabel("" + buttonCount);
        add(b);
      }
    }
    blankRow = size - 1;
    blankCol = size - 1;
```

```
    buttons[blankRow][blankCol].setLabel("");
  }

  /* Code will be given later for the following method:
   *
   *    public void pushTile(int row, int col)
   *
   */
}
```

This code includes a number of new Java features that bear explaining, one of which is the `import` line at the top of the file. This line allows us to use shortened names for the library classes we'll be using. For example, we'll be able to use the short name `Button` rather than the full name, which is `java.awt.Button`.

A more significant difference between the Puzzle class and our earlier Java classes is the apparent nonexistence of a constructor for the `Puzzle` class and the use of the `init` method, which seems to hearken back to our Scheme version of object-oriented programming. Regarding the apparent nonexistence of a constructor, `Puzzle` is in fact using a *default constructor*, which does nothing except invoke the `Applet` constructor. Java writes this do-nothing-extra constructor for us as a shorthand if we don't provide any constructor explicitly. Although the `Puzzle` constructor might seem the natural place to initialize the instance variables for the `Puzzle` class, standard practice dictates that applet initialization be accomplished by overriding the `Applet` class's `init` method. This standard practice derives from the special nature of applets, namely, that they are intended to run inside other programs, most notably in web browsers. In particular, because the browser is in charge of the applet's run-time environment, it needs to set up two-way communications between itself and the applet when it creates the applet. The two-way communication is necessary so that the browser and applet can react in tandem to various events that might occur (such as resizing), as well as to allow the applet access to information it might need (such as the location of image or audio files).

To deal with this two-way communication, the designers of Java have set up a protocol for programmers to follow when writing applets. The only aspect of this protocol that bears on us now concerns the roles of the applet constructor and the `init` method. Specifically, when the browser creates an applet, it first invokes applet's constructor, next sets up the two-way communication between itself and the applet, and then invokes the applet's `init` method. Because the applet's initialization often involves certain aspects of its run-time environment, which requires the two-way communications to have already been established, the standard practice is to wait until the `init` method to do the applet-specific initialization. This is the model we have followed.

A third new feature in the code is the double set of brackets we use for the two-dimensional array of buttons. Actually `buttons` is strictly speaking just a normal one-dimensional array, each element of which is itself another one-dimensional array, which in turn holds `Button`s. This is the same as what we've done in Scheme, where we represented two-dimensional tables as vectors of vectors. However, Java has a handy shorthand; we can say

```
buttons = new Button[size][size];
```

to create the entire structure, consisting in this case of five arrays: four arrays to hold `Button`s and then one "outer" array that contains the four others.

As you can see, we use nested `for` loops to loop over all the columns of all the rows. For each position, we create a new `Button` (temporarily called `b`), store `b` at the appropriate position in the `buttons` array, invoke two methods on `b` to configure it appropriately (`addActionListener` and `setLabel`), and finally `add b` itself to the applet. Let's look more closely at the actual statements in this code.

First, recall that applets are types of `Container`s, so named because they can contain other components such as buttons. Components are added to a container by `Container`'s `add` method. Precisely how these components are arranged in the container is controlled by the container's layout manager, which is specified by `Container`'s `setLayout` method. In our case, we have specified the layout as a 4×4 grid. The `GridLayout` object places the components in the grid from left to right, top to bottom, in the order they are added.

Adding the action listener is how we specify what response should occur when the user clicks on the `Button`. If we omitted that line, the applet would still display the same 4×4 grid of buttons, but we couldn't scramble their labels and then get them back into order; the puzzle would stay permanently in its solved state. We'll say more about the action listener later.

The button's label is set by the `setLabel` method from the `Button` class. The argument to this method is a `String` specifying the new label. For example, at the end we label the bottom right `Button` with an empty string so that it will be blank. How about the rest of the labeling, done inside the nested `for` loops? This takes a bit more explaining. We have a numerical variable, `buttonCount`, which records how many buttons have been created so far. We need to convert that numerical value into the corresponding string of digits, which in the first case is the string `"1"`. This conversion is done by a very strange Java idiom, namely, adding the number onto the empty string: `"" + buttonCount`. This idiom works because of the combination of two facts:

- For `String`s, the plus sign indicates sticking the two `String`s together (like Scheme's `string-append`) rather than numerical addition.

- If the plus sign only has a `String` on one side, whatever is on the other side gets converted to a `String` first, so that it can be appended.

Thus we are asking for the numerical `buttonCount` to be implicitly converted into a `String` and then tacked onto the end of an empty `String`. As roundabout as this may seem, it is in practice a quick and easy idiom to type.

At this point we have finished with the initial presentation of the applet and merely have to make it responsive to the user's pushing on its buttons. Recall that this responsiveness is provided by using the `addActionListener` method on each `Button` to give it its own `TileActionListener`. The idea is that the `Button` notifies the action listener whenever it is pushed. Notice that when we construct the `TileActionListener`, we specify what `Puzzle` it is a part of (by passing in `this`) as well as what row and column its `Button` is in. Here is the code for the `TileActionListener` class:

```
import java.awt.event.*;

public class TileActionListener implements ActionListener {

  private Puzzle puz;
  private int row, col;

  public TileActionListener(Puzzle p, int r, int c){
    puz = p;
    row = r;
    col = c;
  }

  public void actionPerformed(ActionEvent evt){
    puz.pushTile(row, col);
  }
}
```

As you can see, `TileActionListener` isn't a very big class. It stores the `Puzzle` and row and column position in instance variables, and when it is told by the `Button` that an action has been performed (namely, the user pushed the `Button`), the `actionPerformed` method simply invokes the `Puzzle`'s `pushTile` method, passing in the row and column numbers. Thus the real work of shifting all the labels over by one is done in the `Puzzle` class's `pushTile` method.

Although the core of what this `TileActionListener` class does is quite straightforward, it has a couple oddities in the first few lines. A minor one is that this class has a different `import`, to get short names for the `ActionListener` and `ActionEvent` classes. A more interesting one is that `TileActionListener`

`implements ActionListener`—notice the `implements` keyword where we previously have seen `extends`. The reason for this difference is that `ActionListener` isn't strictly speaking a class, it is an *interface*. Interfaces are extremely close relatives of classes, with two differences:

- An interface must be purely abstract. That is, it can only provide a declaration of the method names and types. It can't provide any method implementations or any instance variables to provide a representation. All implementation details are confined to the classes that implement the interface.
- A class can implement as many interfaces as you like, even though it can only extend one superclass.

We won't define any interfaces, so the only place you'll see them is when we `implement` an interface provided by the standard library, like `ActionListener`. We should point out, however, that `LayoutManager` is also an interface; the AWT provides several implementations of this interface, including `GridLayout`. We can add the two interfaces, `ActionListener` and `LayoutManager`, to our UML class diagram as shown in Figure 15.3. The interfaces are labeled as such, and

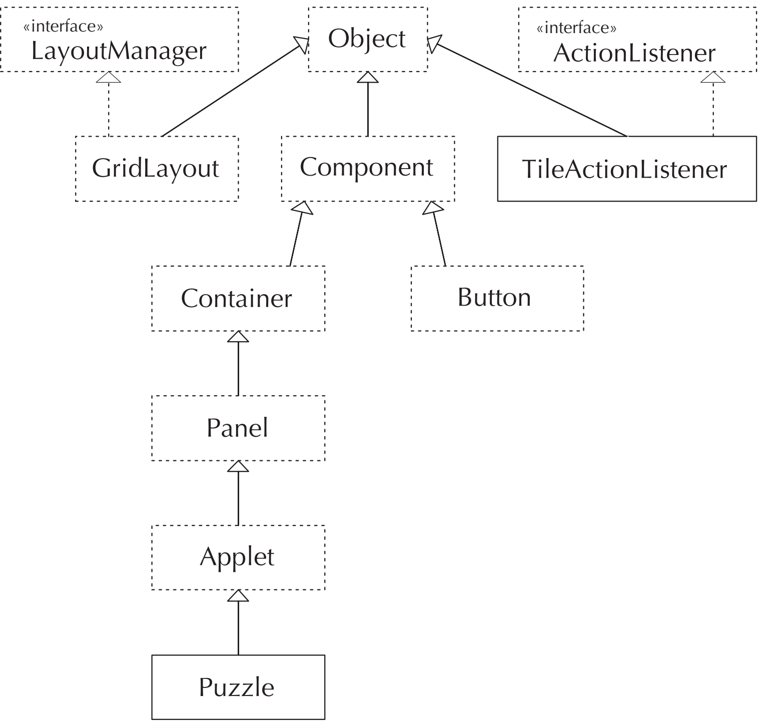


Figure 15.3 The puzzle class hierarchy with interfaces added; dashed boxes still indicate library classes or interfaces, but the dashed arrows are part of the UML notation.

a dashed arrow with a triangular arrowhead is used to connect the specific class (`TileActionListener` or `GridLayout`) to the general interface that it implements.

Now that we have the interfaces in our diagram, we can also add some associations, as shown in Figure 15.4, that will help clarify the relationships between the `Puzzle`, the `Button`s, and the `TileActionListener`s. The `Puzzle` keeps track of the 16 `Button`s so that it can manipulate their labels. Each `Button` is associated with one `TileActionListener`, which is specific to that `Button`. Each of the 16 `TileActionListener`s knows about the `Puzzle`, so that the `TileActionListener`'s `actionPerformed` method can invoke the `Puzzle`'s `pushTile`, as we have seen. So far, the associations are much as we've seen in other programs. But, we have one key new annotation, namely the `:ActionListener` near the arrow from the `Button` class to the `TileActionListener` class. This annotation is our way of showing that the `Button` only makes use of the `ActionListener` interface, not any of the specifics of the `TileActionListener` class. This point is essential because it allows the `Button` class to be general enough to work with other kinds of `ActionListener`s as well.

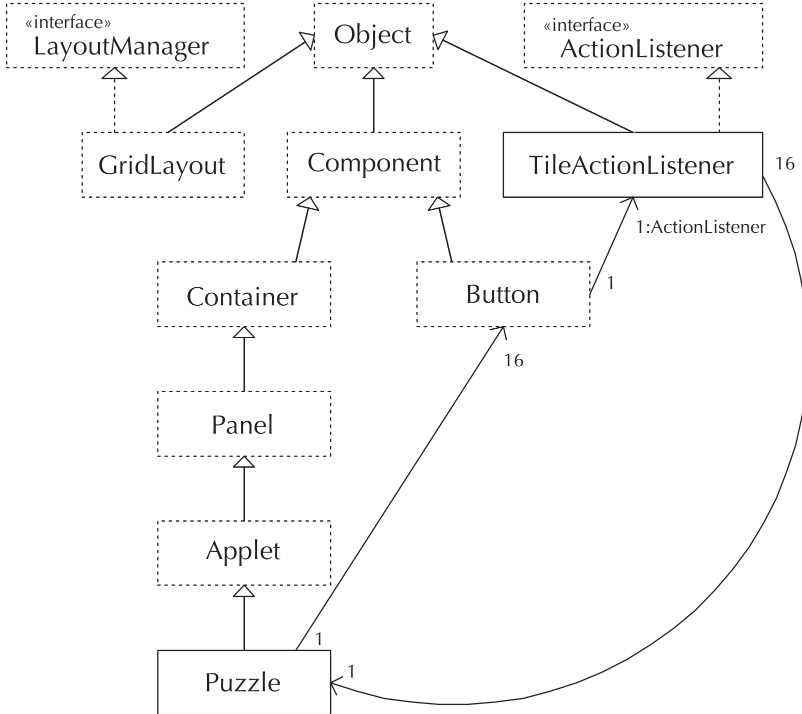


Figure 15.4 The relationships between `Puzzle`, `Button`, and the `TileActionListener` implementation of the `ActionListener` interface

All that is left now is the `pushTile` method in the `Puzzle` class, which takes care of the mechanics of shifting the `Buttons`' labels. To do so, it will use not only the `setLabel` method of the `Button` class, which we saw previously, but also the converse `getLabel` method. The code loops through the affected positions, getting the label from one `Button` and setting it into the neighboring `Button`. If the tile being pushed is in the same row as the blank space, the tiles need shifting to the right or left, whereas if the tile being pushed is in the same column as the blank space, the tiles need shifting down or up. If neither row nor column matches, no shifting happens:

```
public void pushTile(int row, int col){
  if(row == blankRow){
    for( ; blankCol < col; blankCol++){
      buttons[blankRow][blankCol].setLabel
        (buttons[blankRow][blankCol+1].getLabel());
    }

    for( ; blankCol > col; blankCol--){
      buttons[blankRow][blankCol].setLabel
        (buttons[blankRow][blankCol-1].getLabel());
    }
  } else if(col == blankCol){
    for( ; blankRow < row; blankRow++){
      buttons[blankRow][blankCol].setLabel
        (buttons[blankRow+1][blankCol].getLabel());
    }
    for( ; blankRow > row; blankRow--){
      buttons[blankRow][blankCol].setLabel
        (buttons[blankRow-1][blankCol].getLabel());
    }
  }
  buttons[blankRow][blankCol].setLabel("");
}
```

The correctness of the above code critically depends on a feature of `for` loops that we haven't stressed previously. 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 15.5

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, which 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?

The applet itself is now complete; however, in order to view it, we need a *World Wide Web* document that contains it. This document gets written in a special *HyperText Markup Language*, known as HTML. It can contain headings, normal text, etc., as well as a special indication of where the applet should go. We won't delve into HTML here, because this information is widely available and peripheral to our topic. However, the following HTML file would be adequate to show our puzzle applet:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<html>
<head>
<title>Puzzle (original)</title>
</head>
<body>
<h1>Puzzle (original version)</h1>
<applet code="Puzzle" width=150 height=150>
</applet>
</body>
</html>
```

The specifics of how you can get your web browser to access this file and the applet that it contains depend on what sort of computer system you are using. You should be able to get the necessary information from the documentation for your web browser and Java system or from a local expert.

We'll now consider some variations on this applet. The first two just add new features to the puzzle, but one actually changes it to be an entirely different puzzle. We'll also look at some additional variants in the next section, using concurrency.

For our first variant, let's consider adding an "Initialize" button above the grid of tiles, which restores the labels to their starting configuration. That way frustrated users can get a fresh start. To reprogram `Puzzle` to accomplish this task, we will spin off the task of labeling the tiles in their initial state to a method called `initializeTiles`.

More significantly, to make the applet look as shown in Figure 15.5, we need to change our applet's configuration a bit. Rather than consisting of the 16 tile buttons, we'll view the applet as containing a thin control panel along the top and then a big

Figure 15.5 The puzzle applet with the Initialize button added

main panel on the bottom. (As noted in Figure 15.2, `Panel` is the intermediary class between `Container` and `Applet`; it is the most basic `Container` subclass.) The main panel, in turn, will contain the 16 tiles. The control panel, for now, will have only the "Initialize" button. This change is confined to the `init` method, because that is where the various pieces are put together. Here is the new version:

```
import java.awt.*;

public class Puzzle extends java.applet.Applet {

  private int size = 4;         // how many buttons wide and high
  private Button[][] buttons;
  private int blankRow, blankCol;  // position of the blank space

  public void init(){
    setLayout(new BorderLayout());
    Panel controlPanel = new Panel();
    add(controlPanel, "North");
    Panel mainPanel = new Panel();
    add(mainPanel, "Center");

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

```
    mainPanel.setLayout(new GridLayout(size, size));
    buttons = new Button[size][size];
    for(int row = 0; row < size; row++){
      for(int col = 0; col < size; col++){
        Button b = new Button();
        buttons[row][col] = b;
        b.addActionListener(new TileActionListener
                              (this, row, col));
        mainPanel.add(b);
      }
    }
    initializeTiles();
  }

  public void initializeTiles(){
    int buttonCount = 0;
    for(int row = 0; row < size; row++){
      for(int col = 0; col < size; col++){
        buttonCount++;
        buttons[row][col].setLabel("" + buttonCount);
      }
    }
    blankRow = size - 1;
    blankCol = size - 1;
    buttons[blankRow][blankCol].setLabel("");
  }

  /*
   *  Code for pushTile(int row, int col) is unchanged
   */
}
```

As you can see, the applet itself no longer has a `GridLayout`—that has been moved to the `mainPanel`. Instead, the overall layout is now what is called a `BorderLayout`, which is what lets us add the `controlPanel` with a specification that it should form the `"North"` border. We could also, if we wanted, provide borders on the other three sides; we don't, though. The `mainPanel` gets added as the `"Center"` component, which simply means it takes up the remainder of the applet's space.

We still need to write the `InitializeActionListener` class, but it is no big deal at all:

```
import java.awt.event.*;

public class InitializeActionListener implements ActionListener {

  private Puzzle puz;

  public InitializeActionListener(Puzzle p){
    puz = p;
  }

  public void actionPerformed(ActionEvent evt){
    puz.initializeTiles();
  }
}
```

We made a design decision to call `initializeTiles` from `init`, rather than leaving `init` as it was, performing essentially identical actions to what is in the new `initializeTiles` method. We can describe this decision as having chosen *separating out* over *copying out*. This decision involved a trade-off of short-term versus long-term quality considerations:

- Copying the code out while leaving the existing `init` code unchanged would mean not changing that which was already working—and hence (in the short term) would run less risk of breaking it.
- Separating the code out (as we did) makes it more obvious to a reader that `initializeTiles` necessarily is putting the tiles back into the same configuration as `init` puts them in. There is less risk that this equality would be broken if the code is later changed.

Said another way, short-term quality considerations argue for minimizing changes to working code, whereas long-term quality is improved by making those changes necessary to enforce the following dictum:

> **The sameness principle:** Coincidental sameness should be indicated by duplication; necessary sameness should be indicated by sharing.

Exercise 15.6

We should have followed this principle elsewhere in the book but didn't always.

a. Find other examples of where we have adhered to the sameness principle.

b. Find an example of where we have strayed from the principle, and show how to modify our code so as to bring it into compliance.

Exercise 15.7

The sameness principle can be violated in two ways: Code that is coincidentally the same can be shared, and code that is necessarily the same can be duplicated. Compare the dangers inherent in these two forms of violation.

If we want to add additional features, we can just `add` more items to the `controlPanel`. For example, it would be nice if we had 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 `Button`. One aside: When we add a new `Button` 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 preceding `Puzzle` constructor, you'll see we didn't set a layout for the `controlPanel`, just for the applet itself and the `mainPanel`. The solution to this mystery is that each `Panel` when constructed starts out with a default layout, 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.

We have two basic approaches to how the `randomizeTiles` method could work. One would be to literally randomize the 16 labels, by selecting any of the 16 to be on the first tile, then any of the remaining 15 to go on the next tile, etc. A different approach would be to simply 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 15.6 can't be solved because 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

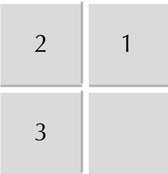


Figure 15.6 From this configuration, no amount of sliding the tiles will put the numbers in order with the blank in the lower right.

in order. A similar but more complicated argument can be made to show that half the configurations in the 15-tile puzzle are also unsolvable.

We therefore make the design decision to write `randomizeTiles` so that it randomly slides tiles around, which 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 lines that pick the random row and random column need some explanation. The procedure `Math.random` generates a random fraction between 0 and 1; multiplying that by `size` (i.e., 4) and then truncating the result to an `int` gives us a random integer from 0 through 3. The notation `(int)` is called a *cast* and converts the number to an `int` by throwing away its fractional part. Other than that magic, the only question you are likely to have is Why 100? The answer is that it seemed like a reasonable number: big enough to do a pretty good job of scrambling but not so large as to take a long time. You can 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 15.8

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" `Button` to the `controlPanel` with a `RandomizeActionListener`.

We'll see a rather different applet 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 `Buttons`. 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 you also have to flip over the four neighboring ones (not counting the diagonal neighbors). If the penny you flip is 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 15.9

Change the puzzle applet to this new puzzle, by doing the following:

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 `Button` 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 `Button` in the pushed position and also `flip` each of its four neighbors, provided that they exist.

15.4 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 goal 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. 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, because the world in which the computer is embedded is concurrent:

- The user is sitting in front of the computer thinking all the time 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—that are 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 applet 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 action listener that when the button is pushed goes into an infinite loop, and 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 action listener is notified, the user interface goes dead until the action listener's `actionPerformed` method has finished responding. Depending on how fast your computer is, you may have noticed this phenomenon 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. 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 concurrent: One part of the program should loop forever, sliding tiles randomly, and 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 applet will now be *multithreaded*.

Rather than attempting a precise definition of threads, let us instead suggest that you think of them as independently executing strands of computations that 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 multithreaded 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 flexibility 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.

We create our poltergeist by defining a subclass of `Thread`. This subclass is called `Poltergeist Thread`. 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`:

```
public class PoltergeistThread extends Thread {

  private Puzzle puz;

  public PoltergeistThread(Puzzle p){
    puz = p;
  }

  public void run(){
    try{
      while(true){
        Thread.sleep(1000); // 1000 milliseconds = 1 second
```

```
        puz.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 1 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 1-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, which the Java system interprets 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:

```
import java.awt.event.*;

public class PoltergeistActionListener implements ActionListener {

  private Puzzle puz;

  public PoltergeistActionListener(Puzzle p){
    puz = p;
  }

  public void actionPerformed(ActionEvent evt){
    new PoltergeistThread(puz).start();
  }
}
```

Note that the `actionPerformed` method creates a new `PoltergeistThread` object and then calls the `start` method on the newly created object. This point is where the concurrency happens: The `start` method immediately returns, so 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 `Button` to the `controlPanel`, you now have an applet 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 15.10

Even a buggy program is worth trying out. Add a `Button` to the `ControlPanel` for firing up a poltergeist, and try it out.

Recall that different Java implementations can braid the same strands of a multithreaded 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 properly, which may leave you wondering why we called the program buggy. The problem is: 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, interesting things happen. Our main focus in this section will be on how you can design a program so 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.

We have 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 two of the poltergeists happens to slide a tile at the same time.

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, because they are initially, and the blank spot only moves from where it is one position at a time toward the tile being pushed, until it reaches that position. Therefore, because 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 2—shooting right past 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 15.11

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. Because a race by definition depends on the timing being just wrong, you could test your program any number of times, never observe any misbehavior, and still have a user run into the problem.

This occurence 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 it 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 that is established by the class's constructor and preserved by all of the class's mutators, so 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 `Button` stored in `buttons[blankRow][blankCol]` has the empty string as its label.
- The remaining `size`$^2 - 1$ `Buttons` 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 to deal 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, whereas the second thread does A and then B. Then even this very simple concurrent system has 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` applet, 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 afterward. 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`. Because `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
}
```

```
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. It is a rule that `synchronized` methods may only be performed in that room, with the door locked. This rule 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. Some other programs, however, have 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, 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 15.12

Consider the `itemVector` in the `ItemList` class, in the Java version of CompuDuds. Although each position in that array can hold an `Item`, it is not necessarily true that they all do. For example, when the array is initially created by the line

```
itemVector = new Item[10];
```

10 different positions can each hold an `Item`, but none of them yet does. (Instead, each holds the special `null` value.) Similarly, when we do a `delete` operation, the vacated position has the special `null` value stored into it, which isn't an `Item`. Thus, when we retrieve an element from the array, as in the line

```
itemVector[index].display();
```

we have in principle the possibility that the retrieved value might not be an `Item` but instead might be the `null` value from when the array was created or from when a `delete` was done. If so, we would get an error when we tried to invoke the `display` method on that non-`Item`.

a. Write a paragraph or two explaining how the design of the `ItemList` class ensures that this will never happen, given that CompuDuds is a single-threaded program.

b. Write a paragraph explaining why the reasoning would break down if an `ItemList` were used from more than one thread and briefly stating what should be done about it.

Having seen how to keep one thread from stepping on another thread's toes, we'll now turn to another important concurrency topic: how one thread can wait for an action in another thread.

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, we have 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*, and the second thread tries to invoke a `synchronized` method on *A*. Each thread waits for the other room to become available. But because 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*.

Suppose we want to change our applet so that rather than having a button that causes a poltergeist to come into existence, it has a checkbox that we can use to turn the poltergeist on or off. A checkbox is a GUI element that switches between an on state and an off state each time you click on it. On some systems, it looks like a box that is empty for off and has a checkmark or X in it for on. On other systems it is a square that has shadows that make it look like it is sticking out for off and recessed in for on.

One way to implement this would be to handle turning the poltergeist on just the same way as we previously handled the button presses (create a thread and start it running) and handle turning the poltergeist off by somehow killing off the thread. Then the next time the checkbox was clicked to turn the poltergeist back on, a new thread would be created, etc.

However, there is another, more interesting, alternative. We can have a single `PoltergeistThread` that starts running at the beginning and is never killed off (at least, not until the whole applet is terminated). However, the poltergeist can be in a dormant state, where it just waits for the checkbox to turn it on. Then it starts doing its usual random pushing of tiles, until such time as the checkbox is clicked again. Then it goes back into the dormant state, once more waiting to be turned on.

To implement this design, we'll give the `PoltergeistThread` three more methods. Two will be `public` methods used by the user interface to exert control: `enable` and `disable`. The `disable` method puts the thread into its dormant state, where it doesn't do any random pushes, and the `enable` state wakes it back up. Finally, we'll add a `private` method called `waitUntilEnabled` that can be used by the thread's main loop to do the actual waiting; here is that main loop:

```
public void run(){
  try{
    while(true){
      Thread.sleep(1000); // 1000 milliseconds = 1 second
      waitUntilEnabled();
      puz.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.
  }
}
```

As you can see, we made just one change from the previous version. Each time around the loop it performs `waitUntilEnabled` before pushing a random tile. If the thread is currently enabled, this call will immediately return and the tile will

be pushed. If the thread is disabled, on the other hand, the `waitUntilEnabled` method won't return until after the `enable` method has been called by the user interface.

Now we have to see how the three new methods are written. We'll add a new instance variable to the `PoltergeistThread` class:

```
private boolean enabled;
```

The two controlling methods can just set this variable appropriately:

```
public void enable(){
  enabled = true;
}

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

Now comes the tricky part: waiting for the `enabled` variable to be set to true. One simple approach (but not a terribly good one) is to simply go into a loop, checking each time around the loop to see if the variable is true yet. To avoid hogging the computer's time too much, we can sleep for a fraction of a second between each check:

```
private void waitUntilEnabled() throws InterruptedException {
  while(!enabled){
    Thread.sleep(100);
  }
}
```

We have two new Java language features in this method. One is the use of the exclamation point to indicate "not." That is, the loop continues so long as the poltergeist is "not enabled," in other words, so long as the `enabled` variable has the value `false`. The other new feature is the phrase "`throws InterruptedException`." You'll recall that the `Thread.sleep` procedure can throw this particular exception. In the `run` method, we handled that by use of a `try` and `catch` construct. Here, however, we have made a different choice. Rather than catching the exception, we've allowed it to continue on out of the `waitUntilEnabled` method so that it can be caught in the caller. Here that makes sense because the caller is `run`, and `run` is in a much better position to do something sensible about being interrupted while waiting. To indicate that an exception of this kind may now emerge from the `waitUntilEnabled` procedure, we need to include the `throws` declaration.

The preceding approach, in which waiting is accomplished by nervously glancing at the controlling variable every fraction of a second, is known as *busy waiting*. We chose, rather arbitrarily, to wait 100 milliseconds (one tenth of a second) between each check. If this time is made too short, the computer will be bogged down doing nothing but checking over and over. If the time is made too long, on the other hand, the program will be very sluggish in responding to changes. In this particular program, the situation isn't unbearable because only one thread is ever waiting, as opposed to lots of threads each checking over and over, and also a bit of sluggishness isn't such a bad thing for a poltergeist that only acts sporadically anyway. In general, however, busy waiting is a terrible way for a thread to wait for some relevant change to occur. (There is also a subtle problem with our specific implementation of busy waiting. Because doing busy waiting at all is a bad idea, we'll describe the subtle problem in the end-of-chapter notes section and move on here to a better alternative.)

So, we need another approach to waiting. What we would really like is to put the thread to sleep not for some predetermined period but rather until the situation changes. How will we know when the situation has changed? It changes when the `enable` method is invoked by the user interface. Therefore, we can have the `enable` method explicitly provide a "wake up call" for the poltergeist thread. Java provides a pair of methods for providing this "wait until awakened" and "wake up" functionality: `wait` and `notify`. Using them, we can change our code as follows:

```
// Warning: these two routines don't work, see below for why.
private void waitUntilEnabled() throws InterruptedException {
  while(!enabled){
    wait();
  }
}

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

As the comment warned you, these methods still aren't quite right. However, they do show the essence of what is needed. The `waitUntilEnabled` method simply uses `wait` where it previously used `Thread.sleep`, and the `enable` method uses `notify` to wake the waiting thread back up.

The problem with the preceding code is that it has a race bug. Suppose the poltergeist thread invokes `waitUntilEnabled` and finds that currently `enabled` is `false`. Therefore, it is going to `wait`. However, in the instant in between when the `while` loop checked the `enabled` flag and when the `wait` is executed, the user clicks the checkbox, turning it on. This invokes the `enable` method. The `enable` method

sets the `enabled` variable to `true` and executes `notify`. The `notify` method finds that there currently isn't any waiting thread to wake up, so it returns without having done anything. Then, just after `notify` has failed to find a waiter to wake up, the poltergeist thread resumes executing, calls `wait`, and goes to sleep. Now the thread is nominally enabled but is sleeping.

This problem can be solved the same way as other races, by using the `synchronized` keyword. In fact, the Java system will force you to do so by signaling an error if you forget. In addition to the `enable` and `waitUntilEnabled` methods, you should add the `synchronized` keyword to `disable`, because that is another method that accesses the shared state. (The details of why `disable` should be `synchronized` are explained in the end-of-chapter notes.) Thus our correct version using `wait` and `notify` is as follows:

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

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

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

You might have this question: The `waitUntilEnabled` method is `synchronized`, so in our metaphor it goes into the lockable room, locks the door, and then if the `enabled` variable is `false`, it `wait`s. If it `wait`s with the room locked, how can the `enable` method get in to change `enabled` to `true`? The answer is that the lock is temporarily unlocked while waiting using `wait` and is relocked before `wait` returns.

This need to relock the lock before returning from `wait` also explains why we need a `while` loop around the `wait`, rather than just an `if`. A brief period occurs where the lock is unlocked, between when `enable` leaves and when `wait` relocks it before continuing. Conceivably `disable` could slip in during this window of opportunity and change `enabled` back to `false`. If so, when `wait` does (later) get the lock and resumes execution, the `while` loop will discover this and `wait` again.

Do we now have a working puzzle applet with a checkbox for turning a poltergeist on and off? If you've stayed awake yourself, you may realize that we're still missing something: the checkbox itself, and the associated listener that actually calls the `enable` and `disable` methods. In the `Puzzle` constructor, we can add the following lines (right after the Initialize and Randomize buttons):

```
PoltergeistThread thread = new PoltergeistThread(this);
thread.disable(); // just to emphasize that it starts disabled
thread.start();
Checkbox poltergeistCheckbox = new Checkbox("Poltergeist");
poltergeistCheckbox.addItemListener
  (new PoltergeistItemListener(thread));
controlPanel.add(poltergeistCheckbox);
```

As you can see, `Checkbox`es don't have `ActionListener`s; they have `ItemListener`s, which is a rather fine distinction that the Java system makes. A `Button` can be "acted upon," with each action independent from any previous ones, whereas a `Checkbox` is an "item" that can "change state" from "selected" to "deselected" or vice versa. `Checkbox`es are initially deselected.

In practical terms, this distinction means that we need to write the `PoltergeistItemListener` class as follows:

```
import java.awt.event.*;

public class PoltergeistItemListener implements ItemListener {

  private PoltergeistThread thread;

  public PoltergeistItemListener(PoltergeistThread t){
    thread = t;
  }

  public void itemStateChanged(ItemEvent evt){
    if(evt.getStateChange() == ItemEvent.SELECTED){
      thread.enable();
    } else {
      thread.disable();
    }
  }
}
```

The ItemEvent that is passed in to itemStateChanged tells us information about the state-change that occurred; in particular, we can tell whether it was a change to selectedness (being turned on) or deselectedness (being turned off) as shown in the preceding code.

Before we leave the puzzle applet entirely, we can make one other improvement to it. Right now if you are viewing this applet in a web browser and have the poltergeist turned on, and then you tell the browser to switch to viewing some other web page, the poltergeist will keep right on sliding tiles, even if you can't see the applet. If you then tell the browser to go back to the page with the applet on it, you'll find that the tiles have moved around in your absence. This is probably not what most users would want. To fix this, we can disable the PoltergeistThread when the user stops viewing the applet. We can know when to do this because the browser will invoke the stop method of an applet whenever it stops showing that applet. The default implementation of this method, inherited from the java.applet.Applet class, doesn't do anything. We can override it with a version that puts our thread on hold:

```
public void stop(){
  thread.disable();
}
```

We'll also need to change where we declare thread. At the moment it is a temporary local variable inside the Puzzle constructor; we'll need to change it to be an instance variable instead.

When the user goes back to looking at the applet, we need to set the poltergeist going again. The browser invokes another method to notify the applet that it is once again being viewed; not surprisingly, it is called start. At first you might think it could just do thread.enable(). However, that would turn the poltergeist "back" on even if it hadn't been on in the first place. To solve this problem, we can check the current state of the checkbox and only re-enable the thread if the checkbox is in the on state:

```
public void start(){
  if(poltergeistCheckbox.getState()){
    thread.enable();
  }
}
```

Because this code uses the poltergeistCheckbox variable, we'll again need to move that declaration so as to turn it into an instance variable. Notice that the Checkbox class has a getState method that returns true or false to indicate whether the box is currently in the on or off state, respectively.

Finally, we can add a destroy method to the Puzzle class, which gets called by the browser when the applet is being completely evicted, as opposed to just temporarily ceasing to be viewed. What we'll do in that case is to entirely kill off the poltergeist thread so that it doesn't keep running when there is no longer an applet for it to mess with. We can kill off the thread using the Thread class's stop method:

```
public void destroy(){
  thread.stop();
}
```

Exercise 15.13

Another possibility, rather than forcibly killing off the poltergeist thread, would be to politely ask it to stop running. We could change the main loop in the run method from while(true) to while(!terminated), with a boolean instance variable called terminated. We would initialize terminated to false and set it to true in a new terminate method we could add to the PoltergeistThread class. Then the Puzzle class's destroy method could call terminate instead of stop. Implement this approach. Here's the tricky part: Make sure it works even if the poltergeist is disabled at the time it is terminated.

15.5 An Application: Simulating Compound Interest

Imagine that you have just started work for a small company that produces Java applets for use in education. One of the company's applets is used to illustrate how compound interest works; it is shown in Figure 15.7. This applet simulates the passage of years at a rate of 1 year per second, displaying information in the scrolling area that occupies the main portion of the applet. 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 applet has three controls. One is a checkbox labeled "Run" that can be used to pause the simulation or resume it. (The applet 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 to year 0. The applet is included on the web site for this book, 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, Mr. Wright, comes to you with an interesting problem concerning the compound-interest simulation applet. Although it generally seems to work properly,

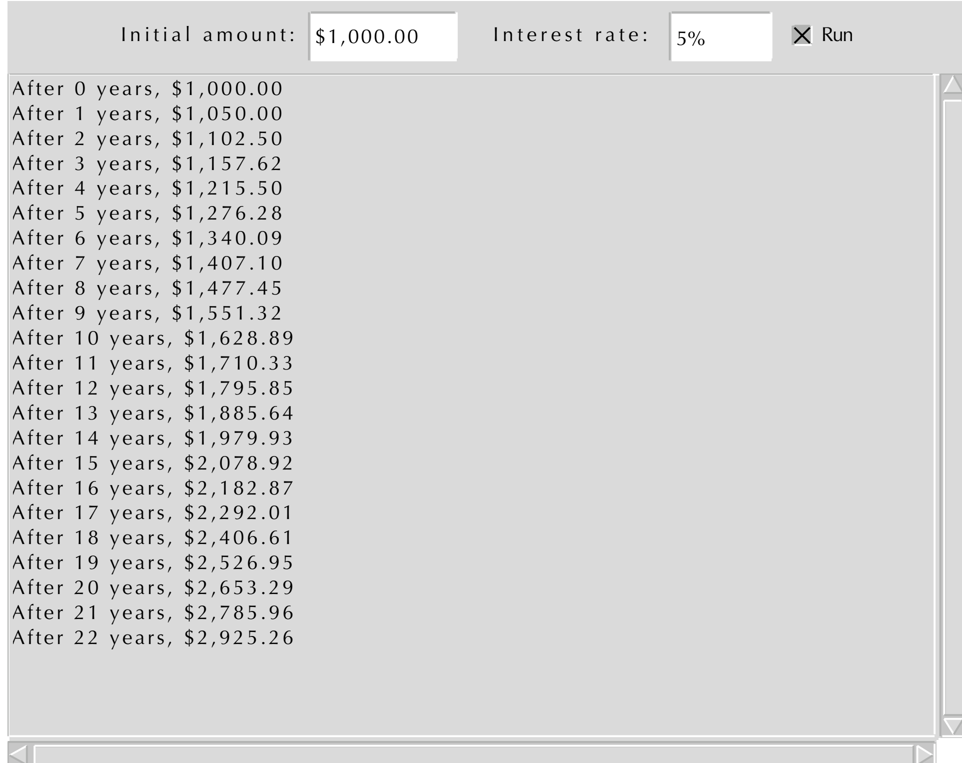


Figure 15.7 Compound interest simulation applet

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.

Because 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 applet 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 the boneheaded programmer who preceded you at the company didn't bother to put "`synchronized`" where it was needed, 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, which 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?

Because of the questions, 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 applet so that these race conditions can be made to repeatably happen, rather than just once in a blue moon, 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, 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 applet 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 methods from the `CompoundingThread` class, `setInitial Amount` 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:

```
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();
  }

  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();
  }
}
```

We have a few new Java features in this code, as usual. Perhaps the most significant is that it uses numbers that aren't integers—this is what the type "`double`" is for. For example, unlike the CompuDuds program, which stored $19.50 as 1950 (the number of cents), the compound interest applet stores it as 19.5. In order to multiply the current amount of money by the multiplier (which might, for example, be 1.05 if the interest rate was 5 percent), the `doYear` method uses `*=`. This operator is analogous to `+=`, in that it uses the old value to compute the new one; here it does so by multiplication. Finally, this code uses a fancy library class, `java.text.NumberFormat`, to format the current amount of money for the output line. For example, if `current` is

19.5, the expression `fmt.format(current)` would evaluate to the string `"$19.50"`. Not only does this expression 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 15.14

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 15.15

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 15.16

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 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 15.17

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 because 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());
  }
}
```

This little class contains some fairly tricky stuff, and although you don't really need to understand it to make another one just like it, we don't want 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`, because 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, which is what the `Double` class is for. We make a `Double` called `value` that holds 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 do the same thing 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 make this declaration with the notation `(Number)`, which is 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 15.18

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

Review Problems

Exercise 15.19

Suppose we add one more method to the `ItemList` class, in the Java version of the CompuDuds program. The method follows, with the nondescriptive name `mystery`:

```
public class ItemList extends Object {

  ... all the existing stuff goes here ...

  public Item mystery(){
    Item soFar = itemVector[0];
    for(int i = 1; i < numItems; i = i + 1){
      if(itemVector[i].price() > soFar.price()){
        soFar = itemVector[i];
      }
    }
    return soFar;
  }
}
```

a. Under what conditions can the `mystery` method be legally invoked to retrieve an `Item`?

b. Describe as precisely as you can what the `mystery` method returns, while retaining an "outsider's" perspective. That is, your description should focus on what the method returns and not how it finds it. Your description should not be in terms of the representation of an `ItemList` (for example, it should not mention the `itemVector`); instead, your description should use terminology that is understandable to someone who is using the `ItemList` class but has not seen the code for it.

c. Give three examples of options that could be added to the user interface that would make use of the `mystery` method. You don't need to show the code, just list what the option would do. It is okay if their usefulness is questionable. You can use existing methods in addition to the new `mystery` method but shouldn't assume any other new methods are added.

Exercise 15.20

Modify the `ItemList` class so that when a deletion leaves the `itemVector` less than one-third full, a new `itemVector` half as big is created. However, the `itemVector`

should never be made any smaller than its original size, 10. See Exercise 14.18 and the text preceding that exercise for more information on this technique.

Chapter Inventory

Vocabulary

applet
concurrency
type
declaration
class method
instance method
array
expression
statement
command-line argument
exception
batch processing
event-driven graphical user interface (GUI)
Abstract Window Toolkit (AWT)
default constructor
interface
World Wide Web
HyperText Markup Language (HTML)
cast
networking
client
server
thread
multithreaded
race
lock
deadlock
busy waiting

Slogans

The sameness principle

Java syntax

```
public
extends
//
/*...*/
private
int
new
this
=
return
void
super
boolean
if...else...
[]
...?...:...
==
for
+=
++
length
--
null
static
/
%
while
true
try...catch...
||
```

`&&`
`import`
`+` for string concatenation and conversion
`implements`
`(`*type*`)`
`synchronized`
`!`
`false`
`throws`
`double`
`*=`

Library Classes and Interfaces

Note that these all are listed with their full name, even though many were used with shortened names. For example, we used `String` rather than the full name `java.lang.String`. The short form is available without needing an `import` directive for those classes that are in the `java.lang` package.

```
java.lang.String
java.lang.System
java.io.IOException
java.lang.NumberFormatException
java.io.BufferedReader
java.io.InputStreamReader
java.awt.Button
java.awt.Checkbox
java.awt.Component
java.awt.TextField
java.applet.Applet
java.awt.Container
java.awt.LayoutManager
java.awt.GridLayout
java.awt.event.ActionListener
java.awt.event.ActionEvent
java.awt.Panel
java.awt.BorderLayout
java.awt.FlowLayout
java.lang.Math
java.lang.Thread
java.lang.InterruptedException
java.awt.event.ItemListener
java.awt.event.ItemEvent
java.text.NumberFormat
java.awt.Label
java.lang.Double
java.lang.Number
```

Nonlibrary Classes

```
Item
OxfordShirt
Chinos
ItemList
CompuDuds
Puzzle
TileActionListener
InitializeActionListener
RandomizeActionListener
PoltergeistThread
PoltergeistActionListener
PoltergeistItemListener
CompoundingThread
Compounder
InitialAmountField
FormattedField
```

Library Methods

`print` (in `java.io.PrintStream`)
`println` (in `java.io.PrintStream`)
`readLine` (in `java.io.BufferedReader`)
`add` (in `java.awt.Container`)
`setLayout` (in `java.awt.Container`)
`setLabel` (in `java.awt.Button`)
`addActionListener` (in `java.awt.Button`)
`getLabel` (in `java.awt.Button`)
`random` (in `java.lang.Math`)
`sleep` (in `java.lang.Thread`)
`start` (in `java.lang.Thread`)
`wait` (in `java.lang.Object`)
`notify` (in `java.lang.Object`)
`getStateChange` (in `java.awt.event.ItemEvent`)
`getState` (in `java.awt.Checkbox`)
`stop` (in `java.lang.Thread`)
`format` (in `java.text.NumberFormat`)
`doubleValue` (in `java.lang.Number`)

Nonlibrary Methods

`price` (in `Item`)
`display` (in `Item`, `OxfordShirt`, `Chinos`, and `ItemList`)
`inputSpecifics` (in `Item`, `OxfordShirt`, and `Chinos`)
`reviseSpecifics` (in `Item`)
`empty` (in `ItemList`)
`totalPrice` (in `ItemList`)
`add` (in `ItemList`)
`choose` (in `ItemList`)
`displayPrice` (in `CompuDuds`)
`main` (in `CompuDuds`)
`inputItem` (in `CompuDuds`)
`inputIntegerInRange` (in `CompuDuds`)
`inputSelection` (in `CompuDuds`)
`init` (in `Puzzle`)
`actionPerformed` (in `TileActionListener`, `InitializeActionListener`, and `PoltergeistActionListener`)
`initializeTiles` (in `Puzzle`)
`randomizeTiles` (in `Puzzle`)
`flip` (in `Puzzle`)
`run` (in `PoltergeistThread` and `CompoundingThread`)
`enable` (in `PoltergeistThread` and `CompoundingThread`)
`disable` (in `PoltergeistThread` and `CompoundingThread`)
`waitUntilEnabled` (in `PoltergeistThread` and `CompoundingThread`)
`itemStateChanged` (in `PoltergeistItemListener`)
`stop` (in `Puzzle` and `Compounder`)
`start` (in `Puzzle` and `Compounder`)
`destroy` (in `Puzzle` and `Compounder`)
`terminate` (in `PoltergeistThread`)
`outputLine` (in `Compounder`)
`clearOutput` (in `Compounder`)
`setInitialAmount` (in `CompoundingThread`)
`setInterestRate` (in `CompoundingThread`)
`setValue` (in `FormattedField`)
`valueEntered` (in `FormattedField`, `InitialAmountField`, and `InterestRateField`)

Sidebars

Nested Calls to Synchronized Methods and Deadlock

Notes

We mentioned that our implementation of busy waiting (Section 15.4) was flawed. The problem is as follows. The Java language specification says that in the absence of `synchronized` constructs, an arbitrarily long delay can occur between when one thread changes a variable and when that change is apparent to other threads. (This permission to delay the news allows more efficient implementations, especially on multiprocessor systems.) The consequence of this specification is that when the main user interface thread sets the `enabled` variable to `true`, there is no guarantee when, if ever, the busy-waiting poltergeist thread will see it as having become `true`. The only way to fix this problem is to use `synchronized`, which guarantees that all variables' values are updated. But the non-busy-waiting version, using `wait` and `notify`, is superior in any case. The same issue explains why in the `wait`/`notify` version the `disable` method needs to be `synchronized`—otherwise the poltergeist thread might never get word of a disablement.

Our discussion of developing graphical user interfaces (GUIs) in Java has been in the context of applets. However, we should point out that nearly everything we said is equally applicable to developing stand-alone Java application programs.

One important source for Java documentation is Sun's Java web site, `http://java.sun.com`. The AWT library classes we used for applets are also documented in a book by Chan and Lee [11], and the language specification is a book by Gosling, Joy, and Steele [23]. Some less definitive but more tutorial sources are the books by Arnold and Gosling [5], Horstmann and Cornell [27], and Campione and Walrath [9].

Leveson and Turner [35] provide a good summary of the Therac-25's problems. They make the point that far more was wrong than just the race bugs we chose to highlight. The software development methodology was badly flawed, the hardware was missing safety interlocks on the theory that the software provided adequate safety assurances, and the procedures for following up on trouble reports were inadequate.