Exercise 2.80 Open the Code Pad in the *better-ticket-machine* project. Type the following in the Code Pad:

```
TicketMachine t1 = new TicketMachine(1000);
t1.getBalance()
t1.insertMoney(500);
t1.getBalance()
```

Take care to type these lines exactly as they appear here; pay particular attention to whether or not there is a semicolon at the end of the line. Note what the calls to `getBalance` return in each case.

Exercise 2.81 Now add the following in the Code Pad:

```
TicketMachine t2 = t1;
```

What would you expect a call to `t2.getBalance()` to return? Try it out.

Exercise 2.82 Add the following:

```
t1.insertMoney(500);
```

What would you expect the following to return? Think carefully about this before you try it, and be sure to use the `t2` variable this time.

```
t2.getBalance()
```

Did you get the answer you expected? Can you find a connection between the variables `t1` and `t2` that would explain what is happening?

2.23 Summary

In this chapter, we have covered the basics of how to create a class definition. Classes contain fields, constructors, and methods that define the state and behavior of objects. Within the body of a constructor or method, a sequence of statements implements that part of its behavior. Local variables can be used as temporary data storage to assist with that. We have covered assignment statements and conditional statements, and will be adding further types of statements in later chapters.

Terms introduced in this chapter:

field, instance variable, constructor, method, method header, method body, actual parameter, formal parameter, accessor, mutator, declaration, initialization, block, statement, assignment statement, conditional statement, return statement, return type, comment, expression, operator, variable, local variable, scope, lifetime

Concept summary

- **object creation** Some objects cannot be constructed unless extra information is provided.
- **field** Fields store data for an object to use. Fields are also known as instance variables.
- **comment** Comments are inserted into the source code of a class to provide explanations to human readers. They have no effect on the functionality of the class.
- **constructor** Constructors allow each object to be set up properly when it is first created.
- **scope** The scope of a variable defines the section of source code from which the variable can be accessed.
- **lifetime** The lifetime of a variable describes how long the variable continues to exist before it is destroyed.
- **assignment statement** Assignment statements store the value represented by the right-hand side of the statement in the variable named on the left.
- **accessor method** Accessor methods return information about the state of an object.
- **mutator method** Mutator methods change the state of an object.
- **println** The method **System.out.println** prints its parameter to the text terminal.
- **conditional statement** A conditional statement takes one of two possible actions based upon the result of a test.
- **boolean expression** Boolean expressions have only two possible values: true and false. They are commonly found controlling the choice between the two paths through a conditional statement.
- **local variable** A local variable is a variable declared and used within a single method. Its scope and lifetime are limited to that of the method.

The following exercises are designed to help you experiment with the concepts of Java that we have discussed in this chapter. You will create your own classes that contain elements such as fields, constructors, methods, assignment statements, and conditional statements.

Exercise 2.83

Below is the outline for a **Book** class, which can be found in the *book-exercise* project. The outline already defines two fields and a constructor to initialize the fields. In this and the next few exercises, you will add features to the class outline.

Add two accessor methods to the class—**getAuthor** and **getTitle**—that return the **author** and **title** fields as their respective results. Test your class by creating some instances and calling the methods.

```
/**
 * A class that maintains information on a book.
 * This might form part of a larger application such
 * as a library system, for instance.
 *
 * @author (Insert your name here.)
 * @version (Insert today's date here.)
 */
public class Book
{
    // The fields.
    private String author;
    private String title;

    /**
     * Set the author and title fields when this object
     * is constructed.
     */
    public Book(String bookAuthor, String bookTitle)
    {
        author = bookAuthor;
        title = bookTitle;
    }

    // Add the methods here...
}
```

Exercise 2.84 Add two methods, **printAuthor** and **printTitle**, to the outline **Book** class. These should print the **author** and **title** fields, respectively, to the terminal window.

Exercise 2.85 Add a field, **pages**, to the **Book** class to store the number of pages. This should be of type **int**, and its initial value should be passed to the single constructor, along with the **author** and **title** strings. Include an appropriate **getPages** accessor method for this field.

Exercise 2.86 Are the **Book** objects you have implemented immutable? Justify your answer.

Exercise 2.87 Add a method, **`printDetails`**, to the **`Book`** class. This should print details of the author, title, and pages to the terminal window. It is your choice how the details are formatted. For instance, all three items could be printed on a single line, or each could be printed on a separate line. You might also choose to include some explanatory text to help a user work out which is the author and which is the title, for example

```
Title: Robinson Crusoe, Author: Daniel Defoe, Pages: 232
```

Exercise 2.88 Add a further field, **`refNumber`**, to the **`Book`** class. This field can store a reference number for a library, for example. It should be of type **`String`** and initialized to the zero length string (**`""`**) in the constructor, as its initial value is not passed in a parameter to the constructor. Instead, define a mutator for it with the following header:

```
public void setRefNumber(String ref)
```

The body of this method should assign the value of the parameter to the **`refNumber`** field. Add a corresponding **`getRefNumber`** accessor to help you check that the mutator works correctly.

Exercise 2.89 Modify your **`printDetails`** method to include printing the reference number. However, the method should print the reference number only if it has been set—that is, if the **`refNumber`** string has a non-zero length. If it has not been set, then print the string **`"ZZZ"`** instead. *Hint:* Use a conditional statement whose test calls the **`length`** method on the **`refNumber`** string.

Exercise 2.90 Modify your **`setRefNumber`** mutator so that it sets the **`refNumber`** field only if the parameter is a string of at least three characters. If it is less than three, then print an error message and leave the field unchanged.

Exercise 2.91 Add a further integer field, **`borrowed`**, to the **`Book`** class. This keeps a count of the number of times a book has been borrowed. Add a mutator, **`borrow`**, to the class. This should update the field by 1 each time it is called. Include an accessor, **`getBorrowed`**, that returns the value of this new field as its result. Modify **`printDetails`** so that it includes the value of this field with an explanatory piece of text.

Exercise 2.92 Add a further **`boolean`** field, **`courseText`**, to the **`Book`** class. This records whether or not a book is being used as a text book on a course. The field should be set through a parameter to the constructor, and the field is immutable. Provide an accessor method for it called **`isCourseText`**.

Exercise 2.93 *Challenge exercise* Create a new project, *heater-exercise*, within BlueJ. Edit the details in the project description—the text note you see in the diagram. Create a class, **`Heater`**, that contains a single field, **`temperature`** whose type is *double-precision floating point*—see Appendix B, Section B.1, for the Java type name that corresponds to this description. Define a constructor that takes no parameters. The **`temperature`** field should be set to the value 15.0 in the constructor. Define the mutators **`warmer`** and **`cooler`**, whose effect is to increase or decrease the value of temperature by 5.0° respectively. Define an accessor method to return the value of **`temperature.`**

Exercise 2.94 *Challenge exercise* Modify your **`Heater`** class to define three new *double-precision floating point* fields: **`min`**, **`max`**, and **`increment`**. The values of **`min`** and **`max`** should be set by parameters passed to the constructor. The value of **`increment`** should be set to 5.0 in the constructor. Modify the definitions of **`warmer`** and **`cooler`** so that they use the value of **`increment`** rather than an explicit value of 5.0. Before proceeding further with this exercise, check that everything works as before.

Now modify the **`warmer`** method so that it will not allow the temperature to be set to a value greater than **`max`**. Similarly modify **`cooler`** so that it will not allow **`temperature`** to be set to a value less than **`min`**. Check that the class works properly. Now add a method, **`setIncrement`**, that takes a single parameter of the appropriate type and uses it to set the value of **`increment`**. Once again, test that the class works as you would expect it to by creating some **`Heater`** objects within BlueJ. Do things still work as expected if a negative value is passed to the **`setIncrement`** method? Add a check to this method to prevent a negative value from being assigned to **`increment`**.

This page intentionally left blank