CHAPTER

2 Understanding Class Definitions

Main concepts discussed in this chapter:

- fields
- constructors
- parameters
- methods (accessor, mutator)
- assignment and conditional statement

Java constructs discussed in this chapter:

field, constructor, comment, parameter, assignment (=), block, return statement, **`void`**, compound assignment operators (`+=`, `-=`), if-statement

In this chapter, we take our first proper look at the source code of a class. We will discuss the basic elements of class definitions: *fields*, *constructors*, and *methods*. Methods contain statements, and initially we look at methods containing only simple arithmetic and printing statements. Later, we introduce *conditional statements* that allow choices between different actions to be made within methods.

We shall start by examining a new project in a fair amount of detail. This project represents a naíve implementation of an automated ticket machine. As we start by introducing the most basic features of classes, we shall quickly find that this implementation is deficient in a number of ways. So we shall then proceed to describe a more sophisticated version of the ticket machine that represents a significant improvement. Finally, in order to reinforce the concepts introduced in this chapter, we take a look at the internals of the *lab-classes* example encountered in Chapter 1.

2.1 Ticket machines

Train stations often provide ticket machines that print a ticket when a customer inserts the correct money for their fare. In this chapter, we shall define a class that models something like these ticket machines. As we shall be looking inside our first Java example classes, we shall keep our simulation fairly simple to start with. That will give us the

opportunity to ask some questions about how these models differ from the real-world versions, and how we might change our classes to make the objects they create more like the real thing.

Our ticket machines work by customers "inserting" money into them and then requesting a ticket to be printed. Each machine keeps a running total of the amount of money it has collected throughout its operation. In real life, it is often the case that a ticket machine offers a selection of different types of ticket, from which customers choose the one they want. Our simplified machines print tickets of only a single price. It turns out to be significantly more complicated to program a class to be able to issue tickets of different values, than it does to offer a single price. On the other hand, with object-oriented programming it is very easy to create multiple instances of the class, each with its own price setting, to fulfill a need for different types of tickets.

2.1.1 Exploring the behavior of a naïve ticket machine

Concept

Object creation: Some objects cannot be constructed unless extra information is provided.

Open the *naíve-ticket-machine* project in BlueJ. This project contains only one class—**`TicketMachine`**—which you will be able to explore in a similar way to the examples we discussed in Chapter 1. When you create a **`TicketMachine`** instance, you will be asked to supply a number that corresponds to the price of tickets that will be issued by that particular machine. The price is taken to be a number of cents, so a positive whole number such as 500 would be appropriate as a value to work with.

Exercise 2.1 Create a **`TicketMachine`** object on the object bench and take a look at its methods. You should see the following: **`getBalance`**, **`getPrice`**, **`insertMoney`**, and **`printTicket`**. Try out the **`getPrice`** method. You should see a return value containing the price of the tickets that was set when this object was created. Use the **`insertMoney`** method to simulate inserting an amount of money into the machine. The machine stores as a balance the amount of money inserted. Use **`getBalance`** to check that the machine has kept an accurate record of the amount just inserted. You can insert several separate amounts of money into the machine, just like you might insert multiple coins or bills into a real machine. Try inserting the exact amount required for a ticket, and use **`getBalance`** to ensure that the balance is increased correctly. As this is a simple machine, a ticket will not be issued automatically, so once you have inserted enough money, call the **`printTicket`** method. A facsimile ticket should be printed in the BlueJ terminal window.

Exercise 2.2 What value is returned if you get the machine's balance after it has printed a ticket?

Exercise 2.3 Experiment with inserting different amounts of money before printing tickets. Do you notice anything strange about the machine's behavior? What happens if you insert too much money into the machine—do you receive any refund? What happens if you do not insert enough and then try to print a ticket?

Exercise 2.4 Try to obtain a good understanding of a ticket machine's behavior by interacting with it on the object bench before we start looking, in the next section, at how the `TicketMachine` class is implemented.

Exercise 2.5 Create another ticket machine for tickets of a different price; remember that you have to supply this value when you create the machine object. Buy a ticket from that machine. Does the printed ticket look any different from those printed by the first machine?

2.2 Examining a class definition

The exercises at the end of the previous section reveal that `TicketMachine` objects only behave in the way we expect them to if we insert the exact amount of money to match the price of a ticket. As we explore the internal details of the class in this section, we shall see why this is so.

Take a look at the source code of the `TicketMachine` class by double-clicking its icon in the class diagram within BlueJ. It should look something like Figure 2.1.

The complete text of the class is shown in Code 2.1. By looking at the text of the class definition piece by piece, we can flesh out some of the object-oriented concepts that discussed in Chapter 1. This class definition contains many of the features of Java that we will see over and over again, so it will pay greatly to study it carefully.

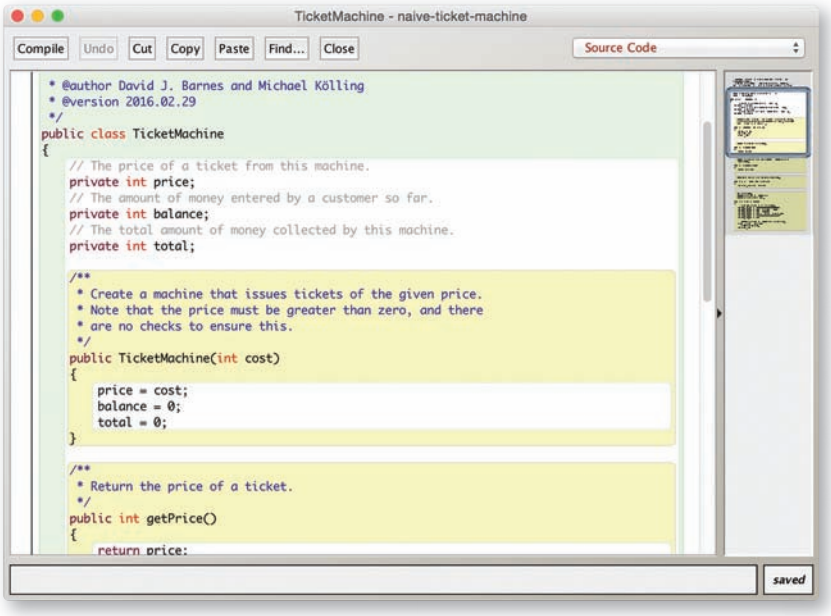


Figure 2.1
The BlueJ editor window

Code 2.1
The `TicketMachine` class

```
/**
 * TicketMachine models a naive ticket machine that issues
 * flat-fare tickets.
 * The price of a ticket is specified via the constructor.
 * It is a naive machine in the sense that it trusts its users
 * to insert enough money before trying to print a ticket.
 * It also assumes that users enter sensible amounts.
 *
 * @author David J. Barnes and Michael Kölling
 * @version 2016.02.29
 */
public class TicketMachine
{
    // The price of a ticket from this machine.
    private int price;
    // The amount of money entered by a customer so far.
    private int balance;
    // The total amount of money collected by this machine.
    private int total;

    /**
     * Create a machine that issues tickets of the given price.
     * Note that the price must be greater than zero, and there
     * are no checks to ensure this.
     */
    public TicketMachine(int cost)
    {
        price = cost;
        balance = 0;
        total = 0;
    }

    /**
     * Return the price of a ticket.
     */
    public int getPrice()
    {
        return price;
    }

    /**
     * Return the amount of money already inserted for the
     * next ticket.
     */
    public int getBalance()
    {
        return balance;
    }

    /**
     * Receive an amount of money from a customer.
     */
```

Code 2.1 continued
The `TicketMachine` class

```
    public void insertMoney(int amount)
    {
        balance = balance + amount;
    }

    /**
     * Print a ticket.
     * Update the total collected and
     * reduce the balance to zero.
     */
    public void printTicket()
    {
        // Simulate the printing of a ticket.
        System.out.println("##################");
        System.out.println("# The BlueJ Line");
        System.out.println("# Ticket");
        System.out.println("# " + price + " cents.");
        System.out.println("##################");
        System.out.println();

        // Update the total collected with the balance.
        total = total + balance;
        // Clear the balance.
        balance = 0;
    }
}
```

2.3 The class header

The text of a class can be broken down into two main parts: a small outer wrapping that simply names the class (appearing on a green background), and a much larger inner part that does all the work. In this case, the outer wrapping appears as follows:

```
public class TicketMachine
{
    Inner part of the class omitted.
}
```

The outer wrappings of different classes all look much the same. The outer wrapping contains the class header, whose main purpose is to provide a name for the class. By a widely followed convention, we always start class names with an uppercase letter. As long as it is used consistently, this convention allows class names to be easily distinguished from other sorts of names, such as variable names and method names, which will be described shortly. Above the class header is a comment (shown as blue text) that tells us something about the class.

Exercise 2.6 Write out what you think the outer wrappers of the **`Student`** and **`LabClass`** classes might look like; do not worry about the inner part.

Exercise 2.7 Does it matter whether we write

```
public class TicketMachine
```

or

```
class public TicketMachine
```

in the outer wrapper of a class? Edit the source of the **`TicketMachine`** class to make the change, and then close the editor window. Do you notice a change in the class diagram?

What error message do you get when you now press the *Compile* button? Do you think this message clearly explains what is wrong?

Change the class back to how it was, and make sure that this clears the error when you compile it.

Exercise 2.8 Check whether or not it is possible to leave out the word **`public`** from the outer wrapper of the **`TicketMachine`** class.

Exercise 2.9 Put back the word **`public`**, and then check whether it is possible to leave out the word **`class`** by trying to compile again. Make sure that both words are put back as they were originally before continuing.

2.3.1 Keywords

The words "public" and "class" are part of the Java language, whereas the word "TicketMachine" is not—the person writing the class has chosen that particular name. We call words like "public" and "class" *keywords* or *reserved words* – the terms are used frequently and interchangeably. There are around 50 of these in Java, and you will soon be able to recognize most of them. A point worth remembering is that Java keywords never contain uppercase letters, whereas the words we get to choose (like "TicketMachine") are often a mix of upper- and lowercase letters.

2.4 Fields, constructors, and methods

The inner part of the class is where we define the *fields*, *constructors*, and *methods* that give the objects of that class their own particular characteristics and behavior. We can summarize the essential features of those three components of a class as follows:

- The fields store data persistently within an object.
- The constructors are responsible for ensuring that an object is set up properly when it is first created.
- The methods implement the behavior of an object; they provide its functionality.

In BlueJ, fields are shown as text on a white background, while constructors and methods are displayed as yellow boxes.

In Java, there are very few rules about the order in which you choose to define the fields, constructors, and methods within a class. In the **`TicketMachine`** class, we have chosen to list the fields first, the constructors second, and finally the methods (Code 2.2). This is the order that we shall follow in all of our examples. Other authors choose to adopt different styles, and this is mostly a question of preference. Our style is not necessarily better than all others. However, it is important to choose one style and then use it consistently, because then your classes will be easier to read and understand.

Code 2.2
Our ordering of fields, constructors, and methods

```
public class ClassName
{
    Fields
    Constructors
    Methods
}
```

Exercise 2.10 From your earlier experimentation with the ticket machine objects within BlueJ, you can probably remember the names of some of the methods—**`printTicket`**, for instance. Look at the class definition in Code 2.1 and use this knowledge, along with the additional information about ordering we have given you, to make a list of the names of the fields, constructors, and methods in the **`TicketMachine`** class. *Hint:* There is only one constructor in the class.

Exercise 2.11 What are the two features of the constructor that make it look significantly different from the methods of the class?

2.4.1 Fields

Concept

Fields store data for an object to use. Fields are also known as instance variables.

Fields store data persistently within an object. The **`TicketMachine`** class has three fields: **`price`**, **`balance`**, and **`total`**. Fields are also known as *instance variables*, because the word *variable* is used as a general term for things that store data in a program. We have defined the fields right at the start of the class definition (Code 2.3). All of these variables are associated with monetary items that a ticket-machine object has to deal with:

- **`price`** stores the fixed price of a ticket;
- **`balance`** stores the amount of money inserted into the machine by a user prior to asking for a ticket to be printed;
- **`total`** stores the total amount of money inserted into the machine by all users since the machine object was constructed (excluding any current balance). The idea is that, when a ticket is printed, any money in the balance is transferred to the total.

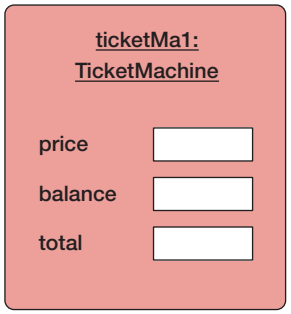
Code 2.3
The fields of the `TicketMachine` class

```
public class TicketMachine
{
    private int price;
    private int balance;
    private int total;

    Constructors and methods omitted.
}
```

Fields are small amounts of space inside an object that can be used to store data persistently. Every object will have space for each field declared in its class. Figure 2.2 shows a diagrammatic representation of a ticket-machine object with its three fields. The fields have not yet been assigned any values; once they have, we can write each value into the box representing the field. The notation is similar to that used in BlueJ to show objects on the object bench, except that we show a bit more detail here. In BlueJ, for space reasons, the fields are not displayed on the object icon. We can, however, see them by opening an inspector window (Section 1.7).

Figure 2.2
An object of class `TicketMachine`



Each field has its own declaration in the source code. On the line above each in the full class definition, we have added a single line of text—a *comment*—for the benefit of human readers of the class definition:

```
// The price of a ticket from this machine.
private int price;
```

Concept

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.

A single-line comment is introduced by the two characters "//", which are written with no spaces between them. More-detailed comments, often spanning several lines, are usually written in the form of multiline comments. These start with the character pair "/*" and end with the pair "*/". There is a good example preceding the header of the class in Code 2.1.

The definitions of the three fields are quite similar:

- All definitions indicate that they are *private* fields of the object; we shall have more to say about what this means in Chapter 6, but for the time being we will simply say that we always define fields to be private.

- All three fields are of type **int**. **int** is another keyword and represents the data type integer. This indicates that each can store a single whole-number value, which is reasonable given that we wish them to store numbers that represent amounts of money in cents.

Fields can store values that can vary over time, so they are also known as *variables*. The value stored in a field can be changed from its initial value if required. For instance, as more money is inserted into a ticket machine, we shall want to change the value stored in the **balance** field. It is common to have fields whose values change often, such as **balance** and **total**, and others that change rarely or not at all, such as **price**. The fact that the value of **price** doesn't vary once set does not alter the fact that it is still called a variable. In the following sections, we shall also meet other kinds of variables in addition to fields, but they will all share the same fundamental purpose of storing data.

The **price**, **balance**, and **total** fields are all the data items that a ticket-machine object needs to fulfill its role of receiving money from a customer, printing tickets, and keeping a running total of all the money that has been put into it. In the following sections, we shall see how the constructor and methods use those fields to implement the behavior of naíve ticket machines.

Exercise 2.12 What do you think is the *type* of each of the following fields?

```
private int count;
private Student representative;
private Server host;
```

Exercise 2.13 What are the *names* of the following fields?

```
private boolean alive;
private Person tutor;
private Game game;
```

Exercise 2.14 From what you know about the naming conventions for classes, which of the type names in Exercises 2.12 and 2.13 would you say are class names?

Exercise 2.15 In the following field declaration from the **TicketMachine** class

```
private int price;
```

does it matter which order the three words appear in? Edit the **TicketMachine** class to try different orderings. After each change, close the editor. Does the appearance of the class diagram after each change give you a clue as to whether or not other orderings are possible? Check by pressing the *Compile* button to see if there is an error message.

Make sure that you reinstate the original version after your experiments!

Exercise 2.16 Is it always necessary to have a semicolon at the end of a field declaration? Once again, experiment via the editor. The rule you will learn here is an important one, so be sure to remember it.

Exercise 2.17 Write in full the declaration for a field of type **`int`** whose name is **`status`**.

From the definitions of fields we have seen so far, we can begin to put a pattern together that will apply whenever we define a field variable in a class:

- They usually start with the reserved word **`private`**.
- They include a type name (such as **`int`**, **`String`**, **`Person`**, etc.)
- They include a user-chosen name for the field variable.
- They end with a semicolon.

Remembering this pattern will help you when you write your own classes.

Indeed, as we look closely at the source code of different classes, you will see patterns such as this one emerging over and over again. Part of the process of learning to program involves looking out for such patterns and then using them in your own programs. That is one reason why studying source code in detail is so useful at this stage.

2.4.2 Constructors

Concept

Constructors allow each object to be set up properly when it is first created.

Constructors have a special role to fulfill. They are responsible for ensuring that an object is set up properly when it is first created; in other words, for ensuring that an object is ready to be used immediately following its creation. This construction process is also called *initialization*.

In some respects, a constructor can be likened to a midwife: it is responsible for ensuring that the new object comes into existence properly. Once an object has been created, the constructor plays no further role in that object's life and cannot be called again. Code 2.4 shows the constructor of the **`TicketMachine`** class.

One of the distinguishing features of constructors is that they have the same name as the class in which they are defined—**`TicketMachine`** in this case. The constructor's name immediately follows the word **`public`**, with nothing in between.[1]

We should expect a close connection between what happens in the body of a constructor and the fields of the class. This is because one of the main roles of the constructor is to initialize the fields. It will be possible with some fields, such as **`balance`** and **`total`**, to set sensible initial values by assigning a constant number—zero in this case. With others, such as the ticket price, it is not that simple, as we do not know the price that tickets from

[1] While this description is a slight simplification of the full Java rule, it fits the general rule we will use in the majority of code in this book.

a particular machine will have until that machine is constructed. Recall that we might wish to create multiple machine objects to sell tickets with different prices, so no one initial price will always be right. You will know from experimenting with creating **`TicketMachine`** objects within BlueJ that you had to supply the cost of the tickets whenever you created a new ticket machine. An important point to note here is that the price of a ticket is initially determined *externally* and then has to be *passed into* the constructor. Within BlueJ, you decide the value and enter it into a dialog box. Part of the task of the constructor is to receive that value and store it in the **`price`** field of the newly created ticket machine, so the machine can remember what that value was without you having to keep reminding it.

Code 2.4
The constructor of the **`TicketMachine`** class

```
public class TicketMachine
{
    Fields omitted.

    /**
     * Create a machine that issues tickets of the given price.
     * Note that the price must be greater than zero, and there
     * are no checks to ensure this.
     */
    public TicketMachine(int cost)
    {
        price = cost;
        balance = 0;
        total = 0;
    }

    Methods omitted.
}
```

We can see from this that one of the most important roles of a field is to remember external information passed into the object, so that that information is available to an object throughout its lifetime. Fields, therefore, provide a place to store long-lasting (i.e., persistent) data.

Figure 2.3 shows a ticket-machine object after the constructor has executed. Values have now been assigned to the fields. From this diagram, we can tell that the ticket machine was created by passing in 500 as the value for the ticket price.

In the next section, we discuss how values are received by an object from outside.

Note In Java, all fields are automatically initialized to a default value if they are not explicitly initialized. For integer fields, this default value is zero. So, strictly speaking, we could have done without setting `balance` and `total` to zero, relying on the default value to give us the same result. However, we prefer to write the explicit assignments anyway. There is no disadvantage to it, and it serves well to document what is actually happening. We do not rely on a reader of the class knowing what the default value is, and we document that we really want this value to be zero and have not just forgotten to initialize it.

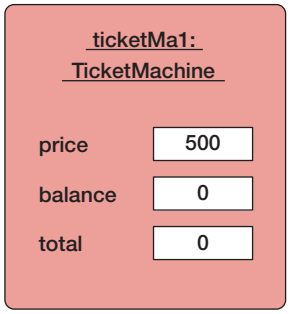


Figure 2.3
A `TicketMachine` object after initialization (created for 500-cent tickets)

2.5 Parameters: receiving data

Constructors and methods play quite different roles in the life of an object, but the way in which both receive values from outside is the same: via *parameters.* You may recall that we briefly encountered parameters in Chapter 1 (Section 1.4). Parameters are another sort of variable, just as fields are, so they are also used to hold data. Parameters are variables that are defined in the header of a constructor or method:

```
public TicketMachine(int cost)
```

This constructor has a single parameter, **`cost`**, which is of type **`int`**—the same type as the **`price`** field it will be used to set. A parameter is used as a sort of temporary messenger, carrying data originating from outside the constructor or method, and making it available inside it.

Figure 2.4 illustrates how values are passed via parameters. In this case, a BlueJ user enters the external value into the dialog box when creating a new ticket machine (shown on the left), and that value is then copied into the **`cost`** parameter of the new machine's constructor. This is illustrated with the arrow labeled (A). The box in the **`TicketMachine`** object in Figure 2.4, labeled "TicketMachine (constructor)," represents additional space for the object that is created only when the constructor executes. We shall call it the *constructor space* of the object (or *method space* when we talk about methods instead of constructors, as the situation there is the same). The constructor space is used to provide space to store the values for the constructor's parameters. In our diagrams, all variables are represented by white boxes.

Concept

The **scope** of a variable defines the section of source code from which the variable can be accessed.

We distinguish between the parameter *names* inside a constructor or method, and the parameter *values* outside, by referring to the names as *formal parameters* and the values as *actual parameters*. So **`cost`** is a formal parameter, and a user-supplied value such as 500 is an actual parameter.

A formal parameter is available to an object only within the body of a constructor or method that declares it. We say that the *scope* of a parameter is restricted to the body of the constructor or method in which it is declared. In contrast, the scope of a field is the whole of the class definition–it can be accessed from anywhere in the same class. This is a very important difference between these two sorts of variables.

Concept

The **lifetime** of a variable describes how long the variable continues to exist before it is destroyed.

A concept related to variable scope is variable *lifetime*. The lifetime of a parameter is limited to a single call of a constructor or method. When a constructor or method is called, the extra space for the parameter variables is created, and the external values copied into that space. Once that call has completed its task, the formal parameters disappear and the values they held are lost. In other words, when the constructor has finished executing, the whole constructor space is removed, along with the parameter variables held within it (see Figure 2.4).

In contrast, the lifetime of a field is the same as the lifetime of the object to which it belongs. When an object is created, so are the fields, and they persist for the lifetime of the object. It follows that if we want to remember the cost of tickets held in the `cost` parameter, we must store the value somewhere persistent—that is, in the `price` field.

Just as we expect to see a close link between a constructor and the fields of its class, we expect to see a close link between the constructor's parameters and the fields, because external values will often be needed to set the initial values of one or more of those fields. Where this is the case, the parameter types will closely match the types of the corresponding fields.

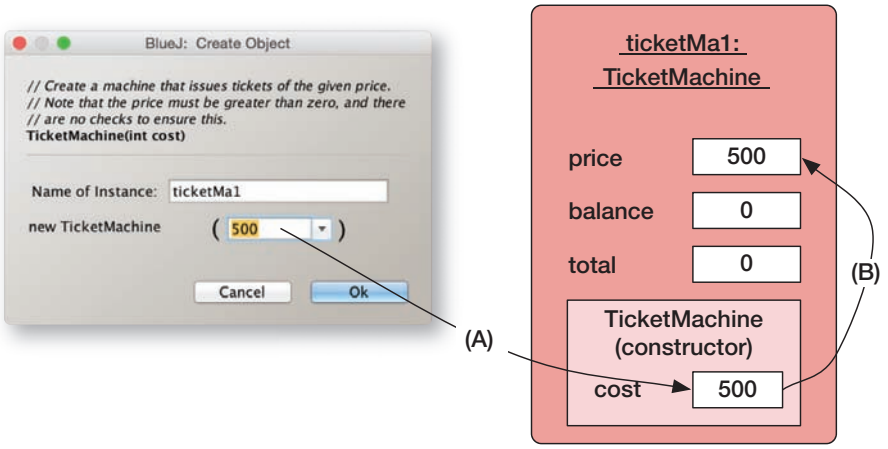


Figure 2.4
Parameter passing (A) and assignment (B)

Exercise 2.18 To what class does the following constructor belong?

```
public Student(String name)
```

Exercise 2.19 How many parameters does the following constructor have, and what are their types?

```
public Book(String title, double price)
```

Exercise 2.20 Can you guess what types some of the **Book** class's fields might be, from the parameters in its constructor? Can you assume anything about the names of its fields?

2.5.1 Choosing variable names

One of the things you might have noticed is that the variable names we use for fields and parameters have a close connection with the purpose of the variable. Names such as **price**, **cost**, **title**, and **alive** all tell you something useful about the information being stored in that variable. This, makes it easier to understand what is going on in the program. Given that we have a large degree of freedom in our choice of variable names, it is worth following this principle of choosing names that communicate a sense of purpose rather than arbitrary and meaningless combinations of letters and numbers.

2.6 Assignment

In the previous section, we noted the need to copy the short-lived value stored in a parameter variable into somewhere more permanent—a field variable. In order to do this, the body of the constructor contains the following *assignment statement*:

```
price = cost;
```

Concept

Assignment statements store the value represented by the right-hand side of the statement in the variable named on the left.

Assignment statements are used frequently in programming, as a means to store a value into a variable. They can be recognized by the presence of an assignment operator, such as "=" in the example above. Assignment statements work by taking the value of what appears on the right-hand side of the operator and copying that value into the variable on the left-hand side. This is illustrated in Figure 2.4 by the arrow labeled (B). The right-hand side is called an *expression*. In their most general form, expressions are things that compute values, but in this case, the expression consists of just a single variable, whose value is copied into the **price** variable. We shall see examples of more-complicated expressions later in this chapter.

One rule about assignment statements is that the type of the expression on the right-hand side must match the type of the variable to which its value is assigned. We have already met three different, commonly used types: **int**, **String**, and (very briefly) **boolean**. This rule means that we are not allowed to store an **int**-type expression in a **String**-type variable, for instance. This same rule also applies between formal parameters and actual parameters: the type of an actual-parameter expression must match the type of the formal-parameter variable. For now, we can say that the types of both must be the same, although we shall see in later chapters that this is not the whole truth.

Exercise 2.21 Suppose that the class **Pet** has a field called **name** that is of the type **String**. Write an assignment statement in the body of the following constructor so that the **name** field will be initialized with the value of the constructor's parameter.

```
public Pet(String petsName)
{
}
```

Exercise 2.22 *Challenge exercise* The following object creation will result in the constructor of the **Date** class being called. Can you write the constructor's header?

```
new Date("March", 23, 1861)
```

Try to give meaningful names to the parameters.

2.7 Methods

The **TicketMachine** class has four methods: **getPrice**, **getBalance**, **insertMoney**, and **printTicket**. You can see them in the class's source code (Code 2.1) as yellow boxes. We shall start our look at the source code of methods by considering **getPrice** (Code 2.5).

Code 2.5
The **getPrice** method

```
public class TicketMachine
{
    Fields omitted.
    Constructor omitted.
    /**
     * Return the price of a ticket.
     */
    public int getPrice()
    {
        return price;
    }
    Remaining methods omitted.
}
```

Concept

Methods consist of two parts: a header and a body.

Methods have two parts: a *header* and a *body*. Here is the method header for **getPrice**, preceded by a descriptive comment:

```
/**
 * Return the price of a ticket.
 */
public int getPrice()
```

It is important to distinguish between method headers and field declarations, because they can look quite similar. We can tell that **getPrice** is a method and not a field because method headers always include a pair of parentheses–"(" and ")"–and no semicolon at the end of the header.

The method body is the remainder of the method after the header. It is always enclosed by a matching pair of curly brackets: "{" and "}". Method bodies contain the *declarations* and *statements* that define what an object does when that method is called. Declarations are used to create additional, temporary variable space, while statements describe the actions of the method. In **getPrice**, the method body contains a single statement, but we shall soon see examples where the method body consists of many lines of both declarations and statements.

Any set of declarations and statements between a pair of matching curly brackets is known as a *block*. So the body of the **TicketMachine** class, the bodies of the constructor, and all of the methods within the class are blocks.

There are at least two significant differences between the headers of the **TicketMachine** constructor and the **getPrice** method:

```
public TicketMachine(int cost)

public int getPrice()
```

- The method has a *return type* of **int**; the constructor has no return type. A return type is written just before the method name. This is a difference that applies in all cases.
- The constructor has a single formal parameter, **cost**, but the method has none—just a pair of empty parentheses. This is a difference that applies in this particular case.

It is an absolute rule in Java that a constructor may not have a return type. On the other hand, both constructors and methods may have any number of formal parameters, including none.

Within the body of **getPrice** there is a single statement:

```
return price;
```

This is called a *return statement*. It is responsible for returning an integer value to match the **int** return type in the method's header. Where a method contains a return statement, it is always the final statement of that method, because no further statements in the method will be executed once the return statement is executed.

Return types and return statements work together. The **int** return type of **getPrice** is a form of promise that the body of the method will do something that ultimately results in an integer value being calculated and returned as the method's result. You might like to think of a method call as being a form of question to an object, and the return value from the method being the object's answer to that question. In this case, when the **getPrice** method is called on a ticket machine, the question is, What do tickets cost? A ticket machine does not need to perform any calculations to be able to answer that, because it keeps the answer in its **price** field. So the method answers by just returning the value of that variable. As we gradually develop more-complex classes, we shall inevitably encounter more-complex questions that require more work to supply their answers.

2.8 Accessor and mutator methods

We often describe methods such as the two "get" methods of **TicketMachine** (**getPrice** and **getBalance**) as *accessor methods* (or just *accessors*). This is because they return information to the caller about the state of an object; they provide access to information about the object's state. An accessor usually contains a return statement in order to pass back that information.

Concept

Accessor methods return information about the state of an object.

There is often confusion about what "returning a value" actually means in practice. People often think it means that something is printed by the program. This is not the case at all— we shall see how printing is done when we look at the **printTicket** method. Rather, returning a value means that some information is passed internally between two different parts of the program. One part of the program has requested information from an object via a method call, and the return value is the way the object has of passing that information back to the caller.

The **get** methods of a ticket machine perform similar tasks: returning the value of one of their object's fields. The remaining methods—**insertMoney** and **printTicket**—have a

Exercise 2.23 Compare the header and body of the **getBalance** method with the header and body of the **getPrice** method. What are the differences between them?

Exercise 2.24 If a call to **getPrice** can be characterized as "What do tickets cost?" how would you characterize a call to **getBalance**?

Exercise 2.25 If the name of **getBalance** is changed to **getAmount**, does the return statement in the body of the method also need to be changed for the code to compile? Try it out within BlueJ. What does this tell you about the name of an accessor method and the name of the field associated with it?

Exercise 2.26 Write an accessor method **getTotal** in the **TicketMachine** class. The new method should return the value of the **total** field.

Exercise 2.27 Try removing the return statement from the body of **getPrice**. What error message do you see now when you try compiling the class?

Exercise 2.28 Compare the method headers of **getPrice** and **printTicket** in Code 2.1. Apart from their names, what is the main difference between them?

Exercise 2.29 Do the **insertMoney** and **printTicket** methods have return statements? Why do you think this might be? Do you notice anything about their headers that might suggest why they do not require return statements?

Concept

Mutator methods change the state of an object.

much more significant role, primarily because they *change* the value of one or more fields of a ticket-machine object each time they are called. We call methods that change the state of their object *mutator methods* (or just *mutators*).

In the same way as we think of a call to an accessor as a request for information (a question), we can think of a call to a mutator as a request for an object to change its state. The most basic form of mutator is one that takes a single parameter whose value is used to directly overwrite what is stored in one of an object's fields. In a direct complement to "get" methods, these are often called "set" methods, although the **TicketMachine** does not have any of those, at this stage.

One distinguishing effect of a mutator is that an object will often exhibit slightly different behavior before and after it is called. We can illustrate this with the following exercise.

Exercise 2.30 Create a ticket machine with a ticket price of your choosing. Before doing anything else, call the **getBalance** method on it. Now call the **insertMoney** method (Code 2.6) and give a non-zero positive amount of money as the actual parameter. Now call **getBalance** again. The two calls to **getBalance** should show different outputs, because the call to **insertMoney** had the effect of changing the machine's state via its **balance** field.

The header of **insertMoney** has a **void** return type and a single formal parameter, **amount**, of type **int**. A **void** return type means that the method does not return any value to its caller. This is significantly different from all other return types. Within BlueJ, the difference is most noticeable in that no return-value dialog is shown following a call to a **void** method. Within the body of a **void** method, this difference is reflected in the fact that there is no return statement.[2]

Code 2.6
The **insertMoney** method

```
/**
 * Receive an amount of money from a customer.
 */
public void insertMoney(int amount)
{
    balance = balance + amount;
}
```

In the body of **insertMoney**, there is a single statement that is another form of assignment statement. We always consider assignment statements by first examining the calculation on the right-hand side of the assignment symbol. Here, its effect is to calculate a value that is the sum of the number in the **amount** parameter and the number in the **balance** field. This combined value is then assigned to the **balance** field. So the effect is to increase the value in **balance** by the value in **amount**.[3]

Exercise 2.31 How can we tell from just its header that **setPrice** is a method and not a constructor?

```
public void setPrice(int cost)
```

Exercise 2.32 Complete the body of the **setPrice** method so that it assigns the value of its parameter to the **price** field.

Exercise 2.33 Complete the body of the following method, whose purpose is to add the value of its parameter to a field named **score**.

```
/**
 * Increase score by the given number of points.
 */
public void increase(int points)
{
  ...
}
```

[2] In fact, Java does allow **void** methods to contain a special form of return statement in which there is no return value. This takes the form

```
return;
```

and simply causes the method to exit without executing any further code.

[3] Adding an amount to the value in a variable is so common that there is a special **compound assignment operator** to do this: **+=**. For instance:

```
balance += amount;
```

Exercise 2.34 Is the **increase** method a mutator? If so, how could you demonstrate this?

Exercise 2.35 Complete the following method, whose purpose is to subtract the value of its parameter from a field named **price**.

```
/**
 * Reduce price by the given amount.
 */
public void discount(int amount)
{
  ...
}
```

2.9 Printing from methods

Code 2.7 shows the most complex method of the class: **printTicket**. To help your understanding of the following discussion, make sure that you have called this method on a ticket machine. You should have seen something like the following printed in the BlueJ terminal window:

```
##################
# The BlueJ Line
# Ticket
# 500 cents.
##################
```

This is the longest method we have seen so far, so we shall break it down into more manageable pieces:

- The header indicates that the method has a **void** return type, and that it takes no parameters.
- The body comprises eight statements plus associated comments.
- The first six statements are responsible for printing what you see in the BlueJ terminal window: five lines of text and a sixth blank line.
- The seventh statement adds the balance inserted by the customer (through previous calls to **insertMoney**) to the running total of all money collected so far by the machine.
- The eighth statement resets the balance to zero with a basic assignment statement, in preparation for the next customer.

Code 2.7
The **print-Ticket** method

```
/**
 * Print a ticket.
 * Update the total collected and reduce the balance to zero.
 */
public void printTicket()
{
    // Simulate the printing of a ticket.
    System.out.println("##################");
    System.out.println("# The BlueJ Line");
    System.out.println("# Ticket");
    System.out.println("# " + price + " cents.");
    System.out.println("##################");
    System.out.println();

    // Update the total collected with the balance.
    total = total + balance;
    // Clear the balance.
    balance = 0;
}
```

Concept

The method **System.out.println** prints its parameter to the text terminal.

By comparing the output that appears with the statements that produced it, it is easy to see that a statement such as

```
System.out.println("# The BlueJ Line");
```

literally prints the string that appears between the matching pair of double-quote characters. The basic form of a call to `println` is

```
System.out.println(something-we-want-to-print);
```

where *something-we-want-to-print* can be replaced by any arbitrary string, enclosed between a pair of double-quote characters. For instance, there is nothing significant in the "#" character that is in the string—it is simply one of the characters we wish to be printed.

All of the printing statements in the **printTicket** method are calls to the **println** method of the **System.out** object that is built into the Java language, and what appears between the round brackets is the parameter to each method call, as you might expect. However, in the fourth statement, the actual parameter to **println** is a little more complicated and requires some more explanation:

```
System.out.println("# " + price + " cents.");
```

What it does is print out the price of the ticket, with some extra characters on either side of the amount. The two "+" operators are being used to construct a single actual parameter, in the form of a string, from three separate components:

- the string literal: **"# "** (*note the space character after the hash*);
- the value of the **price** field (note that there are no quotes around the field name because we want the field's value, not its name);
- the string literal: **" cents."** (note the space character before the word **"cents"**).

When used between a string and anything else, "+" is a string-concatenation operator (i.e., it concatenates or joins strings together to create a new string) rather than an arithmetic-addition operator. So the numeric value of price is converted into a string and joined to its two surrounding strings.

Note that the final call to **println** contains no string parameter. This is allowed, and the result of calling it will be to leave a blank line between this output and any that follows after. You will easily see the blank line if you print a second ticket.

Exercise 2.36 Write down exactly what will be printed by the following statement:

```
System.out.println("My cat has green eyes.");
```

Exercise 2.37 Add a method called **prompt** to the **TicketMachine** class. This should have a **void** return type and take no parameters. The body of the method should print the following single line of output:

```
Please insert the correct amount of money.
```

Exercise 2.38 What do you think would be printed if you altered the fourth statement of **printTicket** so that **price** also has quotes around it, as follows?

```
System.out.println("# " + "price" + " cents.");
```

Exercise 2.39 What about the following version?

```
System.out.println("# price cents.");
```

Exercise 2.40 Could either of the previous two versions be used to show the price of tickets in different ticket machines? Explain your answer.

Exercise 2.41 Add a **showPrice** method to the **TicketMachine** class. This should have a **void** return type and take no parameters. The body of the method should print:

```
The price of a ticket is xyz cents.
```

where ***xyz*** should be replaced by the value held in the **price** field when the method is called.

Exercise 2.42 Create two ticket machines with differently priced tickets. Do calls to their **showPrice** methods show the same output, or different? How do you explain this effect?

2.10 Method summary

It is worth summarizing a few features of methods at this point, because methods are fundamental to the programs we will be writing and exploring in this book. They implement the core actions of every object.

A method with parameters will receive data passed to it from the method's caller, and will then use that data to help it perform a particular task. However, not all methods take parameters; many simply use the data stored in the object's fields to carry out their task.

If a method has a non-**`void`** return type, it will return some data to the place it was called from—and that data will almost certainly be used in the caller for further calculations or program manipulations. Many methods, though, have a **`void`** return type and return nothing, but they still perform a useful task within the context of their object.

Accessor methods have non-**`void`** return types and return information about the object's state. Mutator methods modify an object's state. Mutators often take parameters whose values are used in the modification, although it is still possible to write a mutating method that does not take parameters.

2.11 Summary of the naíve ticket machine

We have now examined the internal structure of the naíve **`TicketMachine`** class in some detail. We have seen that the class has a small outer layer that gives a name to the class, and a more substantial inner body containing fields, a constructor, and several methods. Fields are used to store data that enable objects to maintain a state that persists between method calls. Constructors are used to set up an initial state when an object is created. Having a proper initial state will enable an object to respond appropriately to method calls immediately following its creation. Methods implement the defined behavior of the class's objects. Accessor methods provide information about an object's state, and mutator methods change an object's state.

We have seen that constructors are distinguished from methods by having the same name as the class in which they are defined. Both constructors and methods may take parameters, but only methods may have a return type. Non-**`void`** return types allow us to pass a value out of a method to the place where the method was called from. A method with a non-**`void`** return type must have at least one return statement in its body; this will often be the final statement. Constructors never have a return type of any sort—not even **`void`**.

Before attempting these exercises, be sure that you have a good understanding of how ticket machines behave and how that behavior is implemented through the fields, constructor, and methods of the class.

Exercise 2.43 Modify the constructor of **`TicketMachine`** so that it no longer has a parameter. Instead, the price of tickets should be fixed at 1,000 cents. What effect does this have when you construct ticket-machine objects within BlueJ?

Exercise 2.44 Give the class two constructors. One should take a single parameter that specifies the price, and the other should take no parameter and set the price to be a default value of your choosing. Test your implementation by creating machines via the two different constructors.

Exercise 2.45 Implement a method, **`empty`**, that simulates the effect of removing all money from the machine. This method should have a **`void`** return type, and its body should simply set the **`total`** field to zero. Does this method need to take any parameters? Test your method by creating a machine, inserting some money, printing some tickets, checking the total, and then emptying the machine. Is the **`empty`** method a mutator or an accessor?

2.12 Reflecting on the design of the ticket machine

From our study of the internals of the **`TicketMachine`** class, you should have come to appreciate how inadequate it would be in the real world. It is deficient in several ways:

- It contains no check that the customer has entered enough money to pay for a ticket.
- It does not refund any money if the customer pays too much for a ticket.
- It does not check to ensure that the customer inserts sensible amounts of money. Experiment with what happens if a negative amount is entered, for instance.
- It does not check that the ticket price passed to its constructor is sensible.

If we could remedy these problems, then we would have a much more functional piece of software that might serve as the basis for operating a real-world ticket machine.

In the next few sections, we shall examine the implementation of an improved ticket machine class that attempts to deal with some of the inadequacies of the naïve implementation. Open the *better-ticket-machine* project. As before, this project contains a single class: **`TicketMachine`**. Before looking at the internal details of the class, experiment with it by creating some instances and see whether you notice any differences in behavior between this version and the previous naíve version.

One specific difference is that the new version has an additional method, **`refundBalance`**. Take a look at what happens when you call it.

Code 2.8 A more sophisticated **`TicketMachine`**

```
/**
 * TicketMachine models a ticket machine that issues
 * flat-fare tickets.
 * The price of a ticket is specified via the constructor.
 * Instances will check to ensure that a user only enters
 * sensible amounts of money, and will only print a ticket
 * if enough money has been input.
 *
```

Code 2.8 continued
A more sophisticated `TicketMachine`

```
 * @author David J. Barnes and Michael Kölling
 * @version 2016.02.29
 */
public class TicketMachine
{
    // The price of a ticket from this machine.
    private int price;
    // The amount of money entered by a customer so far.
    private int balance;
    // The total amount of money collected by this machine.
    private int total;

    /**
     * Create a machine that issues tickets of the given price.
     */
    public TicketMachine(int cost)
    {
        price = cost;
        balance = 0;
        total = 0;
    }

    /**
     * @Return The price of a ticket.
     */
    public int getPrice()
    {
        return price;
    }

    /**
     * Return The amount of money already inserted for the
     * next ticket.
     */
    public int getBalance()
    {
        return balance;
    }

    /**
     * Receive an amount of money from a customer.
     * Check that the amount is sensible.
     */
    public void insertMoney(int amount)
    {
        if(amount > 0) {
            balance = balance + amount;
        }
        else {
            System.out.println("Use a positive amount rather than: " +
                               amount);
        }
    }
```

Code 2.8 continued
A more sophisticated `TicketMachine`

```
    /**
     * Print a ticket if enough money has been inserted, and
     * reduce the current balance by the ticket price. Print
     * an error message if more money is required.
     */
    public void printTicket()
    {
        if(balance >= price) {
            // Simulate the printing of a ticket.
            System.out.println("##################");
            System.out.println("# The BlueJ Line");
            System.out.println("# Ticket");
            System.out.println("# " + price + " cents.");
            System.out.println("##################");
            System.out.println();

            // Update the total collected with the price.
            total = total + price;
            // Reduce the balance by the price.
            balance = balance - price;
        }
        else {
            System.out.println("You must insert at least: " +
                               (price - balance) + " more cents.");
        }
    }

    /**
     * Return the money in the balance.
     * The balance is cleared.
     */
    public int refundBalance()
    {
        int amountToRefund;
        amountToRefund = balance;
        balance = 0;
        return amountToRefund;
    }
}
```

2.13 Making choices: the conditional statement

Code 2.8 shows the internal details of the better ticket machine's class definition. Much of this definition will already be familiar to you from our discussion of the naíve ticket machine. For instance, the outer wrapping that names the class is the same, because we have chosen to give this class the same name. In addition, it contains the same three fields to maintain object state, and these have been declared in the same way. The constructor and the two **`get`** methods are also the same as before.

The first significant change can be seen in the **`insertMoney`** method. We recognized that the main problem with the naíve ticket machine was its failure to check certain conditions. One of

those missing checks was on the amount of money inserted by a customer, as it was possible for a negative amount of money to be inserted. We have remedied that failing by making use of a *conditional statement* to check that the amount inserted has a value greater than zero:

```
if(amount > 0) {
    balance = balance + amount;
}
else {
    System.out.println("Use a positive amount rather than: " +
                       amount);
}
```

Concept

A **conditional statement** takes one of two possible actions based upon the result of a test.

Conditional statements are also known as *if-statements*, from the keyword used in most programming languages to introduce them. A conditional statement allows us to take one of two possible actions based upon the result of a check or test. If the test is true, then we do one thing; otherwise, we do something different. This kind of either/or decision should be familiar from situations in everyday life: for instance, if I have enough money left, then I shall go out for a meal; otherwise, I shall stay home and watch a movie. A conditional statement has the general form described in the following *pseudo-code*:

```
if(perform some test that gives a true or false result) {
    Do the statements here if the test gave a true result
}
else {
    Do the statements here if the test gave a false result
}
```

Certain parts of this pseudo-code are proper bits of Java, and those will appear in almost all conditional statements–the keywords **`if`** and **`else`**, the round brackets around the test, and the curly brackets marking the two blocks–while the other three italicized parts will be fleshed out differently for each particular situation being coded.

Concept

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.

Only one of the two blocks of statements following the test will ever be performed following the evaluation of the test. So, in the example from the **`insertMoney`** method, following the test of an inserted amount we shall only either add the amount to the balance or print the error message. The test uses the *greater-than operator*, ">", to compare the value in **`amount`** against zero. If the value is greater than zero, then it is added to the balance. If it is not greater than zero, then an error message is printed. By using a conditional statement, we have, in effect, protected the change to **`balance`** in the case where the parameter does not represent a valid amount. Details of other Java operators can be found in Appendix C. The obvious ones to mention at this point are "<" (less-than), "<=" (less-than or equal-to), and ">=" (greater-than or equal-to). All are used to compare two numeric values, as in the **`printTicket`** method.

The test used in a conditional statement is an example of a *boolean expression.* Earlier in this chapter, we introduced arithmetic expressions that produced numerical results. A boolean expression has only two possible values (**`true`** or **`false`**): the value of **`amount`** is either greater than zero (**`true`**) or it is not greater (**`false`**). A conditional statement makes use of those two possible values to choose between two different actions.

Exercise 2.46 Check that the behavior we have discussed here is accurate by creating a **TicketMachine** instance and calling **insertMoney** with various actual parameter values. Check the balance both before and after calling **insertMoney**. Does the balance ever change in the cases when an error message is printed? Try to predict what will happen if you enter the value zero as the parameter, and then see if you are right.

Exercise 2.47 Predict what you think will happen if you change the test in **insertMoney** to use the *greater-than or equal-to operator:*

```
if(amount >= 0)
```

Check your predictions by running some tests. What is the one situation in which it makes a difference to the behavior of the method?

Exercise 2.48 Rewrite the if-else statement so that the method still behaves correctly but the error message is printed if the boolean expression is true but the balance is increased if the expression is false. You will obviously have to rewrite the condition to make things happen this way around.

Exercise 2.49 In the *figures* project we looked at in Chapter 1 we used a **boolean** field to control a feature of the circle objects. What was that feature? Was it well suited to being controlled by a type with only two different values?

2.14 A further conditional-statement example

The **printTicket** method contains a further example of a conditional statement. Here it is in outline:

```
if(balance >= price) {
    Printing details omitted.
    // Update the total collected with the price.
    total = total + price;
    // Reduce the balance by the price.
    balance = balance - price;
}
else {
    System.out.println("You must insert at least: " +
                       (price - balance) + " more cents.");
}
```

With this if-statement, we fix the problem that the naíve version makes no check that a customer has inserted enough money for a ticket before printing. This version checks that the value in the **balance** field is at least as large as the value in the **price** field. If it is, then it is okay to print a ticket. If it is not, then we print an error message instead.

The printing of the error message follows exactly the same pattern as we saw for printing the price of tickets in the **printTicket** method; it is just a little more verbose.

```
System.out.println("You must insert at least: " +
                   (price - balance) + " more cents.");
```

The single actual parameter to the **println** method consists of a concatenation of three elements: two string literals on either side of a numeric value. In this case, the numeric value is a subtraction that has been placed in parentheses to indicate that it is the resulting value we wish to concatenate with the two strings.

Exercise 2.50 In this version of **printTicket**, we also do something slightly different with the **total** and **balance** fields. Compare the implementation of the method in Code 2.1 with that in Code 2.8 to see whether you can tell what those differences are. Then check your understanding by experimenting within BlueJ.

Exercise 2.51 Is it possible to remove the else part of the if-statement in the **printTicket** method (i.e., remove the word **else** and the block attached to it)? Try doing this and seeing if the code still compiles. What happens now if you try to print a ticket without inserting any money?

The **printTicket** method reduces the value of **balance** by the value of **price**. As a consequence, if a customer inserts more money than the price of the ticket, then some money will be left in the balance that could be used toward the price of a second ticket. Alternatively, the customer could ask to be refunded the remaining balance, and that is what the **refundBalance** method does, as we shall see in section 2.16.

2.15 Scope highlighting

You will have noticed by now that the BlueJ editor displays source code with some additional decoration: colored boxes around some elements, such as methods and if-statements (see, for example, Code 2.8).

These colored annotations are known as *scope highlighting*, and they help clarify logical units of your program. A *scope* (also called a *block*) is a unit of code usually indicated by a pair of curly brackets. The whole body of a class is a scope, as is the body of each method and the *if* and *else* parts of an if-statement.

As you can see, scopes are often nested: the if-statement is inside a method, which is inside a class. BlueJ helps by distinguishing different kinds of scopes with different colors.

One of the most common errors in the code of beginning programmers is getting the curly brackets wrong—either by having them in the wrong place, or by having a bracket missing altogether. Two things can greatly help in avoiding this kind of error:

- Pay attention to proper indentation of your code. Every time a new scope starts (after an open curly bracket), indent the following code one level more. Closing the scope brings

the indentation back. If your indentation is completely out, use BlueJ's "Auto-layout" function (find it in the editor menu!) to fix it.

- Pay attention to the scope highlighting. You will quickly get used to the way well-structured code looks. Try removing a curly bracket in the editor or adding one at an arbitrary location, and observe how the coloring changes. Get used to recognizing when scopes look wrong.

Exercise 2.52 After a ticket has been printed, could the value in the `balance` field ever be set to a negative value by subtracting `price` from it? Justify your answer.

Exercise 2.53 So far, we have introduced you to two arithmetic operators, `+` and `-`, that can be used in arithmetic expressions in Java. Take a look at Appendix C to find out what other operators are available.

Exercise 2.54 Write an assignment statement that will store the result of multiplying two variables, `price` and `discount`, into a third variable, `saving`.

Exercise 2.55 Write an assignment statement that will divide the value in `total` by the value in `count` and store the result in `mean`.

Exercise 2.56 Write an if-statement that will compare the value in `price` against the value in `budget`. If `price` is greater than `budget`, then print the message "Too expensive"; otherwise print the message "Just right".

Exercise 2.57 Modify your answer to the previous exercise so that the message includes the value of your budget if the price is too high.

2.16 Local variables

So far, we have encountered two different sorts of variables: fields (instance variables) and parameters. We are now going to introduce a third kind. All have in common that they store data, but each sort of variable has a particular role to play.

Section 2.7 noted that a method body (or, in general, a *block*) can contain both declarations and statements. To this point, none of the methods we have looked at contain any declarations. The `refundBalance` method contains three statements and a single declaration. The declaration introduces a new kind of variable:

```
public int refundBalance()
{
    int amountToRefund;
    amountToRefund = balance;
    balance = 0;
    return amountToRefund;
}
```

What sort of variable is **amountToRefund**? We know that it is not a field, because fields are defined outside methods. It is also not a parameter, as those are always defined in the method header. The **amountToRefund** variable is what is known as a *local variable*, because it is defined *inside* a method body.

Concept

A **local variable** is a variable declared and used within a single method. Its scope and lifetime are limited to that of the method.

Local variable declarations look similar to field declarations, but they never have **private** or **public** as part of them. Constructors can also have local variables. Like formal parameters, local variables have a scope that is limited to the statements of the method to which they belong. Their lifetime is the time of the method execution: they are created when a method is called and destroyed when a method finishes.

You might wonder why there is a need for local variables if we have fields. Local variables are primarily used as temporary storage, to help a single method complete its task; we think of them as data storage for a single method. In contrast, fields are used to store data that persists through the life of a whole object. The data stored in fields is accessible to all of the object's methods. We try to avoid declaring as fields variables that really only have a local (method-level) usage, whose values don't have to be retained beyond a single method call. So even if two or more methods in the same class use local variables for a similar purpose, it is not appropriate to define them as fields if their values don't need to persist beyond the end of the methods.

In the **refundBalance** method, **amountToRefund** is used briefly to hold the value of the **balance** immediately prior to the latter being set to zero. The method then returns the remembered value of the balance. The following exercises will help to illustrate why a local variable is needed here, as we try to write the **refundBalance** method without one.

It is quite common to initialize local variables within their declaration. So we could abbreviate the first two statements of **refundBalance** as

```
int amountToRefund = balance;
```

but it is still important to keep in mind that there are two steps going on here: declaring the variable **amountToRefund**, and giving it an initial value.

Pitfall A local variable of the same name as a field will prevent the field being accessed from within a constructor or method. See Section 3.13.2 for a way around this when necessary.

Exercise 2.58 Why does the following version of **refundBalance** not give the same results as the original?

```
public int refundBalance()
{
    balance = 0;
    return balance;
}
```

What tests can you run to demonstrate that it does not?

Exercise 2.59 What happens if you try to compile the `TicketMachine` class with the following version of `refundBalance`?

```
public int refundBalance()
{
    return balance;
    balance = 0;
}
```

What do you know about return statements that helps to explain why this version does not compile?

Exercise 2.60 What is wrong with the following version of the constructor of `TicketMachine`?

```
public TicketMachine(int cost)
{
    int price = cost;
    balance = 0;
    total = 0;
}
```

Try out this version in the *better-ticket-machine* project. Does this version compile? Create an object and then inspect its fields. Do you notice something wrong about the value of the `price` field in the inspector with this version? Can you explain why this is?

2.17 Fields, parameters, and local variables

With the introduction of `amountToRefund` in the `refundBalance` method, we have now seen three different kinds of variables: fields, formal parameters, and local variables. It is important to understand the similarities and differences between these three kinds. Here is a summary of their features:

- All three kinds of variables are able to store a value that is appropriate to their defined types. For instance, a defined type of `int` allows a variable to store an integer value.
- Fields are defined outside constructors and methods.
- Fields are used to store data that persist throughout the life of an object. As such, they maintain the current state of an object. They have a lifetime that lasts as long as their object lasts.
- Fields have class scope: they are accessible throughout the whole class, so they can be used within any of the constructors or methods of the class in which they are defined.

- As long as they are defined as **`private`**, fields cannot be accessed from anywhere outside their defining class.
- Formal parameters and local variables persist only for the period that a constructor or method executes. Their lifetime is only as long as a single call, so their values are lost between calls. As such, they act as temporary rather than permanent storage locations.
- Formal parameters are defined in the header of a constructor or method. They receive their values from outside, being initialized by the actual parameter values that form part of the constructor or method call.
- Formal parameters have a scope that is limited to their defining constructor or method.
- Local variables are defined inside the body of a constructor or method. They can be initialized and used only within the body of their defining constructor or method. Local variables must be initialized before they are used in an expression—they are not given a default value.
- Local variables have a scope that is limited to the block in which they are defined. They are not accessible from anywhere outside that block.

New programmers often find it difficult to work out whether a variable should be defined as a field or as a local variable. Temptation is to define all variables as fields, because they can be accessed from anywhere in the class. In fact, the opposite approach is a much better rule to adopt: define variables local to a method unless they are clearly a genuine part of an object's persistent state. Even if you anticipate using the same variable in two or more methods, define a separate version locally to each method until you are absolutely sure that persistence between method calls is justified.

Exercise 2.61 Add a new method, **`emptyMachine`**, that is designed to simulate emptying the machine of money. It should reset **`total`** to be zero, but also return the value that was stored in **`total`** before it was reset.

Exercise 2.62 Rewrite the **`printTicket`** method so that it declares a local variable, **`amountLeftToPay`**. This should then be initialized to contain the difference between **`price`** and **`balance`**. Rewrite the test in the conditional statement to check the value of **`amountLeftToPay`**. If its value is less than or equal to zero, a ticket should be printed; otherwise, an error message should be printed stating the amount left to pay. Test your version to ensure that it behaves in exactly the same way as the original version. Make sure that you call the method more than once, when the machine is in different states, so that both parts of the conditional statement will be executed on separate occasions.

Exercise 2.63 *Challenge exercise* Suppose we wished a single `TicketMachine` object to be able to issue tickets of different prices. For instance, users might press a button on the physical machine to select a discounted ticket price. What further methods and/or fields would need to be added to `TicketMachine` to allow this kind of functionality? Do you think that many of the existing methods would need to be changed as well?

Save the *better-ticket-machine* project under a new name, and implement your changes in the new project.

2.18 Summary of the better ticket machine

In developing a better version of the `TicketMachine` class, we have been able to address the major inadequacies of the naïve version. In doing so, we have introduced two new language constructs: the conditional statement and local variables.

- A conditional statement gives us a means to perform a test and then, on the basis of the result of that test, perform one or the other of two distinct actions.
- Local variables allow us to calculate and store temporary values within a constructor or method. They contribute to the behavior that their defining method implements, but their values are lost once that constructor or method finishes its execution.

You can find more details of conditional statements and the form their tests can take in Appendix D.

2.19 Self-review exercises

This chapter has covered a lot of new ground, and we have introduced a lot of new concepts. We will be building on these in future chapters, so it is important that you are comfortable with them. Try the following pencil-and-paper exercises as a way of checking that you are becoming used to the terminology that we have introduced in this chapter. Don't be put off by the fact that we suggest that you do these on paper rather than within BlueJ. It will be good practice to try things out without a compiler.

Exercise 2.64 List the name and return type of this method:

```
public String getCode()
{
    return code;
}
```

Exercise 2.65 List the name of this method and the name and type of its parameter:

```
public void setCredits(int creditValue)
{
    credits = creditValue;
}
```

Exercise 2.66 Write out the outer wrapping of a class called **Person**. Remember to include the curly brackets that mark the start and end of the class body, but otherwise leave the body empty.

Exercise 2.67 Write out definitions for the following fields:

- a field called **name** of type **String**
- a field of type **int** called **age**
- a field of type **String** called **code**
- a field called **credits** of type **int**

Exercise 2.68 Write out a constructor for a class called **Module**. The constructor should take a single parameter of type **String** called **moduleCode**. The body of the constructor should assign the value of its parameter to a field called **code**. You don't have to include the definition for **code**, just the text of the constructor.

Exercise 2.69 Write out a constructor for a class called **Person**. The constructor should take two parameters. The first is of type **String** and is called **myName**. The second is of type **int** and is called **myAge**. The first parameter should be used to set the value of a field called **name**, and the second should set a field called **age**. You don't have to include the definitions for the fields, just the text of the constructor.

Exercise 2.70 Correct the error in this method:

```
public void getAge()
{
    return age;
}
```

Exercise 2.71 Write an accessor method called **getName** that returns the value of a field called **name**, whose type is **String**.

Exercise 2.72 Write a mutator method called **setAge** that takes a single parameter of type **int** and sets the value of a field called **age**.

Exercise 2.73 Write a method called **printDetails** for a class that has a field of type **String** called **name**. The **printDetails** method should print out a string of the form "The name of this person is", followed by the value of the **name** field. For instance, if the value of the **name** field is "Helen", then **printDetails** would print:

```
The name of this person is Helen
```

If you have managed to complete most or all of these exercises, then you might like to try creating a new project in BlueJ and making up your own class definition for a **Person**. The class could have fields to record the name and age of a person, for instance. If you were unsure how to complete any of the previous exercises, look back over earlier sections in this chapter and the source code of **TicketMachine** to revise what you were unclear about. In the next section, we provide some further review material.

2.20 Reviewing a familiar example

By this point in the chapter, you have met a lot of new concepts. To help reinforce them, we shall now revisit a few in a different but familiar context. Along the way, though, watch out for one or two further new concepts that we will then cover in more detail in later chapters!

Open the *lab-classes* project that we introduced in Chapter 1, and then examine the **Student** class in the editor (Code 2.9).

Code 2.9 The **Student** class

```
/**
 * The Student class represents a student in a student administration system.
 * It holds the student details relevant in our context.
 *
 * @author Michael Kölling and David Barnes
 * @version 2016.02.29
 */
public class Student
{
    // the student's full name
    private String name;
    // the student ID
    private String id;
    // the amount of credits for study taken so far
    private int credits;
```

Code 2.9 continued
The **Student** class

```
/**
 * Create a new student with a given name and ID number.
 */
public Student(String fullName, String studentID)
{
    name = fullName;
    id = studentID;
    credits = 0;
}

/**
 * Return the full name of this student.
 */
public String getName()
{
    return name;
}

/**
 * Set a new name for this student.
 */
public void changeName(String replacementName)
{
    name = replacementName;
}

/**
 * Return the student ID of this student.
 */
public String getStudentID()
{
    return id;
}

/**
 * Add some credit points to the student's accumulated credits.
 */
public void addCredits(int additionalPoints)
{
    credits += additionalPoints;
}

/**
 * Return the number of credit points this student has accumulated.
 */
public int getCredits()
{
    return credits;
}

/**
 * Return the login name of this student. The login name is a combination
 * of the first four characters of the student's name and the first three
 * characters of the student's ID number.
 */
```

Code 2.9 continued
The **Student** class

```
    public String getLoginName()
    {
        return name.substring(0,4) + id.substring(0,3);
    }

    /**
     * Print the student's name and ID number to the output terminal.
     */
    public void print()
    {
        System.out.println(name + ", student ID: " + id + ", credits: " + credits);
    }
}
```

In this small example, the pieces of information we wish to store for a student are their name, their student ID, and the number of course credits they have obtained so far. All of this information is persistent during their time as a student, even if some of it changes during that time (the number of credits). We want to store this information in fields, therefore, to represent each student's state.

The class contains three fields: **name**, **id**, and **credits**. Each of these is initialized in the single constructor. The initial values of the first two are set from parameter values passed into the constructor. Each of the fields has an associated **get** accessor method, but only **name** and **credits** have associated mutator methods. This means that the value of an **id** field remains fixed once the object has been constructed. If a field's value cannot be changed once initialized, we say that it is *immutable*. Sometimes we make the complete state of an object immutable once it has been constructed; the **String** class is an important example of this.

2.21 Calling methods

The **getLoginName** method illustrates a new feature that is worth exploring:

```
public String getLoginName()
{
    return name.substring(0,4) +
           id.substring(0,3);
}
```

We are seeing two things in action here:

- Calling a method on another object, where the method returns a result.
- Using the value returned as a result as part of an expression.

Both **name** and **id** are **String** objects, and the **String** class has a method, **substring**, with the following header:

```
/**
 * Return a new string containing the characters from
 * beginIndex to (endIndex-1) from this string.
 */
public String substring(int beginIndex, int endIndex)
```

An index value of zero represents the first character of a string, so **getLoginName** takes the first four characters of the **name** string and the first three characters of the **id** string, then concatenates them together to form a new string. This new string is returned as the method's result. For instance, if **name** is the string **"Leonardo da Vinci"** and **id** is the string **"468366"**, then the string **"Leon468"** would be returned by this method.

We will learn more about method calling between objects in Chapter 3.

Exercise 2.74 Draw a picture of the form shown in Figure 2.3, representing the initial state of a **Student** object following its construction, with the following actual parameter values:

```
new Student("Benjamin Jonson", "738321")
```

Exercise 2.75 What would be returned by **getLoginName** for a student with **name "Henry Moore"** and **id "557214"**?

Exercise 2.76 Create a **Student** with name **"djb"** and **id "859012"**. What happens when **getLoginName** is called on this student? Why do you think this is?

Exercise 2.77 The **String** class defines a **length** accessor method with the following header:

```
/**
 * Return the number of characters in this string.
 */
public int length()
```

so the following is an example of its use with the **String** variable **fullName**:

```
fullName.length()
```

Add conditional statements to the constructor of **Student** to print an error message if either the length of the **fullName** parameter is less than four characters, or the length of the **studentId** parameter is less than three characters. However, the constructor should still use those parameters to set the **name** and **id** fields, even if the error message is printed. *Hint:* Use if-statements of the following form (that is, having no **else** part) to print the error messages.

```
if(perform a test on one of the parameters) {
     Print an error message if the test gave a true result
}
```

See Appendix D for further details of the different types of if-statements, if necessary.

Exercise 2.78 *Challenge exercise* Modify the **getLoginName** method of **Student** so that it always generates a login name, even if either the **name** or the **id** field is not strictly long enough. For strings shorter than the required length, use the whole string.

2.22 Experimenting with expressions: the Code Pad

In the previous sections, we have seen various expressions to achieve various computations, such as the `total + price` calculation in the ticket machine and the `name.substring(0,4)` expression in the `Student` class.

In the remainder of this book, we shall encounter many more such operations, sometimes written with operator symbols (such as "+") and sometimes written as method calls (such as `substring`). When we encounter new operators and methods, it often helps to try out with different examples what they do.

The Code Pad, which we briefly used in Chapter 1, can help us experiment with Java expressions (Figure 2.5). Here, we can type in expressions, which will then be immediately evaluated and the results displayed. This is very helpful for trying out new operators and methods.

Exercise 2.79 Consider the following expressions. Try to predict their results, and then type them in the Code Pad to check your answers.

```
99 + 3
"cat" + "fish"
"cat" + 9
9 + 3 + "cat"
"cat" + 3 + 9
"catfish".substring(3,4)
"catfish".substring(3,8)
```

Did you learn anything you did not expect from the exercise? If so, what was it?

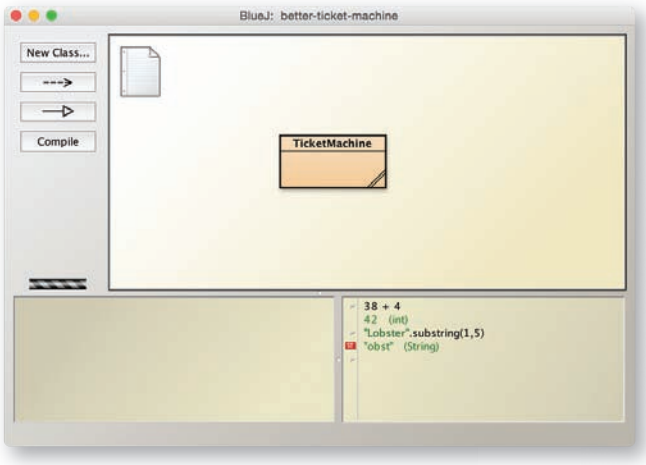


Figure 2.5
The BlueJ Code Pad

When the result of an expression in the Code Pad is an object (such as a **String**), it will be marked with a small red object symbol next to the line showing the result. You can double-click this symbol to inspect it or drag it onto the object bench for further use. You can also declare variables and write complete statements in the Code Pad.

Whenever you encounter new operators and method calls, it is a good idea to try them out here to get a feel for their behavior.

You can also explore the use of variables in the Code Pad. Try the following:

```
sum = 99 + 3;
```

You will see the following error message:

```
Error: cannot find symbol - variable sum
```

This is because Java requires that every variable (**sum**, in this case) be given a type before it can be used. Recall that every time a field, parameter, or local variable has been introduced for the first time in the source, it has had a type name in front of it, such as **int** or **String**. In light of this, now try the following in the Code Pad:

```
int sum = 0;

sum = 99 + 3;
```

This time there is no complaint, because **sum** has been introduced with a type and can be used without repeating the type thereafter. If you then type

```
sum
```

on a line by itself (with no semicolon), you will see the value it currently stores.

Now try this in the Code Pad:

```
String swimmer = "cat" + "fish";

swimmer
```

One again, we have given an appropriate type to the variable **swimmer**, allowing us to make an assignment to it and find out what it stores. This time we chose to set it to the value we wanted at the same time as declaring it.

What would you expect to see after the following?

```
String fish = swimmer;

fish
```

Try it out. What do you think has happened in the assignment?

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`**.