

### 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 naïve 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. A 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 only print tickets of 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 have 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 ticket.

### 2.1.1 Exploring the behavior of a naïve ticket machine

Open the *naive-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 and then use `getBalance` to check that the machine has a record of the amount inserted. You can insert several separate amounts of money into the machine, just like you might insert multiple coins or notes into a real machine. Try inserting the exact amount required for a ticket. 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 check 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 at how the `TicketMachine` class is implemented in the next section.

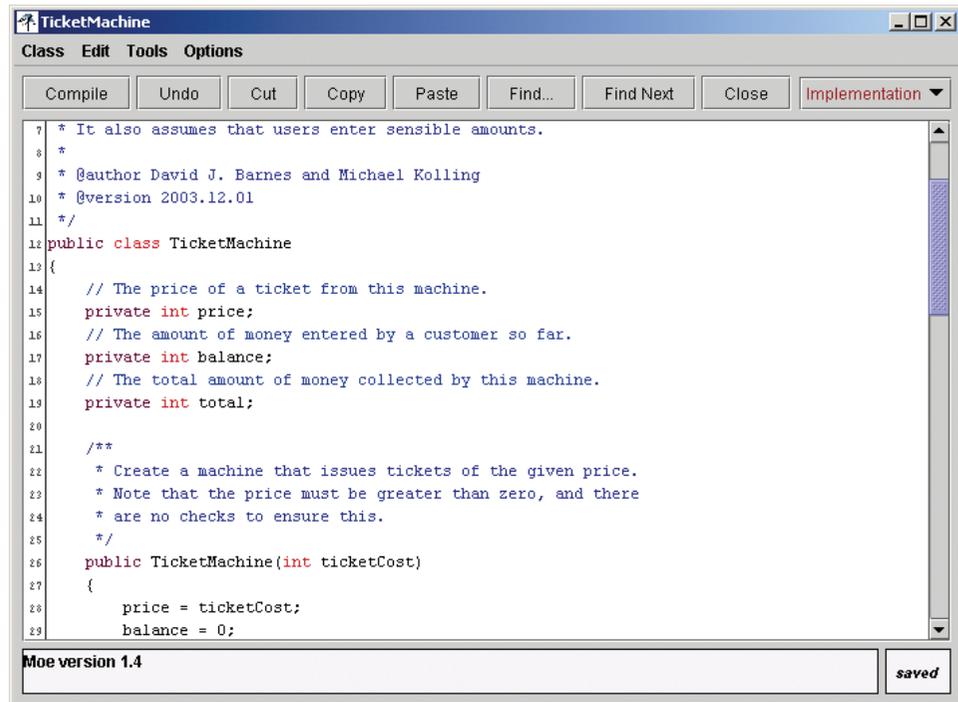
**Exercise 2.5** Create another ticket machine for tickets of a different price. Buy a ticket from that machine. Does the printed ticket look different?

## 2.2 Examining a class definition

Examination of the behavior of `TicketMachine` objects within BlueJ reveals that they only really behave in the way we might expect them to if we insert exactly the correct amount of money to match the price of a ticket. As we explore the internal details of the class in this section, we shall begin to 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. It should look something like Figure 2.1.

**Figure 2.1**  
The BlueJ editor window



```
7  * It also assumes that users enter sensible amounts.
8  *
9  * @author David J. Barnes and Michael Kolling
10 * @version 2003.12.01
11 */
12 public class TicketMachine
13 {
14     // The price of a ticket from this machine.
15     private int price;
16     // The amount of money entered by a customer so far.
17     private int balance;
18     // The total amount of money collected by this machine.
19     private int total;
20
21     /**
22      * Create a machine that issues tickets of the given price.
23      * Note that the price must be greater than zero, and there
24      * are no checks to ensure this.
25      */
26     public TicketMachine(int ticketCost)
27     {
28         price = ticketCost;
29         balance = 0;
30     }
31 }
```

Moe version 1.4 saved

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 we talked about in Chapter 1.

**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 2006.03.30
 */
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 ticketCost)
    {
        price = ticketCost;
        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;
    }
}
```

**Code 2.1**  
**continued**

The  
TicketMachine  
class

```

/**
 * Receive an amount of money in cents from a customer.
 */
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 Fields, constructors, and methods

The source of most classes can be broken down into two main parts: a small outer wrapping that simply names the class, 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 pretty much the same; their main purpose is to provide a name for 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?

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

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 for each object to use.
- The constructors allow each object to be set up properly when it is first created.
- The methods implement the behavior of the objects.

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 to 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 2.9** 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 try 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 2.10** Do you notice any features of the constructor that make it significantly different from the other methods of the class?

## 2.3.1 Fields

### Concept:

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

The `TicketMachine` class has three fields: `price`, `balance`, and `total`. Fields are also known as *instance variables*. We have defined these right at the start of the class definition (Code 2.3). All of the fields are associated with monetary items that a ticket machine object has to deal with:

- The `price` field stores the fixed price of a ticket.
- The `balance` field stores the amount of money inserted into the machine by a user prior to asking for a ticket to be printed.
- The `total` field stores a record of the total amount of money inserted into the machine by all users since the machine object was constructed.

### Code 2.3

The fields of the `TicketMachine` class

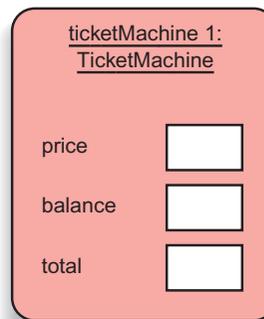
```
public class TicketMachine
{
    private int price;
    private int balance;
    private int total;

    Constructor and methods omitted.
}
```

Fields are small amounts of space inside an object that can be used to store values. Every object, once created, will have some space for every 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.

**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 multi-line 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 5, but for the time being we will simply say that we always define fields to be private.
- All three fields are of type `int`. 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.

Because fields can store values that can vary over time, they are also known as *variables*. The value stored in a field can be changed if we wish to. For instance, as more money is inserted into a ticket machine we shall want to change the value stored in the `balance` field. In the following sections we shall also meet other kinds of variables in addition to fields.

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 naïve ticket machines.

**Exercise 2.11** What do you think is the *type* of each of the following fields?

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

**Exercise 2.12** What are the *names* of the following fields?

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

**Exercise 2.13** 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 2.14** 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 2.15** Write in full the declaration for a field of type `int` whose name is `status`.

## 2.3.2 Constructors

### Concept:

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

The constructors of a class have a special role to fulfill. It is their responsibility to put each object of that class into a fit state to be used once it has been created. This is also called *initialization*. The constructor initializes the object to a reasonable state. 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.

### 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 ticketCost)
    {
        price = ticketCost;
        balance = 0;
        total = 0;
    }

    Methods omitted.
}
```

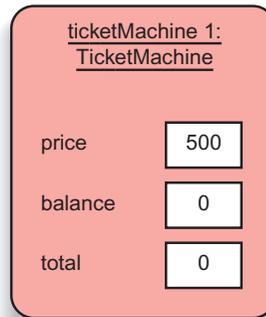
The fields of the object are initialized in the constructor. Some fields, such as `balance` and `total`, can be set to sensible initial values by assigning a constant number, zero in this case. With others, such as the ticket price, it is not that simple, as we do not know the price that tickets from 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 recall 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 *outside* the ticket machine, and then has to be *passed into* a ticket machine object. Within BlueJ you decide the value and enter it into a dialog box. One task of the constructor is to receive that value and store it in the `price` field of the newly created ticket machine so that the machine can remember what that value was without you having to keep reminding it. We can see from this that one of the most important roles of a field is to remember information, so that it is available to an object throughout that object's lifetime.

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.

**Figure 2.3**

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



**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 0. So, strictly speaking, we could have done without setting `balance` and `total` to 0, 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.

## 2.4 Passing data via parameters

The way in which both constructors and methods receive values is via their *parameters*. You may recall that we briefly encountered parameters in Chapter 1. Parameters are defined in the header of the constructor or method:

```
public TicketMachine(int ticketCost)
```

This constructor has a single parameter, `ticketCost`, which is of type `int` – the same type as the `price` field it will be used to set. Figure 2.4 illustrates how values are passed via parameters. In this case, a BlueJ user enters a value into the dialog box when creating a new ticket machine (shown on the left), and that value is then copied into the `ticketCost` parameter of the new machine’s constructor. This is illustrated with the arrow labeled (A). The box in the ticket machine object in Figure 2.4, labeled ‘TicketMachine (constructor),’ is 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 (and other variables that we will come across later).

We distinguish between parameter names inside a constructor or method, and parameter values outside, by referring to the names as *formal parameters* and the values as *actual parameters*. So `ticketCost` is a formal parameter, and a user-supplied value, such as 500, is an actual parameter. Because they are able to store values, formal parameters are another sort of variable. In our diagrams, all variables are represented by white boxes.

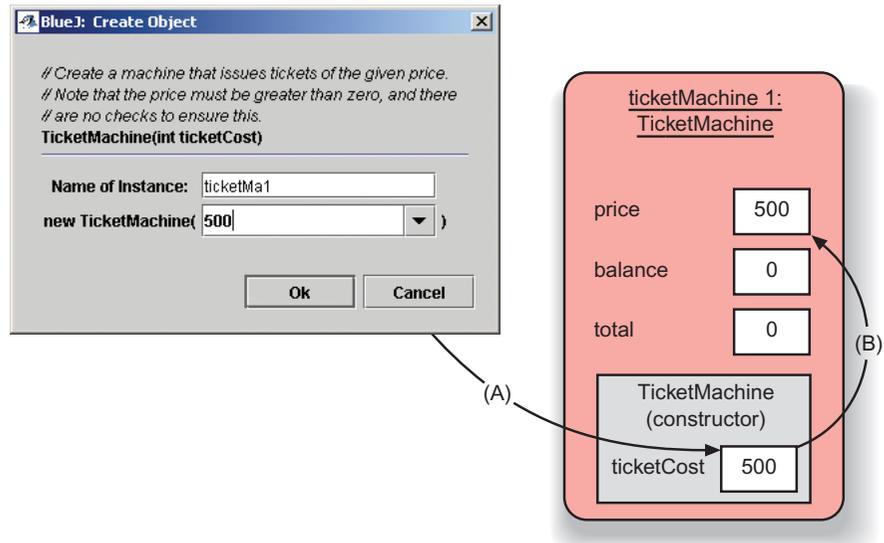
A formal parameter is available to an object only within the body of a constructor or method that declares it. We say that the *scope* of a parameter is restricted to the body of the constructor or method in which it is declared. In contrast, the scope of a field is the whole of the class definition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

### Concept:

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

**Figure 2.4**

Parameter passing  
(A) and assignment  
(B)

**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. 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 (see Figure 2.4) is removed, along with the parameter variables held within it.

In contrast, the lifetime of a field is the same as the lifetime of the object to which it belongs. It follows that if we want to remember the cost of tickets held in the `ticketCost` parameter, we must store the value somewhere more persistent – that is, in the `price` field.

**Exercise 2.16** To what class does the following constructor belong?

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

**Exercise 2.17** How many parameters does the following constructor have and what are their types?

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

**Exercise 2.18** Can you guess what types some of the `Book` class's fields might be? Can you assume anything about the names of its fields?

## 2.5 Assignment

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

```
price = ticketCost;
```

**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 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 a 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*: expressions are things that compute values. In this case, the expression consists of just a single variable but we shall see some examples of more complicated expressions containing arithmetic operations later in this chapter. One rule about assignment statements is that the type of the expression must match the type of the variable to which it is assigned. So far we have met three different types: `int`, `String`, and (very briefly) `boolean`. This rule means that we are not allowed to store an integer-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 2.19** Suppose that the class `Pet` has a field called `name` that is of 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 2.20** *Challenge exercise* What is wrong with the following version of the constructor of `TicketMachine`?

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

Once you have spotted the problem, try out this version in the *naive-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?

## 2.6 Accessor methods

The `TicketMachine` class has four methods: `getPrice`, `getBalance`, `insertMoney`, and `printTicket`. 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`:

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

The first three lines are a comment describing what the method does. The fourth line is also known as the *method signature*.<sup>1</sup> It is important to distinguish between method signatures and field declarations, because they can look quite similar. We can tell that `getPrice` is a method and not a field because it is followed by a pair of parentheses: ‘(’ and ‘)’. Note, too, that there is no semicolon at the end of the signature.

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 happens inside an object when that method is called. In our example above, the method body contains a single statement, but we shall see examples very soon 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 and the bodies of all the methods within the class are blocks.

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

```
public TicketMachine(int ticketCost)

public int getPrice()
```

<sup>1</sup> This definition differs slightly from the more formal definition in the Java language specification where the signature does not include the access modifier and return type.

- The method has a *return type* of `int`, but the constructor has no return type. A return type is written just before the method name.
- The constructor has a single formal parameter, `ticketCost`, but the method has none – just a pair of empty parentheses.

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 a *return statement*. It is responsible for returning an integer value to match the `int` return type in the method's signature. 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.

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.

#### Concept:

**Accessor methods** return information about the state of an object.

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 that state. An accessor usually contains a return statement in order to pass back that information as a particular value.

**Exercise 2.21** Compare the `getBalance` method with the `getPrice` method. What are the differences between them?

**Exercise 2.22** If a call to `getPrice` can be characterized as 'What do tickets cost?', how would you characterize a call to `getBalance`?

**Exercise 2.23** If the name of `getBalance` is changed to `getAmount`, does the return statement in the body of the method need to be changed, too? Try it out within `BlueJ`.

**Exercise 2.24** Define an accessor method, `getTotal`, that returns the value of the `total` field.

**Exercise 2.25** Try removing the return statement from the body of `getPrice`. What error message do you see now when you try compiling the class?

**Exercise 2.26** Compare the method signatures of `getPrice` and `printTicket` in Code 2.1. Apart from their names, what is the main difference between them?

**Exercise 2.27** 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?

## 2.7

## Mutator methods

### Concept:

**Mutator methods** change the state of an object.

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 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 accessors as requests for information (questions), we can think of mutators as requests for an object to change its state.

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 2.28** 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 output because the call to `insertMoney` had the effect of changing the machine’s state via its `balance` field.

The signature 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.<sup>2</sup>

### Code 2.6

The `insertMoney` method

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

<sup>2</sup> 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.

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`.<sup>3</sup>

**Exercise 2.29** How can we tell from just its header that `setPrice` is a method and not a constructor?

```
public void setPrice(int ticketCost)
```

**Exercise 2.30** Complete the body of the `setPrice` method so that it assigns the value of its parameter to the `price` field.

**Exercise 2.31** 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)
{
    ...
}
```

**Exercise 2.32** Can you 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)
{
    ...
}
```

## 2.8 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

<sup>3</sup> 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;
```

**Code 2.7**

The `printTicket` method

```
/**
 * Print a ticket and reduce the
 * current 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;
}
```

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 signature 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.
- 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, ready for the next customer to insert some money.

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. All of these printing statements are calls to the `println` method of the `System.out` object that is built into the Java language. In the fourth statement the actual parameter to `println` is a little more complicated:

**Concept:**

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

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

The two '+' operators are being used to construct a single string parameter from three components:

- the string literal: "# " (note the space character after the hash);
- the value of the `price` field (note there are no quotes around the field 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.

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 2.33** 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 something like:

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

**Exercise 2.34** 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 something like:

```
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 2.35** 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?

**Exercise 2.36** 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 2.37** What about the following version?

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

**Exercise 2.38** Could either of the previous two versions be used to show the price of tickets in different ticket machines? Explain your answer.

## 2.9

### Summary of the naïve ticket machine

We have now examined the internal structure of the naïve ticket machine 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. 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. Accessors provide information about an object's state, and mutators 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 result out of a method. A method with a non-void return type will have a return statement as the final statement of its body. Return statements are only applicable to methods, because 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 2.39** Modify the constructor of `TicketMachine` so that it no longer has a parameter. Instead, the price of tickets should be fixed at 1000 cents. What effect does this have when you construct ticket machine objects within BlueJ?

**Exercise 2.40** 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 this method a mutator or an accessor?

**Exercise 2.41** Implement a method, `setPrice`, that is able to set the price of tickets to a new value. The new price is passed in as a parameter value to the method. Test your method by creating a machine, showing the price of tickets, changing the price, and then showing the new price. Is this method a mutator?

**Exercise 2.42** 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.

## 2.10

### Reflecting on the design of the 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.

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 order to see that we can improve the existing version, 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 naïve version. One specific difference is that the new version has one additional method, `refundBalance`. Later in this chapter we shall use this method to introduce an additional feature of Java, so take a look at what happens when you call it.

## 2.11 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 naïve 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.

### Code 2.8

A more sophisticated ticket machine

```
/**
 * 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.
 * @author David J. Barnes and Michael Kölling
 * @version 2006.03.30
 */
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;
```

**Code 2.8**  
**continued**

A more sophisticated ticket machine

```
/**
 * Create a machine that issues tickets of the given price.
 */
public TicketMachine(int ticketCost)
{
    price = ticketCost;
    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 in cents 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: " +
            amount);
    }
}

/**
 * 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()
{
```

**Code 2.8****continued**

A more sophisticated ticket machine

```

        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) + " cents.");
        }
    }

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

```

The first significant change can be seen in the `insertMoney` method. We recognized that the main problem with the naïve 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: " + 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 word 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. 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
}
```

It is important to appreciate that only one of the sets 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.

### 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.

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`: either the value of `amount` is 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 2.43** 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 2.44** 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 difference does it make to the behavior of the method?

**Exercise 2.45** In the *shapes* 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?

## 2.12 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.");  
}
```

We wish to remedy the fact that the naïve version makes no check that a customer has inserted enough money to be issued with a ticket. 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.

**Exercise 2.46** 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.

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 towards 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 the next section.

**Exercise 2.47** 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 2.48** 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 D to find out what other operators are available.

**Exercise 2.49** Write an assignment statement that will store the result of multiplying two variables, `price` and `discount`, into a third variable, `saving`.

**Exercise 2.50** Write an assignment statement that will divide the value in `total` by the value in `count` and store the result in `mean`.

**Exercise 2.51** 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 2.52** Modify your answer to the previous exercise so that the message if the price is too high includes the value of your budget.

## 2.13 Local variables

The `refundBalance` method contains three statements and a declaration. The declaration illustrates a new sort of variable:

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

### 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.

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. 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;
```

Local variable declarations look similar to field declarations, but they never have `private` or `public` as part of them. 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. Constructors can also have local variables.

Local variables are often used as temporary storage locations to help a method complete its task. In this method `amountToRefund` is used to hold the value of the balance immediately prior to the latter being set to zero. The method then returns the old 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.

**Exercise 2.53** 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 2.54** 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?

Now that you have seen how local variables are used, look back at Exercise 2.20 and check that you understand how, there, a local variable is preventing a field of the same name from being accessed.

**Pitfall** A local variable of the same name as a field will prevent the field being accessed from within a method. See Section 3.12.2 for a way around this when necessary.

## 2.14 Fields, parameters, and local variables

With the introduction of `amountToRefund` in the `refundBalance` method we have now seen three different kinds of variable: 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 variable are able to store a value that is appropriate to their defined type. 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 persists 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: their accessibility extends throughout the whole class, and 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.

**Exercise 2.55** Add a new method, `emptyMachine`, that is designed to simulate emptying the machine of money. It should both return the value in `total` and reset `total` to be zero.

**Exercise 2.56** Is `emptyMachine` an accessor, a mutator, or both?

**Exercise 2.57** 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 still required. Test your version to ensure that it behaves in exactly the same way as the original version.

**Exercise 2.58** *Challenge exercise* Suppose we wished a single `TicketMachine` object to be able to issue tickets with different prices. For instance, users might press a button on the physical machine to select a particular 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 to the new project.

## 2.15

### Summary of the better ticket machine

In developing a more sophisticated 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 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 that their tests can take in Appendix C.

## 2.16

### 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 you do these on paper rather than within BlueJ. It will be good practice to try things out without a compiler.

**Exercise 2.59** List the name and return type of this method:

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

**Exercise 2.60** List the name of this method and the name and type of its parameter:

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

**Exercise 2.61** 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 2.62** 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 2.63** 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 2.64** 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 fields, just the text of the constructor.

**Exercise 2.65** Correct the error in this method:

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

**Exercise 2.66** Write an accessor method called **getName** that returns the value of a field called **name**, whose type is **String**.

**Exercise 2.67** Write a mutator method called **setAge** that takes a single parameter of type **int** and sets the value of a field called **age**.

**Exercise 2.68** 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 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.

## 2.17

## Reviewing a familiar example

By this point in the chapter you have met a lot of new concepts. To help reinforce those concepts, we shall now revisit them in a different but familiar context. 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 2006.03.30
 */
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;

    /**
     * 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;
    }
}
```

**Code 2.9****continued**

The Student class

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

/**
 * 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 newCreditPoints)
{
    credits += newCreditPoints;
}

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

/**
 * Print the student's name and ID number
 * to the output terminal.
 */
```

**Code 2.9**  
**continued**The `Student` class

```
public void print()
{
    System.out.println(name + " (" + id + ")");
}
}
```

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.

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

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

Both `name` and `id` are strings, and the `String` class has an accessor method, `substring`, with the following signature:

```
/**
 * 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, the first three characters of the `id` string, and concatenates them 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.

**Exercise 2.69** 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 2.70** What would be returned by `getLoginName` for a student with the name "Henry Moore" and the `id` "557214"?

**Exercise 2.71** 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 2.72** The `String` class defines a `length` accessor method with the following signature:

```
/**
 * Return the number of characters in this string.
 */
public int 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 C for further details of the different types of `if` statement, if necessary.

**Exercise 2.73** *Challenge exercise* Modify the `getLoginName` method of `Student` so that it always generates a login name, even if either of the `name` and `id` fields is not strictly long enough. For strings shorter than the required length, use the whole string.

## 2.18

## 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 constructors and methods a sequence of statements defines how an object accomplishes its designated tasks. We have covered assignment statements and conditional statements, and will be adding further types of statement in later chapters.

### Terms introduced in this chapter

**field, instance variable, constructor, method, method signature, method body, 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

- **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 where the variable can be accessed.

- **lifetime** The lifetime of a variable describes how long the variable continues to exist before it is destroyed.
- **assignment** Assignment statements store the value represented by the right-hand side of the statement in the variable named on the left.
- **method** Methods consist of two parts: a header and a body.
- **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** 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 2.74** 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 exercise and the next few, you will add further 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 2.75** 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 2.76** Add a further 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 2.77** 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 2.78** 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 signature:

```
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 2.79** 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, 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 2.80** 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 2.81** 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 2.82** *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 integer field, **temperature**. Define a constructor that takes no parameters. The **temperature** field should be set to the value 15 in the constructor. Define the mutators **warmer** and **cooler**, whose effect is to increase or decrease the value of temperature by 5° respectively. Define an accessor method to return the value of **temperature**.

**Exercise 2.83** *Challenge exercise* Modify your **Heater** class to define three new integer 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 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. 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 integer parameter 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**.