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

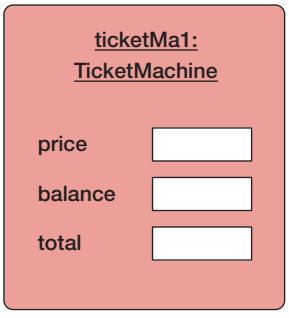
```
public class TicketMachine
{
    private int price;
    private int balance;
    private int total;

    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.