Implementing Classes

8

Outcomes:

By the end of this chapter you should be able to:

- *design classes using the notation of the* ***Unified Modeling Language (UML)***;
- *write the Java code for a specified class*;
- *explain the difference between* **`public`** *and* **`private`** *access to attributes and methods*;
- *explain the meaning of the term* ***encapsulation***;
- *explain the use of the* **`static`** *keyword*;
- *pass objects as parameters*;
- *develop their own* ***collection classes*** *in Java*;
- *identify the advantages of object-oriented programming*.

8.1 Introduction

This chapter is arguably the most important so far, because it is here that you are going to learn how to develop the classes that you need for your programs. You are already familiar with the concept of a class, and the idea that we can create objects that belong to a class; in the last chapter you saw how to create and use objects, and you saw how we could use the methods of a class without knowing anything about how they work.

In this chapter you will look inside the classes you have studied to see how they are constructed, and how you can write classes of your own. We start with the `Oblong` class.

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_8

8.2 Designing Classes in UML Notation

In the last chapter you saw that a class consists of:

- a set of **attributes** (the data);
- a set of **methods** that can access or change those attributes.

When we design a class we must, of course, consider what data the class needs to hold, and what methods are needed to access that data. The `Oblong` class that we develop here will need to hold two items of data—the length and the height of the oblong; these will have to be real numbers, so **`double`** would be the appropriate type for each of these two attributes. You have already seen the methods that we provided for this class in Table 7.1 in the previous chapter.

When we design classes, it is very useful to start off by using a diagrammatic notation. The usual way this is done is by making use of the notation of the **Unified Modeling Language (UML)**.[1] In this notation, a class is represented by a box divided into three sections. The first section provides the name of the class, the second section lists the attributes, and the third section lists the methods. The UML class diagram for the `Oblong` class is shown in Fig. 8.1.

You can see that the UML notation requires us to indicate the names of the attributes along with their types, separated by a colon.

In the last chapter we introduced you to the concept of *encapsulation* or information-hiding. This is the technique of making attributes accessible only to the methods of the same class, and it is this feature of object-oriented languages that has contributed to object-orientation becoming the standard way of programming in today's world. By restricting access in this way, programmers can keep the data in their classes "sealed off" from other classes, because they are the ones in control of how it is actually accessed.

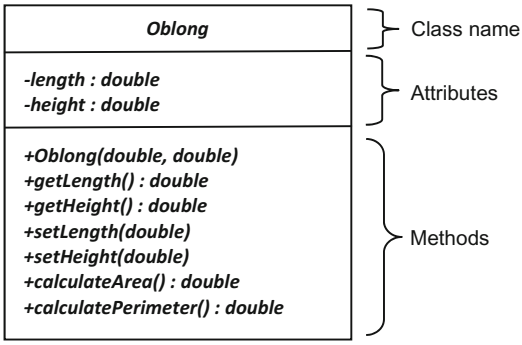


Fig. 8.1 The design of the *Oblong* class

[1] Martina Seidl et al., *UML @Classroom, An Introduction to Object Oriented Modeling*, Springer 2015.

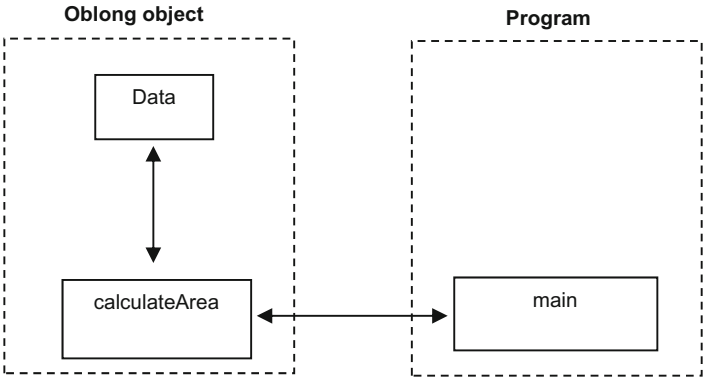


Fig. 8.2 Encapsulation requires data be kept hidden inside an object

The way our `Oblong` class has been set up means that you cannot directly use the `length` and `height` attributes in another program. If you want to find out the area of the oblong in, say, the `main` method of another program then you can't do this by accessing the `length` and `height` data directly, because access to these attributes is denied.

Instead we would, as you know, call the `calculateArea` method of the `Oblong` object. We design our classes like this because doing so means that no-one can inadvertently *change* the values of `length` and `height`—our data is kept secure. If access to these attributes were not restricted in this way, then the `length` and `height` data could inadvertently be changed. Instead we limit access of the `Oblong` class to its methods. This is illustrated in Fig. 8.2.

The plus and minus signs that you can see in the UML diagram in Fig. 8.1 are all to do with this idea of encapsulation; a minus sign means that the attribute or method is **private**—that is, it is accessible *only to methods within the same class*. A plus sign means that it is **public**—it is accessible to methods of other classes. Normally we make the attributes private, and the methods public, in this way achieving encapsulation. You will see how it is done in a Java class in the next section.

Now let's consider the notation for the methods. You can see from the diagram that the parameter types are given in brackets—for example:

+setLength(double)

Here you can see that the `setLength` method requires one **`double`** parameter (in this case the new length value). The return types are placed after the brackets, preceded by a colon—for example:

+getLength() : double

Here you can see that the `getLength` method returns a value of type **`double`** (in this case the current value of the private `length` attribute).

Where there is no return type, nothing appears after the brackets, as in the `setlength` and `setHeight` methods.

The first method, `Oblong`, is the constructor. As we know the constructor always has the same name as the class, and in this case it requires two parameters of type **`double`**:

+Oblong(double, double)

You should note that a constructor *never has a return type*. In fact you will see later that in Java we don't even put the word **`void`** in front of a constructor; if we did the compiler would think it was a regular method.

As you saw in the previous chapter, we have provided our `Oblong` class with methods for reading and writing to the attributes—and it is conventional to begin the name of such methods with `get-` and `set-`respectively. However, it is not always the case that we choose to supply methods such as `setLength` and `setHeight`, which allow us to *change* the attributes. Sometimes we set up our class so that the only way that we can assign values to the attributes is via the constructor. This would mean that the values of the `length` and `height` could be set only at the time a new `Oblong` object was created, and could not be changed after that. Whether or not you want to provide a means of writing to individual attributes depends on the nature of the system you are developing and should be discussed with potential users. However, we believe that it is a good policy to provide write access to only those attributes that clearly require to be changed during the object's lifetime, and we have taken this approach throughout this book. In this case we have included "set" methods for `length` and `height` because we are going to need them in Chap. 10.

8.3 Implementing Classes in Java

8.3.1 The *Oblong* Class

Now that we have the basic design of the `Oblong` class we can go ahead and write the Java code for it. We present the code here—when you have had a look at it we will discuss it.

The *Oblong* class

```
public class Oblong
{
    // the attributes
    private double length;
    private double height;

    // the methods

    // the constructor
    public Oblong(double lengthIn, double heightIn)
    {
        length = lengthIn;
        height = heightIn;
    }

    // this method allows us to read the length attribute
    public double getLength()
    {
        return length;
    }

    // this method allows us to read the height attribute
    public double getHeight()
    {
        return height;
    }

    // this method allows us to write to the length attribute
    public void setLength(double lengthIn)
    {
        length = lengthIn;
    }

    // this method allows us to write to the height attribute
    public void setHeight(double heightIn)
    {
        height = heightIn;
    }

    // this method returns the area of the Oblong
    public double calculateArea()
    {
        return length * height;
    }

    // this method returns the perimeter of the Oblong
    public double calculatePerimeter()
    {
        return 2 * (length + height);
    }
}
```

Let's take a closer look at this. The first line declares the `Oblong` class:

```
public class Oblong
```

Next come the attributes. An `Oblong` object will need attributes to hold values for the length and the height of the oblong, and these will be of type **`double`**. The declaration of the attributes in the `Oblong` class took the following form in our UML diagram:

-length : double
-height : double

In Java this is implemented as:

```
private double length;
private double height;
```

As you can see, attributes are declared like any other variables, except that they are declared *outside* of any method, and they also have an additional word in front of them—the word **`private`**, corresponding to the minus sign in the UML notation. In Java, this keyword is used to restrict the scope of the attributes to methods of this class only, as we described above.

You should note that the attributes of a class are accessible to *all* the methods of the class—unlike *local* variables, which are accessible only to the methods in which they are declared.

Figure 8.1 made it clear which methods we need to define within our `Oblong` class. First comes the constructor. You should recall that it has the same name as the class, and, unlike any other method, it has no return type—not even **`void`**! It looks like this.

```
public Oblong(double lengthIn, double heightIn)
{
    length = lengthIn;
    height = heightIn;
}
```

The first thing to notice is that this method is declared as **`public`**. Unlike the attributes, we want our methods to be accessible from outside so that they can be called by methods of other classes.

In our class we are defining the constructor so that when a new `Oblong` object is created (with the keyword **`new`**) then not only do we get some space reserved in memory, but some other stuff occurs also; in this case two assignment statements are executed. The first assigns the value of the parameter `lengthIn` to the `length` attribute, and the second assigns the value of the parameter `heightIn` to the `height` attribute. We are sticking to our naming convention for attributes here by appending the word 'In' to them, but of course you can use any names for your parameters. Remember once again that the attributes are visible to *all* the methods of the class.

When we define a constructor like this in a class it is termed a *user-defined*[2] constructor. If we don't define our own constructor, then one is automatically provided for us—this is referred to as the **default** constructor. The default constructor takes no parameters and when it is used to create an object—for example in a line like this:

```
Oblong myOblong = new Oblong();
```

[2]Here the word *user* is referring to the person *writing* the program, not the person using it!

then all that happens is that memory is reserved for the new object—no other processing takes place. Any attributes will be given initial values according to the rules that we give you later in Sect. 8.5.

One more thing about constructors: once we have defined our own constructors, this default constructor is no longer automatically available. If we want it to be available then we have to re-define it explicitly. In the `Oblong` case we would define it as:

```
public Oblong()
{
}
```

You can see that just like regular methods, constructers can be overloaded, and we can define several constructors in one class. When we create an object it will be clear from the parameter list which constructor we are referring to.

Now let's take a look at the definition of the next method, `getLength`. The purpose of this method is simply to send back the value of the `length` attribute. In the UML diagram it was declared as:

+getLength() : double

In Java this becomes:

```
public double getLength()
{
    return length;
}
```

Once again you can see that the method has been declared as **`public`** (indicated by the plus sign in UML), enabling it to be accessed by methods of other classes.

The next method, `getHeight`, behaves in the same way in respect of the `height` attribute.

Next comes the `setLength` method:

+setLength(double)

We implement this as:

```
public void setLength(double lengthIn)
{
    length = lengthIn;
}
```

This method does not return a value, so its return type is **`void`**. However, it does require a parameter of type **`double`** that it will assign to the `length` attribute. The body of the method consists of a single line which assigns the value of `lengthIn` to the `length` attribute.

The next method, `setHeight`, behaves in the same way in respect of the `height` attribute.

After this comes the `calculateArea` method:

+calculateArea() : double

We implement this as:

```
public double calculateArea()
{
    return length * height;
}
```

Once again there are no formal parameters, as this method does not need any data in order to do its job; it returns a **`double`**. The actual code is just one line, namely the statement that returns the area of the oblong, calculated by multiplying the value of the `length` attribute by the value of the `height` attribute.

The `calculatePerimeter` method is similar and thus the definition of the `Oblong` class is now complete.

One important thing to note here. Unlike some of the methods we developed in Chap. 5, the methods that we have defined here deal only with the basic *functionality* of the class—they do not include any routines that deal with input or output. That is because the methods of Chap. 5 were only being used by the class in which they were written—but now our methods will be used by other classes that we cannot as yet predict. So when developing a class we should always strive to restrict our methods to the essential functions that define the class (in this case, for example, calculating the area and perimeter of the oblong), and to exclude anything that is concerned with the input or output functions of a program. If we do this, then our class can be used in any sort of application, regardless of whether it is a simple console application like the ones we have developed so far, or a complex graphical application like the ones you will come across later in this book.

8.3.2 The *BankAccount* Class

The UML class diagram for the `BankAccount` class, which we used in the previous chapter, is shown in Fig. 8.3.

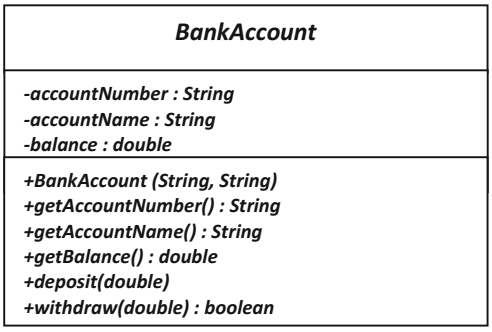


Fig. 8.3 The design of *BankAccount* class

You will notice here that *accountNumber* and *accountName* are declared as *Strings*; it is perfectly possible for the attributes of one class to be objects of another class.

We can now inspect the code for this class:

The *BankAccount* class

```
public class BankAccount
{
    // the attributes
    private String accountNumber;
    private String accountName;
    private double balance;

    // the methods

    // the constructor
    public BankAccount(String numberIn, String nameIn)
    {
        accountNumber = numberIn;
        accountName = nameIn;
        balance = 0;
    }

    // methods to read the attributes
    public String getAccountName()
    {
        return accountName;
    }

    public String getAccountNumber()
    {
        return accountNumber;
    }

    public double getBalance()
    {
        return balance;
    }

    // methods to deposit and withdraw money
    public void deposit(double amountIn)
    {
        balance = balance + amountIn;
    }

    public boolean withdraw(double amountIn)
    {
        if(amountIn > balance)
        {
            return false; // no withdrawal was made
        }
        else
        {
            balance = balance - amountIn;
            return true; // money was withdrawn successfully
        }
    }
}
```

Now that we are getting the idea of how to define a class in Java, we do not need to go into so much detail in our analysis and explanation.

The first three lines declare the attributes of the class, and are as we would expect:

```
private String accountNumber;
private String accountName;
private double balance;
```

Now the constructor:

```
public BankAccount(String numberIn, String nameIn)
{
    accountNumber = numberIn;
    accountName = nameIn;
    balance = 0;
}
```

You can see that when a new object of the `BankAccount` class is created, the `accountName` and `accountNumber` will be assigned the values of the parameters passed to the method. In this case, the `balance` will be assigned the value zero; this makes sense because when someone opens a new account there is a zero balance until a deposit is made.[3]

The next three methods, `getAccountNumber`, `getAccountName` and `getBalance`, are all set up so that the values of the corresponding attributes (which of course have been declared as **`private`**) can be read.

After these we have the `deposit` method:

```
public void deposit(double amountIn)
{
    balance = balance + amountIn;
}
```

This method does not return a value; it is therefore declared to be of type **`void`**. It does however require that a value is sent in (the amount to be deposited), and therefore has one parameter—of type **`double`**—in the brackets. As you would expect with this method, the action consists of adding the deposit to the `balance` attribute of the `BankAccount` object.

Now the `withdraw` method:

```
public boolean withdraw(double amountIn)
{
       if(amountIn > balance)
       {
           return false; // no withdrawal was made
       }
       else
       {
           balance = balance - amountIn;
           return true; // money was withdrawn successfully
       }
 }
```

[3]You would be right in thinking that the balance attribute would automatically be assigned a value of zero if we did not specifically do that here. However it is good practice always to ensure that variables are initialized with the values that we require—particularly because in many other programming languages attributes are not initialized as they are in Java.

The amount is subtracted only if there are sufficient funds—in other words if the amount to be withdrawn is no bigger than the balance. If this is not the case then a value of **`false`** is returned and the method terminates. Otherwise the amount is subtracted from the balance and a value of **`true`** is returned. The return type of the method therefore is **`boolean`**.

8.4 The *static* Keyword

You have already seen the keyword **`static`** in front of the names of methods in some Java classes. A word such as this (as well as the words **`public`** and **`private`**) is called a **modifier**. A modifier determines the particular way a class, attribute or method is accessed.

Let's explore what this **`static`** modifier does. Consider the `BankAccount` class that we discussed in the previous section. Say we wanted to have an additional method which added interest, at the current rate, to the customer's balance. It would be useful to have an attribute called `interestRate` to hold the value of the current rate of interest. But of course the interest rate is the same for any customer—and if it changes, we want it to change for every customer in the bank; in other words for every object of the class. We can achieve this by declaring the variable as **`static`**. An attribute declared as **`static`** is a *class* attribute; any changes that are made to it are made to all the objects in the class. The way this is achieved is by the program creating only one copy of the attribute and making it accessible to all objects.

It would make sense if there were a way to access this attribute without reference to a specific object; and so there is! All we have to do is to declare methods such as `setInterestRate` and `getInterestRate` as **`static`**. This makes a method into a *class* method; it does not refer to any specific object. As you will see in our next program, `BankAccountTester2`, we can call a class method by using the class name instead of the object name.

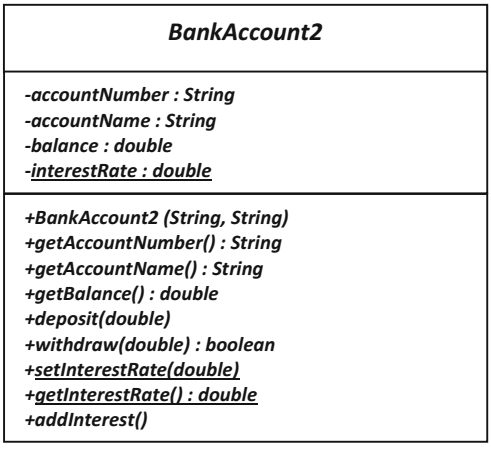


Fig. 8.4 The design of the *BankAccount2* class

We have rewritten our `BankAccount` class, and called it `BankAccount2`. We have included three new methods as well as the new **`static`** attribute `interestRate`. The first two of these—`setInterestRate` and `getInterestRate`—are the methods that allow us to read and write to our new attribute. These have been declared as **`static`**. The third—`addInterest`—is the method that adds the interest to the customer's balance. As can be seen in Fig. 8.4, the UML notation is to underline static attributes and methods.

Here is the code for the class. The new items have been emboldened.

***BankAccount2* - the modified *BankAccount* class**

```
public class BankAccount2
{
    private String accountNumber;
    private String accountName;
    private double balance;
    private static double interestRate;

    public BankAccount2(String numberIn, String nameIn)
    {
        accountNumber = numberIn;
        accountName = nameIn;
        balance = 0;
    }

    public String getAccountName()
    {
        return accountName;
    }

    public String getAccountNumber()
    {
        return accountNumber;
    }

    public double getBalance()
    {
        return balance;
    }

    public void deposit(double amountIn)
    {
    balance = balance + amountIn;
    }

            public boolean withdraw(double amountIn)
            {
                    if(amountIn > balance)
                    {
                            return false;
                    }
                    else
                    {
                            balance = balance - amountIn;
                            return true;
                    }
            }

    public static void setInterestRate(double rateIn)
    {
        interestRate = rateIn;
    }

    public static double getInterestRate()
    {
        return interestRate;
    }

    public void addInterest()
    {
        balance = balance + (balance * interestRate)/100;
    }
}
```

The following program, `BankAccountTester2`, uses this modified version of the `BankAccount` class.

BankAccountTester2

```
public class BankAccountTester2
{
    public static void main(String[] args)
    {
        // create a bank account
        BankAccount2 account1 = new BankAccount2("99786754","Gayle Forcewind");
        // create another bank account
        BankAccount2 account2 = new BankAccount2("99887776","Stan Dandy-Liver");
        // make a deposit into the first account
        account1.deposit(1000);
        // make a deposit into the second account
        account2.deposit(2000);
        // set the interest rate
        BankAccount2.setInterestRate(10);
        // add interest to accounts
        account1.addInterest();
        account2.addInterest();
        // display the account details
        System.out.println("Account number: " + account1.getAccountNumber());
        System.out.println("Account name: " + account1.getAccountName());
        System.out.println("Interest Rate " + BankAccount2.getInterestRate());
        System.out.println("Current balance: " + account1.getBalance());
        System.out.println(); // blank line
        System.out.println("Account number: " + account2.getAccountNumber());
        System.out.println("Account name: " + account2.getAccountName());
        System.out.println("Interest Rate " + BankAccount2.getInterestRate());
        System.out.println("Current balance: " + account2.getBalance());
    }
}
```

Take a closer look at the first four lines of the `main` method of the above program. We have created two new bank accounts which we have called `account1` and `account2`, and have assigned account numbers and names to them at the time they were created (via the constructor). We have then deposited amounts of 1000 and 2000 respectively into each of these accounts.

Now look at the next line:

```
BankAccount2.setInterestRate(10);
```

This line sets the interest rate to 10. Because `setInterestRate` has been declared as a **`static`** method, we have been able to call it by using the class name `BankAccount2`. Because `interestRate` has been declared as a **`static`** attribute this change is effective for any object of the class. Therefore, when we add interest to each account as we do with the next two lines:

```
account1.addInterest();
account2.addInterest();
```

we should expect it to be calculated with an interest rate of 10, giving us new balances of 1100 and 2200 respectively.

This is exactly what we get, as can be seen from the output below:

```
Account number: 99786754
Account name: Gayle Forcewind
Interest Rate 10.0
Current balance: 1100.0

Account number: 99887776
Account name: Stan Dandy-Liver
Interest Rate 10.0
Current balance: 2200.0
```

Class methods can be very useful indeed and we shall see further examples of them in this chapter. Of course, we have always declared our `main` method, and other methods within the same class as the `main` method, as **`static`**—because these methods belong to the class and not to a specific object.

8.5 Initializing Attributes

Looking back at the `BankAccount2` class in the previous section, some of you might have been asking yourselves what would happen if we called the `getInterestRate` method before the interest rate had been set using the `setInterestRate` method. In fact, the answer is that a value of zero would be returned. This is because, while Java does not give an initial value to *local* variables (which is why you get a compiler error if you try to use an uninitialized variable), Java always initializes attributes. Numerical attributes such as **`int`** and **`double`** are initialized to zero; **`boolean`** attributes are initialized to **`false`** and objects are initialized to **`null`**. Character attributes are given an initial Unicode value of zero.

Despite the above, it is nonetheless good programming practice always to give an initial value to your attributes, rather than leave it to the compiler. One very good reason for this is that you cannot assume that every programming language initializes variables in the same way—if you were using C++, for example, the initial value of any variable is completely a matter of chance—and you won't get a compiler error to warn you! In the `BankAccount2` class, it would have done no harm at all to have initialized the `interestRate` variable when it was declared:

```
private static double interestRate = 0;
```

In fact, one technique you could use is to give the `interestRate` attribute some special initial value (such as a negative value) to indicate to the user of this class that the interest rate had not been set. You will see another example where this technique can be used in question 2 of the programming exercises.

8.6 The *EasyScanner* Class

In the previous chapter we used a class called `EasyScanner` that could make keyboard input a lot easier. We have now covered all the concepts you need in order to understand how this class works. Here it is:

The *EasyScanner* class

```
import  java.util.Scanner;

public class EasyScanner
{
    public static int nextInt()
    {
        Scanner keyboard = new Scanner(System.in);
        int i = keyboard.nextInt();
        return i;
    }

    public static double nextDouble()
    {
        Scanner keyboard = new Scanner(System.in);
        double d = keyboard.nextDouble();
        return d;
    }

    public static String nextString()
    {
        Scanner keyboard = new Scanner(System.in);
        String s = keyboard.nextLine();
        return s;
    }

    public static char nextChar()
    {
        Scanner keyboard = new Scanner(System.in);
        char c = keyboard.next().charAt(0);
        return c;
    }
}
```

You can see that we have made every method a **`static`** method, so that we can simply use the class name when we call a method. For example:

```
int number = EasyScanner.nextInt();
```

You can see that the `nextString` method uses the `nextLine` method of the `Scanner` class—but as a new `Scanner` object is created each time the method is called there is no problem about using it after a `nextInt` or a `nextDouble` method as there is with `nextLine` itself.

We will use the `EasyScanner` class later, in Sect. 8.8.2.

8.7 Passing Objects as Parameters

In Chap. 5 it was made clear that when a variable is passed to a method it is simply the *value* of that variable that is passed—and that therefore a method cannot change the value of the original variable. In Chap. 6 you found out that in the case of an

array it is the value of the memory location (a *reference*) that is passed and consequently the value of the original array elements can be changed by the called method.

What about objects? Let's write a little program (`ParameterTest`) to test this out.

ParameterTest

```
public class ParameterTest
{
   public static void main(String[] args)
   {
      // create new bank account
      BankAccount testAccount = new BankAccount("1", "Ann T Dote");
      test(testAccount); // send the account to the test method
      System.out.println("Account Number: " + testAccount.getAccountNumber());
      System.out.println("Account Name: " + testAccount.getAccountName());
      System.out.println("Balance: " + testAccount.getBalance());
   }

   // a method that makes a deposit in the bank account
   static void test(BankAccount accountIn)
   {
      accountIn.deposit(2500);
   }
}
```

The output from this program is as follows:

```
Account Number: 1
Account Name: Ann T Dote
Balance: 2500.0
```

You can see that the deposit has successfully been made—in other words the attribute of the object has actually been changed. This is because what was sent to the method was, of course, a *reference* to the original `BankAccount` object, `testAccount`. Thus `accountIn` is a *copy* of the `testAccount` reference and so points to the original object and invokes that object's methods. So the following line of code:

```
accountIn.deposit(2500);
```

calls the `deposit` method of the original `BankAccount` object.

You might think this is a very good thing, and will make life easier for you as a programmer. However, you need a word of caution here. It is very easy inadvertently to allow a method to change an object's attributes, so you need to take care—more about this in the second semester.

8.8 Collection Classes

In Chap. 7 we introduced you to the idea of a collection class—a class which holds a collection of objects. We showed you how to use the `ArrayList` class to hold a collection of objects of a specific type, and how to use a couple of `ArrayList` methods.

Methods of Java's collection classes are of course not tailored to any specific type. A method such as `remove`, for example, requires us to send in a reference to the object to be removed. Normally, however, we would reference an object such as a bank account by a unique field such as an account number—so any program using an `ArrayList` would need to first search the list to find the object we want, and then to remove it.

While this approach would work perfectly well, it would be a lot more convenient if we could tailor our collection classes so that they provided the specific methods that we need. For example, in the case of a list of bank accounts it would be useful to deposit or withdraw funds from a specific account referenced by its account number, or to remove an account with a particular number.

We can do this quite easily by creating our own collection class with an attribute which is itself a collection, such as `ArrayList`. We have done this below; we have called our collection class `Bank`.

8.8.1 The *Bank* Class

When one object itself consists of other objects, this relationship is called **aggregation**. This association, represented in UML by a diamond, is often referred to as a *part-of* relationship. For example, the association between a car and the passengers in the car is aggregation. **Composition** (represented by a filled diamond) is a special, stronger, form of aggregation whereby the "whole" is actually dependent on the "part". For example, the association between a car and its engine is one of *composition*, as a car cannot exist without an engine. A collection class is an implementation of the aggregation relationship.

The association between the container object, `Bank`, and the contained object, `BankAccount`, is shown in the UML diagram of Fig. 8.5.

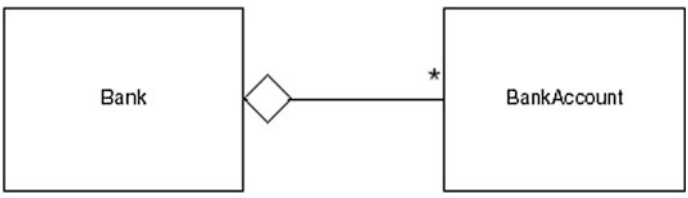


Fig. 8.5 The *Bank* object can contain many *BankAccount* objects

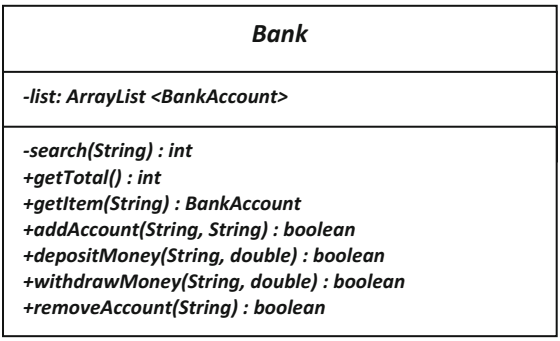


Fig. 8.6 The design of the *Bank* class

The asterisk at the other end of the joining line indicates that the `Bank` object contains *zero or more* `BankAccount` objects. The design for the `Bank` class is now given in Fig. 8.6.

As can be seen in Fig. 8.6, the class will have a single attribute, `list`, which is a collection of `BankAccounts`. Here we have decided to use an `ArrayList` of `BankAccount` objects to store this collection:

-list: ArrayList <BankAccount>

There are seven methods, which are described below:

-search(String) : int
This is what we can term a **helper** method; it will be declared as **`private`** (note the minus sign in the UML notation), because it is not intended for it to be called by other classes. It accepts a `String` representing the account number. It then returns the index of the account with that account number in the `ArrayList`. If the account number does not exist, then a "phoney" index (−999) will be returned to indicate failure.

+getTotal(): int
This method simply returns the total number of accounts currently in the system.

+getItem(String): BankAccount

This method receives a `String` representing an account number, and returns the `BankAccount` with that account number.

If the account number is not valid, a **`null`** value will be returned.

+addAccount(String, String): boolean

This method receives two strings representing the account number and name respectively, and adds an account with these details to the list of accounts. If an account with this number already exists, the new account will not be added and the method will return a value of **`false`**. However, if the operation has been completed successfully a value of **`true`** is returned.

+depositMoney(String, double) : boolean

Accepts a `String`, representing the account number of a particular account, and an amount of money which is to be deposited in that account. Returns **`true`** if the deposit was made successfully, or **`false`** otherwise (no such account number).

+withdrawMoney(String, double) : boolean

Accepts a `String`, representing the account number of a particular account, and an amount of money which is to be withdrawn from that account. Returns **`true`** if the withdrawal was made successfully, or **`false`** otherwise. The reason for the withdrawal not taking place could be that there is no such account number or that there are insufficient funds. In this version of `Bank`, the method does not indicate which of these reasons caused the failure—that is left for you as an end of chapter exercise.

+removeAccount(String) : boolean

Accepts a `String`, representing an account number, and removes that account from the list. Returns **`true`** if the account was removed successfully, or **`false`** otherwise (no such account number).

The code for the `Bank` class is presented below. Take a careful look at it, then we will discuss it.

The *Bank* class

```
import java.util.ArrayList;

public class Bank
{

    ArrayList<BankAccount> list = new ArrayList<>();

    // helper method to find the index of a specified account
    private int search(String accountNumberIn)
    {
        for(int i = 0; i <= list.size() - 1; i++)
        {
            BankAccount tempAccount = list.get(i); // find the account at index i
            String tempNumber = tempAccount.getAccountNumber(); // get account number
            if(tempNumber.equals(accountNumberIn)) // if this is the account we are looking for
            {
                return i; // return the index
            }
        }
        return -999;
    }

    // return the total number of items
    public int getTotal()
    {
        return list.size();
    }


    // return an account with a particular account number
    public BankAccount getItem(String accountNumberIn)
    {
        int index = search(accountNumberIn);
        if(index != -999) // check that account exists
        {
             return list.get(index);
        }
        else
        {
             return null; // no such account
        }
    }

    // add an item to the list
    public boolean addAccount(String accountNumberIn, String nameIn)
    {
        if(search(accountNumberIn) == -999) // check that account does not already exist
        {
            list.add(new BankAccount(accountNumberIn, nameIn)); // add new account
            return true;
        }
        return false;
    }

    // deposit money in a specified account
    public boolean depositMoney(String accountNumberIn, double amountIn)
    {
        BankAccount acc = getItem(accountNumberIn);
        if(acc != null)
        {
            acc.deposit(amountIn);
            return true; // indicate success
        }
        else
        {
            return false; // indicate failure
        }
    }

    // withdraw money from a specified account
    public boolean withdrawMoney(String accountNumberIn, double amountIn)
    {
        BankAccount acc = getItem(accountNumberIn);
        if(acc != null && acc.getBalance() >= amountIn)
        {
            acc.withdraw(amountIn);
            return true; // indicate success
        }
        else
        {
            return false; // indicate failure
        }
    }

    // remove an account
    public boolean removeAccount(String accountNumberIn)
    {
        int index = search(accountNumberIn); // find index of account
        if(index != -999) // if account exists account
        {
            list.remove(index);
            return true; // remove was successful
        }
        else
        {
            return false; // remove was unsuccessful
        }
    }
}
```

As you can see, we have declared and initialized a single attribute, an `ArrayList` which will hold `BankAccount` objects.

```
ArrayList<BankAccount> list = new ArrayList<>();
```

Now the methods. Firstly the `search` method, which is declared as **`private`** because it is there only to assist other methods of the class, rather than to be accessed by other classes:

```
    private int search(String accountNumberIn)
    {
        for(int i = 0; i <= list.size() - 1; i++)
        {
            BankAccount tempAccount = list.get(i); // find the account at index i
            String tempNumber = tempAccount.getAccountNumber(); // get account number
            if(tempNumber.equals(accountNumberIn)) // if this is the account we are looking for
            {
                return i; // return the index
            }
        }
        return -999;
    }
```

You have seen something like this before in Chap. 6 when we searched an integer array—you can see we are using the same technique of sending back a "dummy" value if the account number is not valid.

On each iteration of the loop the account at that index is retrieved using the `get` method of `ArrayList` and assigned to a `BankAccount` object, `tempAccount`. The account number of `tempAccount` is then assigned to a `String` variable `tempNumber`. This is compared with the account number that has been input. If the account number matches, the loop returns the index of that item and terminates. Otherwise, the loop continues to the end of the list (determined by using the `size` method of `ArrayList`). The method then returns the dummy value of −999, indicating that no item with that account number exists.

The next method simply returns the total number of items currently in the list, again using the `size` method of `ArrayList`:

```
    public int getTotal()
    {
        return list.size();
    }
```

Next we have a method to retrieve an account with a particular account number:

```
public BankAccount getItem(String accountNumberIn)
{
    int index = search(accountNumberIn);
    if(index != -999) // check that account exists
    {
         return list.get(index);
    }
    else
    {
         return null; // no such account
    }
}
```

Here you can see how we utilize our `search` method—we use it to find the index of the account with the given account number, then we check that the index is not equal to −999 (in other words that the account exists), and as long it is a valid index we return the relevant account. If the index is not valid a **`null`** value is returned.

Now we come to the `addAccount` method:

```
public boolean addAccount(String accountNumberIn, String nameIn)
{
    if(search(accountNumberIn) == -999) // check that account does not already exist
    {
        list.add(new BankAccount(accountNumberIn, nameIn)); // add new account
        return true;
    }
    return false;
}
```

Once again we use the `search` method, this time to check that an account with this number does *not* already exist—so we are hoping that `search` returns a value −999. If this is the case we use the `add` method of `ArrayList` to add a new `BankAccount` object, which we create from the account number and account name that are received as parameters to the method—the method then returns **`true`**, indicating success.

Should the account already exist, a value of **`false`** is returned, indicating that no new account was added.

Now for the `depositMoney` method.

```
public boolean depositMoney(String accountNumberIn, double amountIn)
{
    BankAccount acc = getItem(accountNumberIn);
    if(acc != null)
    {
        acc.deposit(amountIn);
        return true; // indicate success
    }
    else
    {
        return false; // indicate failure
    }
}
```

The method receives the account number of the account in which we wish to place the money, and the amount to be deposited. We retrieve the correct account with the `getItem` method that we developed earlier. We check that this is not a **`null`** value, and if all is well we use the `deposit` method of `BankAccount` to deposit the money, and return a value of **`true`**, to indicate success. If the account returned was **`null`**, that indicates that there was no account with the account number in question, and in that case a value of **`false`** is returned.

The `withdrawMoney` method is similar except that we need to have an additional check in the **`if`** statement to see whether or not there were sufficient funds for the withdrawal to go ahead:

```
public boolean withdrawMoney(String accountNumberIn, double amountIn)
{
    BankAccount acc = getItem(accountNumberIn);
    if(acc != null && acc.getBalance() >= amountIn)
    {
        acc.withdraw(amountIn);
        return true; // indicate success
    }
    else
    {
        return false; // indicate failure
    }
}
```

As we mentioned above, the method could be improved if there were a way to determine whether the withdrawal was declined because there was no such account or because there were insufficient funds. This is left for the exercises at the end of the chapter.

Finally we have the method that removes an account:

```
public boolean removeAccount(String accountNumberIn)
{
    int index = search(accountNumberIn); // find index of account
    if(index != -999) // if account exists account
    {
        list.remove(index);
        return true; // remove was successful
    }
    else
    {
        return false; // remove was unsuccessful
    }
}
```

As you can see, we make use of the `remove` method of `ArrayList`, which removes an item at a particular index. The index is found by calling the `search` method as before, and as before we first test to make sure the account exists; if it does, then the account is removed and a value of **`true`** is returned—if not the method returns a value of **`false`**.

It is worth considering how much more complex it would have been to remove an item from the collection if we had used an array rather than an `ArrayList`. Since the array might be only partially full, we would have to introduce a variable to keep track of the position of the last item in the array. Then all the items following the one to be removed would have to be shuffled along by one position in order to overwrite the given item. Finally, we would need to make sure when any new item is added, it is added to the end of the reduced array, so we would have to reduce the 'end-of-array' variable by 1. That's rather a lot of work we don't need to do now we have used an `ArrayList` and can do all that with just one call to its `remove` method!

8.8.2 Testing the *Bank* Class

The program below, `BankApplication` uses the `Bank` class—notice that we are using our new `EasyScanner` class here.

BankApplication

```
public class BankApplication
{
    public static void main(String[] args)
    {
        char choice;

        Bank myBank = new Bank();

        // offer menu
        do
        {
            System.out.println();
            System.out.println("1. Create new account");
            System.out.println("2. Remove an account");
            System.out.println("3. Deposit money");
            System.out.println("4. Withdraw money");
            System.out.println("5. Check account details");
            System.out.println("6. Quit");
            System.out.println();
            System.out.print("Enter choice [1-6]: ");

            // get choice
            choice = EasyScanner.nextChar();
            System.out.println();

            // process menu options
            switch (choice)
            {
                case '1': option1(myBank);
                break;
                case '2': option2(myBank);
                break;
                case '3': option3(myBank);
                break;
                case '4': option4(myBank);
                break;
                case '5': option5(myBank);
                break;
                case '6': break;
                default: System.out.println("Invalid entry");
                }
            }
            while (choice != '6');
        }

        // add account
        static void option1(Bank bankIn)
        {
            // get details from user
            System.out.print("Enter account number: ");
            String number = EasyScanner.nextString();
            System.out.print("Enter account name: ");
            String name = EasyScanner.nextString();
            // add account to list
            boolean success = bankIn.addAccount(number, name);
            if(success)
            {
                System.out.println("Account added");
            }
            else
            {
                System.out.println("Account number already exists");
            }
        }
            // remove account
            static void option2(Bank bankIn)
            {
                // get account number of account to remove
                System.out.print("Enter account number: ");
                String number = EasyScanner.nextString();
                // delete item if it exists
                boolean found = bankIn.removeAccount(number);

                if (found)
                {
                    System.out.println("Account removed");
                }
                else
                {
                System.out.println("No such account number");
                }
```

```
        }

        // deposit money in an account
        static void option3(Bank bankIn)
        {
            // get details from user
            System.out.print("Enter account number: ");
            String number = EasyScanner.nextString();
            System.out.print("Enter amount to deposit: ");
            double amount = EasyScanner.nextDouble();

            boolean found = bankIn.depositMoney(number, amount);

            if(found)
            {
                System.out.println("Money deposited");
            }
            else
            {
             System.out.println("No such account");
            }

    }

    // withdraw money from an account
    static void option4(Bank bankIn)
    {
        // get details from user
        System.out.print("Enter account number: ");
        String number = EasyScanner.nextString();
        System.out.print("Enter amount to withdraw: ");
        double amount = EasyScanner.nextDouble();
        boolean ok = bankIn.withdrawMoney(number, amount);

        if(ok)
        {
            System.out.println("Withdrawal made");
        }
        else
        {
            System.out.println("No such account or insufficient funds");
        }

    }

    // check account details
    static void option5(Bank bankIn)
    {
        // get details from user
        System.out.print("Enter account number ");
        String number = EasyScanner.nextString();

        BankAccount account = bankIn.getItem(number);

        if(account != null)
        {
            System.out.println("Account number: " + account.getAccountNumber());
            System.out.println("Account name: " +account.getAccountName());
            System.out.println("Balance: " + account.getBalance());
            System.out.println();
        }
        else
        {
            System.out.println("No such account");
        }
    }
}
```

You are familiar with this sort of menu-driven program, so there is not too much to say about it, except to observe that this is probably the first example of an application which, although not all that complex, could actually be thought of as the kind of application that could be used in a real business environment. Of course, in the outside world such applications are much more sophisticated than this, but they are, in principle, not too different from the sort of thing we have just done. Notice that our application involves a number of classes that we have written ourselves, and have pulled together to form a single application.

It is worth drawing attention to the way that the program makes use of some of the features of the `Bank` class that we incorporated into the `BankApplication`. For example, in `option1` (and similarly in other methods) we make use of the fact that the `addAccount` method of `Bank` returns **`true`** if the new account was successfully added, and **`false`** otherwise:

```
boolean success = bankIn.addAccount(number, name);
if(success)
{
     System.out.println("Account added");
}
else
{
     System.out.println("Account number already exists");
}
```

In a similar way, in `option5`, we use the fact that the `getItem` method returns **`null`** if the account was not found:

```
if(account != null)
{
    System.out.println("Account number: " + account.getAccountNumber());
    System.out.println("Account name: " +account.getAccountName());
    System.out.println("Balance: " + account.getBalance());
    System.out.println();
}
else
{
    System.out.println("No such account");
}
```

We end this chapter with an example program run from `BankApplication`, followed by a few ideas on how our application could be improved.

```
1. Create new account
2. Remove an account
3. Deposit money
4. Withdraw money
5. Check account details
6. Quit

Enter choice [1-6]: 1

Enter account number: 63488965
Enter account name: Mary Land-Cookies
Account added

1. Create new account
2. Remove an account
3. Deposit money
4. Withdraw money
5. Check account details
6. Quit
```

```
Enter choice [1–6]: 1

Enter account number: 98654322
Enter account name: Laura Norder
Account added

1. Create new account
2. Remove an account
3. Deposit money
4. Withdraw money
5. Check account details
6. Quit

Enter choice [1–6]: 1

Enter account number: 12347890
Enter account name: Gary Baldi–Biscuits
Account added

1. Create new account
2. Remove an account
3. Deposit money
4. Withdraw money
5. Check account details
6. Quit

Enter choice [1–6]: 3

Enter account number: 12347890
Enter amount to deposit: 1500
Money deposited

1. Create new account
2. Remove an account
3. Deposit money
4. Withdraw money
5. Check account details
6. Quit

Enter choice [1–6]: 2

Enter account number: 98654322
Account removed
```

```
1. Create new account
2. Remove an account
3. Deposit money
4. Withdraw money
5. Check account details
6. Quit

Enter choice [1–6]: 5

Enter account number 55566777
No such account

1. Create new account
2. Remove an account
3. Deposit money
4. Withdraw money
5. Check account details
6. Quit

Enter choice [1–6]: 5

Enter account number 12347890
Account number: 12347890
Account name: Gary Baldi–Biscuits
Balance: 1500.0

1. Create new account
2. Remove an account
3. Deposit money
4. Withdraw money
5. Check account details
6. Quit

Enter choice [1–6]: 6
```

We should point out that for our application to be useful to any organization, it would need to be able to store the account information even after the application terminates. However, before you are able to achieve this you will have to wait until the second semester, where you will find out how to create files to hold permanent records.

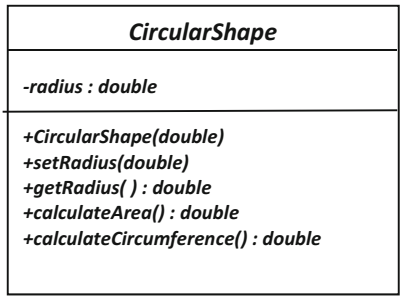
8.9 The Benefits of Object-Oriented Programming

In this chapter and the previous one you have seen how to create classes and use them as data types in your programs. You have seen how the process of building classes enables us to hide data within a class. Programming languages based on classes and objects—in other words object-oriented languages—have brought a number of benefits, and are now the standard. Below we have summarized some of the benefits that this has brought us.

- As we have demonstrated, the ability to encapsulate data within a class has enabled us to build far more secure systems.
- The object-oriented approach makes it far easier for us to *re-use* classes again and again. Having defined a `BankAccount` class or a `Student` class for example, we can use them in many different programs without having to write a new class each time. In the next chapter you will also see how it is possible to refine existing classes to meet additional needs by the technique known as **inheritance**. If systems can be assembled from re-usable objects, this leads to far higher productivity.
- With the object-oriented approach it is possible to define and use classes which are not yet complete. They can then be extended without upsetting the operation of other classes. This greatly improves the testing process. We can easily build prototypes without having to build a whole system before testing it and letting the user of the system see it.
- The object-oriented approach makes it far easier to make changes to systems once they have been completed. Whole classes can be replaced, or new classes can easily be added.
- The object-oriented way of doing things is a far more "natural" approach. We base our programs on objects that exist in the real world—students, bank accounts, customers and so on.
- The modular nature of object-oriented programming improves the whole development process. The modular approach means that the old methodologies whereby systems were first analysed, then designed, and then implemented and tested were able give way to new methods whereby these processed were far more integrated and systems were developed far more rapidly.

8.10 Self-test Questions

1. In question 7 of the programming exercises at the end of Chap. 2 you wrote a program that calculated the area and circumference of a circle. Now consider a class that we could develop for this purpose; we have called it `CircularShape`. Here is the UML design:

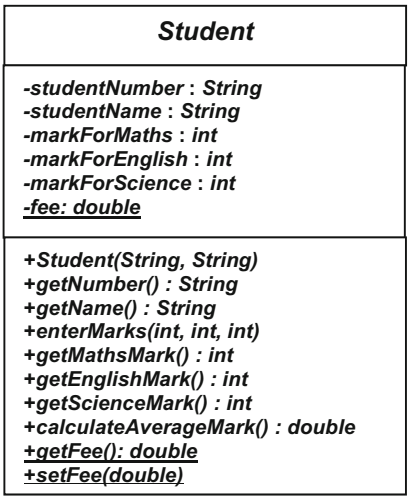


(a) Distinguish between *attributes* and *methods* in this class.
(b) Explain what it meant by the term *encapsulation*, how it is recorded in this UML diagram and how it is implemented in a Java class.
(c) For each method in the `CircularShape` class, determine

- the number of parameters;
- the type of any parameters;
- the return type;
- the equivalent method header in Java.

(d) Add an additional method into this UML diagram, `calculateDiameter`, which calculates and returns the diameter of the circle.
(e) Write the Java code for the `calculateDiameter` method.

2. The UML diagram below represents the design for a `Student` class.



You can see that students have a name, a number, some marks for subjects they are studying and the fee. Methods are then provided to process this data.

(a) What is indicated by the fact that certain attributes and methods have been underlined?
(b) Write the Java code for the parts of the class that have been underlined.

3. Consider the following class:

```
public class SomeClass
{
    private int x;

    public SomeClass( )
    {
        x = 10;
    }

    public SomeClass(int xIn)
    {
        x = xIn;
    }

    public void setX(int xIn)
    {
        x = xIn;
    }

    public int getX()
    {
        return x;
    }
}
```

(a) What would be the output from the following program?

```
public class Test1
{
    public static void main(String[] args)
    {
        SomeClass myObject = new SomeClass();
        System.out.println(myObject.getX());
    }
}
```

(b) What would be the output from the following program?

```
public class Test2
{
    public static void main(String[] args)
    {
        SomeClass myObject = new SomeClass(5);
        System.out.println(myObject.getX());
    }
}
```

(c) Explain why the following program would not compile.

```
public class Test3
{
    public static void main(String[] args)
    {
        SomeClass myObject = new SomeClass(5, 8);
        System.out.println(myObject.getX());
    }
}
```

(d) What would be the output from the following program?

```
public class Test4
{
    public static void main(String[] args)
    {
        int y = 20;
        SomeClass myObject = new SomeClass(5);
        System.out.println(myObject.getX());
        test(y, myObject);
        System.out.println(y);
        System.out.println(myObject.getX());
    }

    static void test(int z, SomeClass classIn)
    {
        z = 50;
        classIn.setX(100);
    }
}
```

4. Consider the `Bank` program from Sect. 8.8.1.

(a) Adapt the `withdrawMoney` method so that it distinguishes the two reasons why the method might fail—namely that there is no account with the given account number, or there is not enough money in the account to make a withdrawal.

A **`boolean`** method would no longer suffice as there is more then one possibility. One solution would be for the method to return an integer—perhaps 1 for success, −1 to indicate that the method failed because there was no such account number, and −2 to indicate that it failed because there were insufficient funds.

(b) Adapt the `BankApplication` program from Sect. 8.8.2 so that option 4 now uses the new version of `withdrawMoney`.

5. Identify some of the reasons why the object-oriented approach has become the norm for programming.

8.11 Programming Exercises

1. (a) Implement the `CircularShape` class that was discussed in self-test question 1 above.

   (b) Add the `calculateDiameter` method into this class as discussed in self-test question 1d and 1e above.

   (c) Write a program to test out your class. This program should allow the user to enter a value for the radius of the circle, and then display the area, circumference and diameter of this circle on the screen by calling the appropriate methods of the `CircularShape` class.

   (d) Modify the tester program above so that once the information has been displayed the user is able to reset the radius of the circle. The area, circumference and diameter of the circle should then be displayed again.

2. (a) Write the code for the `Student` class discussed in self-test question 2 above. You should note that in order to ensure that a **`double`** is returned from the `calculateAverageMark` method you should specifically divide the total of the three marks by 3.0 and not simply by 3 (look back at Chap. 2 to remind yourself why this is the case).

   Another thing to think about is what you choose for the initial values of the marks. If you chose to give each mark an initial value of zero, this could be ambiguous; a mark of zero could mean that the mark simply has not been entered—or it could mean the student actually scored zero in the subject! Can you think of a better initial value?

   You can assume that the fees for the student are set initially to 3000.

   (b) Write a tester class to test out your `Student` class; it should create two or three students (or even better an `ArrayList` of students), and use the methods of the `Student` class to test whether they work according to the specification.

3. A system is being developed for use in a store that sells electrical appliances. A class called `StockItem` is required for this system. An object of the `StockItem` class will require the following attributes:

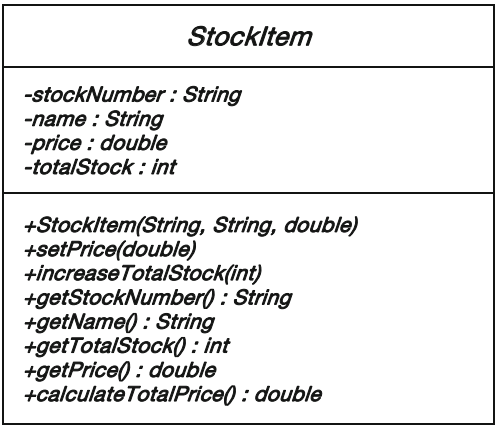
   - a stock number;
   - a name;
   - the price of the item;
   - the total number of these items currently in stock.

   The first three of the above attributes will need to be set at the time a `StockItem` object is created—the total number of items in stock will be set to zero at this time. The stock number and name will not need to be changed after the item is created.

The following methods are also required:

- a method that allows the price to be re-set during the object's lifetime;
- a method that receives an integer and adds this to the total number of items of this type in stock;
- a method that returns the total value of items of this type in stock; this is calculated by multiplying the price of the item by the number of items in stock;
- methods to read the values of all four attributes.

The design of the `StockItem` class is shown in the following UML diagram:



(a) Write the code for the `StockItem` class.

(b) Consider the following program, which uses the `StockItem` class, and in which some of the code has been replaced by comments:

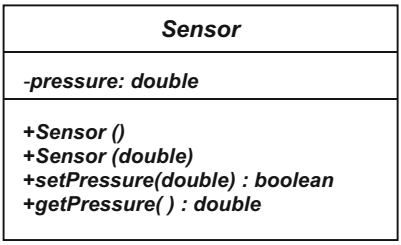
```
import  java.util.Scanner;
public class TestProg
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        Scanner keyboardString = new Scanner(System.in);
        String tempNumber;
        String tempName;
        double tempPrice;

        System.out.print("Enter the stock number: ");
        tempNumber = keyboardString.nextLine();
        System.out.print("Enter the name of the item: ");
        tempName = keyboardString.nextLine();
        System.out.print("Enter the price of the item: ");
        tempPrice = keyboard.nextDouble();

        // Create a new item of stock using the values that were entered by the user

        // Increase the total number of items in stock by 5

        // Display the stock number

        // Display the total price of all items in stock
    }
}
```

Replace the comments with appropriate code.

(c) i. A further attribute, `salesTax`, is required. The value of this attribute should always be the same for each object of the class. Write the declaration for this attribute.

ii. Provide a class method, `setSalesTax`, for this class—it should receive a **`double`** and set the value of the sales tax to this value.

iii. Write a line of code that sets the sales tax for all objects of the class to 10 without referring to any particular object.



4. The class shown below keeps track of a pressure sensor in a laboratory.

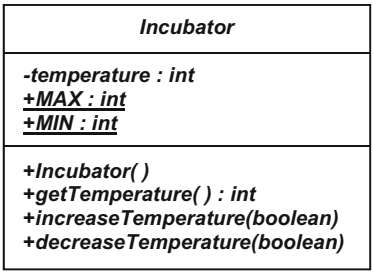
   When a `Sensor` object is created using the first constructor, the initial pressure is set to zero. When it is created using the second constructor it is set to the value of the parameter.

   The pressure should not be set to a value less than zero. Therefore, if the input parameter to the `setPressure` method is a negative number, the pressure should not be changed and a value of `false` should be returned. If the pressure is set successfully, a value of `true` should be returned.

   (a) Write the code for the `Sensor` class.

   (b) Develop a `SensorTester` program to test the `Sensor` class.

5. Consider a class that keeps track of the temperature within an incubator. The UML diagram is shown below:



   When an `Incubator` object is created, the temperature is initially set to 5°.

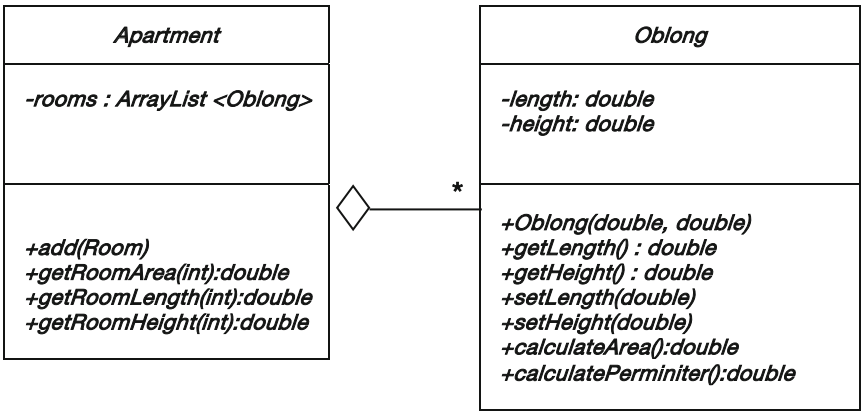
The `increaseTemp` method increases the temperature by 1, and the `decreaseTemp` method decreases the temperature by 1. However, the temperature must never be allowed to rise above a maximum value of 10 nor fall below a minimum value of −10. If an attempt is made to increase or decrease the temperature so it falls outside this range, then an alarm must be raised; the methods in this case should not increase or decrease the temperature but should return a value of **`false`**, indicating that the alarm should be raised. If the temperature is changed successfully, however, a value of **`true`** is returned.

(a) Write the code for the `Incubatotor` class.

(b) Develop a `IncubatorTester` program to test the `Incubator` class.

6. Implement the changes to the `Bank` class and the `BankApplication` program suggested in question 4 of the self-test questions. The source code for the `Bank` class and the `BankApplication` class can be downloaded from the website.

7. (a) In programming Exercise 6 of the last chapter you were asked to develop a program to process a collection of rooms in an apartment. Now consider a collection class, `Apartment`, for this purpose. The `Apartment` class would store a collection of `Oblong` objects, where each `Oblong` object represents a particular room in the apartment. The UML diagram depicting the association between the `Apartment` class and the `Oblong` class is shown below:



The single attribute of the `Apartment` class consists of a collection of `Oblong` objects, `rooms`, which makes use of an `ArrayList`.

The methods of the `Apartment` class are described below:

+add(Room) : boolean
Adds the given room to the list of `rooms`.

+getRoomArea(int) : double
Returns the area of the given room number sent in as a parameter. If an invalid room number is sent in as a parameter this method should send back some dummy value (for example −999).

+getRoomLength(int) : double
Returns the length of the given room number sent in as a parameter. If an invalid room number is sent in as a parameter this method should send back some dummy value (for example −999).

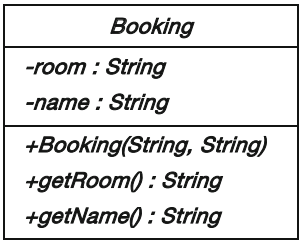
+getRoomHeight(int) : double
Returns the height of the given room number sent in as a parameter. If an invalid room number is sent in as a parameter this method should send back some dummy value (for example −999).

Implement the `Apartment` class.

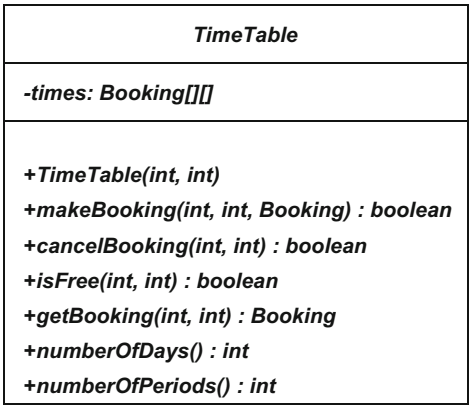
(b) Develop an `ApartmentTester` program to test the `Apartment` class.

8. Consider a scenario in which a university allows lecturers to borrow equipment. The equipment is available for use 5 days a week and for 7 periods during each day. When the equipment is booked for use, the details of the booking (room number and lecturer name) are recorded. When no booking is recorded, the equipment is available for use.

(a) Create a `Booking` class defined in the UML diagram below:



(b) Now a `TimeTable` class is defined to process these bookings. Its UML diagram is given below:



As you can see, the attribute of this class is a two-dimensional array of `Booking` objects. The methods of this class are defined below:

+TimeTable(int, int)
A constructor that accepts the number of days per week and number of periods per day and sizes the timetable accordingly.

You should note that initially all elements in the array will of course have a **`null`** value—*a* **`null`** *value will represent an empty slot.*

+makeBooking(int, int, Booking) : boolean
Accepts the booking details for a particular day and period and, as long as this slot is not previously booked and the day and period numbers are valid, updates the timetable accordingly. Returns **`true`** if the booking was recorded successfully and **`false`** if not.

+cancelBooking(int, int) : boolean
Cancels the booking details for a particular day and period. Returns **`false`** if the given slot was not previously booked or the day and period number are invalid, and **`true`** if the slot was cancelled successfully.

+isFree(int, int) : boolean
Accepts a day and period number and returns **`true`** if the day and period numbers are valid and the given slot is free, and **`false`** otherwise.

+getBooking(int, int) : Booking
Accepts a day and period number and returns the booking for the given slot if the day and period number are valid and the slot has been booked or **`null`** otherwise.

+numberOfDays() : int
Returns the number of days associated with this timetable.

+numberOfPeriods() : int
Returns the number of periods associated with this timetable.
Implement this class in Java.

(c) Write a suitable tester for this class.

9. Add some additional methods such as `nextByte` and `nextLong` to the `EasyScanner` class.