Classes and Objects

7

Outcomes:

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

- *explain the meaning of the term* ***object-oriented***;
- *explain and distinguish between the terms* ***class*** *and* ***object***;
- *create objects in Java*;
- *call the methods of an object*;
- *use a number of methods of the String class*;
- *create and use arrays of objects*;
- *create an* `ArrayList` *and make use of the* `add` *and* `get` *methods of this class*.

7.1 Introduction

In the 1990s it became the norm for programming languages to use special constructs called **classes** and **objects**. Such languages are referred to as **object-oriented programming languages**. In this chapter and the next one we will explain what is meant by these terms, and show you how we can exploit the full power of object-oriented languages like Java.

7.2 Classes as Data Types

So far you have been using data types such as **`char`**, **`int`** and **`double`**. These are simple data types that hold a *single* piece of information. But what if we want a variable to hold more than one related piece of information? Think for example of a book—a book might have a title, an author, an ISBN number and a price—or a

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_7

student might have a name, an enrolment number and marks for various subjects. Types such as **char** and **int** can hold a single piece of information only, and would therefore be completely inadequate for holding all the necessary information about a book or a student. An array would also not do because the different bits of data will not necessarily be all of the same type. Earlier languages such as C and Pascal got around this problem by allowing us to create a type that allowed more than one piece of information to be held—such types were known by various names in different languages, the most common being *structure* and *record*.

Object-oriented languages such as Java and C++ went one stage further however. They enabled us not only to create types that stored many pieces of data, but also to define within these types the methods by which we could process that data. For example a book 'type' might have a method that adds tax to the sale price; a student 'type' might have a method to calculate an average mark.

Do you remember that in the exercises at the end of Chap. 2 you wrote a little program that asked the user to provide the length and height of a rectangle, and then displayed the area and perimeter of that rectangle? In Chap. 5 you were asked to adapt this program so that it made use of separate methods to perform the calculations. Such a program might look like this:

RectangleCalculations

```
import java.util.Scanner;

public class RectangleCalculations
{
       public static void main(String[] args)
       {
          double length, height, area, perimeter;
          Scanner keyboard = new Scanner(System.in);
          System.out.print("What is the length of the rectangle? "); // prompt for length
          length = keyboard.nextDouble(); // get length from user
          System.out.print("What is the height of the rectangle? "); // prompt for height
          height = keyboard.nextDouble(); // get height from user
          area = calculateArea(length, height); // call calculateArea method
          perimeter = calculatePerimeter(length, height); // call calculatePerimeter method
          System.out.println("The area of the rectangle is " + area); // display area
          System.out.println("The perimeter of the rectangle is " + perimeter); // display perimeter
       }

       // method to calculate area
       static double calculateArea(double lengthIn, double heightIn)
       {
          return lengthIn * heightIn;
       }

       // method to calculate perimeter
       static double calculatePerimeter(double lengthIn, double heightIn)
       {
          return 2 * (lengthIn + heightIn);
       }
}
```

Can you see how useful it might be if, each time we wrote a program dealing with rectangles, instead of having to declare several variables and write methods to calculate the area and perimeter of a rectangle, we could just use a rectangle 'type' to create a single variable, and then use its pre-written methods? In fact you wouldn't even have to know how these calculations were performed.

This is exactly what an object-oriented language like Java allows us to do. You have probably guessed by now that this special construct that holds both data and methods is called a **class**. You have already seen a class as the basic unit which

contains our `main` method and any other additional methods. Now we can also use classes to define new 'types'.

You can see that there are two aspects to a class:

- the data that it holds;
- the tasks it can perform.

In the next chapter you will see that the different items of data that a class holds are referred to as the **attributes** of the class; the tasks it can perform, as we have seen, are referred to as the **methods** of the class—you have seen in Chap. 5 how we define methods. However, in Chap. 5, the methods were called only from *within* the class itself. Now we are going to see how to call the methods of *another* class. In fact you have already been doing this without quite realizing it—because you have, since the second chapter, been calling the methods of the `Scanner` class!

7.3 Objects

In order to use the methods of a class you need to create an **object** of that class. To understand the difference between classes and objects, you can think of a class as a blueprint, or template, from which objects are generated, whereas an object refers to an individual *instance* of that class. For example, imagine a system that is used by a bookshop. The shop will contain many hundreds of books—but we do not need to define a book hundreds of times. We would define a book once (in a class) and then generate as many objects as we want from this blueprint, each one representing an individual book.

This is illustrated in Fig. 7.1.

In one program we may have many classes, as we would probably wish to generate many kinds of objects. A bookshop system might have books, customers, suppliers and so on. A university administration system might have students, courses, lecturers etc.

Object-oriented programming therefore consists of defining one or more classes that may interact with each other.

We will now illustrate all of this by creating and using objects of predefined classes—defined either by ourselves or defined by the Java developers and provided as a standard part of the Java Development Kit. We are going to start by considering the example we discussed in Sect. 7.2 where we proposed a single "type" that dealt with calculating the area and perimeter of a rectangle. We are in fact not going to call our new class `Rectangle`, but `Oblong`. The reason for this is that there are in fact more than one `Rectangle` classes defined in the "built-in" Java libraries; now, while it is perfectly possible to have more than one class with the same name, it is better to avoid any possible confusion at this stage. In Chap. 19, you will learn how to avoid naming conflicts, but for now it is better to call our new class by a unique name.

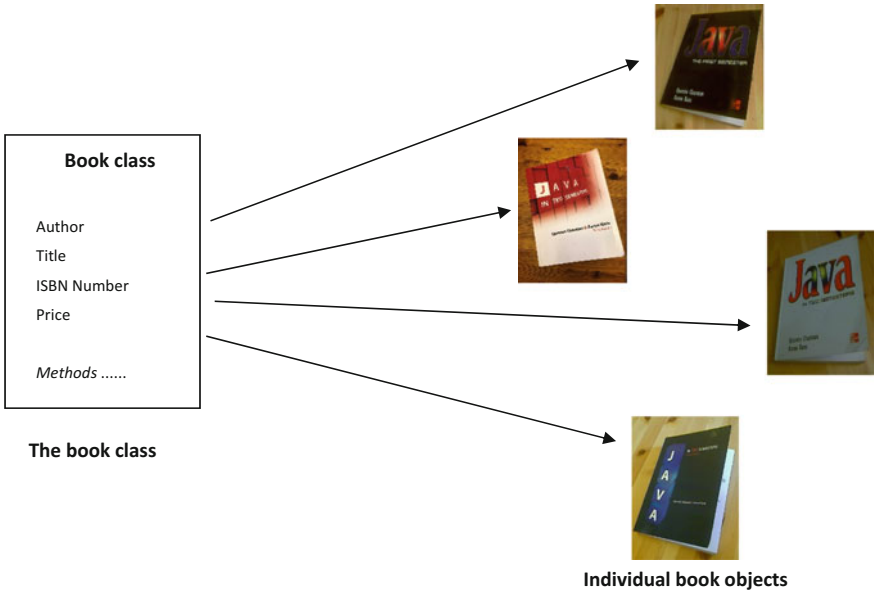


Fig. 7.1 Many objects can be generated from a single class template

And here is a "fun fact" for you—an oblong is not actually the same as a rectangle, because in an oblong the length and the height must be unequal, whereas a rectangle can have four equal sides—in other words it can be a square. Put another way a square is a kind of rectangle but not a kind of oblong.

7.4 The *Oblong* Class

We have written an `Oblong` class for you. The class we have created is saved in a file called `Oblong.java`. and you will need to download it from the website in order to use it. You must make sure that it is in the right place for your compiler to find it. You will need to place it in your project according to the rules of the particular IDE you are using.

In the next chapter we will look inside this class and see how it was written. For now, however, you can completely ignore the program code, because you can use a class without knowing anything about the details.

Once you have been provided with this `Oblong` class, instead of being restricted to making simple declarations like this:

```
int x;
```

you will now be able to make declarations like:

```
Oblong myOblong;
```

You can see that this line is similar to a declaration of a variable; however what we are doing here is not declaring a variable of a primitive type such as **`int`**, but declaring the name of an *object* (`myOblong`) of the *class* (`Oblong`)—effectively we have created a new type, `Oblong`.

You need to be sure that you understand what this line actually does; all it does in fact is to create a variable that holds a **reference** to an object, rather than the object itself. As explained in the previous chapter, a reference is simply a *name* for a location in memory. At this stage we have *not* reserved space for our new `Oblong` object; all we have done is named a memory location `myOblong`, as shown in Fig. 7.2.

Now of course you will be asking the question "How is memory for the `Oblong` object going to be created, and how is it going to be linked to the reference `myOblong`?".

As we have indicated, an object is often referred to as an *instance* of a class; the process of creating an object is referred to as **instantiation**. In order to create an object we use a very special method of the class called a **constructor**.

The constructor is a method that *always has the same name as the class*. When you create a new object this special method is always called; its function is to reserve some space in the computer's memory just big enough to hold the required object (in our case an object of the `Oblong` class).

As we shall see in the next chapter, a constructor can be defined to do other things, as well as reserve memory. In the case of our `Oblong` class, the constructor has been defined so that every time a new `Oblong` object is created, the length and the height are set—and they are set to the values that the user of the class sends in. So every time you create an `Oblong` object you have to specify its length and its height at the same time. Here for example is how you would call the constructor and create an oblong with length 7.5 and height 12.5:

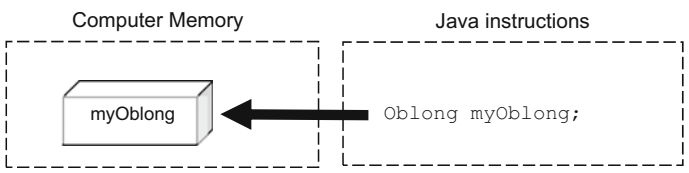


Fig. 7.2 Declaring an object reference

```
myOblong = new Oblong(7.5, 12.5);
```

This is the statement that reserves space in memory for a new `Oblong`. Using the constructor with the keyword **`new`**, reserves memory for a new `Oblong` object. Now, in the case of the `Oblong` class, the people who developed it (in this case it was us!) defined the constructor so that it requires that two items of data, both of type **`double`**, get sent in as parameters. Here we have sent in the numbers 7.5 and 12.5. The location of the new object is stored in the named location `myOblong`. This is illustrated in Fig. 7.3.

Now every time we want to refer to our new `Oblong` object we can use the variable name `myOblong`.

As is the case with the declaration and initialization of simple types, Java allows us to declare a reference and create a new object all in one line:

```
Oblong myOblong = new Oblong(7.5, 12.5);
```

There are all sorts of ways that we can define constructors (for example, in a `BankAccount` class we might want to set the overdraft limit to a particular value when the account is created) and we shall see examples of these as we go along. You have of course already seen another example of this, namely with the `Scanner` class:

```
Scanner keyboard = new Scanner(System.in);
```

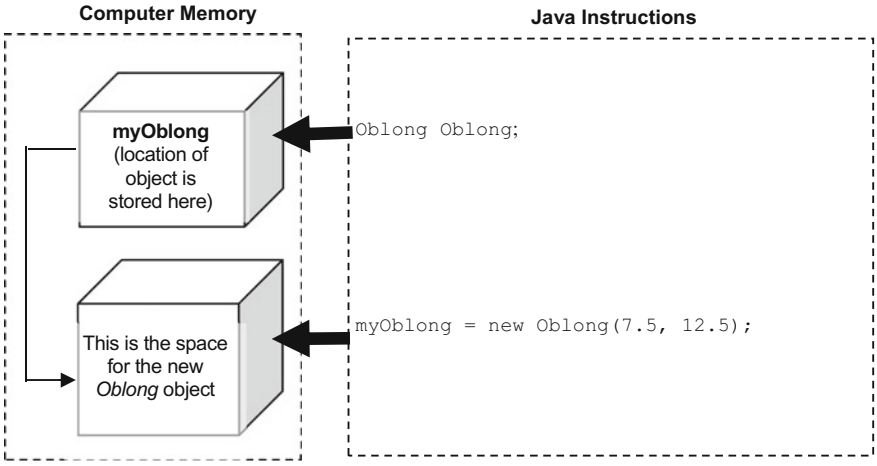


Fig. 7.3 Creating a new object

You can understand now how this line creates a new `Scanner` object, `keyboard`, by calling the constructor. The parameter that we are sending in, `System.in`, represents a keyboard object and by using this parameter we are associating the new `Scanner` object with the keyboard.

You will see in the next chapter that when a class is written we make sure that no program can assign values to the attributes directly. In this way the data in the class is protected. Protecting data in this way is known as **encapsulation**. The *only way* to interact with the data is via the methods of the class.

This means that in order to use a class all we need to know are details about its methods: their names, what the methods are expected to do, and also their **inputs** and **outputs**.[1] In other words you need to know what parameters the method requires, and its return type. Once we know this we can interact with the class by using its methods—and it is important to understand that the *only* way we can interact with a class is via its methods.

Table 7.1 lists all the methods of our `Oblong` class with their inputs and outputs —including the constructor.

As far as our `Oblong` class is concerned we have, as expected, provided two methods which will return values for the area and perimeter of the oblong respectively. However, the class wouldn't be very useful if we did not have some means of giving values to the length and height of the oblong. As you have seen we do this initially via the constructor, but we might also want to be able to change these values during the course of a program. We have therefore provided methods called `setLength` and `setHeight` so that we can *write* values to the attributes. It is very likely that we will want to display these values—we have therefore provided methods to return, or *read*, the values of the attributes. These we have called `getLength` and `getHeight`.

You have used methods of the `Scanner` class on many occasions, for example:

```
x = keyboard.nextInt();
```

You can see that in order to call a method of one class from another class we use the name of the object (in this case `keyboard`) together with the name of the method (`nextInt`) separated by a full stop (often referred to as the **dot operator**).

In the case of the `Oblong` class, we might, for example, call the `setLength` method with a statement such as:

```
myOblong.setLength(5.0);
```

[1] A list of a method's inputs and outputs is often referred to as the method's **interface**—though this should not be confused with the *user interface*, the meaning of which we described in the first chapter.

Table 7.1 The methods of the *Oblong* class

Method	Description	Inputs	Output
`Oblong`	The constructor	Two items of data, both of type **`double`**, representing the length and height of the oblong respectively	Not applicable
`setLength`	Sets the value of the length of the oblong	An item of type **`double`**	None
`setHeight`	Sets the value of the height of the oblong	An item of type **`double`**	None
`getLength`	Returns the length of the oblong	None	An item of type **`double`**
`getHeight`	Returns the height of the oblong	None	An item of type **`double`**
`calculateArea`	Calculates and returns the area of the oblong	None	An item of type **`double`**
`calculatePerimeter`	Calculates and returns the perimeter of the oblong	None	An item of type **`double`**

In Chap. 5, when we called the methods from *within* a class, we used the name of the method on its own. In actual fact, what we were doing is form of shorthand. When we write a line such as

```
demoMethod(x);
```

we are actually saying call `demoMethod`, which is a method of *this* class. In Java there exists a special keyword **`this`**. The keyword **`this`** is used within a class when we wish to refer to an attribute of the class itself, rather than an attribute of some other class. The line of code above is actually shorthand for:

```
this.demoMethod(x);
```

As your programming skills advance, you will find that there are occasions when you actually have to use the **`this`** keyword, rather than simply allowing it to be assumed.

You should be aware of the fact that, just as you cannot use a variable that has not been initialized, you cannot call a method of an object if no storage is allocated for the object; so watch out for this happening in your programs—it would cause a problem at run-time. In Java, when a reference is first created without assigning it to a new object in the same line, it is given a special value of **`null`**; a **`null`** value indicates that no storage is allocated. We can also *assign* a **`null`** value to a reference at any point in the program, and test for it as in the following example:

```
Oblong myOblong; // at this point myOblong has a null value
myOblong = new Oblong(5.0, 7.5); // create a new Oblong object with length 5.0 and height 7.5

// more code goes here

myOblong = null; // re-assign a null value
if(myOblong == null) // test for null value
{
    System.out.println("No storage is allocated to this object");
}
```

In the next section we will write a program that creates an `Oblong` object and then uses the methods described in Table 7.1 to interact with this object.

7.5 The *OblongTester* Program

The following program shows how the `Oblong` class can be used by another class, in this case a class called `OblongTester`. Study the program code and then we will discuss it.

OblongTester

```
import java.util.Scanner;

public class OblongTester
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);

        // declare two variables to hold the length and height of the Oblong as input by the user
        double oblongLength, oblongHeight;

        // declare a reference to an Oblong object
        Oblong myOblong;

        // now get the values from the user
        System.out.print("Please enter the length of your Oblong: ");
        oblongLength = keyboard.nextDouble();
        System.out.print("Please enter the height of your Oblong: ");
        oblongHeight = keyboard.nextDouble();

        // create a new Oblong object
        myOblong = new Oblong(oblongLength, oblongHeight);

        /* use the various methods of the Oblong class to display the length, height, area and
           perimeter of the Oblong */
        System.out.println("Oblong length is " + myOblong.getLength());
        System.out.println("Oblong height is " + myOblong.getHeight());
        System.out.println("Oblong area is " + myOblong.calculateArea());
        System.out.println("Oblong perimeter is " + myOblong.calculatePerimeter());
    }
}
```

Let's analyse the `main` method line by line. After creating the new `Scanner` object, the method goes on to declare two variables:

```
double oblongLength, oblongHeight;
```

As you can see, these are of type **`double`** and they are going to be used to hold the values that the user chooses for the length and height of the oblong.

The next line declares the `Oblong` object:

```
Oblong myOblong;
```

After getting the user to enter values for the length and height of the oblong we have this line of code:

```
myOblong = new Oblong(oblongLength, oblongHeight);
```

Here we have called the constructor and sent through the length and height as entered by the user.

Now the next line:

```
System.out.println("Oblong Length is " + myOblong.getLength());
```

This line displays the length of the oblong. It uses the method of `Oblong` called `getLength`, and as we said in the previous section to do this we use the dot operator to separate the name of the object and the name of the method.

The next three lines are similar:

```
System.out.println("Oblong height is " + myOblong.getHeight());
System.out.println("Oblong area is " + myOblong.calculateArea());
System.out.println("Oblong Perimeter is "+ myOblong.calculatePerimeter());
```

We have called the `getHeight` method, the `calculateArea` method and the `calculatePerimeter` method to display the height, area and perimeter of the oblong on the screen.

You might have noticed that we haven't used the `setLength` and `setHeight` methods—that is because in this program we didn't wish to change the length and height once the oblong had been created—but this is not the last you will see of our `Oblong` class—and in future programs these methods will come in useful.

Now we can move on to look at using some other classes. The first is not one of our own, but the built-in `String` class provided with all versions of Java.

7.6 Strings

You know from Chap. 1 that a string is a sequence of characters—like a name, a line of an address, a car registration number, or indeed any meaningless sequence of characters such as "h83hdu2&e£8". Java provides a `String` class that allows us to use and manipulate strings.

As we shall see in a moment, the `String` class has a number of constructors—but in fact Java actually allows us to declare a string object in the same way as we declare variables of simple type such as **`int`** or **`char`**. You should remember of course that `String` is a class, and starts with a capital letter. For example we could make the following declaration:

```
String name;
```

and we could then give this string a value:

```
name = "Quentin";
```

We could also do this in one line:

```
String name = "Quentin";
```

We should bear in mind, however, that this is actually just a convenient way of declaring a `String` object by calling its constructor, which we would do like this with exactly the same effect:

```
String name = new String("Quentin");
```

You should be aware that the `String` class is the only class that allows us to create new objects by using the assignment operator in this way.

7.6.1 Obtaining Strings from the Keyboard

In order to get a string from the keyboard, you should use the `next` method of `Scanner`. However, a word of warning here—when you do this you should not enter strings that include spaces, as this will give you unexpected results. We will show you in the next section a way to get round this restriction.

Below is a little program that uses the Java `String` class. Some of you might find it amusing (although others might not!).

StringTest

```
import java.util.Scanner;

public class StringTest
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        String name; // declaration of a String
        int age;
        System.out.print("What is your name? ");
        name = keyboard.next(); // the 'next' method is for String input
        System.out.print("What is your age? ");
        age = keyboard.nextInt();
        System.out.println();
        System.out.println("Hello " + name);
        // now comes the joke!!
        System.out.println("When I was your age I was "  + (age + 1));
    }
}
```

One thing to notice in this program is the way in which the + operator is used for two very different purposes. It is used with strings for concatenation—for example:

```
"Hello" + name
```

It is also used with numbers for addition—for example:

```
age + 1
```

Notice that we have had to enclose this expression in brackets to avoid any confusion:

```
System.out.println("When I was your age I was "  + (age + 1));
```

Here is a sample run from the above program:

```
What is your name? Aaron
What is your age? 15

Hello Aaron
When I was your age I was 16
```

7.6.2 The Methods of the *String* Class

The `String` class has a number of interesting and useful methods, and we have listed some of them in Table 7.2.

Table 7.2 Some *String* methods

Method	Description	Inputs	Output
`length`	Returns the length of the string	None	An item of type **`int`**
`charAt`	Accepts an integer and returns the character at that position in the string. Note that indexing starts from zero, not 1! You have been using this method in conjunction with the `next` method of the `Scanner` class to obtain single characters from the keyboard	An item of type **`int`**	An item of type **`char`**
`substring`	Accepts two integers (for example `m` and `n`) and returns a copy of a chunk of the string. The chunk starts at position `m` and finishes at position `n-1`. Remember that indexing starts from zero. (Study the example below.)	Two items of type **`int`**	A `String` object
`concat`	Accepts a string and returns a new string which consists of the string that was sent in joined on to the end of the original string	A `String` object	A `String` object
`toUpperCase`	Returns a copy of the original string, all upper case	None	A `String` object
`toLowerCase`	Returns a copy of the original string, all lower case	None	A `String` object
`compareTo`	Accepts a string (say `myString`) and compares it to the object's string. It returns zero if the strings are identical, a negative number if the object's string comes before `myString` in the alphabet, and a positive number if it comes later	A `String` object	An item of type **`int`**
`equals`	Accepts an object (such as a `String`) and compares this to another object (such as another `String`). It returns **`true`** if these are identical, otherwise returns **`false`**	An object of any class	A **`boolean`** value
`equalsIgnoreCase`	Accepts a string and compares this to the original string. It returns **`true`** if the strings are identical (ignoring case), otherwise returns **`false`**	A `String` object	A **`boolean`** value
`startsWith`	Accepts a string (say `str`) and returns **`true`** if the original string starts with `str` and **`false`** if it does not (e.g. "hello world" starts with "h" or "he" or "hel" and so on)	A `String` object	A **`boolean`** value
`endsWith`	Accepts a string (say `str`) and returns **`true`** if the original string ends with `str` and **`false`** if it does not (e.g. "hello world" ends with "d" or "ld" or "rld" and so on)	A `String` object	A **`boolean`** value
`trim`	Returns a `String` object, having removed any spaces at the beginning or end	None	A `String` object

There are many other useful methods of the `String` class which you can look up. The following program provides examples of how you can use some of the methods listed above; others are left for you to experiment with in your practical sessions.

StringMethods

```
import java.util.Scanner;

public class StringMethods
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        // create a new string
        String str;
        // get the user to enter a string
        System.out.print("Enter a string without spaces: ");
        str = keyboard.next();
        // display the length of the user's string
        System.out.println("The length of the string is " + str.length());
        // display the third character of the user's string
        System.out.println("The character at position 3 is " + str.charAt(2));
        // display a selected part of the user's string
        System.out.println("Characters 2 to 4 are " + str.substring(1,4));
        // display the user's string joined with another string
        System.out.println(str.concat(" was the string entered"));
        // display the user's string in upper case
        System.out.println("This is upper case: " + str.toUpperCase());
        // display the user's string in lower case
        System.out.println("This is lower case: " + str.toLowerCase());
    }
}
```

A sample run:

```
Enter a string without spaces: Europe
The length of the string is 6
The character at position 3 is r
Characters 2 to 4 are uro
Europe was the string entered
This is upper case: EUROPE
This is lower case: europe
```

7.6.3 Comparing Strings

When comparing two objects, such as `Strings`, we should do so by using a method called `equals`. We should *not* use the equality operator (==); this should be used for comparing primitive types only. If, for example, we had declared two strings, `firstString` and `secondString`, we would compare these in, say, an **`if`** statement as follows:

```
if(firstString.equals(secondString))
{
    // more code here
}
```

Using the equality operator (==) to compare strings is a very common mistake that is made by programmers. Doing this will not result in a compilation error, but it won't give you the result you expect! The reason for this is that all you are doing is finding out whether the objects occupy the same address space in memory—what you actually want to be doing is comparing the actual value of the string attributes of the objects.

Notice that an object of type `String` can also be used within a **`switch`** statement to check to see if it is equal to one of several possible `String` values. The simple `StringCheckWithSwitch` program below illustrates this by giving a meaning for three symbols on a game controller for a particular game:

StringCheckWithSwitch

```
import java.util.Scanner;

public class StringCheckWithSwitch
{
    public static void main(String[] args)
    {
       Scanner keyboard = new Scanner (System.in);
       String symbol;
       // get symbol from user
       System.out.println("Enter the symbol(square/circle/triangle)");
       symbol = keyboard.next();
       // use String object in switch
       switch(symbol)
       {
            case "square": System.out.println("ATTACK"); break;
            case "circle": System.out.println("BLOCK"); break;
            case "triangle": System.out.println("JUMP"); break;
            default: System.out.println("Invalid Choice");
       }
    }
}
```

Here is a sample run from the program:

```
Enter the symbol(square/circle/triangle)
triangle
JUMP
```

Here is another sample run from the program:

```
Enter the symbol(square/circle/triangle)
square
ATTACK
```

The `String` class also has a very useful method called `compareTo`. As you can see from Table 7.2 this method accepts a string (called `myString` for example) and compares it to the string value of the object itself. It returns zero if the strings are identical, a negative number if the original string comes before `myString` in the alphabet, and a positive number if it comes later.

The program below provides an example of how the `compareTo` method is used.

StringComparison

```
import java.util.Scanner;

public class StringComparison
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        String string1, string2;
        int comparison;

        // get two strings from the user
        System.out.print("Enter a String: ");
        string1 = keyboard.next();
        System.out.print("Enter another String: ");
        string2 = keyboard.next();

        // compare the strings
        comparison = string1.compareTo(string2);
        if(comparison < 0) // compareTo returned a negative number
        {
            System.out.println(string1 + " comes before " + string2 + " in the alphabet");
        }
        else if(comparison > 0) // compareTo returned a positive number
        {
            System.out.println(string2 + " comes before " + string1 + " in the alphabet");
        }
        else // compareTo returned zero
        {
            System.out.println("The strings are identical");
        }
    }
}
```

Here is a sample run from the program:

```
Enter a String: hello
Enter another String: goodbye
goodbye comes before hello in the alphabet
```

You should note that (as with the `equals` method) the `compareTo` method is case-sensitive—upper-case letters will be considered as coming before lower-case letters (their Unicode value is lower). If you are not interested in the case of the letters, you should convert both strings to upper (or lower) case before comparing them.

If all you are interested in is whether the strings are identical, it is easier to use the `equals` method. If the case of the letters is not significant you can use `equalsIgnoreCase`.

7.6.4 Entering Strings Containing Spaces

As we mentioned above there is a problem with using the `next` method of `Scanner` when we enter strings that contain spaces. If you try this you will see that the resulting string stops at the first space, so if you enter the string "Hello world" for example, the resulting string would actually be "Hello".

To enter a string that contains spaces you need to use the method `nextLine`. Unfortunately however there is also an issue with this. If the `nextLine` method is used after a `nextInt` or `nextDouble` method, then it is necessary to create a separate `Scanner` object (because using the same `Scanner` object will make

your program behave erratically). So, if your intention is that the user should be able to enter strings that contain spaces, the best thing to do is to declare a separate `Scanner` object for string input. This is illustrated below:

StringExample2

```
import java.util.Scanner;

public class StringExample2
{
    public static void main(String[] args)
    {
        double d;
        int i;
        String s;
        Scanner keyboardString = new Scanner(System.in); // Scanner object for string input
        Scanner keyboard = new Scanner(System.in); // Scanner object for all other types of input
        System.out.print("Enter a double: ");
        d = keyboard.nextDouble();
        System.out.print("Enter an integer: ");
        i = keyboard.nextInt();
        System.out.print("Enter a string: ");
        s = keyboardString.nextLine(); // use the Scanner object reserved for string input
        System.out.println();
        System.out.println("You entered: ");
        System.out.println("Double: " + d);
        System.out.println("Integer: " + i);
        System.out.println("String: " + s);
    }
}
```

Here is a sample run from this program:

```
Enter a double: 3.4
Enter an integer: 10
Enter a string: Hello world

You entered:
Double: 3.4
Integer: 10
String: Hello world
```

7.7 Our Own *Scanner* Class for Keyboard Input

It might have occurred to you that using the `Scanner` class to obtain keyboard input can be a bit of a bother.

- it is necessary to create a new `Scanner` object in every method that uses the `Scanner` class;
- there is no simple method such as `nextChar` for getting a single character like there is for the **`int`** and **`double`** types;
- as we have just seen there is an issue when it comes to entering strings containing spaces.

Table 7.3 The input methods of the *EasyScanner* class

Java type	`EasyScanner` method
`int`	`nextInt()`
`double`	`nextDouble()`
`char`	`nextChar()`
`String`	`nextString()`

To make life easier, we have created a new class which we have called `EasyScanner`. In the next chapter we will "look inside" it to see how it is written—in this chapter we will just show you how to use it. The methods of `EasyScanner` are described in Table 7.3.

To make life really easy we have written the class so that we don't have to create new `Scanner` objects in order to use it (that is taken care of in the class itself)—and we have written it so that you can simply use the name of the class itself when you call a method (you will see how to do this in the next chapter). The following program demonstrates how to use these methods.

EasyScannerTester

```
public class EasyScannerTester
{
    public static void main(String[] args)
    {
        System.out.print("Enter a double: ");
        double d = EasyScanner.nextDouble(); // to read a double
        System.out.println("You entered: " + d);
        System.out.println();

        System.out.print("Enter an integer: ");
        int i = EasyScanner.nextInt(); // to read an int
        System.out.println("You entered: " + i);
        System.out.println();

        System.out.print("Enter a string: ");
        String s = EasyScanner.nextString(); // to read a string
        System.out.println("You entered: " + s);
        System.out.println();

        System.out.print("Enter a character: ");
        char c = EasyScanner.nextChar(); // to read a character
        System.out.println("You entered: " + c);
        System.out.println();
    }
}
```

You can see from this program how easy it is to call the methods, just by using the name of the class itself—for example:

```
double d = EasyScanner.nextDouble();
```

In the next chapter you will see how it is possible to do this.

Here is a sample run:

```
Enter a double: 23.6
You entered: 23.6
```

```
Enter an integer: 50
You entered: 50

Enter a string: Hello world
You entered: Hello world

Enter a character: B
You entered: B
```

You are now free to use the `EasyScanner` class if you wish. You can copy it from the website—as usual make sure it is in the right place for your compiler to find it.

7.8 The *Console* Class

In Chap. 1 we talked briefly about the possibility of running a program in a console, the text window provided by operating systems such as Windows™. We will discuss this in more detail in Chap. 19.

A special class called `Console` is provided (in the `java.io` library) as an alternative to `Scanner`, should you wish to use it—you should note however that the `Console` class is not suitable for running programs within an IDE. An example of a little program that uses the `Console` class is shown in Fig. 7.4.

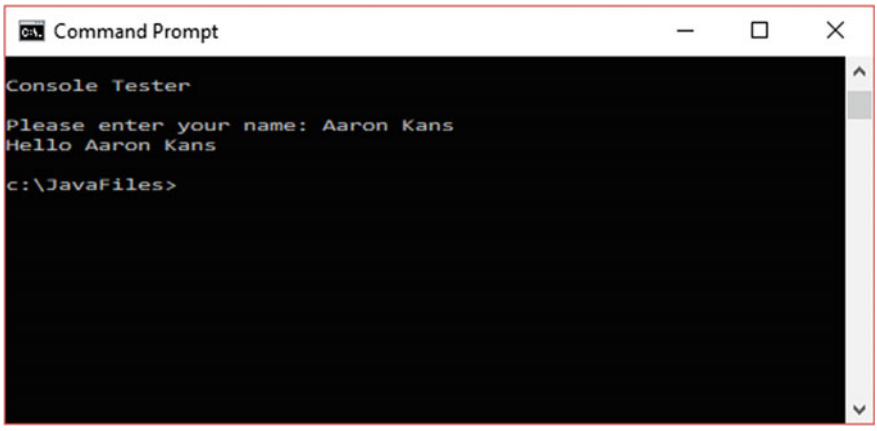


Fig. 7.4 Using the `Console` class

The program below shows how keyboard input is obtained with the `Console` class:

ConsoleTester

```
// demonstration of the console class for keyboard input

import java.io.Console;

public class ConsoleTester
{
    public static void main(String[] args)
    {
          System.out.println();
          System.out.println("Console Tester");
          System.out.println();
          Console con = System.console();
          String name; // declaration of a String
          name = con.readLine("Please enter your name: "); // allow user to enter name
          System.out.println("Hello " + name); // display a message to the user
    }
}
```

You can see from the example in Fig. 7.4 that there is no problem entering a string containing spaces; if, however, you wanted to use the `Console` class for entering **`double`**s or **`int`**s, you would have to enter a string and then convert this to the desired type. You will find out how to do this in Chap. 10.

7.9 The *BankAccount* Class

We have created a class called `BankAccount`, which you can download. This could be a very useful class in a real world application—for example as part of a financial control system. Once again you do not need to look at the details of how this class is coded in order to use it. You do need to know, however, that the class holds three pieces of information—the account number, the account name and the account balance. The first two of these will be `String` objects and the final one will be a variable of type **`double`**.

The methods are listed in Table 7.4.

The methods are straightforward, although you should pay particular attention to the `withdraw` method. Our simple `BankAccount` class does not allow for an overdraft facility, so, unlike the `deposit` method, which simply adds the specified amount to the balance, the `withdraw` method needs to check that the amount to be withdrawn is not greater than the balance of the account; if this were to be the case then the balance would be left unchanged. The method returns a **`boolean`** value to indicate if the withdrawal was successful or not. A **`boolean`** value of **`true`** would indicate success and **`boolean`** value of **`false`** would indicate failure. This enables a program that uses the `BankAccount` class to check whether the withdrawal has been made successfully. You can see how this is done in the program that follows, which makes use of the `BankAccount` class.

Table 7.4 The methods of the *BankAccount* class

Method	Description	Inputs	Output
`BankAccount`	A constructor. It accepts two strings and assigns them to the account number and account name respectively. It also sets the account balance to zero	Two `String` objects	Not applicable
`getAccountNumber`	Returns the account number	None	An item of type `String`
`getAccountName`	Returns the account name	None	An item of type `String`
`getBalance`	Returns the balance	None	An item of type **`double`**
`deposit`	Accepts an item of type **`double`** and adds it to the balance	An item of type **`double`**	None
`withdraw`	Accepts an item of type **`double`** and checks if there are sufficient funds to make a withdrawal. If there are not, then the method terminates and returns a value of **`false`**. If there are sufficient funds, however, the method subtracts the amount from the balance and returns a value of **`true`**	An item of type **`double`**	An item of type **`boolean`**

BankAccountTester

```
import java.util.Scanner;

public class BankAccountTester
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        double amount;
        boolean ok;

        BankAccount account1 = new BankAccount("99786754","Susan Richards");

        System.out.print("Enter amount to deposit: ");
        amount = keyboard.nextDouble();
        account1.deposit(amount);
        System.out.println("Deposit was made");
        System.out.println("Balance = " + account1.getBalance());
        System.out.println();


        System.out.print("Enter amount to withdraw: ");
        amount = keyboard.nextDouble();
        ok = account1.withdraw(amount); // get the return value of the withdraw method
        if(ok)
        {
            System.out.println("Withdrawal made");
        }
        else
        {
            System.out.println("Insufficient funds");
        }
        System.out.println("Balance = " + account1.getBalance());
        System.out.println();
    }
}
```

The program creates a `BankAccount` object and then asks the user to enter an amount to deposit. It then confirms that the deposit was made and shows the new balance.

It then does the same thing for a withdrawal. The `withdraw` method returns a **`boolean`** value indicating if the withdrawal has been successful or not, so we have assigned this return value to a **`boolean`** variable, `ok`:

```
ok = account1.withdraw(amount);
```

Depending on the value of this variable, the appropriate message is then displayed:

```
if(ok)
{
    System.out.println("Withdrawal made");
}
else
{
    System.out.println("Insufficient funds");
}
```

Two sample runs from this program are shown below. In the first the withdrawal was successful:

```
Enter amount to deposit: 1000
Deposit was made
Balance = 1000.0

Enter amount to withdraw: 400
Withdrawal made
Balance = 600.0
```

In the second there were not sufficient funds to make the withdrawal:

```
Enter amount to deposit: 1000
Deposit was made
Balance = 1000.0

Enter amount to withdraw: 1500
Insufficient funds
Balance = 1000.0
```

7.10 Arrays of Objects

In Chap. 6 you learnt how to create arrays of simple types such as **int** and **char**. It is perfectly possible, and often very desirable, to create arrays of objects. There, are, however, some important issues that we need to be aware of. We will illustrate this with a new version of the BankAccountTester from the previous section. In BankAccountTester2, instead of creating a single bank account, we have created several bank accounts by using an array. Take a look at the program, and then we will explain the important issues to you:

BankAccountTester2

```
public class BankAccountTester2
{
    public static void main(String[] args)
    {
        // create an array of references
        BankAccount[] accountList = new BankAccount[3];
        // create three new accounts, referenced by each element in the array
        accountList[0] = new BankAccount("99786754","Susan Richards");
        accountList[1] = new BankAccount("44567109","Delroy Jacobs");
        accountList[2] = new BankAccount("46376205","Sumana Khan");

        // make various deposits and withdrawals
        accountList[0].deposit(1000);
        accountList[2].deposit(150);
        accountList[0].withdraw(500);

        // print details of all three accounts
        for(BankAccount item : accountList)
        {
            System.out.println("Account number: " + item.getAccountNumber());
            System.out.println("Account name: " + item.getAccountName());
            System.out.println("Current balance: " + item.getBalance());
            System.out.println();
        }
    }
}
```

The first line of the main method looks no different from the statements that you saw in the last chapter that created arrays of primitive types:

```
BankAccount[] accountList = new BankAccount[3];
```

However, what is actually going on behind the scenes is slightly different. The above statement does *not* set up an array of BankAccount objects in memory; instead it sets up an array of *references* to such objects (see Fig. 7.5).

At the moment, space has been reserved for the three BankAccount *references* only, *not* the three BankAccount objects. As we told you earlier, when a reference is initially created it points to the constant **null**, so at this point each reference in the array points to **null**.

This means that memory would still need to be reserved for individual BankAccount objects each time we wish to link a BankAccount object to the array. We can now create new BankAccount objects and associate them with elements in the array as we have done with these lines:

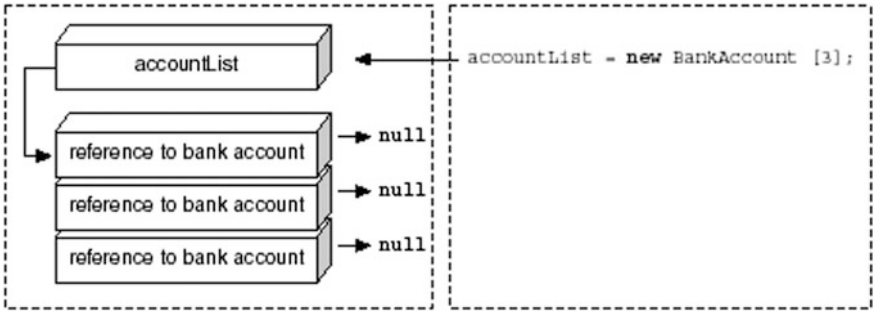


Fig. 7.5 The effect on computer memory of creating an array of objects

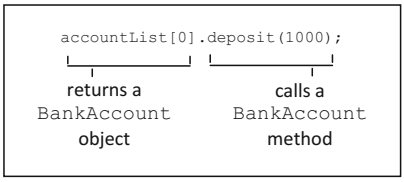
```
accountList[0] = new BankAccount("99786754","Susan Richards");
accountList[1] = new BankAccount("44567109","Delroy Jacobs");
accountList[2] = new BankAccount("46376205","Sumana Khan");
```

Three `BankAccount` objects have been created; the first one, for example, has account number of "99786754" and name "Susan Richards", and the reference at `accountList[0]` is set to point to it. This is illustrated in Fig. 7.6.

Once we have created these accounts, we make some deposits and withdrawals.

```
accountList[0].deposit(1000);
accountList[2].deposit(150);
accountList[0].withdraw(500);
```

Look carefully at how we do this. To call a method of a particular array element, we place the dot operator after the final bracket of the array index. This is made clear below:



Notice that in this case when we call the `withdraw` method we have decided not to check the **`boolean`** value returned.

```
accountList[0].withdraw(500); // return value not checked
```

It is not always necessary to check the return value of a method and you may ignore it if you choose.

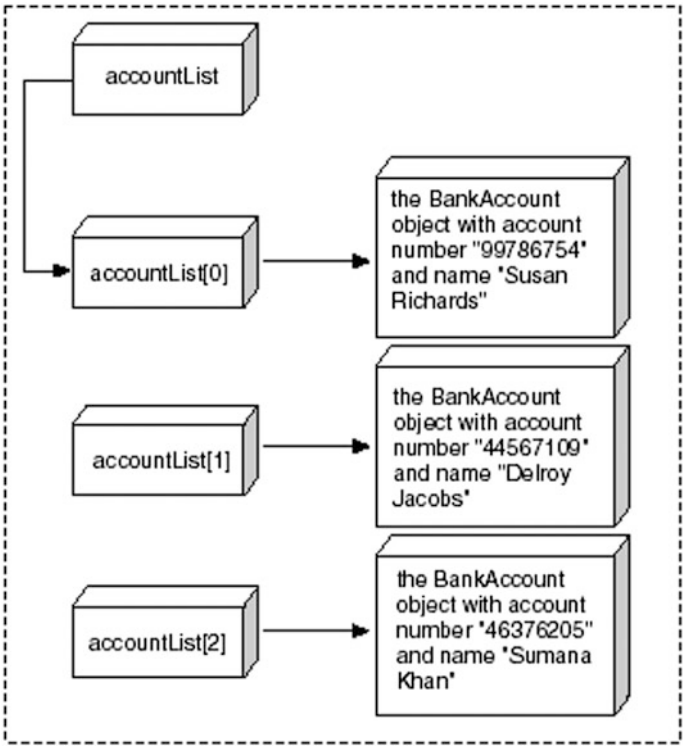


Fig. 7.6 Objects are linked to arrays by reference

Having done this we display the details of all three accounts. As we are accessing the entire array, we are able to use an enhanced **`for`** loop for this purpose; and since we are dealing with an array of `BankAccount` objects here, the type of the items is specified as `BankAccount`.

```
for(BankAccount item : accountList) // type of items is BankAccount
{
    System.out.println("Account number: " + item.getAccountNumber());
    System.out.println("Account name: " + item.getAccountName());
    System.out.println("Current balance: " + item.getBalance());
    System.out.println();
}
```

As you might expect, the output from this program is as follows:

```
Account number: 99786754
Account name: Susan Richards
Current balance: 500.0
```

```
Account number: 44567109
Account name: Delroy Jacobs
Current balance: 0.0

Account number: 46376205
Account name: Sumana Khan
Current balance: 150.0
```

7.11 The *ArrayList* Class

When you studied arrays in Chap. 6, it might have occurred to you that working with arrays can sometimes seem a bit cumbersome. That is because arrays are what we could term a "low-level" construct, which means that they mirror quite closely the workings of the computer itself, and match the way that data is stored in memory. As computer programmers it is vital that you have an understanding of arrays and how they work; however in modern day computing it is not uncommon for a programming language to provide higher level classes whose methods are closer to how we do things in real life; these classes deal with low level detail "behind the scenes", thus enabling the user to keep such detail at arms length.

One example of higher level classes are known as collection classes. These classes allow the programmer to deal with collections such as simple lists (like the `BankAccount` list from the previous section) with methods that will simply add a new item to the end of the list or easily remove an item from a particular position.

Collection classes are examples of classes known as **generic** classes. There are some quite advanced concepts involved in understanding generic classes, so most of this is left until the second semester. In particular we will examine collection classes in depth in Chap. 15. However, here we are going to introduce you to one such collection class called `ArrayList` (which resides in the `java.util` library) and provide you with just enough information to get you started.

You will see in the program that follows that there is some new notation involved. This is because when we declare an object of a collection class we need to indicate the type of object that it will hold, so you will see declarations like this:

```
ArrayList<BankAccount> accountList;
```

The type of object held is indicated within the angle brackets; so `accountList` will hold a list of `BankAccounts`. Similarly, if you wanted to declare an `ArrayList` object called `names` which was to hold `Strings`, you would do so like this:

```
ArrayList<String> names;
```

The program below, `BankAccountTester3`, rewrites `BankAccountTester2` using an `ArrayList` instead of an array.

BankAccountTester3

```
import java.util.ArrayList;

public class BankAccountTester3
{
    public static void main(String[] args)
    {
        // create an array of references
        ArrayList<BankAccount> accountList = new ArrayList<>();

        // create three new accounts, referenced by each element in the array
        accountList.add(new BankAccount("99786754","Susan Richards"));
        accountList.add(new BankAccount("44567109","Delroy Jacobs"));
        accountList.add(new BankAccount("46376205","Sumana Khan"));

        // make various deposits and withdrawals
        accountList.get(0).deposit(1000);
        accountList.get(2).deposit(150);
        accountList.get(0).withdraw(500);

        // print details of all three accounts
        for(BankAccount item : accountList)
        {
            System.out.println("Account number: " + item.getAccountNumber());
            System.out.println("Account name: " + item.getAccountName());
            System.out.println("Current balance: " + item.getBalance());
            System.out.println();
        }
    }
}
```

As you can see the full declaration of `accountList` is as follows:

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

When we call the constructor of a generic class we don't need to re-state the type of object held, so the angle brackets are left empty (this is sometimes referred to as a "diamond"). The compiler infers the type from the first part of the declaration—this is an example of **type inference** (more about this in Chap. 13).

The other thing you will notice is that we don't need to specify how many items an `ArrayList` will hold. It expands and contracts dynamically as we add and remove items.

To add items to the list we use the `add` method of `ArrayList`, which takes as a parameter the object that we are adding:

```
accountList.add(new BankAccount("99786754","Susan Richards"));
accountList.add(new BankAccount("44567109","Delroy Jacobs"));
accountList.add(new BankAccount("46376205","Sumana Khan"));
```

To make deposits and withdrawals, we need to retrieve the individual items that we require. We do this with the `get` method of `ArrayList`, which takes as a parameter the particular index (starting with zero, as with arrays):

```
accountList.get(0).deposit(1000);
accountList.get(2).deposit(150);
accountList.get(0).withdraw(500);
```

Finally we display the items—the enhanced for loop works nicely with classes such as `ArrayList`, so the code for displaying the bank accounts is the same as in the original program:

```
for(BankAccount item : accountList)
{
    System.out.println("Account number: " + item.getAccountNumber());
    System.out.println("Account name: " + item.getAccountName());
    System.out.println("Current balance: " + item.getBalance());
    System.out.println();
}
```

One final word about collection classes such as `ArrayList`. These classes cannot be used to hold primitive types such as **`int`** and **`double`**. But don't worry—there is a way around this using what are known as **wrapper** classes. These will be introduced to you in Chap. 9.

7.12 Self-test Questions

1. Examine the program below and then answer the questions that follow:

```
public class SampleProgram
{
     public static void main(String[] args)
     {
         Oblong oblong1 = new Oblong(3.0, 4.0);
         Oblong oblong2 = new Oblong(5.0, 6.0);
         System.out.println("The area of oblong1 is " + oblong1.calculateArea());
         System.out.println("The area of oblong2 is " + oblong2.calculateArea());
     }
}
```

   (a) By referring to the program above distinguish between a *class* and an *object*.

   (b) By referring to the program above explain the purpose of the *constructor*.

   (c) By referring to the program above explain how you call the method of one class from another class.

   (d) What output would you expect to see from the program above?

2. (a) Write the code that will create two `BankAccount` objects, `acc1` and `acc2`. The account number and account name of each should be set at the time the object is created.

   (b) Write the lines of code that will deposit an amount of 200 into `acc1` and 100 into `acc2`.

(c) Write the lines of code that attempt to withdraw an amount of 150 from `acc1` and displays the message "WITHDRAWAL SUCCESSFUL" if the amount was withdrawn successfully and "INSUFFICIENT FUNDS" if it was not.

(d) Write a line of code that will display the balance of `acc1`.

(e) Write a line of code that will display the balance of `acc2`.

3. In what way does calling methods from the `EasyScanner` class differ from calling methods from the other classes you have met (`BankAccount`, `Oblong`, `String` and `Scanner`)?

4. Consider the following fragment of code that initializes one string constant with a password ("java") and creates a second string to hold the user's guess for the password. The user is then asked to enter their guess:

```
String final PASSWORD = "java"; // set password
String guess; // to hold user's guess
System.out.print("Enter guess: ");
```

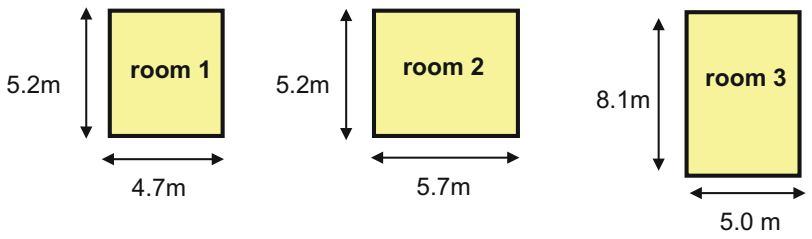
(a) Write a line of code that uses the `EasyScanner` class to read the guess from the keyboard.

(b) Write the code that displays the message "CORRECT PASSWORD" if the user entered the correct password and "INCORRECT PASSWORD" if not.

5. How do arrays of objects differ from arrays of primitive types?

6. (a) Declare an array called `rooms`, to hold three `Oblong` objects. Each `Oblong` object will represent the dimensions of a room in an apartment.

(b) The three rooms in the apartment have the following dimensions:
Add three appropriate `Oblong` objects to the `rooms` array to represent these 3 rooms.



(c) Write the line of code that would make use of the `rooms` array to display the area of room 3 to the screen.

7. Repeat the previous question using and `ArrayList` instead of an array.

7.13 Programming Exercises

In order to tackle these exercises make sure that the classes `Oblong`, `BankAccount` and `EasyScanner` have been copied from the website and placed in the correct directory for your compiler to access them.

1. (a) Implement the program given in self-test question 1 and run it to confirm your answer to part (d) of that question.
   (b) Adapt the program above so that the user is able to set the length and height of the two oblongs. Make use of the `EasyScanner` class to read in the user input.

2. Consider a program to enter and confirm a suitable code name for an agent. Declare two string objects, called `codeName` and `confirm` and then

   (a) Prompt to get the user to enter a suitable name into the `codeName` string;

   (b) Use a **`while`** loop to ensure that the string entered is greater than 6 characters in length, if it is not print "INVALID CODENAME" and ask the user to re-enter a code name;

   (c) Once a valid code name has been entered ask the user to re-enter the code name into the `confirm` string and then use an **`if else`** statement to ensure that the string entered matches the original code name; if it does, print a message "CODE NAME CONFIRMED" otherwise print a message saying "CODE NAME MIS-MATCH";

   (d) Use the `charAt` method to ensure that the code name ends with an 'X' character;

   (e) Finally use the `startsWith` method to ensure that, as well as being greater than 6 characters in length, the code name entered also starts with the words "Agent".

3. Adapt the `StringComparison` program from Sect. 7.6.3, which compares two strings, in the following ways:

   (a) Rewrite the program so that it ignores case;

   (b) Rewrite the program, using the `equals` method, so that all it does is to test whether the two strings are the same;

   (c) Repeat (b) using the `equalsIgnoreCase` method;

   (d) Use the `trim` method so that the program ignores leading or trailing spaces.

4. Design and implement a program that performs in the following way:

- When the program starts, two bank accounts are created, using names and numbers which are written into the code;
- The user is then asked to enter an account number, followed by an amount to deposit in that account;
- The balance of the appropriate account is then updated accordingly—or if an incorrect account number was entered a message to this effect is displayed;
- The user is then asked if he or she wishes to make more deposits;
- If the user answers does wish to make more deposits, the process continues;
- If the user does not wish to make more deposits, then details of both accounts (account number, account name and balance) are displayed.

5. Write a program that creates an array of `Oblong` objects to represent the dimensions of rooms in an apartment as described in self test question 6. The program should allow the user to:

   - Determine the number of rooms;
   - Enter the dimensions of the rooms;
   - Retrieve the area and dimensions of any of the rooms.

6. Repeat the previous question making use of the `ArrayList` class.