54

Objects and Classes

In Java, we place all our code in "classes". Each class contains one or more named blocks of code known as methods, and the group of methods provides all the functionality that's required to handle data (or objects) of a particular type.

Java would be very limited if data could only be held in the eight different primitive types of variables. Although they can form the fundamental building blocks, you really want something much more. You want to be able to define your own data type, create variables to contain data of that type, and then perform operations on the variables, secure in the knowledge that all of the operations are sensible for that type of data.

54.1 Using an instance of a class

You declare a variable to be not an **int** or a **double** but, perhaps, a "**Film**":

```
Film Kids;
```

There you go, a variable that can contain a film, and the name of the variable is "**Kids**".

Just like the declaration

```
int number;
```

tells Java that number is an **int**, but doesn't set any value into it, so the same thing applies to our **Kids** variable. Let's put something into our **Kids** variable, which we'll do by running a method that can be used for any variable of type **Film**.

```
Kids = new Film("Shrek",133);
```

"Create an object of type **Film**; the two parameters are 'Shrek' and '133', and store the **Film** into the variable called '**Kids**'."

There's going to be a certain number of things that we can do on a variable of type **Film**,[1] and if we want to do those things we need to do them by method calls. For example:

```
System.out.println("The kids are watching "+ Kids.getcalled());
```

will output

```
The kids are watching Shrek
```

This evening, the adults are going to watch "Road to Perdition" and the children will be watching "Shrek". Let's put both films into a class, and see how much earlier the adults will be through:

Figure 297 Running "Eve"

```
bash-2.04$ java Eve
The kids are watching Shrek
The big kids are watching Road to Perdition
The adults will be through 16 minutes ahead
bash-2.04$
```

[1] and we need it defined somewhere, but that's a story later in this module

And here's the class that we wrote:

```
public class Eve {
        public static void main(String [] args) {

        // Set up two film objects

        Film Kids;
        Film Adults;

        Kids = new Film("Shrek",133);
        Adults = new Film("Road to Perdition",117);

        System.out.println("The kids are watching "+
                        Kids.getcalled());
        System.out.println("The big kids are watching "+
                        Adults.getcalled());

        int before = Kids.getminutes() - Adults.getminutes();

        System.out.println("The adults will be through "+
                                before+ " minutes ahead");
        }
}
```

Looks good. We've used two objects of type **Film**, set them up, called information back, etc. As programmers of this application, we needed to know what methods we could call in the **Film** class, and what the parameters were to the methods, but we didn't need to know how the class works internally.

Chances are that the **Film** class is also in use by some other programmers and applications. Good; there's no point in rewriting code, and there's no point in everyone having to learn the low-level details about how to handle films. Provided that the **Film** class was written and tested well and does what's needed, it's a building block that has a very wide use.

Some detail of using an object

"**new**" is a reserved word in Java. When our program says "**new Film**" it's an instruction for Java to set up the memory to hold everything about a film, and to pass back some sort of variable[1] through which the information can be later accessed.

The **getcalled** and **getminutes** methods are names that just happen to be defined in class **Film** as methods that can be called on an object of type **Film**, and the syntax for the call is **objectname.method()**. If the method has any extra parameters, they'll be passed in within the round brackets as you can see (for example) in the **println** calls. **println** is a method in just the same way that **getcalled** is, it's just that **println** is a part of the standard library and **getcalled** isn't.

[1] sometimes referred to as an instance

54.2 Writing your own class

In order for the application we've just looked at to work, someone had to write and provide the Film class, not too hard as it's just another class:

```
public class Film {

        String called;
        int minutes;

        public Film (String title, int length) {
                called = title;
                minutes = length;
                }

        public int getminutes() {
                return minutes;
                }

        public String getcalled() {
                return called;
                }

        public void setminutes(int length) {
                minutes = length;
                }
        }
```

First, note there's no **`main`** method. The **`Film`** class isn't written to run as a stand-alone application, so it doesn't need such a method. What does it have?

It has a method with the same name as the name of the class – **`"Film`**". By definition in Java, this is the **`constructor`** method that is run when the **`new`** keyword is used on the class. The constructor does not have a return type specified like all the other methods in this and previous examples.[1]

What information do we want to be able to store about each and every film object that we create? In our simple example, the title and the length in minutes. If we were to declare variables to grab this information within our constructor, those variables would be lost as soon as our constructor finished running – a sorry state of affairs. Instead, we have declared all the variables we want in each object at the very top of the class:

```
String called;
int minutes;
```

For each object of type **`Film`** that we create, two new variables ("**`called`**" and "**`minutes`**") will be created. When we call any of the other methods that we have defined on the class, it will choose which **`called`** or **`minutes`** variable to use depending on which instance of the class we have called the method on.

[1] `getminutes` returns an `int`, `setminutes` is specified as void to state that it has nothing to return, etc.

Exercise

You're going to be working with vehicles (cars, buses, train carriages etc.). Create a class of type Vehicle with three parameters – a String for the owner's name, the number of passengers it can take, and the maximum speed it can travel. Also write some get methods to let you retrieve data.

Write a test program in which you create three Vehicles – a car with a maximum capacity of five and a top speed of 70 m.p.h, a railway carriage with a maximum capacity of 45 and a top speed of 120 miles per hour, and a bus with a capacity of 51 and a top speed of 55 m.p.h.

You have 30 people to convey 100 miles. How long is it going to take you to do that with your train carriage, and how long would it take you with your bus?

54.3 Enhancements to the basic class structure

Class or static methods

How many films do we have? In our example, we had two but it's not immediately obvious how we could find that out from a method call to the class. We need a new type of variable or method – a class variable or method, which applies to the class as a whole and not to any particular instance of the class. Such variables and methods are referred to as **`static`**, since Java always uses the same variable or method no matter which member of the class they're called on, or indeed if they're called on the class as a whole.

Direct variable access

It can be a real pain to have to write **`get`** and **`set`** methods for every property, so Java allows you to directly access the variables that relate to each instance of a class, provided that they have open enough privileges.[1]

Object oriented design purists may not be thrilled about this, feeling that everything should be done through a method. In practical use it works fine, and if you're particularly concerned you can declare variables private rather than public which will stop any users in their tracks.

this

Very often, you'll want to give a parameter the same variable name as a variable in the current instance of your class; our earlier example was somewhat fiddled when we wrote:

```
        public Film (String title, int length) {
                called = title;
                minutes = length;
                }
```

as it would have been much cleaner if we had been able to use the word "**`title`**" for both the parameter and the internal variable within the instance.

If we declare a parameter with the same name as such an instance variable, then any references we make will be to the parameter. We can force Java to use the instance variable by writing "**`this`**" in front of the variable name when appropriate.

Overloading

In Java, when you call a method you must get the number and type of parameters correct; it's not just the name that identifies the method, but the name *and* the parameter list. This means that you can write more than one method with the same name, very useful if you want to have a method which you specify "short" and leave it to default the final parameter(s). This can be done equally well with the **`constructor`**, **`static`** or **`dynamic`** methods.

Important to note: In Java, when you call a method you must get the number and type of parameters correct.

An example

We've modified our earlier example to include all the extra facilities that we've talked about since the last exercise – static methods and variables, this, overloading, and direct access to variables. Here's the object class (now called "Film2"):

```
public class Film2{

        String called;
        public int minutes;
        static int count = 0;
```

[1] that word "public" that we've been using is more than enough!

```
public Film2 (String called, int minutes) {
        filmset(called, minutes);
        }

public Film2 (String called) {
        filmset(called, 125);
        }

private void filmset(String called, int minutes) {
        this.called = called;
        this.minutes = minutes;
        count++;
        }

public String getcalled() {
        return called;
        }

public void setminutes(int length) {
        minutes = length;
        }

public static int getcount() {
        return count;
        }

}
```

Here's our test harness:

```
public class Eve2 {

        public static void main(String [] args) {

        // Set up two film objects

        Film2 Kids;
        Film2 Adults;

        Kids = new Film2("Shrek");
        Adults = new Film2("Road to Perdition",117);

        System.out.println("The kids are watching "+
                        Kids.getcalled());
        System.out.println("The big kids are watching "+
                        Adults.getcalled());

        int before = Kids.minutes - Adults.minutes;

        System.out.println("The adults will be through "+
                                before+ " minutes ahead");

        System.out.println("You have a total of " +
                                Film2.getcount() + " films");

        }
}
```

And here's the result from running it:

```
bash-2.04$ java Eve2
The kids are watching Shrek
The big kids are watching Road to Perdition
The adults will be through 8 minutes ahead
You have a total of 2 films
bash-2.04$
```

Figure 298 Running public class Eve2

54.4 Naming conventions

It's common practice to use different capitalisation standards to differentiate between primitive and object variables, and the names of methods.

Primitive variables are usually written all lowercase.

Classes and object variables start with an uppercase letter, and are then continue in lowercase.

Method names start with a lowercase letter, but have capitals for the start of each intermediate word in the method name. Examples:

`numberofobjects`	primitive
`Numberofobjects`	object
`numberOfObjects`	method

Important to note: Primitive variables are usually written all lowercase. Classes and object variables start with an uppercase letter, and are then continue in lowercase. Method names start with a lowercase letter, but have capitals for the start of each intermediate word in the method name.

Exercise

Modify your vehicle class so that you can call a method to ask how many seats (in total) you have created, and test that method by adding a total seat report to the test harness.

Add in a second constructor that takes no parameters at all; if you call up a vehicle and don't specify anything, it's to be a car with four seats, top speed 70 m.p.h. and have an "unknown" owner.