

Notes from Well House Consultants

These notes are written by Well House Consultants and distributed under their Open Training Notes License. If a copy of this license is not supplied at the end of these notes, please visit

*<http://www.wellho.net/net/whcotnl.html>
for details.*

1.1 Well House Consultants

Well House Consultants provides niche training, primarily but not exclusively in Open Source programming languages. We offer public courses at our training centre and private courses at your offices. We also make some of our training notes available under our "Open Training Notes" license, such as we're doing in this document here.

1.2 Open Training Notes License

With an "Open Training Notes License", for which we make no charge, you're allowed to print, use and distribute these notes provided that you retain the complete and unaltered license agreement with them, including our copyright statement. This means that you can learn from the notes, and have others learn from them too.

You are NOT allowed to charge (directly or indirectly) for the copying or distribution of these notes, nor are you allowed to charge for presentations making any use of them.

1.3 Courses presented by the author

If you would like us to attend a course (Java, Perl, Python, PHP, Tcl/Tk, MySQL or Linux) presented by the author of these notes, please see our public course schedule at

<http://www.wellho.net/course/index.html>

If you have a group of 4 or more trainees who require the same course at the same time, it will cost you less to have us run a private course for you. Please visit our onsite training page at

<http://www.wellho.net/course/otc.html>

which will give you details and costing information

1.4 Contact Details

Well House Consultants may be found online at

<http://www.wellho.net>

graham@wellho.net

technical contact

lisa@wellho.net

administration contact

Our full postal address is

404 The Spa

Melksham

Wiltshire

UK SN12 6QL

Phone +44 (0) 1225 708225

Fax +44 (0) 1225 707126

Extending Classes and More

Learn how to implement inheritance in Java, and how Java handles Polymorphism. Java also provides the concepts of abstract classes and interfaces, which are also covered by this module.

<i>Extended classes</i>	4
<i>Abstract Classes</i>	7
<i>The universal superclass.</i>	11
<i>Interfaces</i>	12
<i>The final modifier</i>	15

Let's say that we have a class called "Film". There are certain properties that will apply to any and every film. Each will have a title and a running length, for example. But there will be other features that apply only to certain types of film, or apply in different ways.

For example, how much does it cost to see a film? Let's write a `getcost` method. We know it's going to vary. We may have a "unitprice" variable, but that's per day if you're hiring the film from Blockbuster, a one-off purchase price if you're getting it from Amazon, and a per-person rate if you're seeing it at the cinema. Watching on BBC may be regarded as being free if you don't want to get involved in apportioning out the license fee.

How then am I going to handle an application that needs to make use of several different types of films? I could write a class called "Film" then take a series of copies and modify each copy. But what a nightmare that would be if I were to then want to make some changes to my original code. Fortunately, there's an alternative.

2.1 Extended classes

I'm going to write a new class called "HireFilm", but rather than copy the code from Film into it, I'm going to declare it to extend Film. Then all the code of the Film class will be available in my HireFilm class, together with any extras I provide. If there's something that I really don't want in the original Film class when it's extended, I can override it in the extended class.

Some terminology:

The class on which my new class is based (Film) is called the **BASE CLASS**.

The new class (HireFilm) which extends my base class is called the **SUBCLASS**.

You might have expected the subclass and the base class to be one and the same as they're both "lower" or "down" words, but that's not the case. The subclass has that name because its members are a subset of members of the base class.

Encapsulation

Once I've extended my class and I'm calling it from other classes, it's transparent to me in terms of which methods are in which class.

Here's the application class:

```
public class Hires {

    public static void main(String [] args) {

        // Set up a series of film objects

        HireFilm Watch[] = new HireFilm[2];
        Watch[1] = new HireFilm("Road to Perdition",117,4.95F);
        Watch[0] = new HireFilm("The Truth about Cats and Dogs",93,2.95F);

        for (int i=0; i<Watch.length; i++) {
            String Name = Watch[i].getcalled();
            float payments = Watch[i].getcost(5);
            float valuemeasure = 1.0F / Watch[i].getcostperminute(5);
            System.out.println(Name);
            System.out.println("Cost is "+payments+
                " value factor is "+valuemeasure);
        }
    }
}
```

Figure 1 Running public class Hires

```
bash-2.04$ java Hires
The Truth about Cats and Dogs
Cost is 2.95 value factor is 31.525425
Road to Perdition
Cost is 4.95 value factor is 23.636366
bash-2.04$
```

You might expect to find `getcalled`, `getcost` and `getcostperminute` methods in the `HireFilm` class, but it turns out that's not the case:

```
public class HireFilm extends Film {

    float cost;

    public HireFilm (String title, int length, float pounds) {
        super (title,length);
        this.cost = pounds;
    }

    public float getcost (int people) {
        return cost;
    }

    public float getcostperminute (int people) {
        return cost/minutes;
    }
}
```

Where is `getcalled`? It must be inherited from the class `Film` ... or perhaps from some class that `Film` inherits from. This is the same class `Film` that you've probably seen in an earlier module, since a good base class can be extended without change but it bears repeating here to make the example complete:

```
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;
    }

}
```

The `super()` call is an interesting one. You cannot inherit a constructor into a subclass,¹ but you really don't want to repeat the constructor in all subclasses. Java will automatically call the no-parameter constructor of the base class when you create an object in the subclass unless your very first call is to `super`, which is still a call to the constructor, but perhaps to a constructor with parameters as in our example.

Exercise

Write a class of railway train which extends your class of vehicle. There's an extra parameter over the "vehicle" class - the number of carriages in the train. Write a test program in which you create a 2 coach train and a three coach train, and report the total capacity.

Alternative exercise

Grab and compile the "Lecture" application from the course profile. Application is "Lecture", base class "people", and subclasses "Student" and "Lecturer".

A Lecturer makes 20000 plus 500 pounds for each year of his age. A student makes 4000 pounds from part time work. Extend the application to add a table of earnings on the team list, and print out the earnings total too.

¹ all the right variables have to be set up at the time the object is created

2.2 Abstract Classes

Now extend the application that gives value factors for films to include not only hire films, but also purchased films and films seen at the cinema. If I write each of the classes to include a `getcost` method and a `getcostperminute` method, will that work? Yes, as far as it goes it will. I'll be able call `getcost` on my cinema film and have it multiply my `unitprice` by the number of people who will be watching. But

...

What I really want is to have a whole array of films of various types, call methods on any film and have Java select the right method depending on the type of film. With the class `film` as it stands I can't do this but I have another trick up my sleeve.

If I declare `Film` to be an abstract class, and define methods which must be in all my subclasses, I can achieve the desired result. This does mean altering my `film` class, and that I can no longer create just a `film`, I have to create a `film` of a particular type.

Getting your design right

Design matters. A quick reminder here that you must make every effort to design your class structure correctly – have the right methods in the right level of object, have your classes and subclasses set up to reflect the real-world object that you're modelling as closely as possible. Draw diagrams. Use UML or a more formal design method if appropriate. Consider software tools such as Rose; costly for sure, but much cheaper than doing the whole job twice over.

Design of the demonstration, then:

1. An application in which we define and analyse the films that we're going to entertain our young cousins with when they visit us for their summer holiday. We'll call this the "Babysitter" class.
2. An abstract class called "BaseFilm" that includes handling of the title and price of the film, and abstracts the handling of how much it costs from the subclasses.
3. subclasses called "Tv", "Hire" and "Cinema" which define the actual films and objects that we create.

The results we want to see:¹

```
bash-2.04$ java Babysitter
Shrek is 133 minutes long and costs you 4.95
The latest Lord of the Rings is 167 minutes long and costs you 29.699999
The Truth about Cats and Dogs is 93 minutes long and costs you 2.95
Mary Poppins is 114 minutes long and costs you 0.0
Chicken Run is 115 minutes long and costs you 0.0

Longest film is The latest Lord of the Rings at 167 minutes
Total run time is 622
Total number of films 5
Total expenditure 37.6
bash-2.04$
```

Let's look at the base class first, since that shows you the syntax of defining an abstract class for the first time:

```
public abstract class BaseFilm {

    String called;
    int minutes;
```

¹ you have remembered to start by doing your Use Case, haven't you?

```
public static int count=0;
static int totlength=0;

public abstract float getcost(int npersons);
public abstract float getcostperminute(int npersons);

public BaseFilm (String title, int length) {
    called = title;
    minutes = length;
    count++;
    totlength += length;
}

public int getminutes() {
    return minutes;
}

public String getcalled() {
    return called;
}

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

public static int gettotminutes() {
    return totlength;
}
}
```

The abstract method definitions can be placed anywhere outside the methods of the class but within the class definition; the names given to the parameters are just provided to complete the syntax and are not referenced here.

Having looked at the base class, let's jump all the way on to the main application code; after all, the base class declares all the methods we'll be using so we do have a complete API specification. If any extra public access is added to a particular type of film, we won't easily be able to use it!

```
public class Babysitter {

    public static void main(String [] args) {

        // Set up a series of film objects

        BaseFilm Watch[] = new BaseFilm[5];
        Watch[0] = new Hire("Shrek",133,4.95F);
        Watch[1] = new Cinema("The latest Lord of the Rings",167,4.95F);
        Watch[2] = new Hire("The Truth about Cats and Dogs",93,2.95F);
        Watch[3] = new Tv("Mary Poppins",114);
        Watch[4] = new Tv("Chicken Run",115);

        BaseFilm Longest = null; int longtime = 0;
        float expenditure = 0.0F;

        for (int i=0; i<Watch.length; i++) {
            BaseFilm Current=Watch[i];
```

```

        int mins = Current.getminutes();
        if (mins > longtime) {
            longtime=mins;
            Longest = Current;
        }
        float currentcost = Current.getcost(6);
        expenditure += currentcost;
        System.out.println(Current.getcalled() +
            " is "+Current.getminutes()+
            " minutes long and costs you "+ currentcost);
    }
    System.out.println();
    System.out.println("Longest film is " +
        Longest.getcalled() +
        " at "+longtime+" minutes");
    System.out.println("Total run time is " +
        Longest.gettotminutes()); // Call on any film or the class
    System.out.println("Total number of films " +
        BaseFilm.count); // Call on any film or the class
    System.out.println("Total expenditure " +
        expenditure);
    }
}

```

One of the great powers of an object oriented language is the ability it provides for you to call a method and have it decide at run time which particular flavour it's to run based on the data. Even where your code has just a single call to a method within a loop, different methods may be run each time through that loop at runtime. There's an example in the code above – there are three different getcost methods used, but just one call. This is known as polymorphism; you don't declare polymorphism (just like you don't declare encapsulation), they're automatically there as features of Java.

Let's complete the example by looking at the source code of the three subclasses:

```

public class Cinema extends BaseFilm {

    float cost;

    public Cinema (String title, int length, float pounds) {
        super (title,length);
        this.cost = pounds;
    }

    public float getcost (int people) {
        return cost * people;
    }

    public float getcostperminute (int people) {
        return cost / minutes * people;
    }

}

```

```
public class Hire extends BaseFilm {  
  
    float cost;  
  
    public Hire (String title, int length, float pounds) {  
        super (title,length);  
        this.cost = pounds;  
    }  
  
    public float getcost (int people) {  
        return cost;  
    }  
  
    public float getcostperminute (int people) {  
        return cost/minutes;  
    }  
  
}
```

```
-----  
public class Tv extends BaseFilm {  
  
    public Tv (String title, int length) {  
        super (title,length);  
    }  
  
    public float getcost (int people) {  
        return 0.0F;  
    }  
  
    public float getcostperminute (int people) {  
        return 0.0F;  
    }  
  
}
```

Exercise

Write a class of road coach which extends your class of vehicle. Unlike the train which you wrote earlier in this module, your coach can only be one vehicle long so the constructor will be a little different.

A train has a staff of 1, plus an extra person for every 3 carriages. A coach has a staff of one. You have a two car train, a four car train, and 2 road coaches which you are going to use to take people to a pop festival. Write an application to work out the numbers you can take and the number of staff you'll need.

2.3 The universal superclass

If you define a class and don't tell Java that it extends anything at all, then it's taken to extend the standard base class `Object`. Since you can nest inheritance, this means that all your classes will inherit from `Object`, albeit indirectly in many cases.

`Object` – the universal superclass – includes a number of methods that might be of some use to you, and also a number of methods which you'll want to override in some circumstances.

<code>equals</code>	To see if two objects are the same object
<code>clone</code>	To copy an object
<code>getClass</code>	To find the class of an object
<code>toString</code>	To return an object in string form

The `clone` method allows us to duplicate an object. In other words to take a copy of all its elements, resulting in a new object. This is a different operation to just creating a new instance variable and copying an object reference. If you modify a cloned object, you are not altering the original but if you modify an object that you've set up via a regular assignment from another, you will be altering the original.

Newcomers to Java often want to find out what type of object is held in a variable, and they devour `getClass`. But they're often wrong. If you get your design right, you shouldn't need to find out what type of object you have in this way as you should be calling methods that have been defined as abstract to ensure that your application chooses the correct code to run.

In the base class `Object`, `toString` returns a simple text string describing your object (type of object and memory address), and `equals` tells you if two objects are one and the same. Beware, these methods are often overridden even in standard classes. For example

<code>equals</code> on a <code>String</code>	compares the contents of the <code>String</code>
<code>equals</code> on a <code>File</code>	compares the file name and returns true if two objects point to the same disk file, even if via a different path

`toString` is intentionally designed to be overridden! If you call `println` on an object, it calls the `toString` method internally so that the object is described in the best possible way.

Here's a demonstration class and application to show you some uses of `Object`. Note that we've added a static `main` method within our class – a useful way of providing a test harness for the class without ending up with too many classes.

```
public class Book {

    String Title;
    String ISBN;
    String Owner;

    public Book(String Title, String ISBN, String Owner) {
        this.Title = Title;
        this.ISBN = ISBN;
        this.Owner = Owner;
    }

    public String toString() {
        String Report =
            "\tTitle: "+Title+
            "\n\tISBN: "+ISBN+
            "\n\tOwner: "+Owner;
        return Report;
    }
}
```

```

    public static void main(String [] args) {

        Book [] OurLib;
        OurLib = new Book[3];
        Book Favourite = new Book("Java in a Nutshell", "0-596-00283-1", "Well
House");

        OurLib[0] = new Book("Java in a Nutshell", "0-596-00283-1", "Well
House");
        OurLib[1] = new Book("Java Foundation Classes", "0-596-00113-
4", "Well House");
        OurLib[2] = Favourite;

        for (int i=0; i<OurLib.length; i++) {
            System.out.println("\nBook "+i+OurLib[i]);
            if (Favourite.equals(OurLib[i])) {
                System.out.println("This is my favourite");
            }
            System.out.println(OurLib[i].getClass());
        }
    }
}

```

Figure 2 Running public class Book

```

bash-2.04$ java Book

Book 0 Title: Java in a Nutshell
      ISBN: 0-596-00283-1
      Owner: Well House
class Book

Book 1 Title: Java Foundation Classes
      ISBN: 0-596-00113-4
      Owner: Well House
class Book

Book 2 Title: Java in a Nutshell
      ISBN: 0-596-00283-1
      Owner: Well House
This is my favourite
class Book
bash-2.04$

```

2.4 Interfaces

In Java, every class inherits from exactly one other class, either a class you define as being its base class, or from Object.

In other languages such as C++ and Perl, a class can be defined to inherit from more than one class. Such multiple inheritance makes for a complex and slower language and is rarely the best solution to requirements. It's quite common to have the following conversation on a Java course:

Trainee: "Does Java have multiple inheritance?"

Tutor: "No. Why do you want it?"

Trainee: "Because C++ has it."

Tutor: "But have you ever used it?"

Trainee:"Come to think of it, I've never had the need."
 There's a variant on the final line too.
 Trainee:"Yes, but it turned out to be a bad idea for my particular needs."

But there will be times that you want to write a whole series of classes all of which have one or more methods that they all implement but that they don't inherit or abstract from a base class. Such methods can be easily written, but if you then want to use polymorphism to call the methods you'll have to provide a definition which you can do in the form of an interface.

You define the subclass in which the actual code resides as implementing an interface, and you create a separate interface class to define what's necessary. You can think of an interface as being similar to an abstract class but:

1. There is no code in an interface definition.
2. The syntax of an interface is different to that of an abstract class.
3. You can declare a class as implementing as many interfaces as you like (and it will also extend one class – Object or a class of your choice).

Example ... we'll define an insurable interface

```
public interface insurable {
    void setRisk(String What);
    String getRisk();
}
```

and two classes which implement that interface:

```
public class Car implements insurable {

    String Risklevel;

    public Car() {
        Risklevel = "Third Party, fire and theft";
    }

    public void setRisk(String Level) {
        Risklevel = Level;
    }

    public String getRisk() {
        return Risklevel;
    }

}
```

```
-----

public class House implements insurable {

    String Risklevel;

    public House() {
        Risklevel = "";
    }

    public void setRisk(String Level) {
        if (Risklevel.equals("")) {
            Risklevel = Level;
        }
    }

}
```

```

        } else {
            Risklevel = Risklevel + "\n" + Level;
        }
    }

    public String getRisk() {
        return Risklevel;
    }
}

```

and a main application that uses both Car and House objects.

```

public class Ipay {

// Demonstration of an Interface

    public static void main(String [] args) {

        insurable [] Ours = new insurable[3];

        Ours[0] = new Car();
        Ours[1] = new House();
        Ours[2] = new Car();

        Ours[1].setRisk("Subsidence");
        Ours[1].setRisk("Flood");
        Ours[2].setRisk("Comprehensive");
        Ours[2].setRisk("Any Named Driver");

        for (int i=0; i<Ours.length; i++) {
            System.out.println(
                Ours[i].getClass() + "\n" +
                Ours[i].getRisk() + "\n" );
        }
    }
}

```

```

bash-2.04$ java Ipay
class Car
Third Party, fire and theft

class House
Subsidence
Flood

class Car
Any Named Driver

bash-2.04$

```

Figure 3 Running public class Ipay

You'll notice that we are allowed to declare an array to hold objects of type insurable, even though we can't create an object of that type. After all, we're looking at an interface and not a class.

2.5 The final modifier

A final modifier may be used to fix the value of a static member of a class. It may also be used to prevent any subclasses overriding a method you have defined in your class.

Whilst it is a good idea to use final modifiers as a security check, they do not add functionality. Rather, they give rise to compiler or run-time errors when another method attempts an illegal override, or to modify a fixed value.

 **Exercise**

License

*These notes are distributed under the **Well House Consultants Open Training Notes License**. Basically, if you distribute it and use it for free, we'll let you have it for free. If you charge for its distribution of use, we'll charge.*

3.1 Open Training Notes License

Training notes distributed under the **Well House Consultants Open Training Notes License** (WHCOTNL) may be reproduced for any purpose PROVIDE THAT:

- This License statement is retained, unaltered (save for additions to the change log) and complete.
- No charge is made for the distribution, nor for the use or application thereof. This means that you can use them to run training sessions or as support material for those sessions, but you cannot then make a charge for those training sessions.
- Alterations to the content of the document are clearly marked as being such, and a log of amendments is added below this notice.
- These notes are provided "as is" with no warranty of fitness for purpose. Whilst every attempt has been made to ensure their accuracy, no liability can be accepted for any errors of the consequences thereof.

Copyright is retained by Well House Consultants Ltd, of 404, The Spa, Melksham, Wiltshire, UK, SN12 6QL - phone number +44 (1) 1225 708225. Email contact - Graham Ellis (graham@wellho.net).

Please send any amendments and corrections to these notes to the Copyright holder - under the spirit of the Open Distribution license, we will incorporate suitable changes into future releases for the use of the community.

If you are charged for this material, or for presentation of a course (Other than by Well House Consultants) using this material, please let us know. It is a violation of the license under which this notes are distributed for such a charge to be made, except by the Copyright Holder.

If you would like Well House Consultants to use this material to present a training course for your organisation, or if you wish to attend a public course is one is available, please contact us or see our web site - <http://www.wellho.net> - for further details.

Change log
Original Version, Well House Consultants, 2004

Updated by: _____ on _____
Updated by: _____ on _____
Updated by: _____ on _____
Updated by: _____ on _____
Updated by: _____ on _____
Updated by: _____ on _____

License Ends.