CHAPTER: 09

Inheritance

Objectives:

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

- *explain the term **inheritance**;*
- *design inheritance structures using UML notation;*
- *implement inheritance relationships in Java;*
- *distinguish between **method overriding** and **method overloading**;*
- *explain the term **type cast** and implement this in Java;*
- *explain the use of the* **`abstract`** *modifier when applied to classes and methods;*
- *explain the use of the* **`final`** *modifier, when applied to classes and methods;*
- *describe the way in which all Java classes are derived from the* `Object` *class.*

9.1 Introduction

One of the greatest benefits of the object-oriented approach to software development is that it offers the opportunity for us to *reuse* classes that have already been written – either by ourselves or by someone else. Let's look at a possible scenario. Say you wanted to develop a software system and you have, during your analysis, identified the need for a class called `Employee`. You might be aware that a colleague in your organization has already written an `Employee` class; rather than having to write your own class, it would be easier to approach your colleague and ask her to let you use her `Employee` class.

So far so good, but what if the `Employee` class that you are given doesn't quite do everything that you had hoped? Perhaps your employees are part-time employees, and you want your class to have an attribute like `hourlyPay`, or methods like `calculateWeeklyPay` and `setHourlyPay`, and these attributes and methods do not exist in the `Employee` class you have been given.

You may think it would be necessary to go into the old class and start messing about with the code. But there is no need, because object-oriented programming languages provide the ability to extend existing classes by adding attributes and methods to them. This is called **inheritance**.

9.2 Defining inheritance

Inheritance is the sharing of attributes and methods among classes. We take a class, and then define other classes based on the first one. The new classes *inherit* all the attributes and methods of the first one, but also have attributes and methods of their own. Let's try to understand this by thinking about the `Employee` class.

Say our `Employee` class has two attributes, `number` and `name`, a user-defined constructor, and some basic *get*- and *set*- methods for the attributes. We now define our `PartTimeEmployee` class; this class will *inherit* these attributes and methods, but can also have attributes and methods of its own. We will give it one additional attribute, `hourlyPay`, some methods to access this attribute and one additional method, `calculateWeeklyPay`.

This is illustrated in figure 9.1 which uses the UML notation for inheritance, namely a triangle.

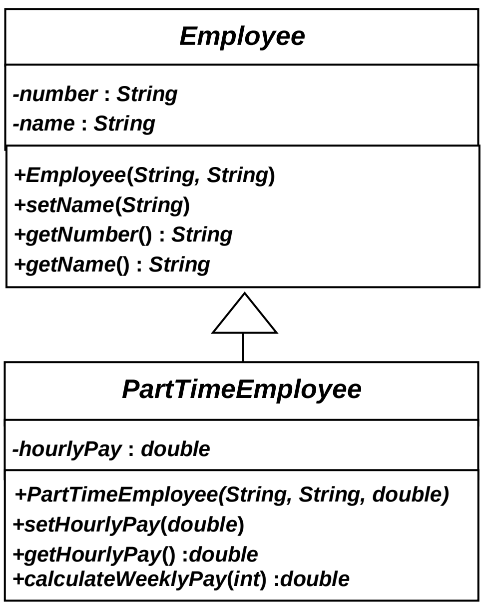

Fig 9.1. An inheritance relationship

You can see from this diagram that an inheritance relationship is a *hierarchical* relationship. The class at the top of the hierarchy – in this case the `Employee` class – is referred to as the **superclass** (or **base class**) and the `PartTimeEmployee` as the **subclass** (or **derived class**).

The inheritance relationship is also often referred to as an *is-a-kind-of* relationship; in this case a `PartTimeEmployee` *is a kind of* `Employee`.

9.3 Implementing inheritance in Java

The code for the `Employee` class is shown below:

The *Employee* class

```
public class Employee
{
   private String number;
   private String name;
   public Employee(String numberIn, String nameIn)
   {
       number = numberIn;
       name = nameIn;
   }
   public void setName(String nameIn)
   {
       name = nameIn;
   }
   public String getNumber()
   {
       return number;
   }
   public String getName()
   {
       return name;
   }
}
```

There is nothing new here, so let's get on with our `PartTimeEmployee` class. We will present the code first and analyse it afterwards.

The *PartTimeEmployee* class

```
public class PartTimeEmployee extends Employee // this class is a subclass of Employee
{
   private double hourlyPay; // this attribute is unique to the subclass
   // the constructor
   public PartTimeEmployee(String numberIn, String nameIn, double hourlyPayIn)
   {
      super(numberIn, nameIn); // call the constructor of the superclass
      hourlyPay = hourlyPayIn;
   }
   // these methods are also unique to the subclass
   public double getHourlyPay()
   {
      return hourlyPay;
   }
   public void setHourlyPay(double hourlyPayIn)
   {
      hourlyPay = hourlyPayIn;
   }
   public double calculateWeeklyPay(int noOfHoursIn)
   {
      return noOfHoursIn * hourlyPay;
   }
}
```

The first line of interest is the class header itself:

```
public class PartTimeEmployee extends Employee
```

Here we see the use of the keyword **`extends`**. Using this word in this way means that the `PartTimeEmployee` class (the *subclass*) inherits all the attributes and methods of the `Employee` class (the *superclass*). So although we haven't coded them, any object of the `PartTimeEmployee` class will have, for example, an attribute called `name` and a method called `getNumber`. A `PartTimeEmployee` is now a *kind of* `Employee`.

But can you see a problem here? The attributes have been declared as **`private`** in the superclass so although they are now part of our `PartTimeEmployee` class, none of the `PartTimeEmployee` class methods can directly access them – the subclass has only the same access rights as any other class!

There are a number of possible ways around this:

1 We could declare the original attributes as **`public`** – but this would take away the whole point of encapsulation.

2 We could use the special keyword **`protected`** instead of **`private`**. The effect of this is that anything declared as **`protected`** is accessible to the methods of any subclasses. There are, however, two issues to think about here. The first is that you have to anticipate in advance when you want your class to be able to be inherited. The second problem is that it weakens your efforts to encapsulate information within the class, since, in Java, **`protected`** attributes are also accessible to any other class in the same package (you will find out much more about the meaning of the word **package** in chapter 20).

The above remarks notwithstanding, this is a perfectly acceptable approach to use, particularly in situations where you are writing a class as part of a discrete application, and you will be aware in advance that certain classes will need to be subclassed. You will see an example of this in section 9.4.

Incidentally, in a UML diagram a **`protected`** attribute is indicated by a hash symbol, #.

3 The other solution, and the one we will use now, is to leave the attributes as **`private`**, but to plan carefully in advance which `get-` and `set-` methods we are going to provide.

After the class header we have the following declaration:

```
private double hourlyPay;
```

This declares an attribute, `hourlyPay`, which is unique to our subclass – but remember that the attributes of the superclass, `Employee`, will be inherited, so in fact any `PartTimeEmployee` object will have *three* attributes.

Next comes the constructor. We want to be able to assign values to the number and name at the time that the object is created, just as we do with an `Employee` object; so our constructor will need to receive parameters that will be assigned to the `number` and `name` attributes.

But wait a minute! How are we going to do this? The `number` and `name` attributes have been declared as **`private`** in the superclass – so they aren't accessible to objects of the subclass. Luckily there is a way around this problem. We can call the constructor of the superclass by using the keyword **`super`**. Look how this is done:

```
public PartTimeEmployee(String numberIn, String nameIn, double hourlyPayIn)
{
   super(numberIn, nameIn); // call the constructor of the superclass
   hourlyPay = hourlyPayIn;
}
```

After calling the constructor of the superclass, we need to perform one more task – namely to assign the third parameter, `hourlyPayIn`, to the `hourlyPay` attribute. Notice, however, that the line that calls **`super`** has to be the first one - if we had written our constructor like this it would not compile:

```
/* This version of the constructor would not compile - the call to super has to be
   the first instruction */

public PartTimeEmployee(String numberIn, String nameIn, double hourlyPayIn)
{
    hourlyPay = hourlyPayIn;
   super(numberIn, nameIn); // this call should have been the first instruction
}
```

The remaining methods of `PartTimeEmployee` are new methods specific to the subclass:

```
public double getHourlyPay()
{
   return hourlyPay;
}
public void setHourlyPay(double hourlyPayIn)
{
   hourlyPay = hourlyPayIn;
}
public double calculateWeeklyPay(int noOfHoursIn)
{
   return noOfHoursIn * hourlyPay;
}
```

The first two provide read and write access respectively to the `hourlyPay` attribute. The third one receives the number of hours worked and calculates the pay by multiplying this by the hourly rate. Program 9.1 demonstrates the use of the `PartTimeEmployee` class.

Program 9.1

```
import java.util.*;
public class PartTimeEmployeeTester
{
  public static void main(String[] args)
  {
     Scanner keyboard = new Scanner(System.in);
     Scanner keyboardString = new Scanner(System.in);
```

```
      String number, name;
      double pay;
      int hours;
      PartTimeEmployee emp;
      // get the details from the user
      System.out.print("Employee Number? ");
      number = keyboardString.nextLine();
      System.out.print("Employee's Name? ");
      name = keyboardString.nextLine();
      System.out.print("Hourly Pay? ");
      pay = keyboard.nextDouble();
      System.out.print("Hours worked this week? ");
      hours = keyboard.nextInt();
      // create a new part-time employee
      emp = new PartTimeEmployee(number, name, pay);
      // display part-time employee's details, including the weekly pay
      System.out.println();

      // the next two mwthods have been inhreted from the Employee class
      System.out.println(emp.getName());
      System.out.println(emp.getNumber());

      System.out.println(emp.calculateWeeklyPay(hours));
   }
}
```

Here is a sample test run:

Employee Number? ***A103456***

Employee's Name? ***Mandy Lifeboats***

Hourly Pay? ***15.50***

Hours worked this week? ***20***

Mandy Lifeboats

A103456

310.0

We can now move on to look at another inheritance example; let's choose the `Rectangle` class that we developed in the last chapter.

9.3.1 Extending the *Rectangle* class

We are going to define a new class called `ExtendedRectangle`, which extends the `Rectangle` class. First, let's remind ourselves of the `Rectangle` class itself.

The *Rectangle* class – a reminder

```
public class Rectangle
{
    // the attributes are declared first
```

```
    private double length;
    private double height;

    // then the methods

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

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

    // the next method allows us to read the height attribute
    public double getHeight()
    {
       return height;
    }
    // the next method allows us to write to the length attribute
    public void setLength(double lengthIn)
    {
       length = lengthIn;
    }
    // the next method allows us to write to the height attribute
    public void setHeight(double heightIn)
    {
       height = heightIn;
    }
    // this method returns the area of the rectangle
    public double calculateArea()
    {
       return length * height;
    }

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

The original `Rectangle` class had the capability of reporting on the perimeter and area of the rectangle. Our extended class will have the capability of sending back a string representation of itself composed of a number of symbols such as asterisks – for example:

```
*****
*****
*****
```

Now at first glance you might think that this isn't a string at all, because it consists of several lines. But if we think of the instruction to start a new line as just another character – which for convenience we could call <NEWLINE> – then our string could be written like this.

```
*****<NEWLINE>*****<NEWLINE>*****
```

In Java we are able to represent this <NEWLINE> character with a special character that looks like this:

```
'\n'
```

This is one of a number of special characters called **escape characters**, which are always introduced by a backslash (\). Another useful escape character is `'\t'` which inserts a tab[1].

Our `ExtendedRectangle` class will need an additional attribute, which we will call `symbol`, to hold the character that is to be used to draw the rectangle. We will also provide a `setSymbol` method, and of course we will need a method that sends back the string representation. We will call this method `draw`. The new constructor will accept values for the length and height as before, but will also receive the character to be used for drawing the rectangle.

The design is shown in figure 9.2.

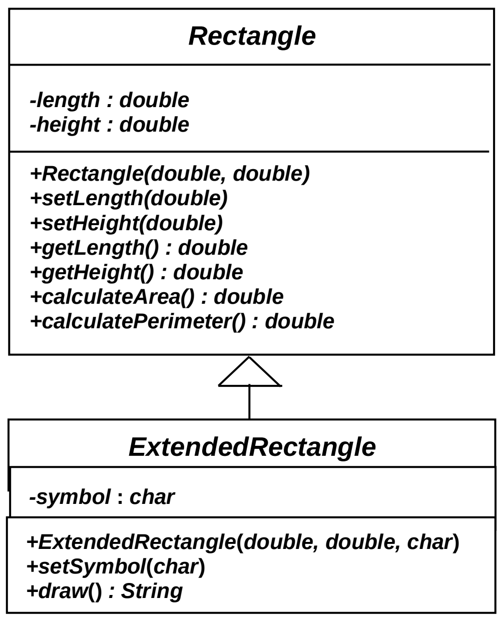

Fig 9.2. The *Rectangle* hierarchy

Now for the implementation. As well as those aspects of the code that relate to inheritance, there is an additional new technique used in this class – this is the technique known as **type casting**. Take a look at the complete code first – then we can discuss this new concept along with some other important features of the class.

The *ExtendedRectangle* class
`public class ExtendedRectangle extends Rectangle`

[1] You would also have to place a backslash in front of a double quote (`\"`), a single quote (`\'`) or another backslash (`\\`) if you wanted any of these to be output as part of a string. This is because the compiler would interpret these as having a special meaning such as terminating the string. You will see an example of this in section 10.5.

```
{
   private char symbol;

   // the constructor
   public ExtendedRectangle(double lengthIn, double heightIn, char symbolIn)
   {
      super(lengthIn, heightIn);
      symbol = symbolIn;
   }

   public void setSymbol(char symbolIn)
   {
      symbol = symbolIn;
   }

   public String draw()
   {
      String s = new String(); // start off with an empty string
      int l, h;
      /* in the next two lines we type cast from double to integer so that we are able to count how
         many times we print the symbol */
      l = (int) getLength();
      h = (int) getHeight();
      for (int i = 1; i <= h; i++)
      {
         for (int j = 1; j <= l; j++)
         {
               s = s + symbol; // add the symbol to the string
         }
         s = s + '\n'; // add the <NEWLINE> character
      }
      return s; // return the string representation
   }
}
```

So let's take a closer look at all this. After the class header – which **`extend`**s the `Rectangle` class – we declare the additional attribute, `symbol`, and then define our constructor:

```
public ExtendedRectangle(double lengthIn, doubleheightIn, char symbolIn)
{
    super(lengthIn, heightIn);
    symbol = symbolIn;
}
```

Once again we call the constructor of the superclass with the keyword **`super`**. After the constructor comes the `setSymbol` method – which allows the symbol to be changed during the rectangle's lifetime – and then we have the `draw` method, which introduces the new concept of **type casting**:

```
public String draw()
{
    String s = new String(); // start off with an empty string
    int l, h;
    l = (int) getLength();
    h = (int) getHeight();
    for (int i = 1; i <= h; i++)
    {
        for (int j = 1; j <= l; j++)
        {
            s = s + symbol; // add a symbol to end of the string
        }
```

```
        s = s + '\n'; // add a new line to the string
    }
    return s;
}
```

Inspect the code carefully – notice that we have declared two local variables of type **`int`**. In order to understand the purpose of these two variables, `l` and `h`, we need to explore this business of type casting, which means forcing an item to change from one type to another.

The `draw` method is going to create a string of one or more rows of stars or crosses or whatever symbol is chosen. Now the dimensions of the rectangle are defined as **`double`**s. Clearly our `draw` method needs to be dealing with whole numbers of rows and columns – so we must convert the length and height of the rectangle from **`double`**s to **`int`**s. There will obviously be some loss of precision here, but that won't matter in this particular case.

As you can see from the above code, type casting is achieved by placing the new type name in brackets before the item you wish to change. This is illustrated in figure 9.3.

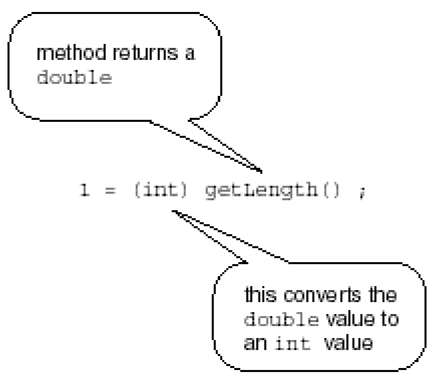

Fig 9.3. Type casting

Program 9.2 uses the `ExtendedRectangle` class. It creates a rectangle of length 10 and height 5, with an asterisk as the symbol; it then draws the rectangle, changes the symbol to a cross, and draws it again.

Program 9.2

```
public class ExtendedRectangleTester
{
   public static void main(String[] args)
   {
      ExtendedRectangle extRectangle = new ExtendedRectangle(10.2,5.3,'*');
      System.out.println(extRectangle.draw());
      extRectangle.setSymbol('+');
      System.out.println(extRectangle.draw());
   }
}
```

The output from program 9.2 is shown below:

```
**********
**********
**********
**********
**********

++++++++++
++++++++++
++++++++++
++++++++++
++++++++++
```

9.4 Method overriding

In chapter 5 you were introduced to the concept of polymorphism - the idea that we can have different methods with the same name, but whose behaviour is different. You saw in that chapter that one way of achieving polymorphism was by method *overloading*, which involves methods of the same class having the same name, but being distinguished by their parameter lists.

Now we are going to explore another way of achieving polymorphism, namely by **method overriding**. In order to do this we are going to extend the `BankAccount` class that we developed in the previous chapter. You will recall that the class we developed there did not provide any overdraft facility - the `withdraw` method was designed so that the withdrawal would take place only if the amount to be withdrawn did not exceed the balance.

Now let's consider a special account which is the same as the original account, but allows holders of the account to be given an overdraft limit and to withdraw funds up to this limit. We will call this account `GoldAccount`. Since a `GoldAccount` ***is a kind of*** `BankAccount`, we can use inheritance here to design the `GoldAccount` class. In addition to the attributes of a `BankAccount`, a `GoldAccount` will need to have an attribute to represent the overdraft limit, and should have `get`− and `set`− methods for this attribute. As far as the methods are concerned, we need to reconsider the `withdraw` method. This will differ from the original method, because, instead of checking that the amount to be withdrawn does not exceed the balance, it will now check that the amount does not exceed the total of the balance plus the overdraft limit. So what we are going to do is to re-write - or *override* - the `withdraw` method in the subclass.

The code for the `BankAccount` class and the `GoldAccount` class appear below in figure 9.4. You will notice that we have made a small change to the original `BankAccount` class. The balance attribute has a hash sign (#) in front of it instead of a minus sign. You will remember from our previous discussion that this means access to the attribute is **`protected`**, rather than **`private`**. The reason why we have decided to make this change is explained below.

You will also notice that the `withdraw` method appears in both classes - this, of course, is because we are going to override it in the subclass.

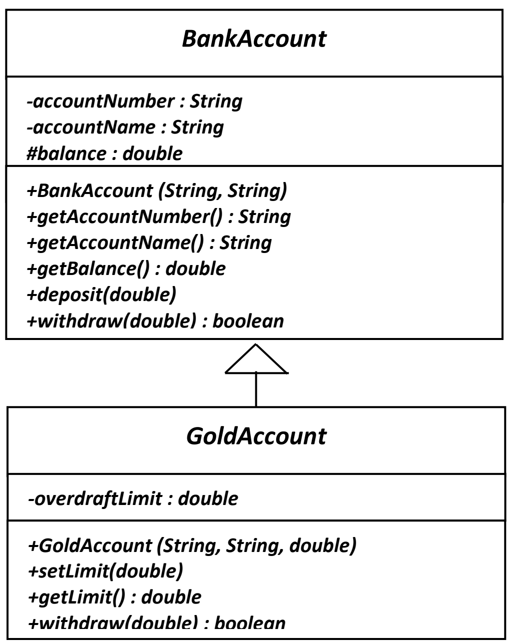

Fig 9.4. The UML diagram for the *BankAccount* hierarchy.

You might already be thinking about how to code the `withdraw` method in the `GoldAccount` class. If you are doing this, you will probably have worked out that this method is going to need access to the `balance` attribute, which of course was declared as **`private`** in the `BankAccount` class, and (for good reason) was not provided with a `set`– method.

When we developed the `BankAccount` class in chapter 8, we developed it as a stand-alone class, and we didn't think about how it might be used in a larger application where it could be refined. Had we known about inheritance at that point we might have given the matter a little more thought, and realised that it would be useful if any sub-classes of `BankAccount` that were developed in the future had access to the `balance` attribute. As we explained in section 9.3, we can achieve that by declaring that attribute as **`protected`** instead of **`private`**. That is what we have done here. The version of `BankAccount` that we are going to use in this chapter is therefore exactly the same as the previous one, with the single difference that the declaration of the `balance` attribute now looks like this, with the keyword **`protected`** replacing **`private`**:

```
   protected double balance;
```

This new version of the `BankAccount` class is available on the CD, in the Chapter 9 folder.

Here is the code for the `GoldAccount` class:

The *GoldAccount* class

```
public class GoldAccount extends BankAccount
{
   private double overdraftLimit;

   public GoldAccount(String numberIn, String nameIn, double limitIn)
   {
      super(numberIn, nameIn);
      overdraftLimit = limitIn;
   }

   public void setLimit(double limitIn)
   {
      overdraftLimit = limitIn;
   }


   public double getLimit()
   {
      return overdraftLimit;
   }

   public boolean withdraw(double amountIn)
   {
      if(amountIn > balance + overdraftLimit) // the customer can withdraw up to the overdraft limit
      {
         return false; // no withdrawal was made
      }
      else
      {
         balance = balance - amountIn; // balance is protected so we have direct access to it
         return true; // money was withdrawn successfully
      }
   }

}
```

The thing that we are interested in here is the `withdraw` method. As we show below the test in the **`if`** statement differs from the original method in the `BankAccount` class, in order to take account of the fact that customers with a gold account are allowed an overdraft:

`withdraw` method in `BankAccount` class

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

`withdraw` method in `GoldAccount` class

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

When we dealt with method *overloading* in chapter 5 we told you that the methods with the same name *within* a class are distinguished by their parameter lists. In the case of method *overriding*, the methods have the same parameter list but belong to different classes - the superclass and the subclass. In this case they are distinguished by the *object with which they are associated*. We illustrate this in program 9.3.

Program 9.3

```
public class OverridingDemo
{
   public static void main(String[] args)
   {
      boolean ok;
      //declare a BankAccount object
      BankAccount bankAcc = new BankAccount("123", "Ordinary Account Holder");
      //declare a GoldAccount object
      GoldAccount goldAcc = new GoldAccount("124", "Gold Account Holder", 500);

      bankAcc.deposit(1000);
      goldAcc.deposit(1000);

      ok = bankAcc.withdraw(1250); // the withdraw method of BankAccount is called
      if(ok)
      {
         System.out.print("Money withdrawn. ");
      }
      else
      {
         System.out.print("Insufficient funds. ");
      }
      System.out.println("Balance of " + bankAcc.getAccountName() + " is " + bankAcc.getBalance());
      System.out.println();

      ok = goldAcc.withdraw(1250); // the withdraw method of GoldAccount is called
      if(ok)
      {
         System.out.print("Money withdrawn. ");
      }
      else
      {
         System.out.print("Insufficient funds. ");
      }
      System.out.println("Balance of " + goldAcc.getAccountName() + " is " + goldAcc.getBalance());
      System.out.println();
   }
}
```

In this program we create an object of the `BankAccount` class and an object of the `GoldAccount` class (with an overdraft limit of 500), and deposit an amount of 1000 in each:

```
BankAccount bankAcc = new BankAccount("123", "Ordinary Account Holder");
GoldAccount goldAcc = new GoldAccount("124", "Gold Account Holder", 500);
bankAcc.deposit(1000);
goldAcc.deposit(1000);
```

Next we attempt to withdraw the sum of 1250 from the `BankAccount` object and assign the return value to a **`boolean`** variable, `ok`:

```
ok = bankAcc.withdraw(1250);
```

The `withdraw` method that is called here will be that of `BankAccount`, because it is called via the `BankAccount` object, `bankAcc`.

Once this is done we display a message showing whether or not the withdrawal was successful, followed by the balance of that account:

```
if(ok)
{
    System.out.print("Money withdrawn. ");
}
    else
{
    System.out.print("Insufficient funds. ");
}
System.out.println("Balance of " + bankAcc.getAccountName() + " is "
                                                        + bankAcc.getBalance());
```

Now the `withdraw` method is called again, but in this case via the `GoldAccount` object, `goldAcc`:

```
ok = goldAcc.withdraw(1250);
```

This time it is the `withdraw` method of `GoldAccount` that will be called, because `goldAcc` is an object of this class. The appropriate message and the balance are again displayed.

The output from this program is shown below:

`Insufficient funds. Balance of Ordinary Account Holder is 1000.0`

`Money withdrawn. Balance of Gold Account Holder is -250.0`

As we would expect, the withdrawal from `BankAccount` does not take place - the balance is 1000, and since there is no overdraft facility a request to withdraw 1250 is denied.

In the case of the `GoldAccount`, however, a withdrawal of 1250 would result in a negative balance of

250, which is allowed, because it is within the overdraft limit of 500.

9.5 Abstract classes

Let's think again about our `Employee` class. Imagine that our business expands, and we now employ full-time employees as well as part-time employees. A full-time employee object, rather than having an hourly rate of pay, will have an annual salary. It might also need a method that calculates the monthly pay (by dividing the annual salary by 12).

Figure 9.5 shows the structure of an employee hierarchy with the two types of employee, the full-time and the part-time employee.

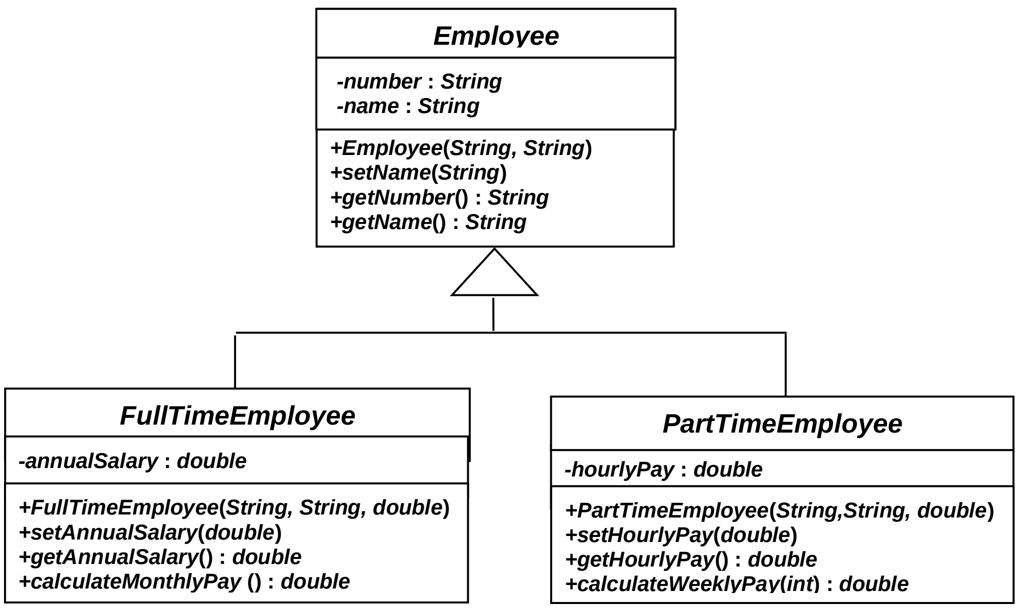

Fig 9.5. An inheritance relationship showing the superclass *Employee* and the subclasses *FullTimeEmployee* and *PartTimeEmployee*

Notice how the two subclasses contain the attributes and methods appropriate to the class. If you think about this a bit more, it will occur to you that *any* employee will always be either a full-time employee or a part-time employee. There is never going to be a situation in which an individual is just a plain old employee! So users of a program that included all these classes would never find themselves creating objects of the `Employee` class. In fact it would be a good idea to prevent people from doing this – and, as you might have guessed, there is a way to do so, which is to declare the class as **`abstract`**. Once a class has been declared in this way it means that you are not allowed to create objects of that class. In order to make our employee class abstract all we have to do is to place the keyword **`abstract`** in the header:

```
public abstract class Employee
```

The Employee class simply acts a basis on which to build other classes. Now, if you tried to create an object of the Employee class you would get a compiler error.

We have already seen the code for Employee and PartTimeEmployee; Here is the code for the FullTimeEmployee class:

The *FullTimeEmployee* class

```
public class FullTimeEmployee extends Employee
{
   private double annualSalary;

   public FullTimeEmployee(String numberIn, String nameIn, double salaryIn)
   {
      super(numberIn,nameIn);
      annualSalary = salaryIn;
   }

   public void setAnnualSalary(double salaryIn)
   {
      annualSalary = salaryIn;
   }

   public double getAnnualSalary()
   {
      return annualSalary;
   }

   public double calculateMonthlyPay()
   {
      return annualSalary/12;
   }
}
```

As we said before, an inheritance relationship is often referred to as an "*is-a-kind-of*" relationship. A full-time employee is a *kind of* employee, as is a part-time employee. Therefore an object that is of type PartTimeEmployee is also of type Employee - an object is the type of its class, and also of any of the superclasses in the hierarchy.

Let's see how this relationship works in a Java program. Imagine a method which is set up to receive an Employee object. If we call that method and send in a FullTimeEmployee object or a PartTimeEmployee object, either is absolutely fine - because both are *kinds of* Employee. We demonstrate this in program 9.4

Program 9.4

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

```
    {
       FullTimeEmployee fte = new FullTimeEmployee("A123", "Ms Full-Time", 25000);
       PartTimeEmployee pte = new PartTimeEmployee("B456", "Mr Part-Time",30);
       testMethod(fte); // call testMethod with a full-time employee object
       testMethod(pte); // call testMethod with a part-time employee object
    }

    static void testMethod(Employee employeeIn) // the method expects to receive an Employee object
    {
       System.out.println(employeeIn.getName());
    }
}
```

In this program `testMethod` expects to receive an `Employee` object. It calls the `getName` method of `Employee` in order to display the employee's name.

In the `main` method, we create two objects, one `FullTimeEmployee` and one `PartTimeEmployee`:

```
FullTimeEmployee fte = new FullTimeEmployee("A123", "Ms Full-Time", 25000);
PartTimeEmployee pte = new PartTimeEmployee("B456", "Mr Part-Time",30);
```

We then call `testMethod` twice - first with `FullTimeEmployee` object and then with the `PartTimeEmployee` object:

```
testMethod(fte); // call testMethod with a full-time employee object
testMethod(pte); // call testMethod with a part-time employee object
```

The method accepts either object, and calls the `getName` method. The output is, as expected:

```
Ms Full-Time
Mr Part-Time
```

9.6 Abstract methods

In program 9.4 we conveniently gave our objects the names "Ms Full-Time" and "Mr Part-Time" so that we could easily identify them in our output. In fact, it wouldn't be a bad idea - particularly for testing purposes - if every `Employee` type actually had a method that returned a string telling us the kind of object we were dealing with. Adding such a method - we could call it `getStatus` - would be simple. For the `FullTimeEmployee` the method would look like this:

```
public String getStatus()
{
   return "Full-Time";
}
```

For the PartTimeEmployee, getStatus would look like this:

```
public String getStatus()
{
   return "Part-Time";
}
```

It would be very useful if we could say to anyone using any of the Employee types, that we *guarantee* that this class will have a getStatus method. That way, a developer could, for example, write a method that accepts an Employee object, and call that object's getStatus method, even without knowing anything else about the class.

As you have probably guessed, we *can* guarantee it! What we have to do is to write an **abstract** method in the superclass - in this case Employee. Declaring a method as **abstract** means that any subclass is *forced* to override it - otherwise there would be a compiler error. So in this case we just have to add the following line into the Employee class:

```
public abstract String getStatus();
```

You can see that to declare an **abstract** method, we use the Java keyword **abstract**, and we define the header, but no body - the actual implementation is left to the individual subclasses. Of course, **abstract** methods can only be declared in **abstract** classes - it wouldn't make much sense to try to declare an object if one or more of its methods were undefined.

Now, having defined the **abstract** getStatus method in the Employee class, if we tried to compile the FullTimeEmployee or the PartTimeEmployee class (or any other class that extends Employee) without including a getStatus method we would be unsuccessful.

Once we have added the different getStatus methods into the Employee classes, we could re-write program 9.4 using the getStatus method in testMethod. We have done this in program 9.5:

Program 9.5

```
public class EmployeeTester2
{
   public static void main(String[] args)
   {
      FullTimeEmployee fte = new FullTimeEmployee("A123", "Ms Full-Time", 25000);
      PartTimeEmployee pte = new PartTimeEmployee("B456", "Mr Part-Time",30);
      testMethod(fte); // call testMethod with a full-time employee object
      testMethod(pte); // call testMethod with a part-time employee object
   }

   static void testMethod(Employee employeeIn) // the method expects to receive an Employee object
   {
      System.out.println(employeeIn.getStatus());
   }
}
```

In program 9.5 it was clear at the time the program was compiled which version of getStatus was being referred to. The first time that the tester method is called a FullTimeEmployee object is sent in, so the getStatus method of FullTimeEmployee is called; The second time that the tester method is called, a PartTimeEmployee object is sent in, so the getStatus method of PartTimeEmployee is called. But now have a look at program 9.6, where, incidentally, we have made use of our EasyScanner class for input.

Program 9.6

```
public class EmployeeTester3
{

   public static void main(String[] args)
   {
      Employee emp; // a reference to an Employee

      char choice;
      String numberEntered, nameEntered;
      double salaryEntered, payEntered;

      System.out.print("Choose (F)ull-Time or (P)art-Time Employee: ");
      choice = EasyScanner.nextChar();

      System.out.print("Enter employee number: ");
      numberEntered = EasyScanner.nextString();

      System.out.print("Enter employee name: ");
      nameEntered = EasyScanner.nextString();

      if(choice == 'F' || choice == 'f')
      {
         System.out.print("Enter annual salary: ");
         salaryEntered = EasyScanner.nextDouble();

         // create a FullTimeEmployee object
         emp = new FullTimeEmployee (numberEntered, nameEntered, salaryEntered);
      }

      else
      {
         System.out.print("Enter hourly pay: ");
         payEntered = EasyScanner.nextDouble();

         // create a PartTimeEmployee object
         emp = new PartTimeEmployee (numberEntered, nameEntered, payEntered);
      }

      testMethod(emp); // call tester with the object created
   }


   static void testMethod(Employee employeeIn)
   {
      System.out.println(employeeIn.getStatus());
   }
}
```

In this program, we call testMethod only once, and allow the user of the program to decide whether a FullTimeEmployee object is sent in as a parameter, or a PartTimeEmployee object. You can see that at the beginning of the program we have declared a reference to an Employee:

```
Employee emp;
```

Although Employee is an **abstract** class, it is perfectly possible to declare a reference to this class - what we would not be allowed to do, of course, is to create an Employee *object*. However, as you will see in a moment, we can point this reference to an object of any subclass of `Employee`, since such objects, like `FullTimeEmployee` and `PartTimeEmployee`, are kinds of `Employee`.

You can see that we request the employee number and name from the user, and then ask if the employee is full-time or part-time. In the former case we get the annual salary and then create a `FullTimeEmployee` object which we assign to the `Employee` reference, `emp`.

```
if(choice == 'F' || choice == 'f')
{
      System.out.print("Enter annual salary: ");
      salaryEntered = input.nextDouble();

      // create a FullTimeEmployee object
      emp = new FullTimeEmployee (numberEntered, nameEntered, salaryEntered);
}
```

In the latter case we request the hourly pay and then assign `emp` to a new `PartTimeEmployee` object:

```
else
{
      System.out.print("Enter hourly pay: ");
      payEntered = input.nextDouble();

      // create a PartTimeEmployee object
      emp = new PartTimeEmployee (numberEntered, nameEntered, payEntered);
}
```

Finally we call the `testMethod` with `emp`:

```
testMethod(emp);
```

The `getStatus` method of the appropriate `Employee` object will then be called

Here are two sample runs from this program:

```
Choose (F)ull-Time or (P)art-Time Employee: F
Enter employee number: 123
Enter employee name: Robertson
Enter annual salary: 23000
Full-Time

Choose (F)ull-Time or (P)art-Time Employee: P
Enter employee number: 876
Enter employee name: Adebayo
Enter hourly pay: 25
Part-Time
```

As you can see, we do not know *until the program is run* whether the `getStatus` method is going to be called with a `FullTimeEmployee object` or a `PartTimeEmployee` object - and yet when the `getStatus` method is called, the correct version is executed.

The technique which makes it possible for this decision to be made at run-time is quite a complex one, and differs slightly from one programming language to another - it is explained in detail in chapter 24.

9.7 The `final` modifier

You have already seen the use of the keyword **`final`** in chapter 2, where it was used to modify a variable and turn it into a constant. It can also be used to modify a class and a method. In the case of a class it is placed before the class declaration, like this:

```
public final class SomeClass
{
   // code goes here
}
```

This means that the class cannot be subclassed. In the case of a method it is used like this:

```
public final void someMethod()
{
   // code goes here
}
```

This means that the method cannot be overridden.

9.8 The *Object* class

One of the very useful things about inheritance is the *is-a-kind-of* relationship that we mentioned earlier. For example, when the `ExtendedRectangle` class extended the `Rectangle` class it became a kind of `Rectangle` - so, we can use `ExtendedRectangle` objects with any code written for `Rectangle` objects. When the `PartTimeEmployee` class extended the `Employee` class it became a kind of `Employee`. We have seen in section 9.5 that, in Java, if a method of some class expects to receive as a parameter an object of another class (say, for example, `Vehicle`), then it is quite happy to receive instead an object of a *subclass* of `Vehicle` – this is because that object will be *a kind of* `Vehicle`.

In Java, every single class that is created is in fact derived from what we might call a special “super superclass”. This super superclass is called `Object`. So every object in Java is in fact *a kind of* `Object`. Any code written to accept objects of type `Object` can be used with objects of any type.