04b Inheritance (Example Program)

Inheritance: a Programming Example

In the following example an Employee base class is used and the sub classes Director, Salaried Worker and Hourly Paid Worker are derived from it.

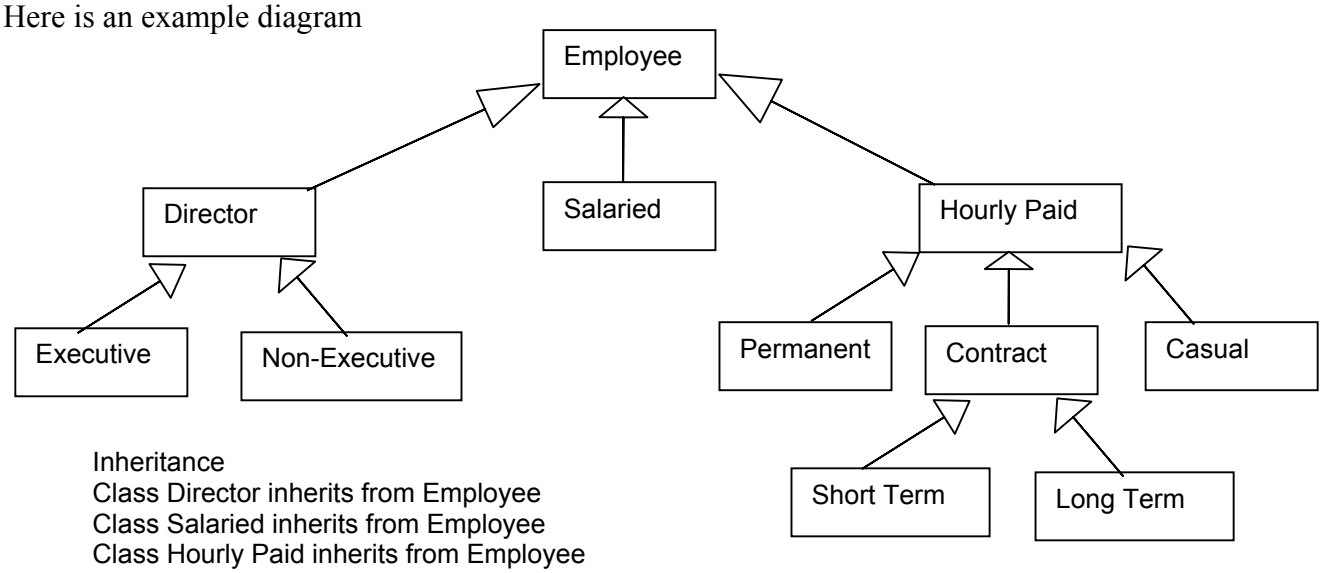
Employee attributes might include details which are applicable to all employees, such as their name and position. However, other attributes might only apply to certain members of staff, such as 'department number' (members of the board may not he in any particular department), or `salary' (which would not apply to some workers who are hourly paid).

Without inheritance we would have to either create completely separate classes for 'Director' and 'Hourly paid worker' (and all other types of employee, duplicating many attributes and methods for each class), or create a large general class of 'employee'. In the latter case we would have to include all possible attributes and methods of any employee in the class, meaning many would not be applicable to particular employee 'objects'. This leads to much redundancy (empty attributes) and unnecessary complication.

Using inheritance, we can put all the attributes and methods which apply to all employees into a base class (an abstract 'employee'). We can then inherit from this class to create more specialised classes for various employee categories. These would add their own attributes and methods to those inherited from the employee class.

In this way the basic class can be built on to cater for all types of employee which exist now or may exist in the future. 'Employee' is an example of an `abstract' class, since it would not be appropriate to instantiate objects from it. To represent the employees that exist in the real world, some further detail must be supplied by derived classes before useable objects can be instantiated.

Here is an example diagram



Inheritance
Class Director inherits from Employee
Class Salaried inherits from Employee
Class Hourly Paid inherits from Employee

Inheritance is a key feature of object-orientation, because it allows us to implement 'polymorphism' (see later), itself a mechanism for generalising the processes in our programs to work with disparate sets of objects. In addition to the benefits outlined above, whereby inheritance reduces duplication and allows existing classes to be reused, inheritance, allows us to treat all objects in a class hierarchy in the same way and ultimately gives us control over large sets of dynamic objects.

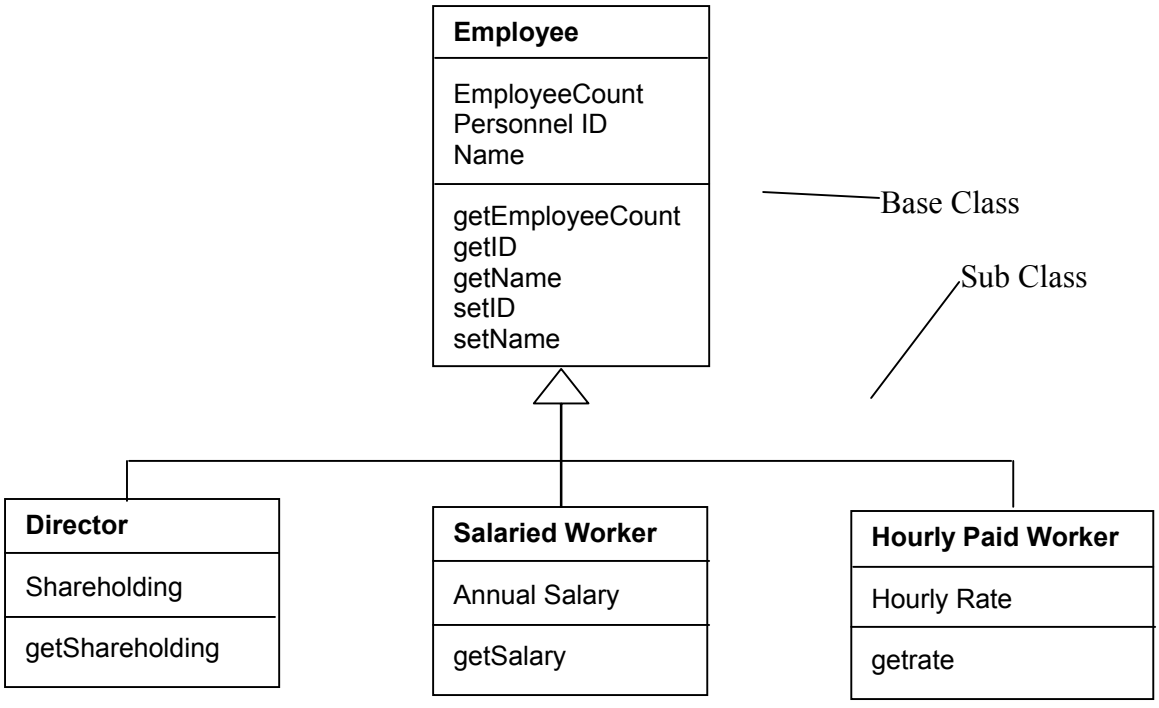
The Employee Example

We referred earlier to the use of inheritance in a set of classes representing employees. In order to code a simplified example based on this scenario we will assume that the following attributes are appropriate to three different types of employee.

Class	Attributes
Director	personnel ID, name, shareholding
Salaried Worker	personnel ID, name, department number, annual salary
Hourly Paid Worker	personnel ID, name, factory section, hourly rate

Although these attributes are a small subset of the likely data in a real employee records system, they should be enough to demonstrate the application of inheritance. In this case, we can use inheritance to avoid duplicating the shared attributes ('personnel ID', pooling the common attributes into an 'abstract' base class called 'employee'.

We can extend this class by inheriting from it into the three 'concrete' classes which add their own class-specific attributes.



The UML notation used here to indicate inheritance is a rather more formalised version rather than the arrows seen earlier. A single outlined arrow head points to the base class with lines drawn to all derived classes. This version of the notation is typically seen CASE tools. Note that the attribute 'employee count' is underlined to show that it is a (static) attribute, likewise the `get employee count' method.

In previous code examples we have typically dealt with one, or at most two, simple classes, and for the sake of simplicity we have seen header files containing both class and method definitions.

> This approach, whilst acceptable for very short examples, does not scale up to larger systems due to the potential for linker errors where header files included in more than one other file, and can also lead to excessive compilation time.
>
> The following example (comprising as it does four classes) demands a more flexible structure. In C++ This would involve separating the class definitions (in header files) from the definitions of their methods (in separate CPP files), with the 'main' function in a separate file again. This means that each CPP file must be separately compiled before the whole program is linked together. You will need to use the facilities of your particular compiler to link the OBJ files generated by compilation into an executable, but modern development environments have simple tools to do this. In this example, we still have a single header file, but the method definitions and 'main' appear in separate CPP files.

The Visual Basic version of the program has 5 files.

Employee	The parent class Employee from which the child classes inherit.
Director	Additional attributes for the Director class and associated Methods
Salaried	Additional attributes for the Salaried class and associated Methods
Hourly Paid	Additional attributes for the Hourly Paid class and associated Methods
The module	Containing Sub Main() –the program sing the classes created.

Employee Class

```
Public Class  Employee
   'Class Employee

   'Attributes
   Public Shared employee_count As Integer       'static varieble
   Dim personnel_id As Integer
   <VBFixedString(10)> Public name As String

   'Methods
   Sub New()   'employee constructor
      ' Increment the employee count
      employee_count = employee_count + 1

      ' generate a unique id from the employee count
      personnel_id = employee_count

      ' get the name from the keyboard
      Console.Write("Enter employee name :")
      name = Console.ReadLine()
   End Sub

   ' class method to return the employee count
   Public Shared Function getEmployeeCount() As Integer
      Return employee_count
   End Function

   ' selector method to return the personnel id
   Function getID() As Integer
      Return personnel_id
   End Function

   ' selector method to return the employee's name
   Function getName() As String
      Return name
   End Function

End Class 'Employee
```

Attributes

Methods

Director Class

```
 ' Class 'Director', inherits from 'Employee'

 Class Director
    Inherits Employee

    'Attributes
    Dim shareholding As Integer

    ' Methods
    Sub New()     'constructor
       ' get the shareholding from the keyboard
       Console.Write("Enter shareholding for {0}  :", name)
       shareholding = Console.ReadLine()
    End Sub

    Function getShareholding() As Integer
       Return shareholding
    End Function

 End Class
```

One added attribute

Methods

Salaried Class

```
Class Salaried
   'Class 'Salaried', inherits from 'Employee'
   Inherits Employee

   'Attributes
   Dim annual_salary As Decimal

   ' Methods
   Sub New() 'Salaried constructor
      ' get the annual salary from the keyboard
      Console.Write("Enter annual salary for {0} :", name)
      annual_salary = Console.ReadLine()
   End Sub

   Function getSalary() As Decimal
      Return annual_salary
   End Function

End Class
```

Hourly Paid Class

```
Public Class HourlyPaid
   'Class 'HourlyPaid', inherits from 'Employee'
   Inherits Employee

   'Attributes
   Dim hourly_rate As Decimal

   'Methods
   Sub New()    'HourlyPaid constructor
      'get the hourly rate from the keyboard
      Console.Write("Enter hourly rate for {0} :", name)
      hourly_rate = Console.ReadLine()
   End Sub

   Function getRate() As Decimal
      Return hourly_rate
   End Function
End Class
```

Main Module

```
'   Program to demonstrate objects of the derived classes of 'EMPLOYEE'
Module Module1

    Sub main()
        '// use constants to size the arrays used to contain employees
        Const MAX_DIRECTORS = 2
        Const MAX_SALARIED = 2
        Const MAX_HOURLY_PAID = 4
        '// use constants to set a standard dividend and working hours in a year
        Const DIV_PER_SHARE = 3.5
        Const HOURS_IN_YEAR = 40 * 52
        '// `i' is used as a loop index
        Dim i As Integer

        '// create 3 arrays of the different employee types using the constants
        '// The constructors will be called here, asking for keyboard input
        Console.WriteLine("Directors:")
        Dim Directors(MAX_DIRECTORS) As Director
        For i = 1 To MAX_DIRECTORS
            Directors(i) = New Director
        Next

        Console.WriteLine()
        Console.WriteLine("Salaried workers:")
        Dim Salaried(MAX_SALARIED) As Salaried
        For i = 1 To MAX_SALARIED
            Salaried(i) = New Salaried
        Next

        Console.WriteLine()
        Console.WriteLine("Hourly paid workers:")
        Dim Hourly_Paid(MAX_HOURLY_PAID) As HourlyPaid
        For i = 1 To MAX_HOURLY_PAID
            Hourly_Paid(i) = New HourlyPaid
        Next

        '// initialise temporary stores
        Dim total_dividend As Decimal = 0
        Dim total_salary As Decimal = 0
        Dim total_wages As Decimal = 0

        '// iterate through the directors array, working out their total dividends
        For i = 1 To MAX_DIRECTORS
            Console.WriteLine("Processing DIRECTOR, ID number {0}", Directors(i).getID())
            total_dividend = total_dividend + (Directors(i).getShareholding() * DIV_PER_SHARE)
        Next

        '// iterate through the salaried array, working out total salaries
        For i = 1 To MAX_SALARIED
            Console.WriteLine("Processing salaried worker, ID number {0}", Salaried(i).getID())
            total_salary = total_salary + Salaried(i).getSalary()
        Next

        '// iterate through the hourly paid array, working out total wages
        For i = 1 To MAX_HOURLY_PAID
            Console.WriteLine("Processing hourly paid worker, ID number {0}", Hourly_Paid(i).getID())
            total_wages = total_wages + Hourly_Paid(i).getRate() * HOURS_IN_YEAR
        Next
```

Continued .. .. ..

.. .. .. Continued

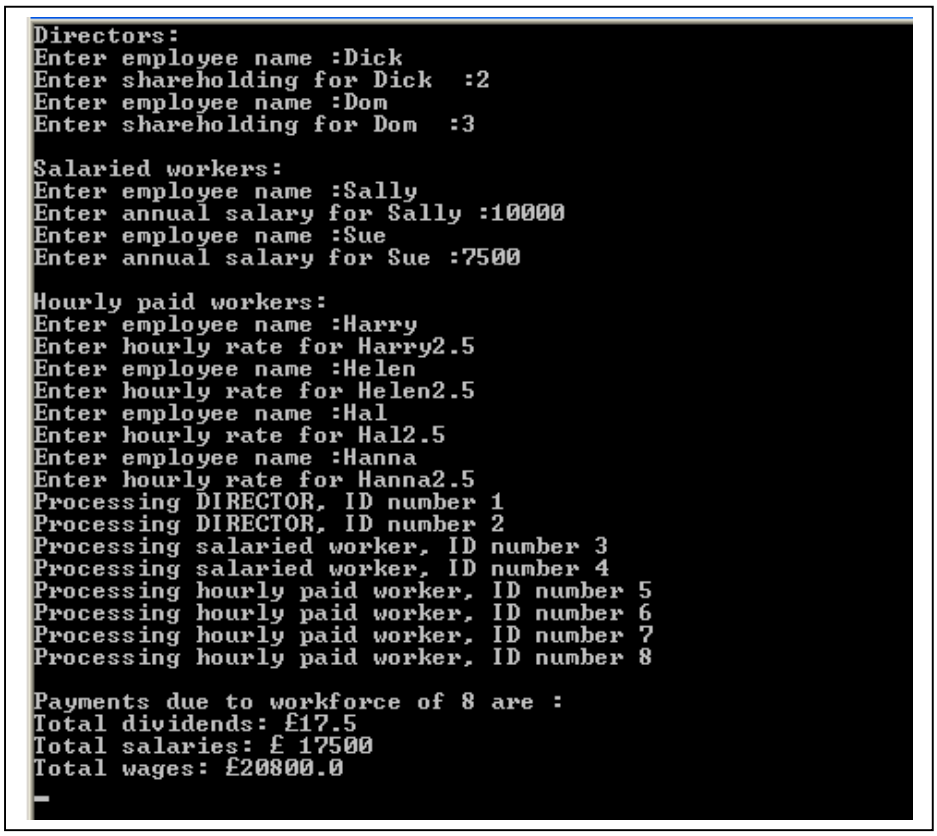
```
    '// display the results
    Console.WriteLine()
    Console.WriteLine("Payments due to workforce of {0} are :", Employee.getEmployeeCount())
    Console.WriteLine("Total dividends: £{0}", total_dividend)
    Console.WriteLine("Total salaries: £ {0}", total_salary)
    Console.WriteLine("Total wages: £{0}", total_wages)

    Console.ReadKey()

  End Sub

End Module
```

When the program is run the screen shows:



Exercise

1) Create the VB.Net program for the employee shown here.
   Run & test the program.

2) In the processing section display the name of the Employee as well as their ID.

3) Improve the final printout to show nicely formatted results.