The purpose of this lab is to get practice at using inheritance.

You'll be setting up an **Employee** class - that has two subclasses :

HourlyEmployee and **SalesEmployee**

Do the lab parts in order.

Part 1 – Employee

Create a new class called **Employee**. It needs attributes :

- firstName (String),
- surName (String),
- staffNumber (int)
- annualSalary (double)

Use encapsulation (as covered last week) – using Eclipse "refactoring" to generate the attribute getters and setters so that you don't have to write them all! If you are using a different IDE, see if you can find out where the generate getter/ setter function is if it has one.

Include a Constructor that sets up all attribute values when an object of Employee is created.

Include a method – calculatePay() – which returns the monthly pay. This is calculated as the annualSalary divided by 12.

Test your code by creating another class (call it Control), putting a "main" method into the Control class – and then from the main method, instantiate an Employee object, print it (using System.out.println(objectName) which calls the toString() method of your object - and call the calculate pay method for the object. How can you see the result for calculatePay()? Print it out?

Part 2 – Hourly Employee (subclass of Employee)

An hourly employee is a type of employee who is paid only for the hours that they work each week. As well as being an employee, they have extra attributes

- hoursWorked - double
- hourlyRate - double

Their pay is calculated differently to employees, and they have an hourly rate. Their annual salary is always set as zero.

Create a new class called **HourlyEmployee** that inherits from Employee.
Self check: how many attributes does an object of type of HourlyEmployee class have?

Include a Constructor to set up all the attributes of an HourlyEmployee object. Hint: You will need the `super` keyword here.

Pay is calculated differently for an hourly employee, so we need to Include a method – **calculatePay()** – which returns the pay as *hoursWorked multiplied by the hourlyRate*. This method is **overriding** the Employee method of calculating pay.

Test your code by instantiating an HourlyEmployee object, and call the calculate pay method for the object.

Part 3 –Employee on Sales Commission (subclass)

An employee on sales commission is type of employee who has a low annual salary, and whose salary is then topped up by commission or reward earned on each sale.

Create a SalesEmployee which inherits from the Employee class. As well as being an employee, it has a specific attribute of type double called:

- commissionEarned

Self check: How many attributes does an object of type of SalesEmployee class have?

Include a Constructor to set up all the attributes of SalesEmployee .

To pay for a SalesEmployee is calculated as annual salary divided by 12, plus commissionEarned. Add a calculatePay() method to **override** the Employee method for calculating pay.

Test your code by instantiating an SalesEmployee *object, calling the calculate pay method for the object and printing the result out to the console.*

----------Part 4 will be covered in class – not as part of the graded lab. But go ahead and try it if you wish to! ----------

Part 4 – Polymorphism - WE will cover this in class

Right now, you have three classes set up – Employee, and two subclasses of Employee: HourlyEmployee and SalesEmployee. They all have their own ways for calculating pay - so each of the three classes has their own method for it.

Do up a sketch out the UML diagram for your three classes..

Now, you'll implement code to demo polymorphism: The right behaviour (i.e method) is executed, based on dynamic checking of the object type at run time.

In your Control class, main method : Instantiate an array that will hold Employee objects

Instantiate each of the entries of the **myEmployees** array with a variety of employees, hourly employees and sales employees.

e.g. myEmployee[0] = new Employee(.., .., ….etc);
e.g. myEmployee[1] = new HourlyEmployee(.., .., ….etc);

Then calculate pay for each of the objects in the array – as shown in class- either as a loop to go through all entries of employees[i].calculatePay() (better)

OR if you don't know how to run an array loop in java, just call each objects calculate pay method:

The right version of `calculatePay()` will get called – because the system detects the object type, and executes the right behaviour for that object type:

Dynamic binding is finding the right calculatePay() method to execute based on type of employee object.

Polymorphism = an Employee object having many forms.