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 **CommissionEmployee**

Do the lab parts in order.

Part 1 – Employee

Note: As per last week, you can use the same java project and package within the project as you did last week – or set up a *new* package within the same project. Or set up a new project and package. It's up to you.

Create a new class called **Employee**. It needs attributes
firstName (String),
surName (String),
staffNumber (int) ,
annualSalary (double)

Include a Contructor that takes in all attribute values when an object of Employee is created.

Include a method – calculatePay() – which returns the monthly pay i.e. the annualSalary divided by 12.

As always, encapsulate the attributes and put in getters and setters (get help on this under the Refactor menu as demo'd in class).

Add a toString() method to your Employee class that returns a String showing the object attributes in a clear way i.e. "This Employee is called .. and has a salary of … " etc.

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)

An hourly employee is someone who is paid only for the hours that they work each week. So, their salary 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. It has extra attributes of *hoursWorked* and *hourlyRate* – both of type double.
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 *hours worked multipled by the hourly rate*. This method is **overriding** the employee method of calculating pay.

Add a **toString()** method for HourlyEmployee. Hint: You will need the `super` keyword here to avoid rewriting code.

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

Part 3 –Employee on Commission (subclass)

An employee on commission is someone who has a low annual salary, and whose salary is then topped up by commission earned on sales.

Create a CommissionEmployee which inherits from the Employee class. It has a specific attribute of "commissionEarned" for the month.

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

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

Employees on commission have their own salary calculations, which is different to ordinary employees. To do this, include a method – calculatePay() – which returns pay calculated as annual salary divided by 12, plus the commissionEarned. This method will be **overriding** the Employee method for calculating pay.

Add a toString() method for CommissionEmployee. Hint: You will need the `super` keyword here to avoid rewriting code.

Test your code by instantiating an CommissionEmployee *object, printing it, and calling the calculate pay method for the object.*

If you comment out the toString() method in the CommissionEmployee class, and print the object again, what happens? What is being called and why?

Part 4 – Polymorphism

Right now, you have three classes set up – Employee, and two subclasses of Employee: HourlyEmployee and CommissionEmployee. They all have their own "rules" for calculating pay - so each of them 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 : Create an array that will hold Employee objects:

```
Employee [] myEmployees = Employee[somelength]
```

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

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

Then calculate pay for each of the objects in the array – as shown in class- either as a loop to got 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:

```
employees[0].calculatePay() ;
employees[1].calculatePay() ;…. Etc
```

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. Polymorphism = an Employee object having many forms.