LAB 2: BUILDING THE ZOO EXAMPLE

Aims:

- Implement some of the object-oriented examples we saw in lecture 2 (to remind yourself, go back to lecture 2 slides)
- Gain more experience of object oriented programming
    - Remember that familiarity with this will not only help you write better software, but will aid understanding of existing projects

Coverage:

- In Part 1 we cover:
    - setters, getters, static variables, ArrayLists, and some Eclipse tips/shortcuts
- In Part 2 (second half of this worksheet) we cover:
    - Composition versus aggregation
    - Writing the employee classes for the Zoo example, including using an abstract class and an interface
    - Understanding Packages

LAB 2 - PART 1

THE ZOO CLASS

First we need to write the Zoo class, and to do that, we need to create a new project in Eclipse. Look at last week's sheet if you forgot how to do this.

Implement the example from the lecture, first of all create a new Zoo class and add the code below. (Remember you can use Eclipse to generate getters/setter for you if you want: add the private `location` field, then go to Source menu→Generate Getters and Setters…)

- If you are not sure about why you need setters/getters, please ask a helper

```
2  public class Zoo {
3      // Private fields
4      private String location;
5
6      public String getLocation() {
7          return location;
8      }
9
10     public void setLocation(String location) {
11         this.location = location;
12     }
13
14     // Constructors
15     public Zoo(String location){
16         this.location=location;
17     }
18
19     public Zoo(){
20         this("Unknown");
21     }
22 }
```

- Add a private variable to store the number of enclosures (ie. the total number of fields, cages, tanks etc.) the zoo has.
- Add a line to the default constructor to set this to a default value (say, 30 enclosures). Add a parameter to the other constructor to read in any value for this.

TIPS TO TRY

- CTRL+space half way through a variable, set of parameters or method name gives you a list of possible completions
- Source menu→Correct indentation fixes the indentation on any lines you have selected
- CTRL+F11 will run the application (saves clicking the green arrow each time!)

- Add a function called `buildNewEnclosure` which adds one to this variable.

Now so far we have just designed the class. Let's now actually create a real instance of it in our program (the object itself).

- Add a `ZooApp` class and `main` method, as before.
- Create 2 new Zoo objects. For one, use the default constructor (ie. don't pass any parameters on construction). For the second, pass an exotic location of your choice, and number of enclosures.

Let's now add a function which prints to the screen information about the zoo. Rather unexcitingly, let's call it printInfo

- Add the `printInfo` method to the Zoo class. It should print to screen the location, and number of enclosures. Should this be public or private?
- Maybe pause here, and have a look at some useful **shortcuts** in Eclipse. Or use Google and try and find some others. They will help you write code a lot faster, and these labs are the perfect place to try them out. We will introduce some in these worksheets, but there are many more out there e.g.:
    - https://dzone.com/articles/effective-eclipse-shortcut-key

THE CONCEPT OF A COMPANY OWNING LOTS OF ZOOS

So you can see above we can have multiple version of our zoo classes. Why would we want to do this? Suppose you are a company, MegaZoo Corp. who is opening five zoos around the world, in London, Tokyo, New York, Paris and Beeston.

- In your `main` code, create five zoo objects to represent this. You can set the enclosure numbers as you wish.
- Print out the info for all 5 zoos.

Suppose you want to assign each Zoo an individual Zoo ID number.

- Think about how you could do this WITHIN the objects themselves
- So we want the first zoo created to have ID 1, second zoo ID 2 etc.
- When you have thought of an idea, then follow the example below…

You can do this using the `static` variable type. These variables are shared and accessible between all objects of the same type. First we need to count how many zoos we have. So we can share our zoo count (called *numberOfZoos* below) between all the objects.

Add a `static int` variable called numberOfZoos:

```
public class Zoo {

    private String location;
    private int numberOfEnclosures;
    public static int numberOfZoos;
```

You can actually access static variables *without* first creating an object. They don't need an object as they are really related to the class itself – all objects of the same class share the variable. Look - I can set its value to 0 **without** having to create an object first!

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

    Zoo.numberOfZoos = 0;
```

(The variable is public so I can do this. If it were private I could not access it outside the class like this, and would have to use a setter method.)

Remember, static means the variable is shared/accessible between all object instances. Let's see what that means if we want to use it to count how many zoos there are.

- Every time we create a new zoo, let's add one to this shared variable.

    - Task: You need to work out where in the zoo class you need to handle this increment of our new variable, and add code to do it.

Now print out the number of zoos after the 5 we have created:

```
//our 5 new zoos:
Zoo zL = new Zoo("London", 10);
Zoo zT = new Zoo("Tokyo", 20);
Zoo zN = new Zoo("New York", 35);
Zoo zP = new Zoo("Paris", 40);
Zoo zB = new Zoo("Beeston", 42);

System.out.println("Number of zoos =" + Zoo.numberOfZoos);
```

Your output should be: `Number of zoos =5`

You can use this now to set an individual ID number (private int) per Zoo in the constructor (zooID = numberOfZoos?)

So static variables are an easy way to share data between objects of the same type. Make sure you are happy with this idea – it's an important concept. Please ask a helper if you would like more explanation.

ADDING ACTUAL ENCLOSURES WE CAN PUT ANIMALS IN: USING COLLECTIONS

So we have the concept of a zoo, but at the moment it's not very interesting. We need a new class to represent our enclosures or compounds, such as tanks, cages, fields etc.

- Add a new Enclosure class:

```
public class Enclosure {

}
```

Think back to the lecture where the idea of Compounds (same as enclosures) was introduced. One zoo can have many enclosures. So how do we represent this?

- We can use a *collection*
- Java has built-in data structures for handling lists etc.

We will use an ArrayList. Add this to the list of variables in the Zoo class:

```
public class Zoo {

    private String location;
    private int numberOfEnclosures;

    public static int numberOfZoos;

    private ArrayList<Enclosure> enclosures;
```

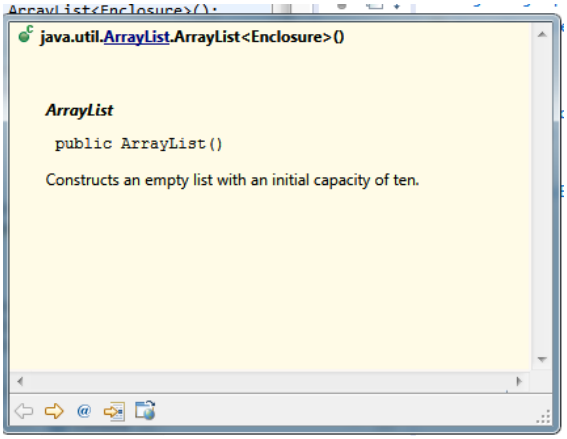
Note Eclipse is not happy! It has underlined it in red – it doesn't know what an ArrayList is. We need to add an import statement to tell Java how to find the relevant library.

A neat trick in Eclipse in this case is to go to Source→Organize Imports. It should automatically add the correct import statement at the top of the file for you!

```
1 import java.util.ArrayList;
```

An ArrayList is really just an array, but suitable for storing objects, and with a number of helper methods. It is dynamic – that is, it can grow as the number of elements grows.

- To see more about what an ArrayList is, position the mouse inside the word ArrayList, and press F2. Click the underlined link in the first line of the pop up for even more info:

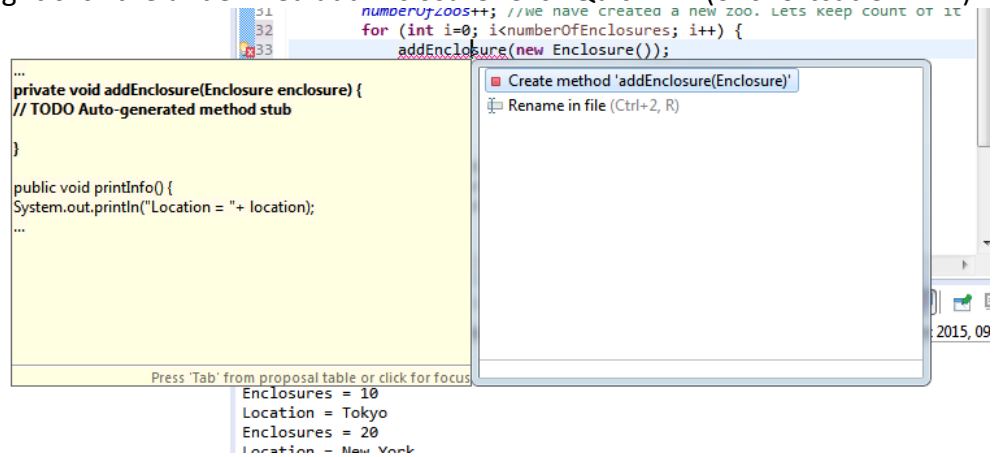


- Now add the highlighted text below to the Zoo constructor. This will create all our Enclosure objects for us:

```
public Zoo(String location, int numberOfEnclosures) {
    //main constructor
    this.location = location;
    this.numberOfEnclosures = numberOfEnclosures;
    numberOfZoos++; //We have created a new zoo. Lets keep count of it
    for (int i=0; i<numberOfEnclosures; i++) {
        addEnclosure(new Enclosure());
    }
    }
```

Note how we can create **new** objects inline like this. Also note, we haven't written an addEnclosure method yet! Don't panic. Eclipse can help!

- Right click the underlined addEnclosure. Click Quick Fix. (or shortcut CTRL+1)



- Double click create method.

As if by magic, the following method should be produced. Note Eclipse has figured out the parameter type it needs:

```
private void addEnclosure(Enclosure enclosure) {
    // TODO Auto-generated method stub

}
```

- This autogeneration is useful for Test Driven Development which we will introduce later, as you can write the test before you have written the code, and generate the basic code outline from the test code itself.

Now you just need to add the actual code inside the method:

```
private void addEnclosure(Enclosure enclosure) {
    this.enclosures.add(enclosure);

}
```

If you run this, you will get an error as we haven't created our **enclosures** object yet.

- Add this line to the constructor. It instantiates our ArrayList object and gets it ready to accept objects of type Enclosure.

```
public Zoo(String location, int numberOfEnclosures) {
    //main constructor
    this.location = location;
    this.numberOfEnclosures = numberOfEnclosures;

    enclosures = new ArrayList<Enclosure>();

    numberOfZoos++; //We have created a new zoo. Lets keep cou
    for (int i=0; i<numberOfEnclosures; i++) {
        addEnclosure(new Enclosure());
    }
    }
```

- Now the code should compile! You are creating a number of new Enclosure objects, equal to the numberOfEnclosures variable.

Challenges:

- Using the same idea, change the Enclosures class so you can add animals, by using an ArrayList again. You will need to create a class Animal.

```
2  public class Animal {
3
4  }
```

And you will need a method to add animal objects to your new ArrayList:

```
public void addAnimal(Animal animal){
    animals.add(animal);
}
```

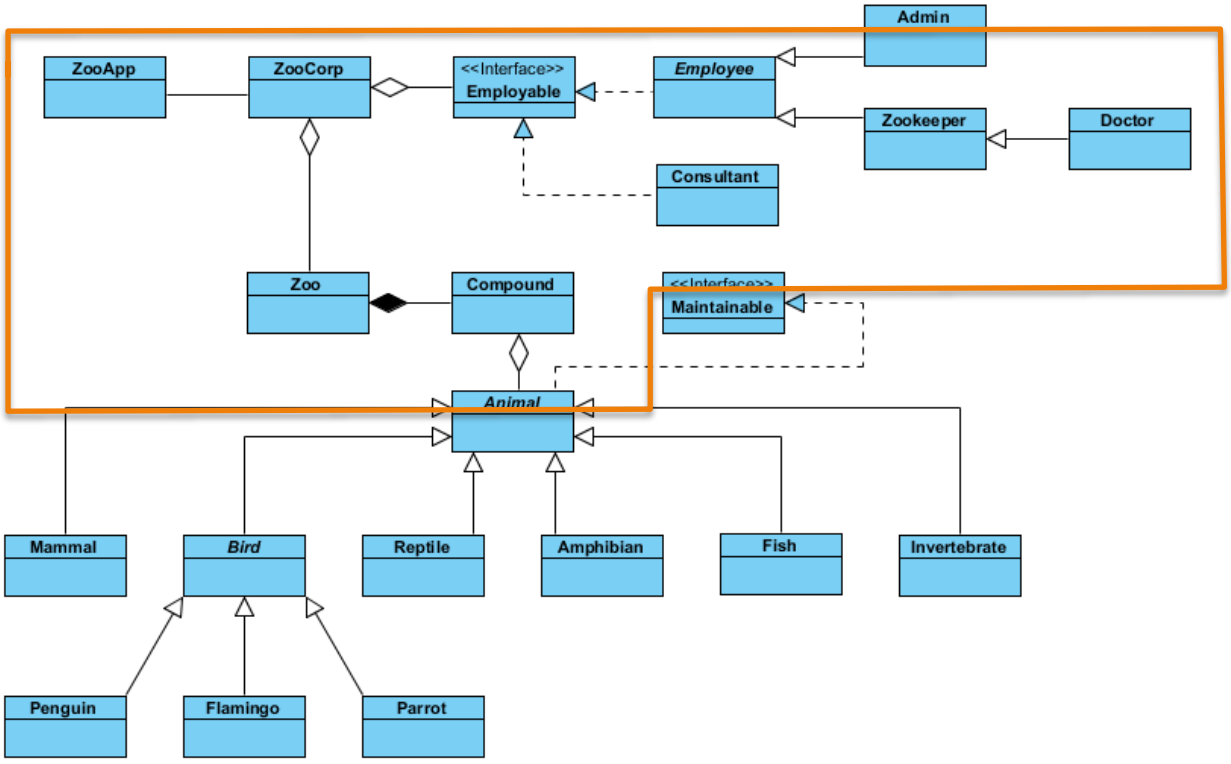
- We will look at how we can use abstract classes and inheritance with animals in Part 2.

LAB 2 - PART 2

Part 2 builds on the code developed in Part 1.

ADDING SOME EMPLOYEES

In the Lecture, we showed you the diagram below which can represent the Zoo:



You're task in the rest of this lab is to modify the existing code, and implement the new employee section above (in the orange box). We will introduce the main steps below:

ZooCorp:

You will need to modify the ZooApp class to use a new ZooCorp object. ZooCorp represent a company which may own lots of Zoos. However, a Zoo can exist before its bought by the company, so it makes sense to use an **aggregation** (Note the open diamond shape).

So let's allow Zoo objects to exist outside of the ZooCorp class. It makes sense to have a ZooCorp Constructor which accepts a Zoo, so let's add that as an option. A good data structure to use might be the ArrayList as we used in Part 1 e.g.

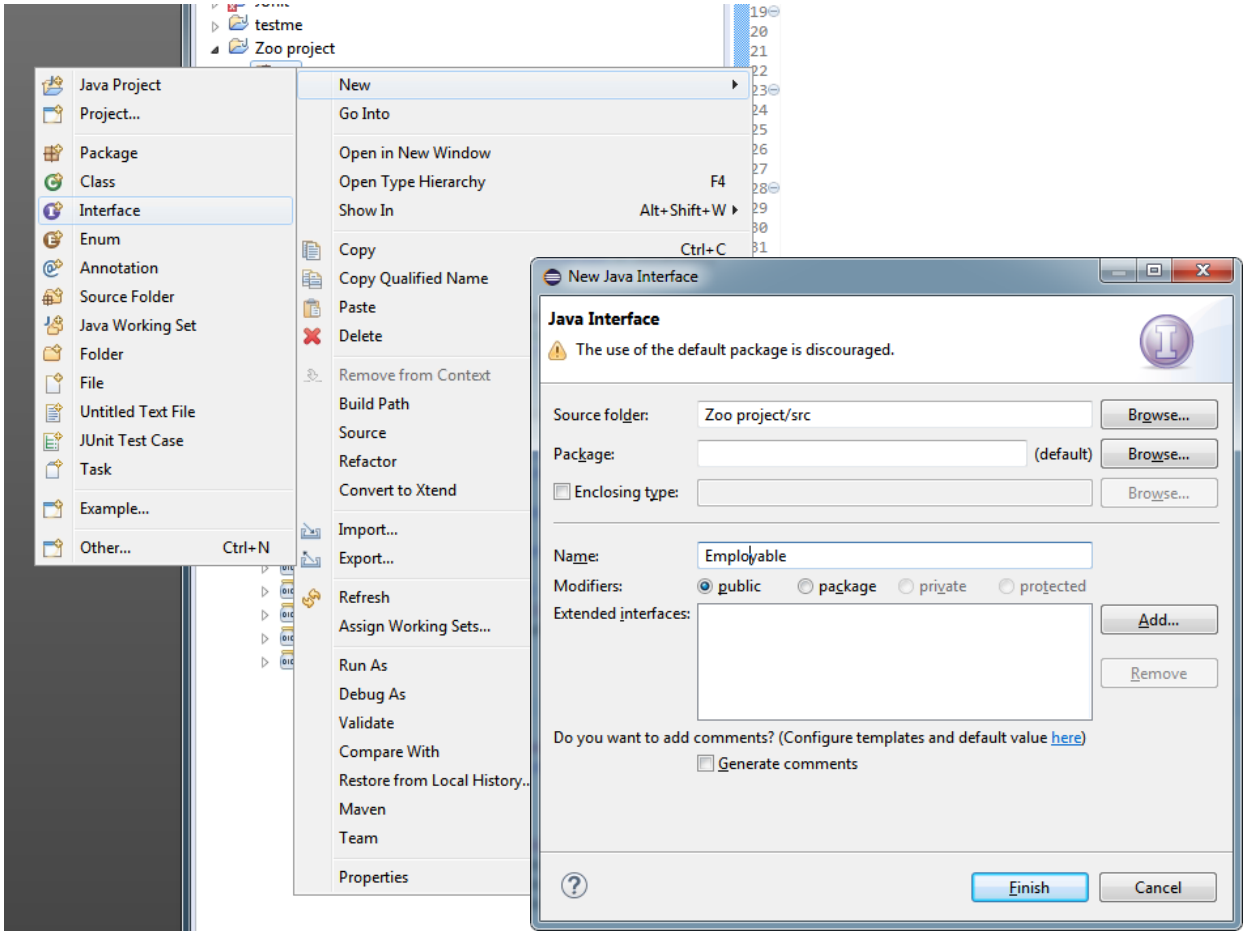
```
private ArrayList<Zoo> zoos;
```

Task

- Write a constructor which takes a Zoo object as a parameter and stores it in an ArrayList.
- Then, write a method called addZoo which also takes a Zoo object as a parameter and again adds it to the ArrayList.

WRITING THE EMPLOYEE CLASSES

Look again at the class diagram above. To write the employee classes, we need to write an interface, as they are written using one called Employable. We'll think about the details of this in a while, but for now let's just create one:



Right click the source and select New→Interface, and fill in the pop up as above.

If you remember, an interface is simply a blueprint for your class: it lists the method names that the class *must* implement. This ensures that any class which uses that interface has at least those methods accessible publicly, for other parts of the program to use.

Why use interfaces? Think of it like this. Whatever TV you have at home, you can be sure it has an on/off button. So, to represent this, in programming you could write an interface called Switchable which has two methods, `switchOn`() and `switchOff`(). Then *any* brand of TV can implement this interface, and you know you can turn it on and off, using the methods it provides.

Here, we are interested in objects which are *employable* – as they represent staff at ZooCrop. All Employable objects should implement the following methods, so add them to the interface:

- void setEmployeeID(int number)
- int getEmployeeID()
- void setEmployeeName(String name)
- String getEmployeeName()
- int getSalary()
- setSalary (int salary)

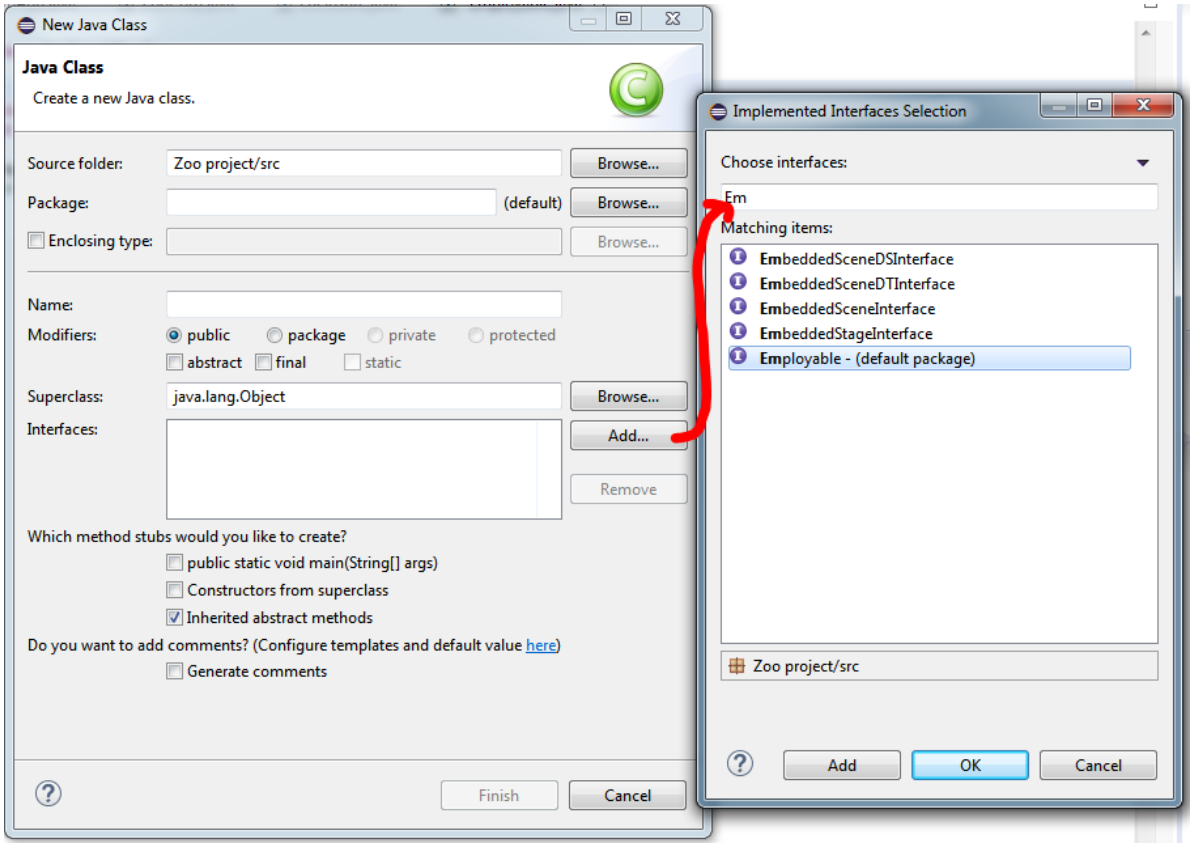
Remember in an interface you just list the method signatures, NOT any code e.g.

```
public void setEmployeeID(int number);
```

It is the job of the class *implementing* the interface to provide the code. We don't care *how* the function works at this stage, just what goes into it (parameters), and what returns from it.

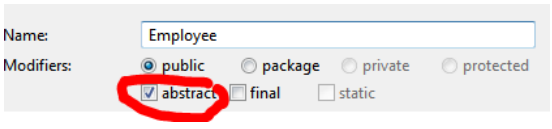
To create a class which implements the interface:

Create a new class as before, but this time, next to the Interfaces box, click Add…:



Start typing Employable, and you will see that your new Employable interface appears in the list. Select and press OK.

Name the class Employee, and then **set the class to abstract**:



Note you can use interfaces in a non-abstract class just fine. But let's do it here to match our diagram above, and to see what using an abstract class can add….

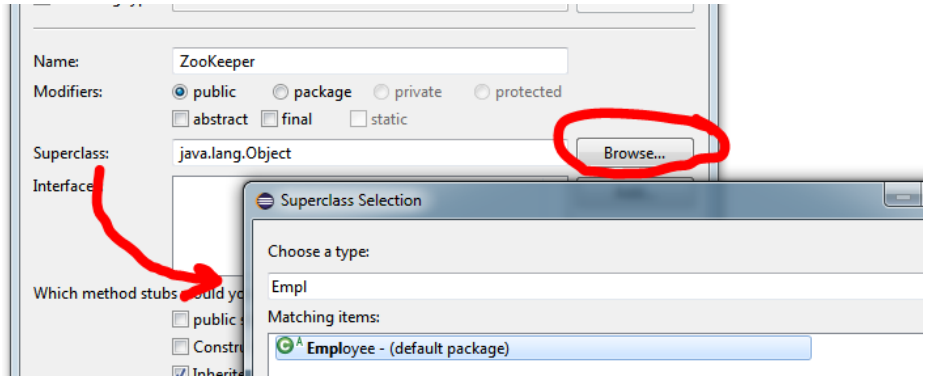
Press Finish. Eclipse will now do something useful – it will create 'stubs' of all the methods you need to implement in the interface.

So, Employee is an **abstract** class. This means we can't create an object directly from this class. This wouldn't make much sense – you can't have a generic Employee, everyone has a particular job (Zoo keeper, admin, till operator etc.) But an abstract class is more than just an interface – we can define some generic behaviour here which all employee objects can use. For example, actually provide implementations for all the methods so far, as they are all very general in nature, in this abstract class:

- Write the code for the stubs. You will need to add some variables.
- Note, **interfaces** do not define the required variables; this is an *implementation* choice we can now choose in the abstract class.

Now let's **add a ZooKeeper class which inherits from this abstract class**:

- Go to the create new class dialog box, and set the name to ZooKeeper and Superclass to Employee:



This will now compile, as the ZooKeeper has access to all the methods we have coded in the abstract class, Employee.

Let's show why you can't code everything in an abstract class. Now add another method to the Employable interface:

- Add: public int calculateChristmasBonus() to the Employable interface

```
public interface Employable {


    public void setEmployeeID(int number);
    public int getEmployeeID();

    void setEmployeeName(String name);
    public String getEmployeeName() ;

    public void setSalary(int number);
    public int getSalary();

    public int calculateChristmasBonus();

}
```

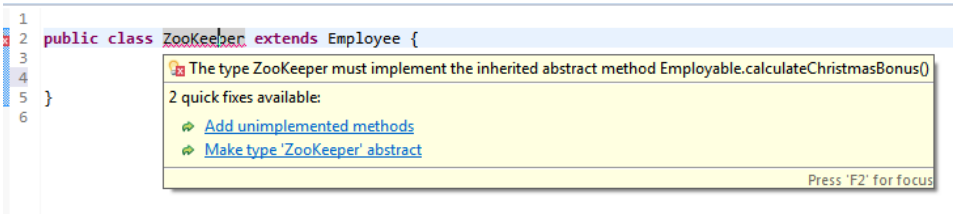
When you implement this, different roles in the Zoo have different bonuses:

- For a ZooKeeper, the Christmas bonus equation is (salary x 0.05+100)
- For an Admin role, the bonus is (salary x 0.08)

So you can't add the implementation into the generic abstract class.

The code will no longer compile, as you need an implementation of this (because the interface says you need one!)

- Go to your ZooKeeper class. Click the red underlined name of the class, and here Eclipse can help us! Click Add unimplemented methods:



Hey presto! The required method stub appears:

```
public class ZooKeeper extends Employee {

    @Override
    public int calculateChristmasBonus() {
        // TODO Auto-generated method stub
        return 0;
    }


}
```

- For your ZooKeeper class, add the bonus equation above
- Now add an Admin class, which also inherits from Employee, and implement the different Admin bonus equation.

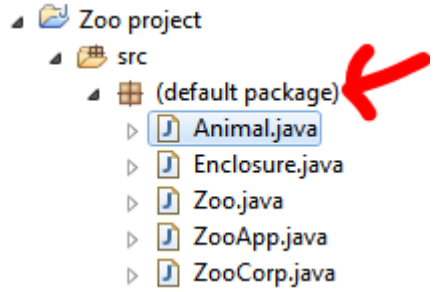
So, the **interface** says there HAS to be a method called calculateChirstmasBonus which return an integer. But we can't put a generic calculation in the **abstract class** Employee because the exact calculation changes depending on the Employee type. So we put the final implementation of this in our actual person classes (ZooKeeper, Admin etc.).

Tasks

- To tidy up, add a constructor for ZooKeeper and Admin to set a name on creation.
  - Could you put this constructor in the abstract class instead? Think about how
- Modify ZooCorp, as you did for Zoos, to accept Employable people, and maintain an ArrayList of employees.

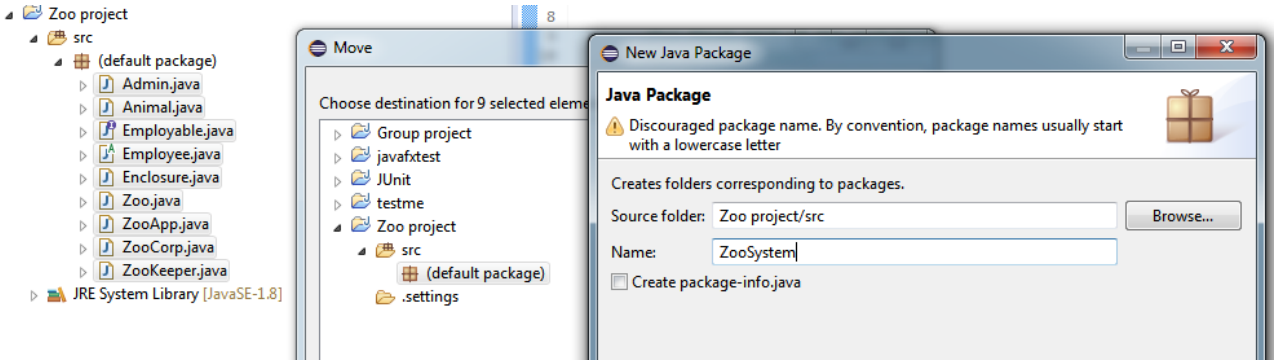
PACKAGES

You may have noticed everything we have written so far is getting put in a Default Package in Eclipse (unless you have set your own up already):



What is this and what does it mean?

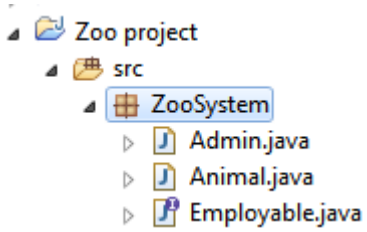
- Let's move all our files to a new package first. Select all your java files related to the zoo work (e.g. use shift-clicking or similar.) Right click the selection, choose Refactor, then Move…



- Choose Create Package, and fill in a package name, e.g. in the example above.

When you're done, you might notice two changes:

1. In the browser, your code now appears under your new package name instead of the Default:



2. All your files now have a new line at the top of each file:

```
package ZooSystem;
```

This collects all the classes and methods together into this new package. It will help prevent naming clashes, and organise who can access the members of the package. Suppose you later build a system for organising staff at a theme park: you may also have employee classes there, but if they were all in the same package, you would have to give them separate names. If they are in separate packages, classes can have identical names and they will not clash.

To use a class inside a package from *outside* the package, we use naming like this:

```
ZooSystem.Admin
```

Or you can import a whole package:

```
import ZooSystem.*;
```

So *that's* where `import` comes from!

There is more on packages here:

https://docs.oracle.com/javase/tutorial/java/package/packages.html

---

This ends the REFRESHER part of the course.

The aims of the labs so far have been to:

1. Give you more experience of concepts from Year 1 which are important for maintainable software design, and understanding existing systems
2. Give you further practical OO programming experience which will be useful in the remainder of this module, including coursework 2
3. Provide experience of using Eclipse

This addresses the following outcomes form the module catalogue:

To build on first year programming modules and further develop programming ability and experience; Design and write object-oriented programs; understand the complex ideas of programming solutions and relate them to particular problems.