

Lab 2: Building the Zoo – Part 1

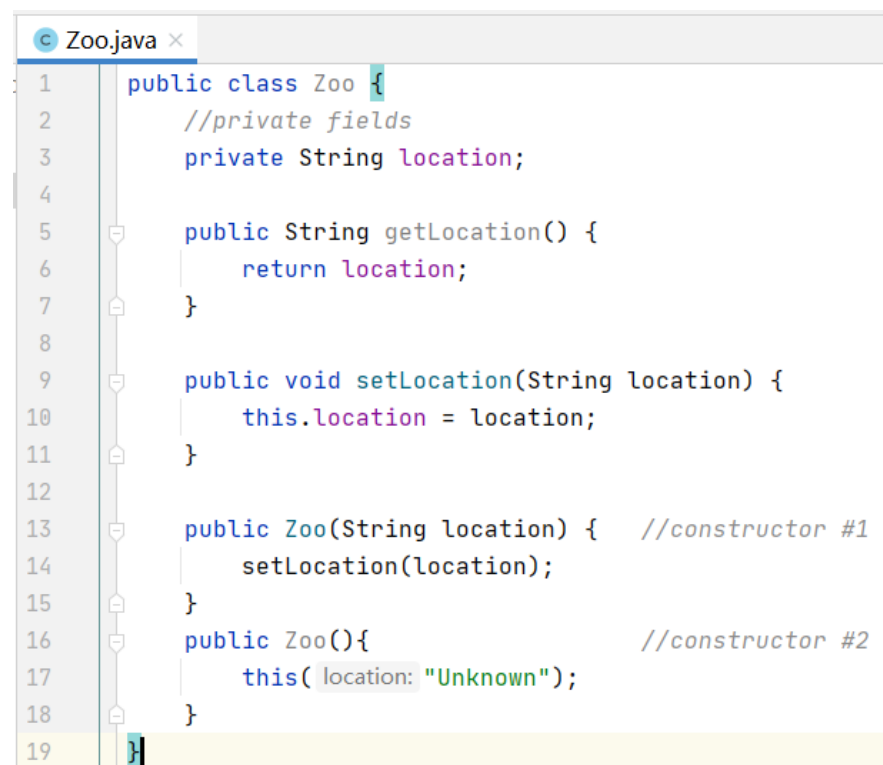
Part 1 Aims

1. Implement some of the object-oriented examples.
2. Gain more experience in OOP.
3. Covers setters, getters, static variables, ArrayLists and some IntelliJ tips/shortcuts.

You might not be able to finish it all during the lab, so please continue with this at home and ask questions during the next lab session.

The Zoo class

First, before creating the **Zoo** class, create a new Java project in IntelliJ, named “Lab2_Zoo”. Note: Create an empty project; do not “Add sample code”. In the project, create the Zoo class. Implement the example shown below.



```
1 public class Zoo {
2     //private fields
3     private String location;
4
5     public String getLocation() {
6         return location;
7     }
8
9     public void setLocation(String location) {
10        this.location = location;
11    }
12
13    public Zoo(String location) { //constructor #1
14        setLocation(location);
15    }
16    public Zoo(){ //constructor #2
17        this( location: "Unknown");
18    }
19 }
```

===

Task:

- a. Add a private variable, named “numCompounds”, to store the number of

- compounds (i.e. the total number of fields, cages, tanks etc.) that the zoo has.
- b. Modify the 1st constructor to read in an initialization value for this variable. Modify the 2nd constructor to set a default value (e.g., 30 compound) for this variable.
 - c. Add a function called **buildNewCompound** which adds one to this variable, and return nothing.

===

Tip: In the menu, use “Code – Auto-Indent Lines” to fix the indentation on any selected lines. (Shortcut for Windows: Ctrl+Alt+I) For Mac users, the list of shortcuts is found here:

https://www.jetbrains.com/help/idea/Reference_Keymap_Idea_Osx.html#keymap=secondary_intellij_osx

The shortcuts will help to write code a lot faster, and these labs are the perfect place to try them out.

So far, the class has been designed. Let’s create a real instance of it in the program (the object itself).

===

Task:

- d. Beside the Zoo class, add a **ZooApp** class, in which create a *main* method.
- e. In the main method, create two new Zoo objects. First, use the default constructor (i.e., don’t pass any parameters on construction) and for the second, pass an exotic location and number of compounds as parameters.
- f. Add a **printInfo** method to the Zoo class. It should print the location and number of Compounds to the console window. Should this be public and private?
- g. Run and verify the above code.

===

A company owning several Zoos

Multiple Zoo objects can be created. Why is this wanted? Suppose you own a company, *MegaZoo Corp.*, which is to open 5 zoos around the world, in London, New York, Paris, Tokyo, and Ningbo.

===

Task:

- a. In the main method, create 5 Zoo objects, set the compound numbers as you wish.
- b. Print out the information for all 5 zoos.

===

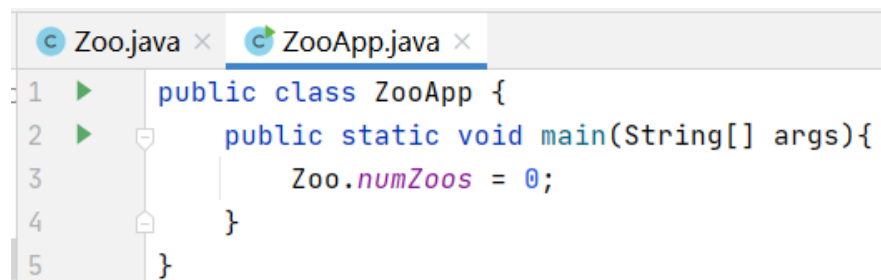
Suppose you want to assign each Zoo an individual Zoo ID number.

- Think about how you could achieve this using static variables.
- The first zoo created is assigned to ID 1, second zoo as ID 2 etc.

The static variables are shared and accessible between all objects of the same type. First, we need to count the number of zoos we have, so we can share the zoo count (called **numZoos**) between all the objects.

```
public class Zoo {
    //private fields
    private String location;
    private int numCompounds;
    public static int numZoos;
```

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. Set its value to 0 without having to create an object first!



```
1 public class ZooApp {
2     public static void main(String[] args){
3         Zoo.numZoos = 0;
4     }
5 }
```

The variable is initialized as public. If it were private, the variable cannot be accessed outside the class and you would have to use a setter method.

Remember, 'static' means the variable is shared/accessible between all object instances. Let's see how it works to count how many zoos there are.

- Every time a new zoo is created, it adds 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 it.

Now, print out the number of zoos after 5 zoos are created.

```
Zoo zL = new Zoo( location: "London", numCompounds: 10);
Zoo zT = new Zoo( location: "Tokyo", numCompounds: 20);
Zoo zN = new Zoo( location: "New York", numCompounds: 35);
Zoo zP = new Zoo( location: "Paris", numCompounds: 40);
Zoo zB = new Zoo( location: "Beeston", numCompounds: 42);
System.out.println("numZoos:" + Zoo.numZoos);
```

The output should be: "Number of zoos = 5".

So static variables are an easy way to share data between objects of the same type. Make sure you understand this important concept.

Adding actual compounds: Using Collections

Now, we need a new class to represent our enclosures or compounds, such as tanks, cages, fields etc.

- Create a new “Compound” class.

```
public class Compound {  
  
}
```

One Zoo can have many compounds, how to represent this?

- 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:



The screenshot shows an IDE with three tabs: Zoo.java, Compound.java, and ZooApp.java. The Zoo.java tab is active, showing the following code:

```
1 import java.util.ArrayList;  
2  
3 public class Zoo {  
4     //private fields  
5     private String location;  
6     private int numCompounds;  
7     public static int numZoos;  
8     private int ZooID;  
9     private ArrayList<Compound> compounds;  
10
```

Note that the ArrayList library is automatically inserted in the first line in IntelliJ IDE. If not, put cursor on the word ArrayList, and click on “import class”. Also, you need to be aware that if using other IDE, it may not always act like this. You might need to manually insert the import clause.

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

- To check more about what an ArrayList is, position the cursor on the word ArrayList. Then its definition will automatically appear. Or position the cursor on the ArrayList and press ‘ctrl’ (if Windows), then the method head will appear. Click the underlined link to see its class definition.

Now, add the *for* loop below to the Zoo constructor. This will create all our Compound objects for us.

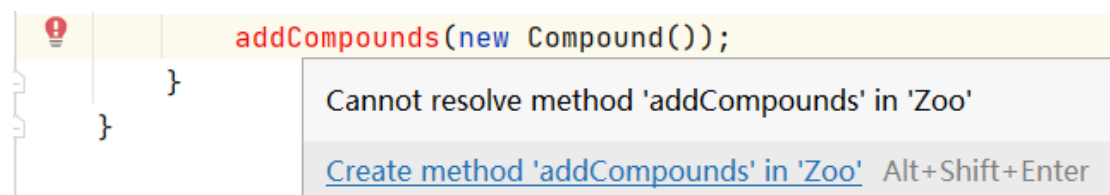
```

} public Zoo(String location, int numCompounds) { //constructor #1
    setLocation(location);
    setNumCompounds(numCompounds);
    numZoos++;
    zooID = numZoos;
    for (int i=0; i<numCompounds; i++){
        addCompounds(new Compound());
    }
}
}

```

Note how new objects can be created inline. Also note, the “addCompound” method haven’t been written yet! Don’t panic. The IntelliJ IDE can help.

Put cursor on the word “addCompounds”, then a pop-up shows as below.



Click on the link, then an **addCompound()** method will be generated. Now, add the necessary code into this method.

```

} private void addCompounds(Compound compound) {
    this.compounds.add(compound);
}
}

```

Note that the ArrayList “compounds” has not been initialized yet, but only declared. Running it, you probably would get a “NullPointerException” error. To deal with this, in the constructor, add the following line to create a “compounds” object.

```

    setLocation(location);
    setNumCompounds(numCompounds);
    compounds = new ArrayList<Compound>();
    numZoos++;
    zooID = numZoos;

```

Now the project should compile!

===

Task: Extending the Zoo Project

- c. Using the same idea, change the “Compound” class to add animals, by using ArrayList again. Here, create a class “Animal”.
- d. You also need a method in the **Compound** class to add animal objects to the new ArrayList.

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

- e. Look at how abstract classes and inheritance with animals is used in Part 2.

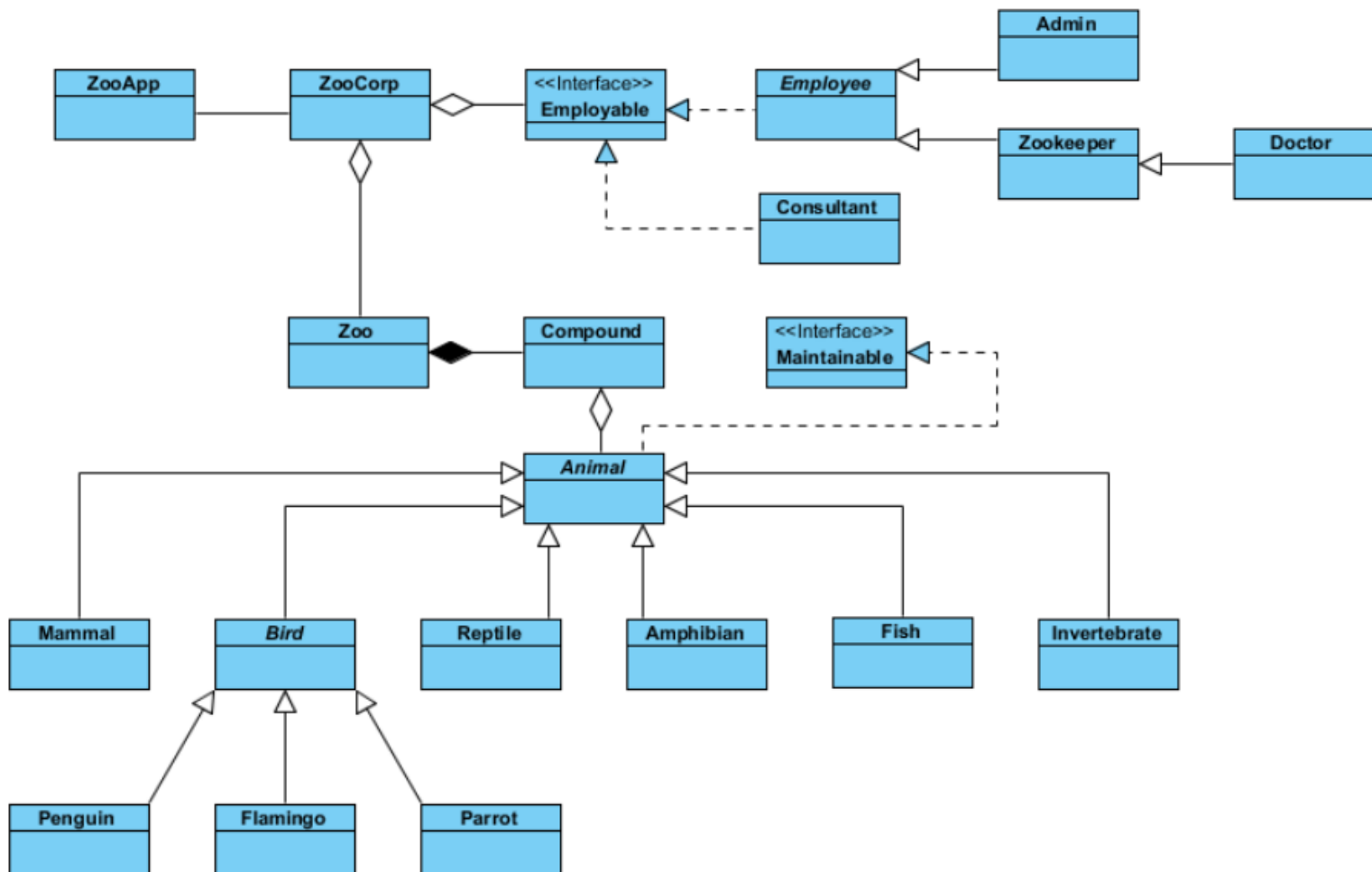
===

Lab 2: Building the Zoo – Part 2

Part 2 Aims

1. Composition vs. Aggregation
2. Coding the “Employee” classes for the Zoo example, using abstract class and interface. Practicing polymorphism.
3. Understanding packages.

Adding ZooCorp



You will need to modify the “ZooApp” class to use a new “**ZooCorp**” object. ZooCorp represent a company which owns multiple Zoos. However, let’s assume that in this project, a Zoo can exist before it is being bought by the company, so it makes sense to use an **aggregation** (shown by open diamond shape).

So, let’s allow Zoo objects to exist outside of 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 ArrayList as well.

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

===

Task: Extending the ZooCorp Class

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

an ArrayList. Of course, the ArrayList should be initialized in the constructor.
===

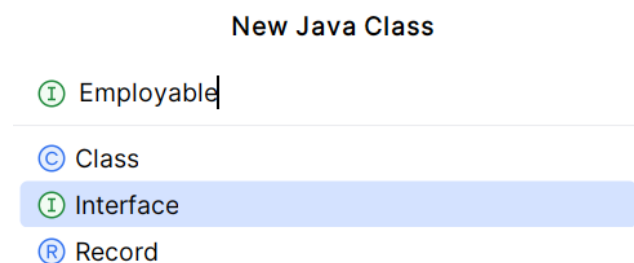
Notice the **composition** relationship between Zoo and Compound. In this project, a Compound cannot exist individually outside the definition of Zoo. Namely, the Compound lives and dies with the Zoo. In order to code the composition relationship, make sure that **any Composition object can only be created and initiated within a Zoo object**. Check the Zoo.java reference source code for Lab 2 uploaded in Moodle. Especially, look at Zoo's constructor: does it accept a Compound object in parameters?

A nice explanation about aggregation, composition, and association can be found here <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/>

Adding some employees

Look again at the class diagram above. To write the Employee classes, you need to write an *interface*, called **Employable**. We'll think about the details of this in a while, but for now let's create one.

Right click the "src" folder and select New – Java Class, fill in the pop up as below.



An interface is simply a blueprint for the 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 interface? Think of it this way. Each TV (whatever brand) at home surely has an on/off button. So, to represent it in your code, you need to write an interface called **Switchable** which has two methods "*switchOn*" and "*switchOff*". Then any brand of TV can implement this interface, and can turn it on and off, although the detailed methods used can vary.

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

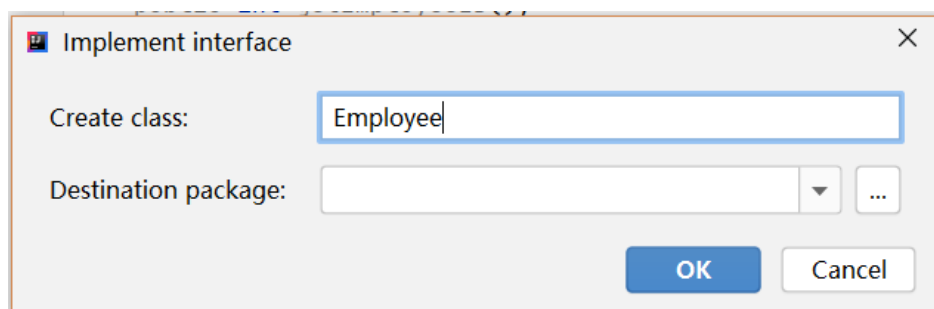

```

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

```

Remember that the interface lists the method signatures, NOT implementations. **Each one ends with a semi-colon ‘;’**. It is the job of the class implementing the interface to provide the code. There is no concern about 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, in the `Employable.java`, put the cursor on the interface name, `Employable`, right click, and click the “Show Context Actions” menu, then “Implement Interface”. Leave the `Employee` class in the same package as other classes now.



In the next pop-up, select all methods to implement. Keep everything as is, and click OK.

In the generated **Employee** class, we add the highlighted keyword, ‘*abstract*’, to make this class an *abstract class* to match the UML diagram above.

```

1 package com.ae2dms.zoolab2;
2
3 public abstract class Employee implements Employable { 2 usages
4     private int ID; 2 usages
5     private String name; 2 usages
6     private int salary; 2 usages

```

Now, “**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 (Zookeeper, admin, operator etc.). But an abstract class is more than just an interface – we can define some generic behavior here which all

employee objects can use.

===

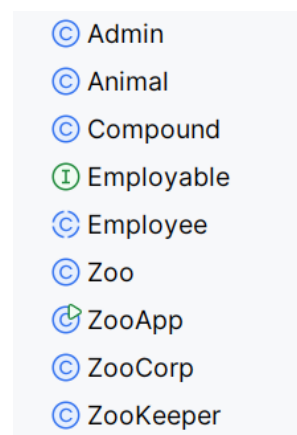
Task:

- c. Write the code for the generated methods/stubs of the Employee class. You will need to add some variables, such as an employee's ID, name, and salary.

===

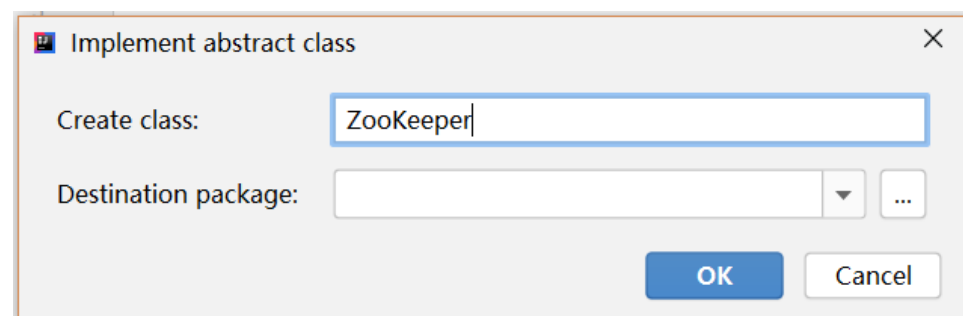
Note, interfaces do not define the required variables (except they are static, final). A general practice is not to put any variables in interface.

Also, note the symbols that IntelliJ uses to represent Interface and Abstract class.



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

- In the editor window of Employee class, put cursor on the class name "Employee", right click and choose the "Show Context Actions" option.
- Then, in the next dialog choose "Implement abstract class", and fill the name "ZooKeeper" in the pop-up window. Click OK.



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

```
1 public class ZooKeeper extends Employee {  
2  
3 }
```

Let's see 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.

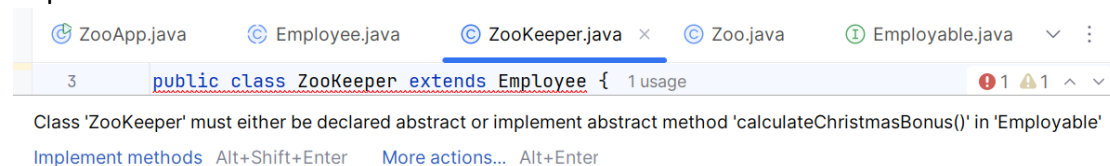
```
public interface Employable {  
    public void setEmployeeID(int number);  
    public int getEmployeeID();  
    public void setEmployeeName(String name);  
    public String getEmployeeName();  
    public int getSalary();  
    public void setSalary(int salary);  
    public int calculateChristmasBonus();  
}
```

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 + 90".

The **interface** says there HAS TO BE a method called *calculateChristmasBonus()* which returns 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 implementation of this in our actual person classes (ZooKeeper, Admin etc.).

Go to the **ZooKeeper** class, you will find the class is now in an error state (underlined in red). Put the cursor on the underlined words, the IDE tells you that the "calculateChristmasBonus" method is not implemented in this class. To solve this issue, either make this ZooKeeper class abstract (to wait for sub-classes to implement it), or implement the method in this class.



Click "Implement methods".

===

Task:

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

- To tidy up, add a constructor for “ZooKeeper” and “Admin” to set a name on creation.
- Think about how to put this constructor in the abstract class instead.
- Modify ZooCorp to accept Employable people and maintain an ArrayList of employees, similar as you did for Zoos.
- In ZooCorp, add a “printZooInfo” method to print the information of all Zoos with **Iterator** (Refer to pp. 43 of lecture 2 slides). Add a “getAllEmployee” method to return all the employees.

===

You can now test your ZooCorp as below. Notice how the object instantiation works based on inheritance in lines 23, 24, and 30. Also notice the dynamic binding (polymorphism) in line 33.

```

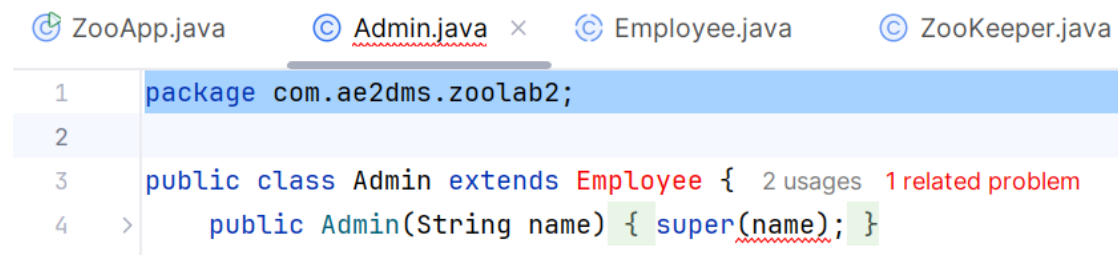
16      ZooCorp zooCorp = new ZooCorp(zL);
17      zooCorp.addZoo(zT);
18      zooCorp.addZoo(zN);
19      zooCorp.addZoo(zP);
20      zooCorp.addZoo(zB);
21      zooCorp.printZooInfo(); //print all zoo information added.
22
23      Employable zk1 = new ZooKeeper( name: "John"); //Can also use Employee
24      Employable am1 = new Admin( name: "Jane");
25      zooCorp.addStaff(zk1);
26      zooCorp.addStaff(am1);
27
28      Iterator<Employable> iter = zooCorp.getAllEmployee().iterator();
29      while(iter.hasNext()){
30          Employable e = iter.next();
31          e.setSalary(100);
32          System.out.println("Christmas bonus for "
33              + e.toString() + " : " + e.calculateChristmasBonus());
34      }

```

Understanding packages

You may have found that everything coded so far has not been placed in any packages. Let's now move all the classes to a new package. Right click “src”, then “New – Package”, and give a name to it, e.g., *com.ae2dms.zoolab2*. An empty package now appears in the src folder.

Drag a class, e.g., **Admin**, into the package. In the pop-up window, click “Refactor”. Then in the Admin.java file, a new line is automatically generated.



```
1 package com.ae2dms.zoolab2;
2
3 public class Admin extends Employee { 2 usages 1 related problem
4 >     public Admin(String name) { super(name); }
```

Note that errors occur for the Admin class now, since the **Employee** class and its constructor cannot be found in this package. Let's move all the rest of the classes into this package, by repeating the above process. Check that all the files in the package should now have this line at the beginning.

```
package com.ae2dms.zoolab2;
```

This collects all the classes and methods together into new package. It will help prevent naming clashes and organize who can access the members of the package. Suppose you later build a system for organizing staff at a theme park, such as Universal Studio Beijing: 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 create a class inside a package from **outside** the package, we use naming like this:

```
com.ae2dms.zoolab2.Admin
```

Or you can import the whole package,

```
import com.ae2dms.zoolab2.*;
```

So that's where import comes from! More information about package can be found at <https://docs.oracle.com/javase/tutorial/java/package/packages.html>.

This ends the '**refresher**' part of the course.

The aims of this laboratory exercise so far have been to:

1. Give you move experience of concepts from previous year 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 the coursework.

3. Provide experience of using IntelliJ.

This addresses the following outcomes from 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.
