

LAB 2: BUILDING THE ZOO EXAMPLE

Aims:

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

Coverage:

- In Part 1 (first half of this worksheet) we cover:
 - setters, getters, static variables, ArrayLists, and IDE shortcuts
- In Part 2 (second half of this worksheet) we cover:
 - Aggregation vs Composition
 - Writing the Employee classes for the Zoo example, including an abstract class and an interface

We release the labsheets on Mondays, for you to start working on this right away. This work goes beyond the lab session time, so we encourage you to finish this work, and use the lab time for support.

Questions channels on Teams. Tag one of our teaching support group in the questions channel. Ask generic questions on the peer support channel. We aim to respond as quickly as possible.

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 IntelliJ. Look at last week's sheet if you forgot how to do this.

-> To implement the example from the lecture, first create a new package and call it whatever you like (e.g. "com.COMP2013").

-> Add a module-info.java file by right clicking on the [src] folder in the Project window and choosing {New > module-info.java} from the appearing context menu.

-> Then, in the module-info.java file type "opens" and the name you have chosen as your package name (i.e. like you did in the previous lab).

```

1 module ZooAppProject {
2     opens com.COMP2013;
3 }
  
```

-> Inside the package create a new Zoo class and add the following code. (Remember that you can also use IntelliJ to generate getters/setter for you if you want: add the private "location" field of type String, hover over the field, click it, and chose "More Actions..." from the appearing context menu.

-> Then chose the action to generate getters and setters. You can also auto-generate constructors. An alternative approach is to use {Code > Generate ...} from the menu bar. If you are not sure about why you need constructors, getters, and setters, check Lecture 1B slides/recording.

```

1 package com.COMP2023;
2
3 public class Zoo {
4     private String location;
5
6     public Zoo(String location) {
7         this.setLocation(location);
8     }
9
10    public Zoo() {
11        this.setLocation("unknown");
12    }
13
14    public String getLocation() {
15        return location;
16    }
17
18    public void setLocation(String location) {
19        this.location = location;
20    }
21 }

```


- Add a private variable to store the number of compounds the zoo has.
- Add a line to the default constructor to set this to a default value (say, 30 compounds). Add a parameter to the other constructor to read in any value for this.
- Add a method called "buildNewCompound()" 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 two 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 compounds.

Let's now add a method, 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 return a String with the zoo's location and number of compounds. **Should this be public or private?**
- Add code to the ZooApp class to make use of it for both zoos. Then do a test run.



```

1 package com.COMP2023;
2
3 public class ZooApp {
4     public static void main(String[] args) {
5
6         System.out.println("Hello world!");
7         Zoo zoo1 = new Zoo();
8         Zoo zoo2 = new Zoo( location: "London", numCompound: 43);
9         System.out.println(zoo1.printInfo());
10        System.out.println(zoo2.printInfo());
11    }
12 }
13

```

Maybe pause here, and have a look at some useful **shortcuts** in IntelliJ. They will help you to write code a lot faster, and these labs are the perfect place to try them out. You can find some suggestions for which are the best online.

THE CONCEPT OF A COMPANY OWNING LOTS OF ZOOS

As you saw in the previous step, we can have multiple version of our zoo classes. Why would we want to do this? Suppose you are a company who owns five zoos around the world, in London, Tokyo, New York, Paris, and Beeston.

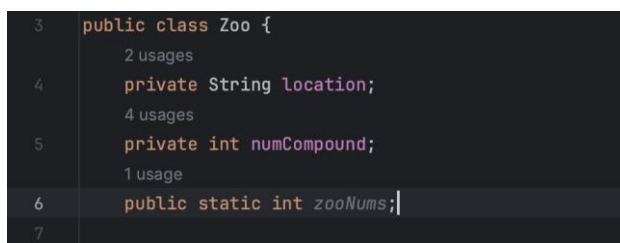
- In your main code, create five zoo objects to represent this. You can set the compound numbers as you wish.
- Print out the info for all five zoos.

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

- Think about how you could do this **WITHIN** the objects themselves
- 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 "numZoo" below) between all the objects.

Add a static int variable called "numZoo":

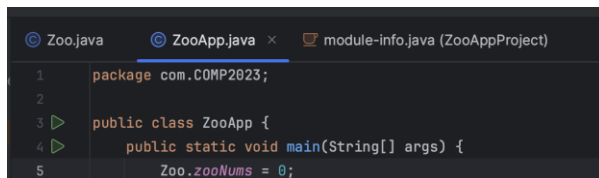


```

3 public class Zoo {
4     private String location;
5     private int numCompound;
6     public static int numZoo;
7

```

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!



```

1 package com.COMP2023;
2
3 public class ZooApp {
4     public static void main(String[] args) {
5         Zoo.zooNums = 0;

```

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

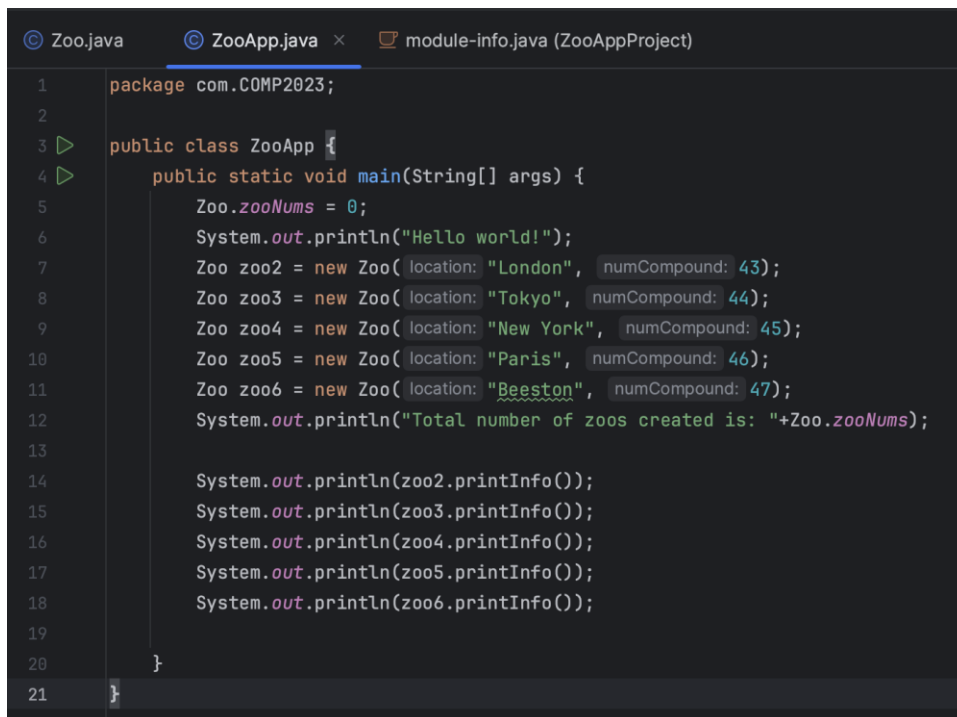
Remember, static means the variable is shared/accessible between all object instances. Let's see what that means if we want to use it to count how many zoos there are.

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

Task: Zoo ID Increment

- 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 five we have created:

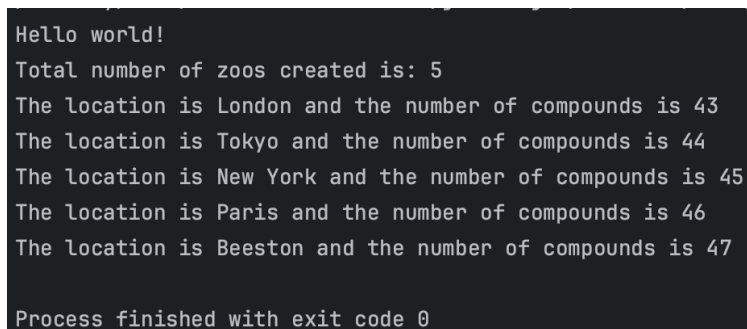


```

1 package com.COMP2023;
2
3 public class ZooApp {
4     public static void main(String[] args) {
5         Zoo.zooNums = 0;
6         System.out.println("Hello world!");
7         Zoo zoo2 = new Zoo( location: "London", numCompound: 43);
8         Zoo zoo3 = new Zoo( location: "Tokyo", numCompound: 44);
9         Zoo zoo4 = new Zoo( location: "New York", numCompound: 45);
10        Zoo zoo5 = new Zoo( location: "Paris", numCompound: 46);
11        Zoo zoo6 = new Zoo( location: "Beeston", numCompound: 47);
12        System.out.println("Total number of zoos created is: "+Zoo.zooNums);
13
14        System.out.println(zoo2.printInfo());
15        System.out.println(zoo3.printInfo());
16        System.out.println(zoo4.printInfo());
17        System.out.println(zoo5.printInfo());
18        System.out.println(zoo6.printInfo());
19
20    }
21 }

```

Your output should be:



```

Hello world!
Total number of zoos created is: 5
The location is London and the number of compounds is 43
The location is Tokyo and the number of compounds is 44
The location is New York and the number of compounds is 45
The location is Paris and the number of compounds is 46
The location is Beeston and the number of compounds is 47

Process finished with exit code 0

```

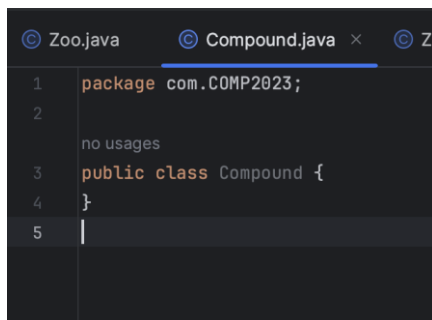
- You can use this now to set an individual unique ID number ("private int zooId") per Zoo in the constructor ("zooID=numZoo;"). For the "zooID" you should only create a getter but not a setter (or create one but set it to private). **Why is this the case?**

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 lab helper if you would like more explanation.

ADDING ACTUAL COMPOUNDS, WE COULD 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 compounds (or enclosures), such as tanks, cages, fields etc.

- Add a new Compound class:



```

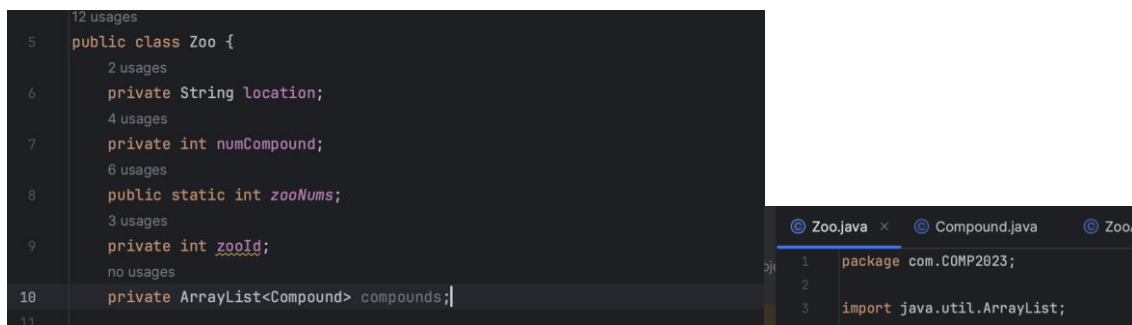
1 package com.COMP2023;
2
3 no usages
4 public class Compound {
5 }

```

Think back to the lecture where the idea of compounds was introduced. One zoo can have many compounds. **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:



```

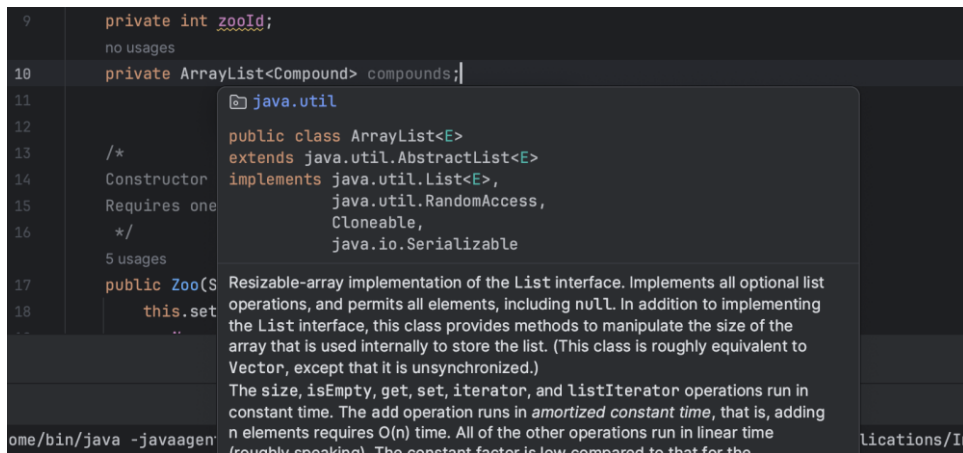
12 usages
5 public class Zoo {
6     2 usages
7     private String location;
8     4 usages
9     private int numCompound;
10    6 usages
11    public static int zooNums;
12    3 usages
13    private int zooId;
14    no usages
15    private ArrayList<Compound> compounds;

```

Note that if IntelliJ is not happy is because it doesn't know what an ArrayList is, and therefore printed it red. Hover over it, and choose "Import Class" from the appearing context menu. This will automatically add the required import statement to tell Java how to find the relevant library. Sometimes this is automatic.

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 wait. A popup with detailed information (parts of the related Javadoc) will appear:



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

```

public Zoo(String location, int numCompound) {
    this.setLocation(location);
    this.setCompounds(numCompound);
    zooNums++;
    this.setZooId(zooNums);
    for (int i=0; i<numCompound; i++){
        addCompound(new Compound());
    }
}

```

- Note how we can create **new** objects inline like this. Also note, we haven't written an addCompound() method yet! Don't panic. IntelliJ can help! Hover over the red "addCompound" and in the appearing popup choose "Create method 'addCompound' in 'Zoo'"



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

- This auto-generation 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:

```

1 usage
private void addCompound(Compound compound) {
    this.compounds.add(compound);
}

```

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

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

```
5 usages
public Zoo(String location, int numCompound) {
    this.setLocation(location);
    this.setCompounds(numCompound);
    compounds = new ArrayList<Compound>();
    zooNums++;
    this.setZooId(zooNums);
    for (int i=0; i<numCompound; i++){
        addCompound(new Compound());
    }
}
```

Note that I am using set methods wherever possible.

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

Task: Extend the Zoo project

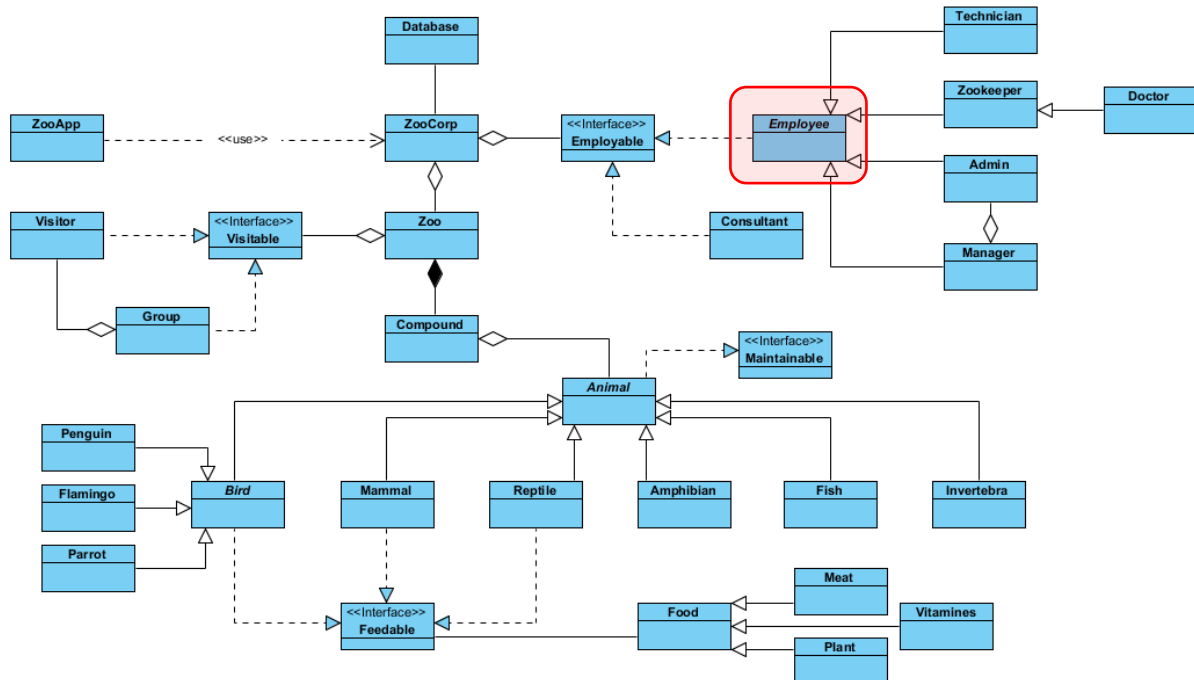
- Using the same idea, change the "Compounds" class so you can add animals, by using an ArrayList again. You will need to create a class Animal and you will need a method to add animal objects to your new ArrayList.
- Note: We will look at how we can use abstract classes and inheritance with animals in Part 2 of this lab.

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:



Your task in the rest of this lab is to modify the existing code and implement the employee section. 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 many zoos. However, a zoo can exist before it's 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 object, so let's add that as an option. A good data structure to use might be the ArrayList again, as we used in Part 1 e.g.

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

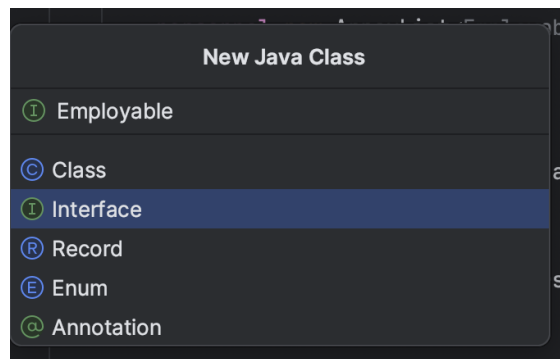
Task: Create and extend the ZooCorp class

- Create a class ZooCorp
- Write a constructor for ZooCorp objects, which takes a Zoo object as an argument and stores it in a newly created ArrayList.
- Then, write a method called addZoo(), which also takes a Zoo object as an argument and 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 will think about the details of this in a while, but for now let's just create one:

- In the Project window right click on "com.COMP2013" and choose {New > Java Class > Interface}



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:

```

1  package com.COMP2013;
2
3  public interface Employable {
4      public void setEmployeeID(int number);
5      public int getEmployeeID();
6      public void setEmployeeName(String name);
7      public String getEmployeeName();
8      public int getSalary();
9      public void setSalary(int salary);
10     public int calculateChristmasBonus();
11 }
12
  
```

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

It is the job of the class *implementing* the interface to provide the code. We don't care *how* the method 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 Employee and then **set the class to abstract**. Then add "implements Employable" after the class name.

```
public abstract class Employee implements Employable {
```

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

Hover over Employable, right click on it and choose the option {Generate...} in the appearing context menu. Next choose {Implement Methods...} from the appearing popup and select all methods for implementation. Click {OK}.

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 (zookeeper, admin, etc.). However, an abstract class is more than just an interface – we can define some generic behaviour here, which all employee objects can use. Provide implementations for all the methods we have so far, as they are all very general in nature, in this abstract class:

Task

- 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 Employee class: Then add "extends Employee" after the class name. This will now compile, as the Zookeeper has access to all the methods we have coded in the abstract class, Employee.

```
3 public class Zookeeper extends Employee {
4
1 usage
```

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

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

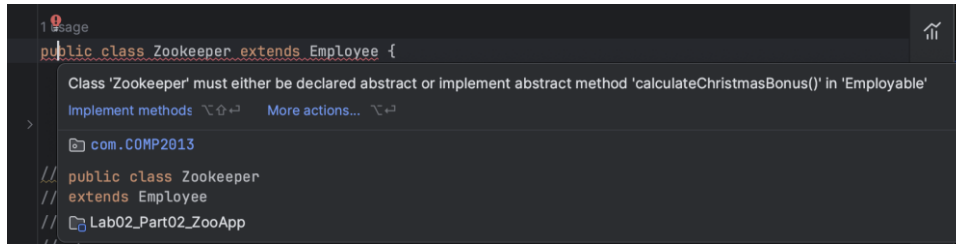
```
public int calculateChristmasBonus();
```

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 IntelliJ can help us! Click "Implement methods" in the appearing popup.



Hey presto! The required method stub appears:

```

public class Zookeeper extends Employee {
    2 usages
    @Override
    public int calculateChristmasBonus() {
        return 0;
    }
}

```

- For your Zookeeper class, add the bonus equation ($salary \times 0.05 + 100$)
- Now add an Admin class, which also inherits from Employee, and implement the different Admin bonus equation ($salary \times 0.08$)

So, the **interface** says there HAS to be a method called calculateChristmasBonus which returns an integer. However, 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: Tidy up...

- To tidy up, add a constructor for Zookeeper and Admin, to set a name on creation.
 - Could you put this constructor in the abstract Employee class instead? How?
- Modify ZooCorp, as you did for Zoo, to accept Employable people, and maintain an ArrayList of employees.
- Write code in the main to test everything. Below is an example, how this might look like.

```

1 public static void main(String[] args) {
2     Zoo.numZoos=0;
3     Zoo zoo1=new Zoo( location: "London", numCompounds: 20);
4     ZooCorp megaZoo=new ZooCorp(zoo1);
5     Zoo zoo2=new Zoo( location: "Tokyo", numCompounds: 15);
6     megaZoo.addZoo(zoo2);
7     Zoo zoo3=new Zoo( location: "New York", numCompounds: 26);
8     Zoo zoo4=new Zoo( location: "Paris", numCompounds: 18);
9     Zoo zoo5=new Zoo( location: "Beeston", numCompounds: 5);
10    System.out.println(zoo1.printInfo());
11    System.out.println(zoo2.printInfo());
12    System.out.println(zoo3.printInfo());
13    System.out.println(zoo4.printInfo());
14    System.out.println(zoo5.printInfo());
15    zoo5.buildNewCompound();
16    System.out.println(zoo5.printInfo());
17    System.out.println("Number of Zoos:"+Zoo.numZoos);
18    Employee sue=new Zookeeper( name: "Sue");
19    Employee bob=new Admin( name: "Bob");
20    megaZoo.addStaff(sue);
21    megaZoo.addStaff(bob);
22    System.out.println(sue.getClass().getSimpleName()+" "+sue.getEmployeeName()+" "+sue.calculateChristmasBonus());
23    System.out.println(bob.getClass().getSimpleName()+" "+bob.getEmployeeName()+" "+bob.calculateChristmasBonus());
24 }
25 }

```

That's all for this labsheet. We hope you enjoyed it!