**Institute of Technology Tralee**

**Computing Department**

**Object Oriented Programming**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

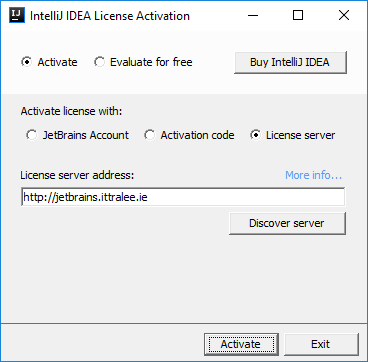
**Practical 5 – Access Modifiers & Static**

At this stage you have seen the full development of an **instantiable class** and the **driver class** that was used to test its functionality. You have also met a number of important OO features such as **classes**, **objects**, **method overloading** and **overriding** to name a few. In addition, you have been given a quick introduction to the use of the **Git/GitHub** VCS, which you will use for the remainder of the module. This time, we will look at the concept of **access modifiers**, why and how we use them and we will also look at the **static** keyword and see why sometimes it is necessary when designing classes to make certain attributes/methods static.

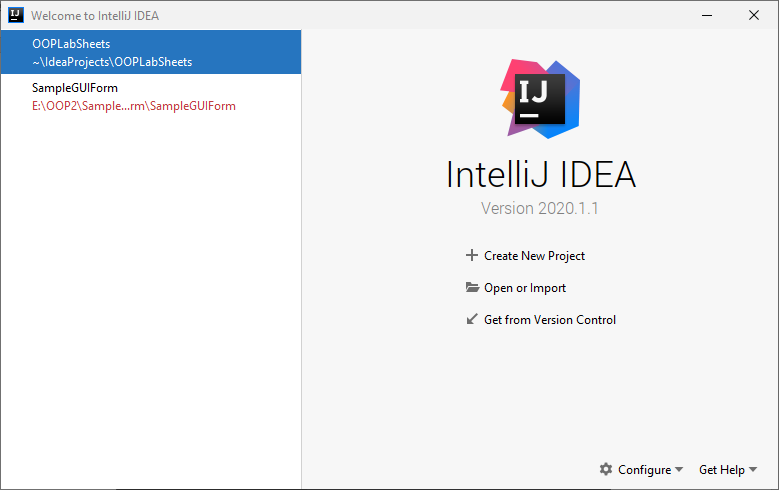
**Getting into IntelliJ**

Click on the **Search Windows** button on the taskbar (it looks like a magnifying glass) and type in the letters “in” - hopefully you will get a match for **IntelliJ IDEA**.

You will see the screen below. Simply select **License Server** 🡪 **Discover Server** 🡪 **Activate**

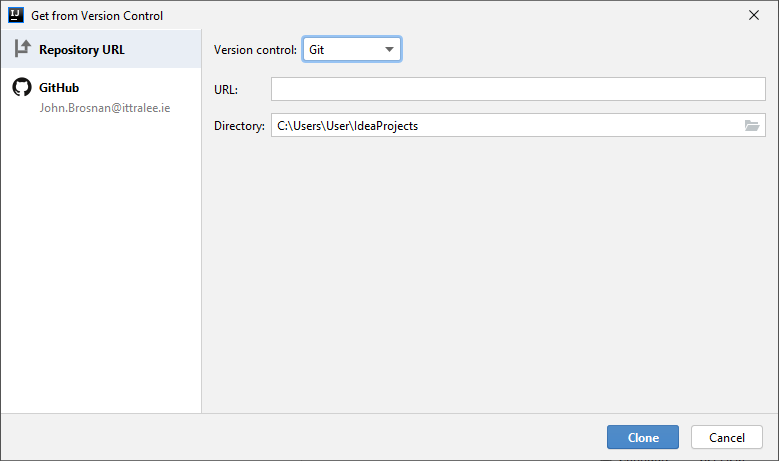


When completed, the IntelliJ **Integrated Development Environment** (**IDE**) should launch for you, after a few seconds. Once this happens, you are ready for coding!



To begin with above, you will see options to “Create New Project”, “Open or Import” etc. As you were introduced to VCS in the last sheet, and since you should really have the latest version of your **OOPLabSheets** project “pushed” to GitHub, Click “**Get from Version Control**” and see if you can now clone your **OOPLabSheets** project locally (If you haven’t your latest version pushed to GitHub, just copy it from your X: drive to some location on C: or work directly with it from the memory stick).

Now a “Clone Repository” window pops up and you can select the repository that you wish to “check out” (download) from GitHub. IntelliJ will remember previously “pushed” repositories, so you can just pick off the one you want. You can also decide where you want the repository to be located locally, I am choosing the same location as the original,



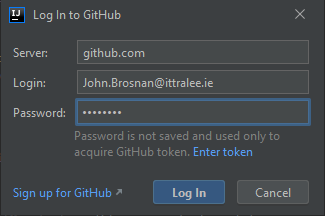
Recall that you **may need to rename** the local repository to something else if you get an error message at this point (I suggest a date e.g. OOPLabSheets26-09-20)

**Pushing an IntelliJ Project to GitHub**

**At the end of the lab session** you should really “**push**” your version-controlled OOPLabSheets project to GitHub, so that it then becomes a remote repository and a back-up of your work. I suggest you back it up to X: or memory stick also.

To do this, within IntelliJ just select **VCS**, then **Git**, then **Push** or, even better, just press the Git “Push” green arrow at the top of the IntelliJ window.

At this point, a dialog may pop up requesting your GitHub account details (IntelliJ might remember these also though). Once supplied, you can press the **Log In** button



If everything goes to plan, you will get a “Pushed 1 commit to origin master” message at the bottom-right of the IntelliJ window:

Now, for proof that the project is actually on GitHub, you can just **view your GitHub repositories list** and you should see the **OOPLabSheets** project listed. You can click into this then to make sure the latest files are definitely there.

**Setting up your Folder Structure**

As you know by now, my preferred approach is to create a new folder for each lab sheet. In IntelliJ this can be done by adding a new **package** to the project. The package will be given an appropriate name, I will call it **labsheet5** here. Recall that **a Java package is simply a way to store related classes together** and essentially a **package is just a folder**. We will talk about packages further in this module but, for now, we will just create a package called **labsheet5** for this IntelliJ project and our intention will be to store all the related classes that we create and use for this lab sheet together within that package (folder). Right-click on the name of the project i.e. **OOPLabSheets** and select **New**🡪**Package.**

You will now be given the opportunity to enter the name of the package, so you can enter **labsheet5**. As soon as you click **OK**, an icon for the newly created package appears in the left-side window, listed as part of the project’s contents. The package is currently empty, but you will be adding some sub-packages to it as the lab goes on.

**Access Modifiers in Java**

Throughout year 1 you will have seen the keyword **public** (every time we created a class and the main() they were both public) and towards the end of semester 2, you saw the keyword **private**, when we created some private inner classes for event-handling on GUIs.

These keywords are examples of **access modifiers** in Java and are the two most widely used ones. There are two other access modifiers, called **protected** and **package** (package access is indicated by **no keyword**). We will discuss protected access later on in the module when we discuss the important topic of **inheritance**. For now, we will just examine the other three kinds of access modifier.

|  |  |  |  |
| --- | --- | --- | --- |
| **Access Modifier** | **Within Class** | **Within Package** | **Global** |
| public | Yes | Yes | Yes |
| package/default | Yes | Yes | No |
| private | Yes | No | No |

The table above indicates the level of visibility of **classes**, **methods** and **attributes** that are defined by one of the three access modifiers. You can see that the level of visibility is **highest for public access** and **lowest for private access**.

***public access***

So we see that classes, methods and attributes defined to be public are **available everywhere** (globally).

Instantiable classes in Java are typically made public since you want them to be available everywhere, for the purposes of **software reuse**. Likewise, all top-level classes defined within the Java API are public, because they need to be available globally from every “client” application.

When it comes to methods, typically methods within an instantiable class are defined to be public, since it is required to make the vast majority of these methods available everywhere. So, in our Computer class, which itself was made public, we created a number of methods within the class, all of which were made public, meaning all were available to any other class that wished to use the Computer class.

In terms of attributes, it is rare to see a public attribute defined within an instantiable class. The reasons for this were mentioned when we were developing the Computer class. If an attribute of an instantiable class is made public, then it is possible to put an object created from the class into an “inconsistent state”. For example, if we had created a Computer object, referenced by c2, and the speed attribute was defined public (or package access) then we would be able to do the following:

c2.speed = -2.75;

which would place the object into an “inconsistent state”, bypassing any error-checking that might have been available within a mutator for example.

In addition it is considered a **dodgy programming practice** to make attributes of a class public because the **implementation of a particular class can change over time** (often for general improvement and performance reasons) and if client code is dependent on the public attributes defined in a previous version of the class, and those attributes no longer exist in the latest version of the class, then the client code breaks down and requires alteration.

These are exactly the reasons **why we define all attributes in instantiable classes private** wherever possible.

However, there are times when having a public attribute is exactly what is needed. For example, within the Math class of the Java API, there are two attributes defined as **constants**, and both of these are **public**. These are **PI** and **E** and are used to represent those mathematical constants. The following is a partial **definition of the Java API Math class** and looks as follows

:

:

/\*\*

\* The class <code>Math</code> contains methods for performing basic

\* numeric operations such as the elementary exponential, logarithm,

\* square root, and trigonometric functions.

\*

\* other comments omitted for brevity here \*/

public final class Math {

:

:

**public static final double E = 2.7182818284590452354;**

**public static final double PI = 3.14159265358979323846;**

public static double sin(double a) {

//code to find sine of an angle in radians

}

public static double cos(double a) {

//code to find cosine of an angle in radians

}

public static double tan(double a) {

//code to find tangent of an angle in radians

}

:

:

:

}

So if we needed to calculate the area of a circle, we might have had:

circleArea = Math.PI \* radius \* radius;

so we are able to access the constant attribute PI from any class outside of the Math class since it is **public**. Importantly though, because these attributes are **final**, we cannot possibly alter their values anyway, so cannot put them into an “inconsistent state”.

***package/default access***

There is **no keyword** associated with package/default access, so if you don’t see the keyword public, private or protected when defining a class, method or attribute, then you know it must have **package** **access**.

Classes, methods and attributes that are defined to have package access are **available within their own class and also within other classes that belong to the same package**.

We cannot access any class, method or attribute defined to be package-access outside of the package. I will get you to prove this to yourself shortly in an exercise.

In first year, we always created our “container” classes as public classes, but it would have been perfectly acceptable to create them as package access classes, because there was never any intention at the time to re-use any of the classes from elsewhere.

In OOP, you don’t tend to see a lot of use for the package access modifier, most of the time you end up using public or private access, as required.

***private access***

Classes, methods and attributes that are defined to have private access are **only** **available within their own class**

Private classes are not so common in OOP. Actually, you cannot define the top-level class in a Java file to be private at all (**prove this now** within IntelliJ by attempting to make your Computer or Book class private), but you can certainly define a private class for an application, as we did last year for our GUI programs.

A private class cannot be accessed outside of the class in which it is defined though, so it is of no use to the outside world, and can only be used internally within an application.

Private methods are more commonplace, and tend to exist as “**utility**” methods within OOP classes, which are just defined to “help” other methods defined (normally public methods). We will see some examples shortly and I will get you to attempt an exercise that creates and uses one. But again, such methods are **not available outside of the class** in which they are defined.

Private attributes are **seen everywhere** in OOP. It is recommended that, if possible, all attributes defined in an instantiable class are private, for reasons cited earlier in relation to public access attributes.

**Tips on Choosing an Access Modifier**

If other programmers use your class, you want to ensure that errors from misuse cannot happen. Access modifiers can help you do this.

* Use the most restrictive access level that makes sense for a particular class, method or attribute. Use private unless you have a good reason not to.
* Avoid public attributes except for constants.

**Exercise 1**

It was mentioned earlier that “we cannot access any class, method or attribute defined to be package-access outside of the package”. Try to prove this now by first of all creating a new package within your **labsheet5** package called **exercise1**.

Now **copy** the **Book.java** file and the **TestBook.java** file from the **exercise4** package within the **labsheet4** package into this exercise1 package. Open the file **Computer.java** that you have in the package **exercise1** of the **labsheet3** package and modify it so that the Computer class access modifier changes from public to package access (simply **remove the public keyword** from the class definition to do this) as follows:

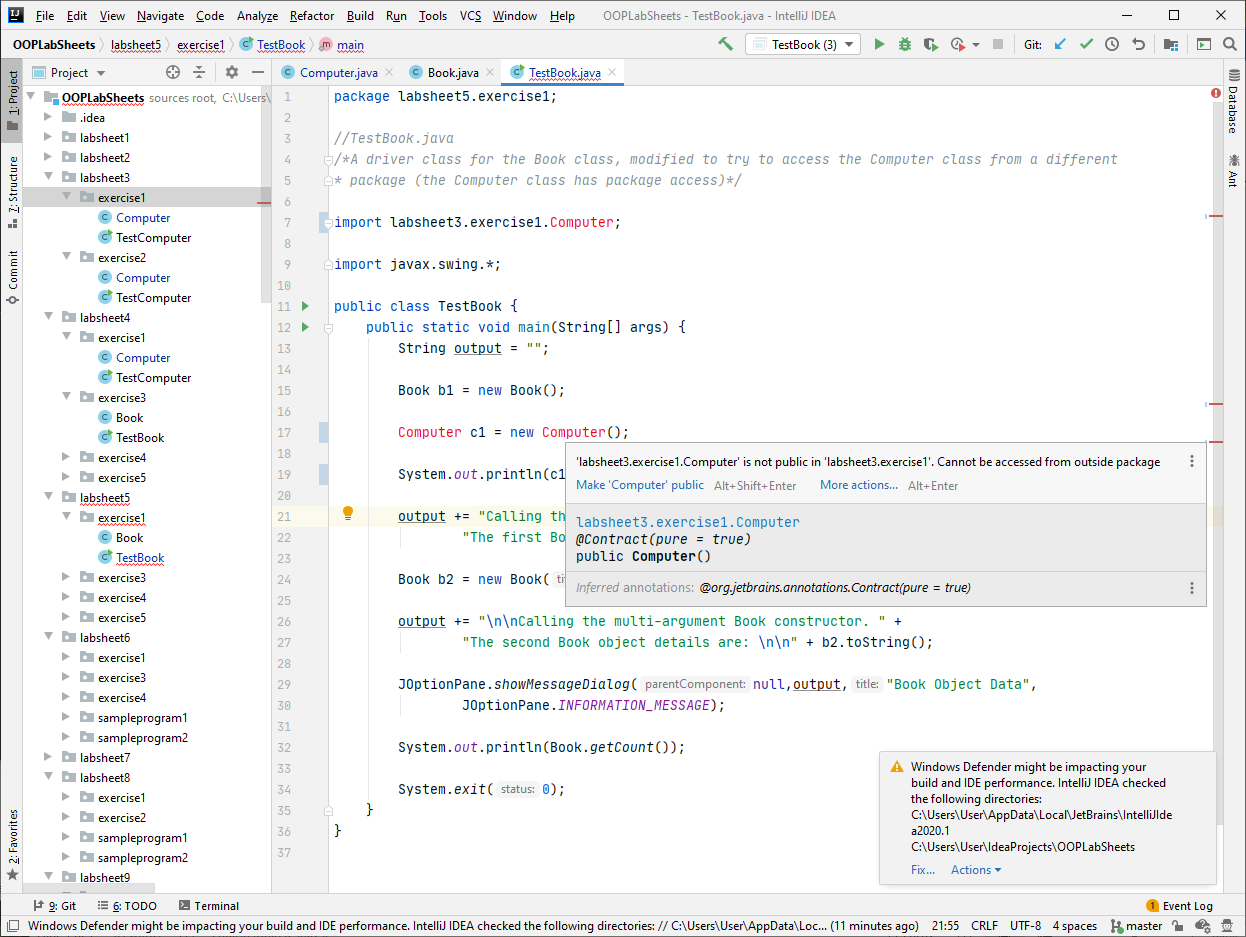
package labsheet3.exercise1;  
  
class Computer {  
 private String manufacturer;  
 private String type;

//rest of class omitted

As the TestBook class and the Computer class are in different packages, in order to access the Computer class from the TestBook class, you would need to import the package it belongs to within the TestBook class by adding the following **import statement**:

import labsheet3.exercise1.Computer;

But you can see below that the word “Computer” is in red, and hovering over it confirms that we are in trouble trying to access the class, since it has only package access.



It is great practice to experiment with your code like this to see what you can and can’t do. You will learn a huge amount extra by doing this.

**Exercise 2**

On a similar theme to the last exercise, make the Computer class in the package labsheet3 public again, but give its **getManufacturer**() accessor **package access** rather than public access.

Now, in the class **TestBook**, you will see that the “red” has disappeared from the import statement, meaning the Computer class is available now. Try adding the following code now to the main() in TestBook.

Computer c1 = new Computer();  
  
System.*out*.println(c1.getManufacturer());

What do you expect to happen?

**Exercise 3**

Within your labsheet5 package create another package called **exercise3** to store the following classes:

An instantiable class called **Thermometer** is required which will have the following three **private** attributes:

* *Current temperature* – an integer
* *Maximum temperature* – an integer
* *Minimum temperature* – an integer

It will have the following **public** **methods**

* a no-argument constructor
* a 1-argument constructor
* accessor methods for each of the attributes
* mutator methods for each of the attributes
* a toString() method which returns the state of the object’s attributes as a String
* a method called **setTemperature**() which will be used to set the current temperature of the thermometer as well as update the maximum/minimum temperatures via a call to updateMaxMinTemp()

It will have the following **private** **method**

* a method called **updateMaxMinTemp**() which will be called within the setTemperature() method, and will contain code to determine, based on the value of the currentTemperature attribute, whether the max/min temperatures should change (so it’s like when a thermometer sensor takes a temperature reading, and the current temperature value changes, then the max/min values must be checked to see if the latest temperature reading requires a change in their values). Design-wise it **makes sense for this method to be private** because we would never change the max/min temperatures for a thermometer directly, they only change as a result of a change in the current temperature reading.

You can take it that the **no-argument constructor** will give a Thermometer object the following initial state for the three attributes:

* *Current temperature* – zero
* *Maximum temperature* – Integer.MIN\_VALUE
* *Minimum temperature* – Integer.MAX\_VALUE

The **1-argument constructor** will give a Thermometer object the following initial state for the three attributes:

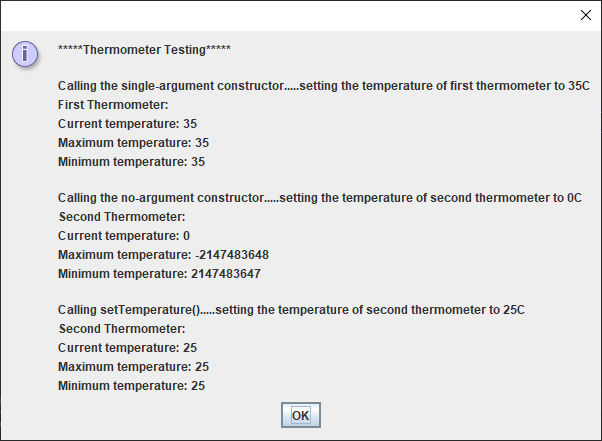
* *Current temperature* – value of argument passed in
* *Maximum temperature* – value of argument passed in
* *Minimum temperature* – value of argument passed in

So all three attributes will share the same value in this case (just like when the temperature sensor takes the first temperature reading)

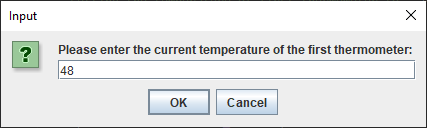
You should code your class to **maximise software reuse**

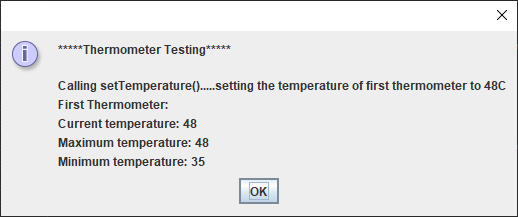
You must also write a **minimalistic driver class called TestThermometer within a file TestThermometer.java** that will test all the functionality of your class.

The output of your driver should appear similar to the following:



After this dialog closes, the user is prompted to enter a value for the temperature of the first thermometer as follows:





**Revision – Static and Instance Methods**

If a method is **static** then we just **call the method directly on the class it belongs to** as follows:

text=JOptionPane.showInputDialog(“This is a static method”);

The method we have used so often, showInputDialog() is a static method. Therefore we **should call it directly on the class it belongs to** rather than on an instance (object) of that class. An object of the class does not need to exist in advance of calling a static method. In other words, we shouldn’t really do this:

JOptionPane dialogRef = new JOptionPane();

text=dialogRef.showInputDialog(“This is a static method”);

because now we have created an instance of JOptionPane() – using the new keyword – and then called the method on that instance. This code is still actually valid but **not conventional** since most professionals never create an instance when calling a static method.

There were numerous examples of **instance methods** in the GUI section of the SP2 module. The code:

jFrameWindow.setLayout(flowLayout);

jFrameWindow.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

shows 2 examples of instance methods called setLayout() and setDefaultCloseOperation().

These are instance methods rather than static methods because **an instance of (an object of) JFrame must exist before they can be called**. The instance in this case is being “pointed” to by the object reference **jFrameWindow**.

Interestingly though, see that the setDefaultCloseOperation() takes as an argument **JFrame.EXIT\_ON\_CLOSE**. This is a **constant** attribute (how can I tell this?) defined within the JFrame class but it is **static**, because we are **able to refer to it directly on its class**.

Last year we never went into the specifics about *why* some methods are static and some are instance methods. It really is an **OO design decision** and we will get into it a little more now.

**Static Attributes and Methods**

For the instantiable classes you have seen so far, each object created from the classes have had their **own set of values for the attributes** defined for the class, which is absolutely normal and generally necessary.

However, there are times when you would prefer to define an attribute so that it is **shared** among all objects that will be created from a given class. The classic example is where you have an instantiable BankAccount class, and you wish to define an attribute called interestRate, which will have some numeric value associated with it.

So we might do the following within the BankAccount class:

private static double *interestRate*;

The keyword **static** applied here means that the attribute interestRate is now shared among all BankAccount objects that will end up being created. There is **only one copy** of this shared attribute.

As in the case of non-static attributes (often called **instance attributes/variables**), because the attribute is private, it cannot be accessed directly outside of the BankAccount class

So, if we have a driver class called TestBankAccount, then we might try to do the following:

BankAccount account1 = new BankAccount();

account1.interestRate = 0.75;

but, of course, it will fail due to the private nature of the attribute.

Instead, as in the case of non-static attributes, we end up trying to modify/access the attribute via mutator/accessor methods.

So we would define a method setInterestRate() to allow us to change the value of the static attribute and a method getInterestRate() to allow us to access its value.

The difference now is that, because these methods are referring to a static attribute, they themselves **must be defined static**, as it will result in a **syntax error** otherwise. This is the very same reason why, in first year, when you created your own user-defined methods within a class, those methods needed to be defined static, since they were being called from the main() method, which itself is defined static.

So, for example, the setInterestRate() method here would look as follows:

public static void setInterestRate(double interestRate) {  
 BankAccount.*interestRate* = interestRate;  
}

You might also be wondering why the line of code within the method here isn’t

this.*interestRate* = interestRate;

as it usually is for non-static attributes. It’s not so obvious, but this would cause a syntax error also, because the **this reference cannot be accessed from a static method**. Instead we need to refer to the interestRate attribute by preceding it with the name of the class itself. In a way it makes sense, because the **attribute belongs to the class** itself rather than a specific instance of the class. In fact static attributes are sometimes called **class attributes**.

Because the setInterestRate() and getInterestRate() methods are static then, outside of the BankAccount class, they **should be called as static methods** directly on the class as follows:

BankAccount.setInterestRate(0.25); //set interest rate for all BankAccount objects to 0.25

System.out.println(BankAccount.getInterestRate()); //display the interest rate

Even though they could actually be called on objects created from BankAccount also (but this is **not considered good programming practice**).

**Exercise 4**

Within your **labsheet5** package create another package called **exercise4** to store the following classes:

An instantiable class called BankAccount is required which will have the following three **private** attributes:

* *owner* – a String
* *number* – an integer
* *interest rate* – a double which will also be static

It will have the following **public** **methods**

* a no-argument constructor
* a 3-argument constructor
* accessor methods for each of the attributes
* mutator methods for each of the attributes
* a toString() method which returns the state of the object’s attributes as a String

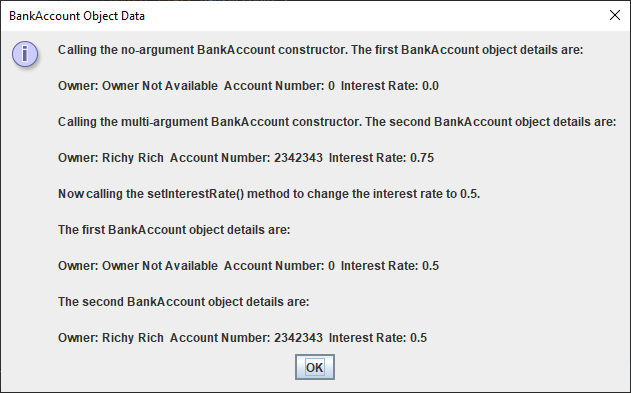
You can take it that the **no-argument constructor** will give a BankAccount object the following initial state for the three attributes:

* *owner* – “Owner not Available”
* *number* – 0
* *interest rate* – 0

You should code your class to **maximise software reuse**

You must also write a **minimalistic driver class called TestBankAccount within a file TestBankAccount.java** that will test all the functionality of your class.

The output of your driver should appear similar to the following:



The key thing to note from the output is that the call to setInterestRate() has the effect of **changing the interest rate for each object** of the class, since it is shared among them all.

Can top-level classes themselves be defined static? **Try this out** now in IntelliJ using your BankAccount class.

If you had a Person class defined, would it make sense for the age attribute defined within this class to be static? Explain briefly.

**Another use of static Attributes**

Sometimes it is desirable to know the number of objects that have been created from a given class. So, going back to the Book class from **Exercise 1** we will pretend that this class needs another attribute called **count**, which will allow a library to determine exactly how many Book objects they have in stock.

We would define the new attribute as follows:

private static int *count*;

and then, to increment this count value, we could define a **private** method called incrementCount() as follows:

private static void incrementCount() {  
 *count*++;  
}

Notice here that I choose to make this method **private** because there is no intention of incrementing the value of the count attribute outside of the Book class itself. It would be wrong to do so as it would lead to logical errors. We only wish to increase this attribute “internally”, within the multi-argument constructor method as follows:

public Book(String title, double price, String ISBN, int pages) {  
 setTitle(title);  
 setPrice(price);  
 setISBN(ISBN);  
 setPages(pages);  
 *incrementCount*(); //increment the ID attribute  
}

Now, when a Book object gets created in the driver class, incrementCount() is called and the count attribute increases by 1. So if we want to know at any point in the driver program how many Book objects exist currently, we can say:

System.*out*.println(Book.*getCount*());

and we will be given the total count of Book objects created.

We could take this idea one step further and use it to allow us to **auto-increment** a particular attribute of a class. So, imagine that the library want to be able to automatically assign an “internal” ID number to each Book object in stock. It would require the creation of an additional instance attribute (since each Book would have its own individual ID number). We could call it **ID** and it would be an **integer**. This would have its own mutator and accessor and the toString() would be altered to take account of the new attribute also.

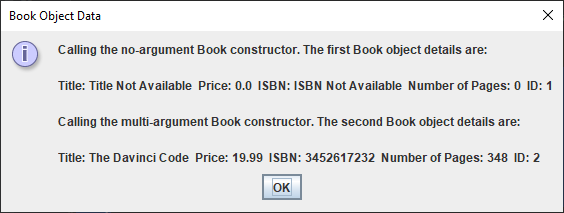
Now everytime we create a Book object, we wish the ID attribute of that Book object to be in-line with the value of the static count attribute discussed earlier.

To do this, you could have the following in the Book class multi-argument constructor:

public Book(String title, double price, String ISBN, int pages) {  
 setTitle(title);  
 setPrice(price);  
 setISBN(ISBN);  
 setPages(pages);  
 *incrementCount*();  
 setID(*count*);  
}

notice here that the Book constructor still takes the same 4 arguments as before, and that ID is not passed in at all, since this is **automatically** given a value here internally, rather than it coming from the “outside”.

Then, from the TestBook class you would see the following output



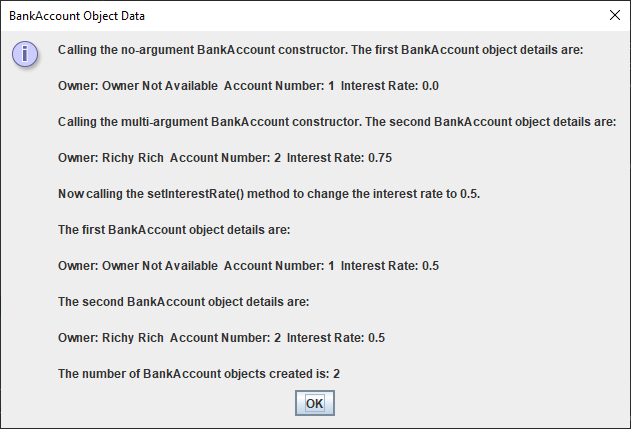
**Exercise 5**

Within your **labsheet5** package create another package called **exercise5** and **copy** the BankAccount and TestBankAccount classes from exercise4 package to this package.

In the BankAccount class you created earlier in Exercise 4, you will have seen that one of the attributes is called number, and that a value can currently be assigned for this from outside the class. You should now **modify** the BankAccount class so that value given to the number attribute will instead be assigned internally, based on the value of an additional static attribute called count, which will be used to keep track of the number of BankAccount objects that have been created. Refer to the last section as a guide when trying to make the necessary modifications here.

You will need to make some changes to the driver class code also in order to get things working as required and to make sure the class is fully tested.

If all goes according to plan, your driver TestBankAccount will produce the following output:



You can see that now the number attribute is set automatically based on the value of the static attribute count. Also you can see the total number of BankAccount objects created here was 2, this information being obtained via a call to the static method **getCount**().

**Some Important OOP Terms Covered in this Lab Sheet**

You **need to be able to recognise and explain various OO features, concepts and terms** (for the mini-project presentation and, more importantly, for the final written exam). It is important to be able to explain various OO concepts well because you could easily be asked for such explanations at **interviews** in year 3 and for **job applications** generally going forward. In a nutshell, you need to be able to “speak” in OO terms.

Therefore, this section is designed to give some definitions/explanations for some of the concepts introduced in this lab sheet.

**Access modifiers** – These are an OO feature which determine the level of visibility/access of **classes**, **methods** and **attributes** from other classes. The level of visibility/access is **highest for public access** and **lowest for private access**. There are two other access modifiers called **package/default access** and **protected access**. It is recommended when designing OO classes/methods/attributes to always **choose the lowest level of visibility/access** we can get away with.

**Instance method** – This is a method which **can only be called on an instance** (object) of a class. These are the predominant kind of method found in OOP, the other kind being static methods. **Mutators** and **accessors** defined within classes are typically instance methods as is the toString().

**static attributes and methods** – if a method or attribute is defined static, then **only one copy of it exists** and this copy is shared among all the objects that are created from the class. It is a **design decision** to choose to define an attribute or method static and it makes perfect sense to create a static attribute if the value of that particular property will be shared by every object created from the class, as it **saves memory**. Static methods that are public can be called directly on the class, there is no need to create an object at all e.g. Math.pow(2,3) where pow() is a static method within the Java API Math class. static attributes can also be used in classes to **keep track of the number of objects created** from a class, which in turn can be used to **auto-increment an attribute** defined within a class.