**Institute of Technology Tralee**

**Computing Department**

**Object Oriented Programming**

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

**Practical 4 – Classes & Objects**

There were a lot of **OO terms and concepts** covered in the last lab sheet. Most will be mentioned again throughout the module, so you will become familiar with them in time. Although we designed and created a working **instantiable class** in the last lab sheet, there are a few things we still need to discuss about it. However, we begin this lab sheet by taking a very quick break from OO and looking at the idea of a **Version Control System** (VCS) and why it is so useful in the software industry. You will be introduced to a particular VCS called **Git** and you will get the chance to create your own online **GitHub** account which you will be able to use for storing your OOP lab work.

**Version Control System (VCS)**

Version Control Systems are a category of software tools which **help a software team manage changes to source code** over time. The software involved **keeps track of every modification** to the code in a special type of database. If a mistake is made, developers are able to “**turn back the clock**” and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members. In this way, VCS acts to **protect source code** from catastrophe, human error and unintended consequences.

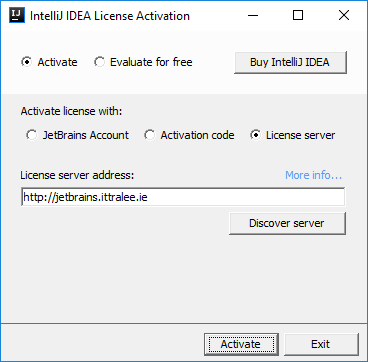
VCS is now considered an essential part of the modern software team’s professional practices. In this module we will use a particular VCS called **Git** at a very basic, introductory, level.

Hopefully, you will have Git installed on your home machines at this stage and integrated with IntelliJ (if not, read the document **InstallingGitAtHome.docx** and install it as soon as you can). It is already installed in the labs and integrated with IntelliJ, so we are spared that work 😊

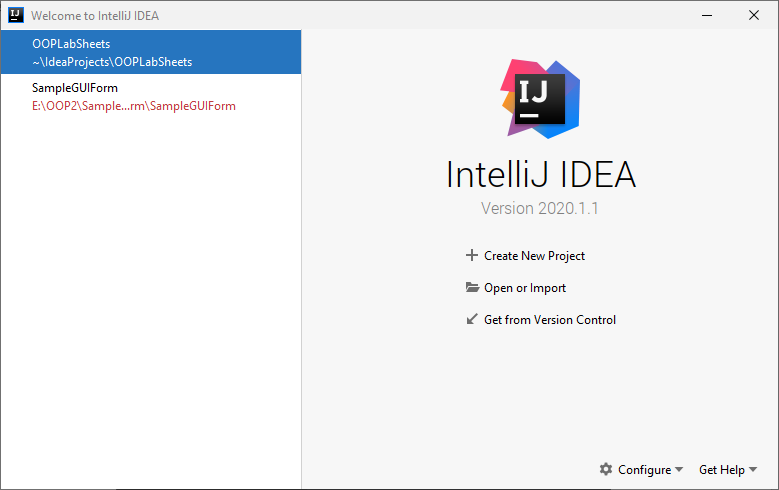
**Getting into IntelliJ**

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!

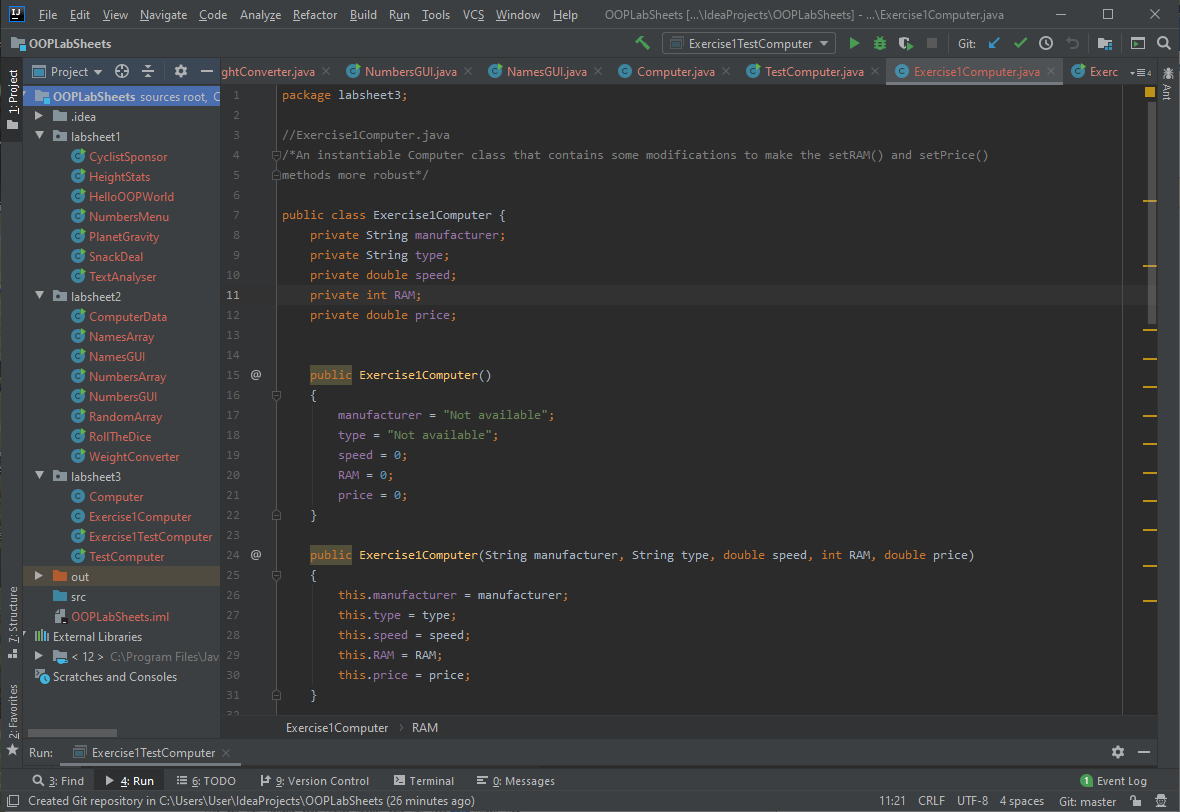


To begin with above, you will see options to “Create New Project”, “Import Project” etc. Click “**Open**” and try to navigate your way to the location where you have stored the project called **OOPLabSheets**. It may be on your X: drive or memory stick. If it is on X: drive, then I suggest you copy the project to the desktop and then just navigate to it (as working from X: can be iffy at times). If you are working from memory stick you can work directly from there if you wish and just navigate to the location of the **OOPLabSheets** project on your memory stick.

Mine is located on the C: drive of my machine at C:\Users\User\IdeaProjects. This is the default location for newly created IntelliJ projects. Once selected, the project opens in IntelliJ.

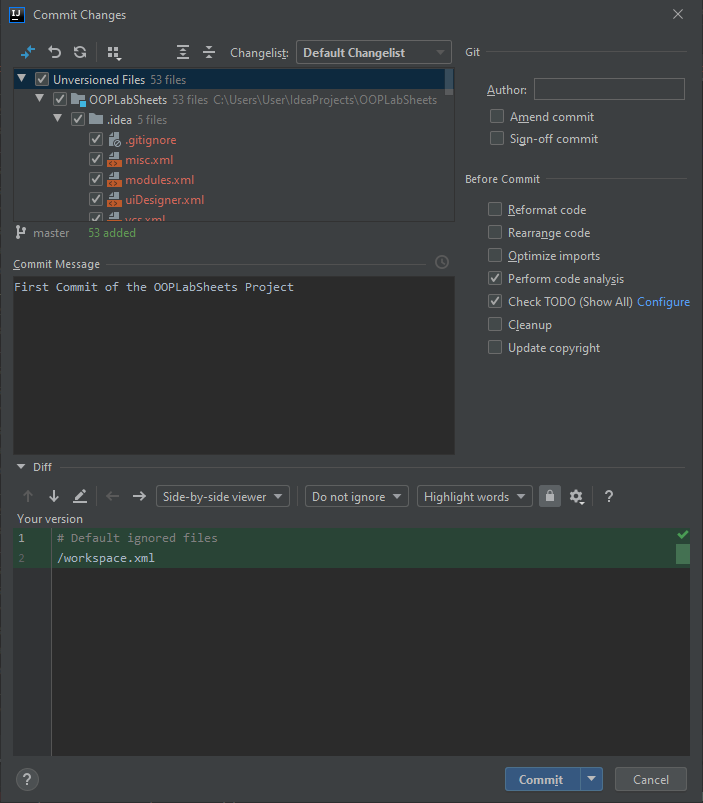
**Using Git for the First Time**

Over the past few labs sheets we have created a number of Java files within the **OOPLabSheets** project. Now we wish to use Git to enable us to perform version control on our project files for the first time. Currently, my IntelliJ window looks as follows:



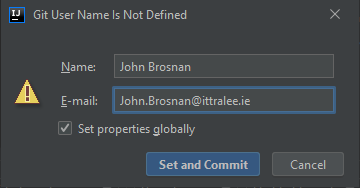
You’ll notice a lot of “red” on the left and also in the tabs at the top. This colour indicates that there are many files (all of them in fact!) that **have not been subjected to version control** yet.

To right this wrong, we can simply click on the **green “tick” icon** near the top of the IntelliJ window. This is the **Git “commit” icon**. When you perform a commit, you are basically saving any changes you have made to the **local Git repository**. The name of the repository here will be **OOPLabSheets.git** Now a “**Commit Changes**” window appears and, in my case, I am told that there are 53 unversioned files in total. So I just tick the top box and all the others get ticked automatically, since my intention here is to commit all 53 files, but you could select just particular ones if you wished.

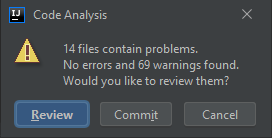


There is a “**commit message**” text-area below this and you need to provide a message to give information about the commit that is taking place. So I am just indicating here that this is my first commit of the OOPLabSheets project. With this done, I can now click the blue **Commit** button at the bottom of the window.

During the commit process, I received an error dialog indicating that my username/email have not been set, so I entered values for these:



And then pressed the “Set and Commit” button. At this point I get the following dialog



This window appears because “**Code Analysis**” is performed on the code files being committed by default. It confirms that there are no errors in my files (phew!) but I do get 69 warnings. If you are interested in seeing these warnings, you can choose to “Review” them (you can learn a lot by doing this) but I will be lazy here and just go and commit instead.

At this point I get a “**commit successful**” message popping up at the bottom of the IntelliJ window and you’ll notice that all the files that were originally “red” are now “white”. So at last all 53 files in my project have been versioned.

If I do further work on any of these files in the future, and commit the changes made, then I will have a record of all the changes to these files stored within a local Git repository. So my advice is that you would use Git within IntelliJ to commit your changes on a regular basis, it only takes seconds.

There is a lot of functionality available with the Git software and how it can be used to proper effect, but that is well beyond the scope of this module. No doubt you will learn more about it over the years to come and you should develop your expertise in version controlling your software throughout your academic career here.

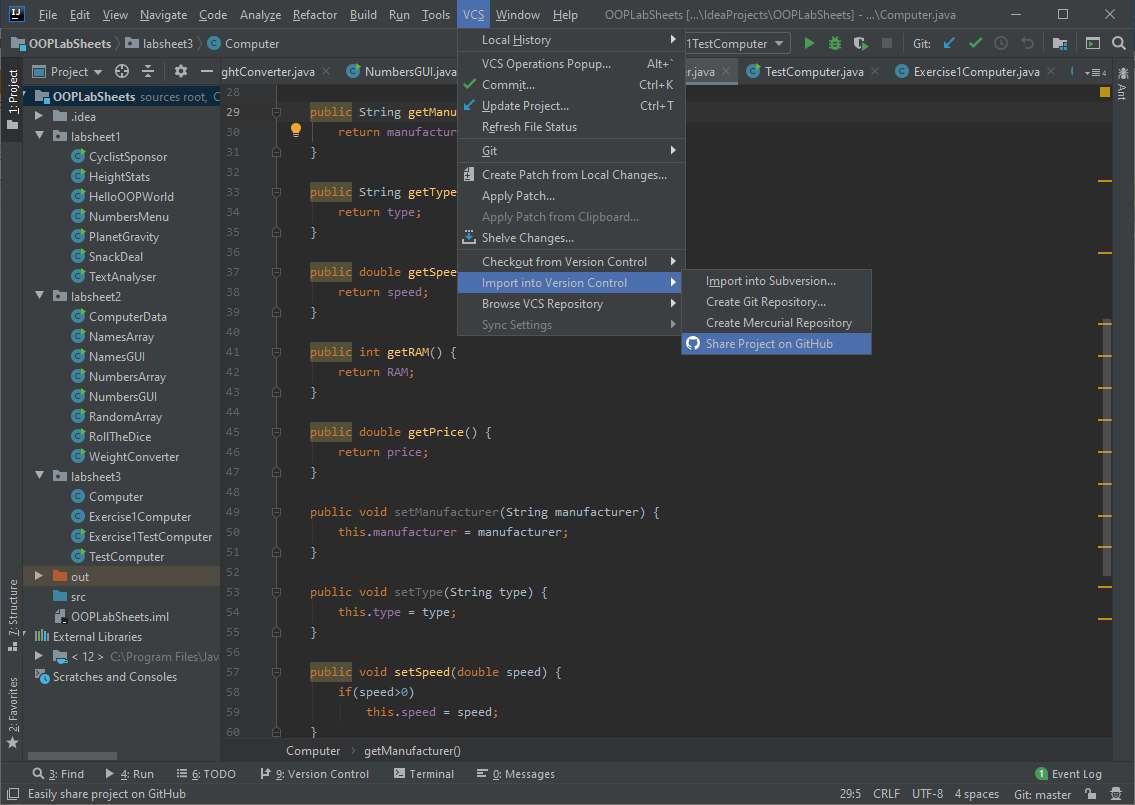
**Committing an IntelliJ Project to GitHub**

All the work done with **Git** has been **local**, on the C: drive of the local machine. **GitHub**, on the other hand, is a **remote** Web-based repository for storing projects. Free accounts are available for storing Git-controlled software and some of you may already have one. If not, you will create one shortly.

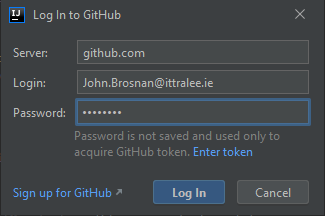
Having a remote repository of your software projects is a good idea, **allowing you to keep a portfolio of your work to show to potential employers**. In combination, Git and GitHub allow you to collaborate on software development using so-called **branches** and **pull requests** to develop your software – if you get the chance, see <https://guides.github.com/activities/hello-world/> for a quick tutorial in your own time but don’t worry about trying to remember any of the specifics.

To create your GitHub account now, just go to <http://github.com> and sign up. Once you have created your account, you will see you have the ability to create a new repository on GitHub. However, instead we will just “**push**” our version-controlled OOPLabSheets project to GitHub, so that it then becomes a remote repository.

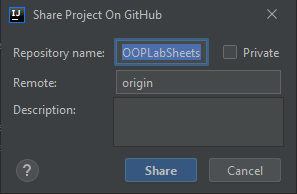
To do this, within IntelliJ just select **VCS**, then **Import into Version Control**, then **Share Project on GitHub**



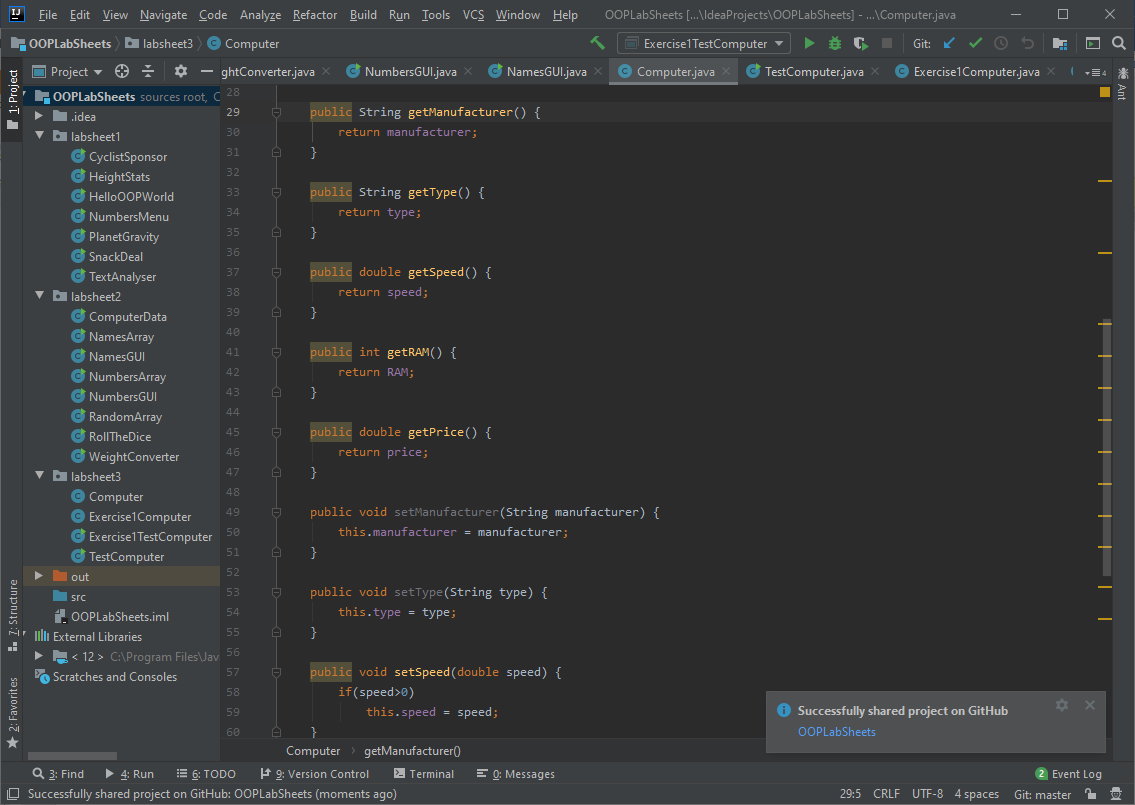
Now a dialog pops up requesting your GitHub account details. Once supplied, you can press the **Log In** button



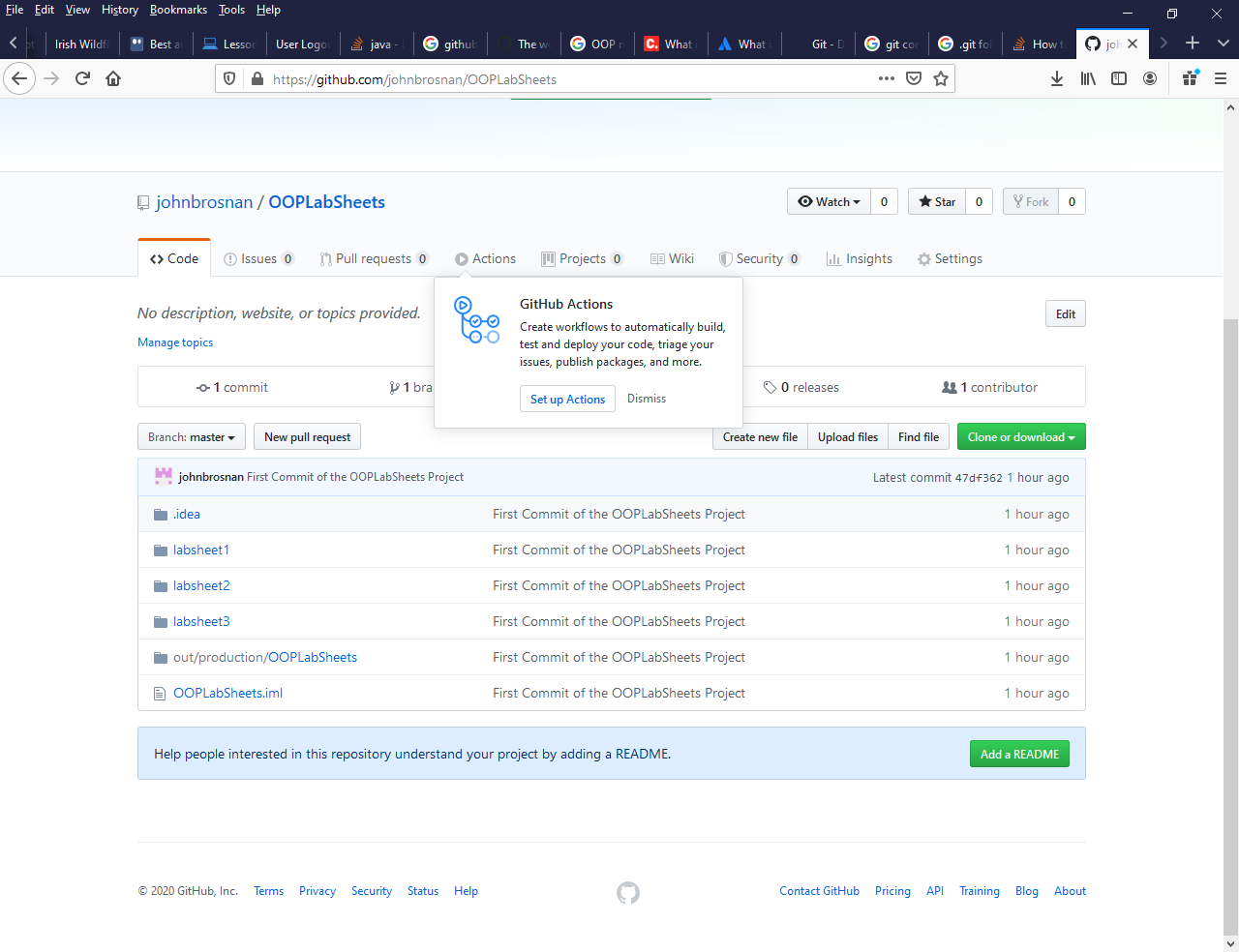
Now another dialog pops up requesting a name for the remote repository, it defaults to the name of the project itself, but you could call it something different if you wished. Now click **Share**



If everything goes to plan, you will get a “Successfully shared project on GitHub” message at the bottom-right of the IntelliJ window:



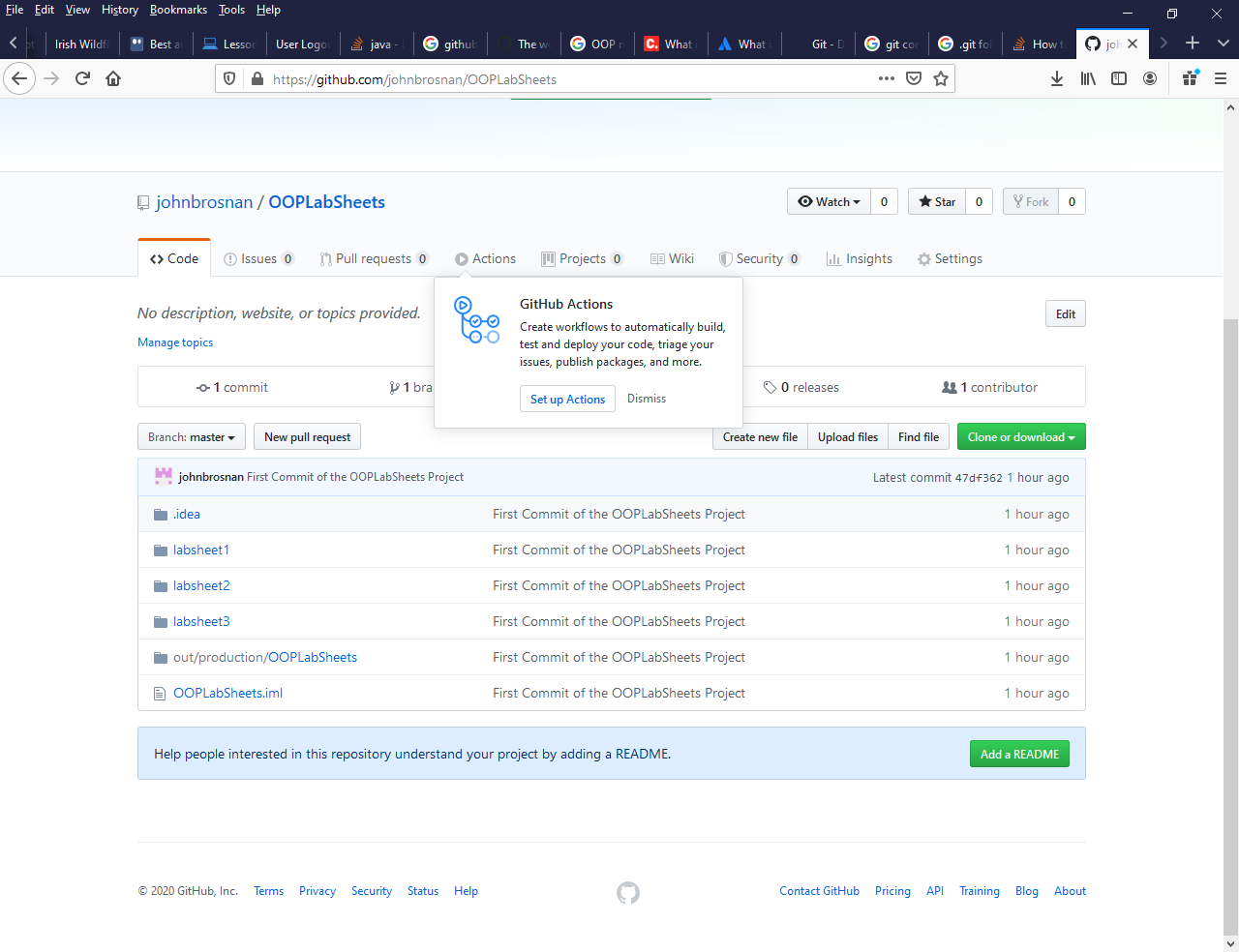
Now, for proof that the project is actually on GitHub, you can just **view your GitHub repositories list** and you should see the OOPLabSheets project listed. You can click into this then and, in my case, I see the following:



So now our local project has been stored on GitHub and we can **access it from anywhere**. We can also **share the link to a repository** with others and they could access it. This is the way you will deliver your mini-project in December.

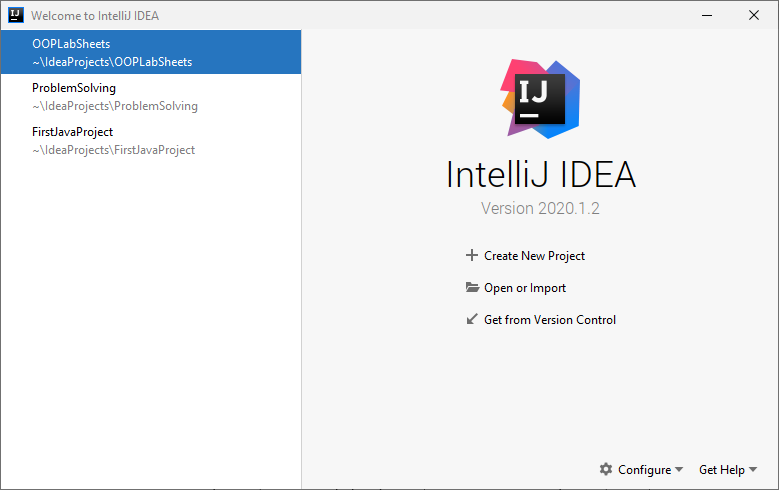
**Pulling an IntelliJ Project Directly from GitHub**

This is just the reverse action. A project has been “pushed” to GitHub and now we want to **download it locally** so that we can work on it.

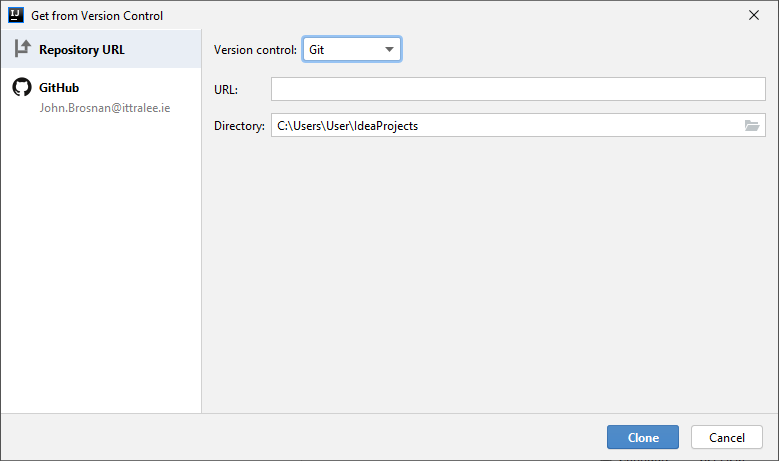


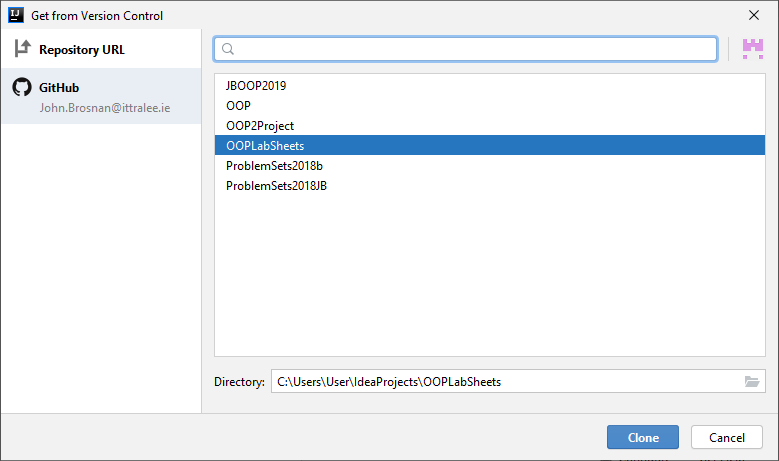
Referring to the window above in GitHub, you can see a green “**Clone or download”** button. Press this and a “**Clone with HTTPS**” dialog pops up. You can now press the “**Open in Desktop**” or the “**Download ZIP**” button. The “Open in Desktop” option will download all the source code as well as the Git repository (giving you a **clone** of the what is on GitHub) whereas the “Download Zip” option will only download the source code, meaning the Git repository won’t be locally available. Should you choose the “Open in Desktop” option then an attempt will be made to open the project within an environment called **GitHub Desktop** (which you would need to have installed on your machine, I don’t myself).

Instead of downloading from GitHub however, the **quicker route** is to use IntelliJ directly for the task. You can “**Get from Version Control**” as indicated in the window below



Now a “**Get from version Control**” window appears and, in the “**Repository URL**” tab, the “**Version Control**” option will default to “**Git**”. In the URL text-field you can enter the GitHub URL for a 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, if it is available to you. Alternatively, you could click on the “**GitHub**” tab on the left and, if you are connected to GitHub, you will be presented with a list of your previously “pushed” GitHub repositories. In my case, this list looks as follows:



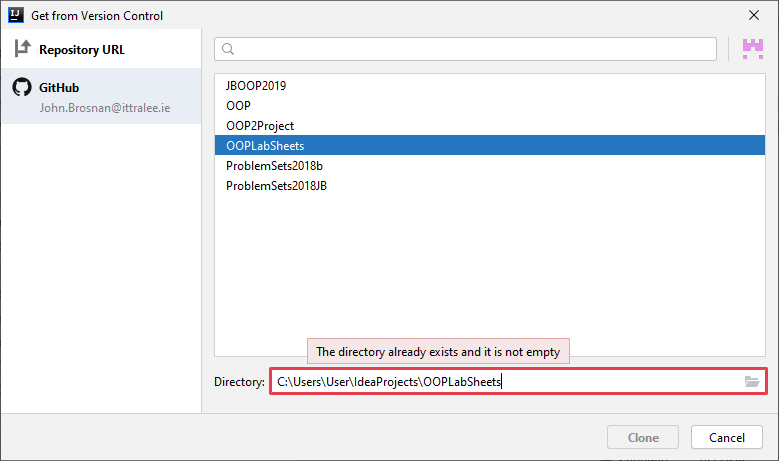


I would be selecting the **OOPLabSheets** repository at this point.

You can also decide where you want the repository to be located locally, I am choosing the same location as the original, on my C: drive

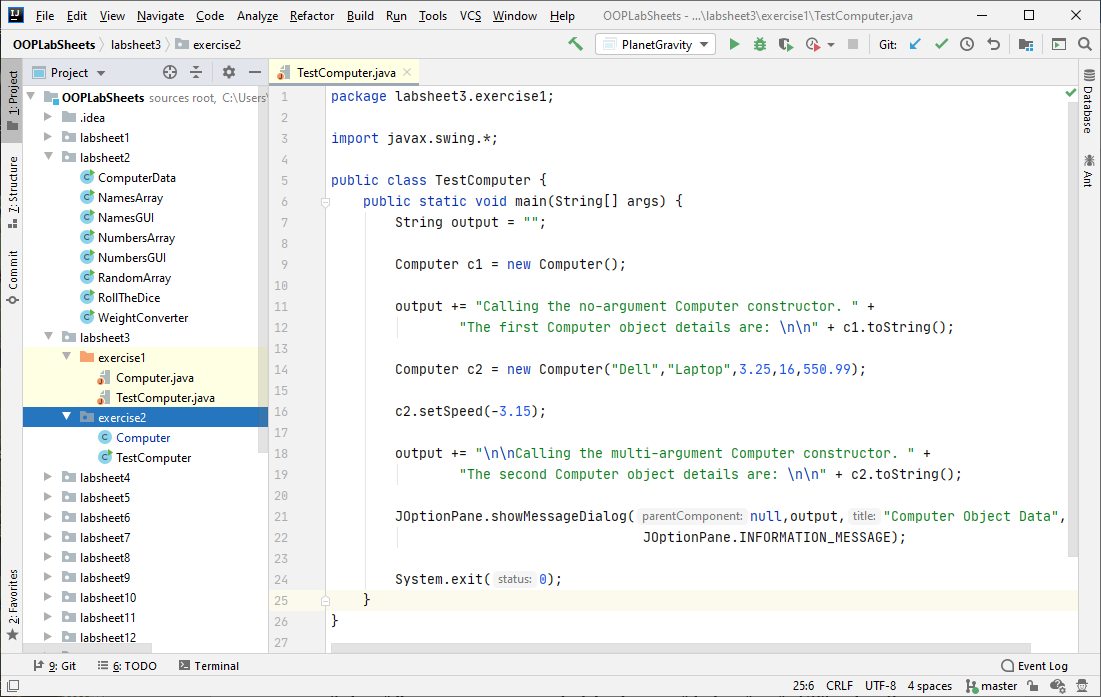
When you have the repository you want selected, you can just press the **Clone** button.

When I click the **Clone** button I get the following:



I am being told that cloning is not possible because I already have an OOPLabSheets folder at the location I am trying to clone to, so cloning won’t automatically overwrite an existing repository, which is a good thing really. So, you can **simply rename the original repository** if you wish to keep onto it just in case (give it a date e.g. OOPLabSheets21-09-20) and then your clone can go ahead.

Once cloned, you will be asked if you wish to open the project. Do this and what you see within IntelliJ will be the downloaded, version-controlled, project you “pushed” to GitHub earlier.



**Backing Up Your Work via GitHub**

So hopefully you can see, at a very high level, some of the benefits of a VCS such as Git and GitHub. I suggest that you work on your lab sheets locally in each lab session, **commit** your changes locally at the end of the session using Git, then **backup** your repository on either memory stick or X: drive and finally “**push**” your repository remotely to GitHub (some students forget to do this in class but it’s not a problem, just “push” the next time you get the chance).

Then the next time you want to work on the project, you can **clone** it from GitHub within IntelliJ and, once cloned locally, you can work with it again in a version-controlled manner.

It is intricate for sure, but once you do these things on a semi-regular basis, they will become second nature.

**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 **labsheet4** 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 **labsheet4** 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 **labsheet4**. 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.

**Developing the Instantiable Computer Class Further**

After the last lab sheet, we have our first, working, instantiable Java class created. However, there are some other features to talk about now in relation to the class.

OO is all about **software reuse**. However, at the moment the Computer class doesn’t maximise this feature. It works alright but there are opportunities to reuse code that we haven’t taken advantage of, and some of these will improve the code also, making it more robust.

For example, in the **multi-argument constructor** we have:

public Computer(String manufacturer,String type,double speed,int RAM,double price) {  
 this.manufacturer = manufacturer;  
 this.type = type;  
 this.speed = speed;  
 this.RAM = RAM;  
 this.price = price;  
}

Although the code is perfect, we are **altering the values of the attributes directly** within the constructor. You might recall that the ability to do this can potentially put an object into an “**inconsistent state**”, since there are **no checks or balances** present to prevent it. You’ll also recall that we added some very basic test code to some of the mutators (setters) in order to pick up negative numeric values and prevent certain attributes being set to these values, such as RAM, speed and price.

There would be nothing to stop us, in a driver program, from doing the following:

Computer c2 = new Computer("Dell","Laptop",-3.25,-16,-550.99);

and the values of the three attributes would be set to negative values. We can prevent this now by modifying the constructor so that it only **alters the attributes indirectly**, via calls to the mutators, as follows:

public Computer(String manufacturer,String type,double speed,int RAM,double price) {  
 setManufacturer(manufacturer);  
 setType(type);  
 setSpeed(speed);  
 setRAM(RAM);  
 setPrice(price);  
}

The calls to the mutators now mean that there is no possibility that a Computer object can have negative values for any of the numeric attributes.

What about the **no-argument constructor**? Well it looks as follows currently:

public Computer() {  
 manufacturer = "Not available";  
 type = "Not available";  
 speed = 0;  
 RAM = 0;  
 price = 0;  
}

Again, the attributes of the class are being **altered directly** here, but there is an important difference. Here the attributes are being set to values dictated by the class designer, and so the values will surely be proper if the class designer is worth their salt 😊

So you will often see the no-argument constructor written in this form, where the attributes are altered directly. However, we will try to **maximise software reuse** in this module and so we will now write it as:

public Computer()  
{  
 setManufacturer("Not available");  
 setType("Not available");  
 setSpeed(0);  
 setRAM(0);  
 setPrice(0);  
}

So again, we are using the mutators to **indirectly alter** the values of the attributes. To be fair, both techniques are correct, but it is good to reuse the mutators and the penalty in terms of execution time is small.

Believe it or not, we can actually code the no-argument constructor in a much sleeker fashion, by using the **this** reference we touched on in the last lab sheet. We said the this reference is effectively a “pointer” to the object itself (every object has its own “this” reference). We used it when writing the original multi-argument constructor, as you can see above. Can you recall why it was needed there?

The this reference can also be used when writing instantiable classes as a **shortcut to** **refer to the name of the classes constructor**. So, we can code the no-argument constructor as simply:

public Computer() {  
 this("Not available","Not available",0,0,0);  
}

It may look strange at first glance, but this no-argument constructor is just immediately calling (reusing) the multi-argument constructor and passing in the values you see to the constructor when it is called. The multi-argument constructor executes, it calls each of the five mutators in turn and sets each of the attributes to one of the values passed in.

So I will always ask you to try to write your no-argument constructor in this fashion, in order to **maximise software reuse**. But it is also important that you can “read” and recognise the other forms also of course.

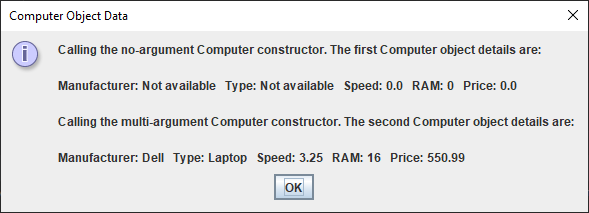
**Exercise 1**

From IntelliJ, within the **labsheet4** package, now create another package called **exercise1** by right-clicking on labsheet4, then selecting **New->Package**

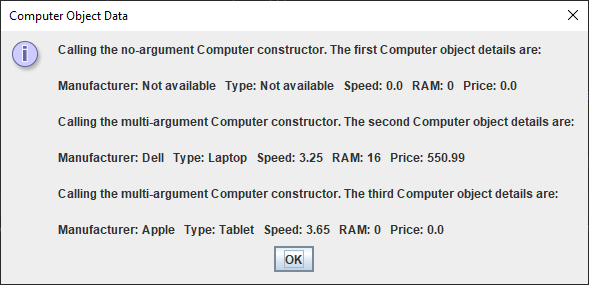


Now **copy** the **Computer.java** file and the **TestComputer.java** file from the **labsheet3** **exercise1** package to the **labsheet4** **exercise1** package. Alter the multi-argument and no-argument constructors as discussed to prove that the Computer class still compiles and runs even though we have changed the way these constructors are written.

The driver should still produce the same output as before



Now **modify** the Computer class **setRAM**() and **setPrice**() methods so that negative values will not be accepted for those attributes (you did this as an exercise last time, but see if you can recall it). Next **modify the driver** so that it creates a third Computer object with the values “Apple”, “Tablet”,3.65,-16,-500.25 and add some additional code so that the state of this object gets displayed. This is to ensure the test code within the altered mutators are preventing invalid values for the RAM and price attributes. The program should now produce the following output:



You see the RAM and Price values for the 3rd Computer object are both zero, proving the alterations worked as required.

**The Minimalistic Driver Class**

As mentioned last time, the driver class is there to enable us to **test** out the functionality of our instantiable class to ensure it is working correctly as an individual unit – this is referred to as **unit testing** in the software testing world. This means that our **driver needs to test every public method** defined within the instantiable class.

We would like to do the testing using as little code as possible naturally. The driver you have now written for Exercise 1 **almost** does this. It tests the no-argument constructor (which in turn tests out the multi-argument constructor, which in turn tests out the 5 mutators, with default “good” values), then it tests out the multi-argument constructor with a set of “good” user-supplied values. Then it tests setSpeed() directly, by passing it a “bad” value to see if the test code within that mutator is working. It then tests out the setRAM() and setPrice() methods indirectly by calling the multi-argument constructor with “bad” values for the RAM and price attributes. Along the way the toString() method is being called a few times and we finally test this when the value of the variable output is displayed onto the message dialog.

Can we write the driver in a shorter manner here? The only thing we can improve really is to remove the direct call to setSpeed(-3.15) and alter the creation of the 3rd Computer object instead as follows:

Computer c3 = new Computer("Apple","Tablet",-3.65,-16,-500.25);

So that we are still testing the setSpeed() mutator with a “bad” value as required.

**The Default Constructor**

We created two constructor methods for our Computer class. It is normal for every instantiable class to contain at least one constructor method, since we need to be able to create objects (instances) from an instantiable class.

However, **if we do not provide any constructor for a class, then the Java runtime will automatically supply a no-argument constructor** which will initialise the attributes of the class to default values of **zeroes for numerics**, **false for boolean** attributes and **null** for **object references**.

So in our Computer class, if we did not define any constructors, Java would provide a no-argument one and set the RAM, speed and price attributes to **zero**, and the manufacturer and type attributes both to **null** (since they are **String** object references).

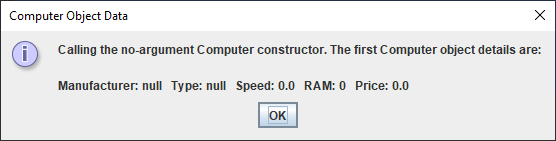
If an object reference is null, it just means that it is currently “pointing” to no object.

**Exercise 2**

To test this out now, modify the Computer class by commenting out the two constructors and modify the TestComputer class by commenting out the parts indicated below:

public class TestComputer {  
 public static void main(String[] args) {  
 String output = "";  
  
 Computer c1 = new Computer();  
  
 output += "Calling the no-argument Computer constructor. " +  
 "The first Computer object details are: \n\n" + c1.toString();  
  
 /\*Computer c2 = new Computer("Dell","Laptop",3.25,16,550.99);  
  
 c2.setSpeed(-3.15);  
  
 output += "\n\nCalling the multi-argument Computer constructor. " +  
 "The second Computer object details are: \n\n" + c2.toString();  
  
 Computer c3 = new Computer("Apple","Tablet",3.65,-16,-500.25);  
  
 output += "\n\nCalling the multi-argument Computer constructor. " +  
 "The third Computer object details are: \n\n" + c3.toString();\*/  
  
 JOptionPane.*showMessageDialog*(null,output,"Computer Object Data",  
 JOptionPane.*INFORMATION\_MESSAGE*);  
  
 System.*exit*(0);  
 }  
}

Now recompile the classes and run the driver to see the result. You should see the following:



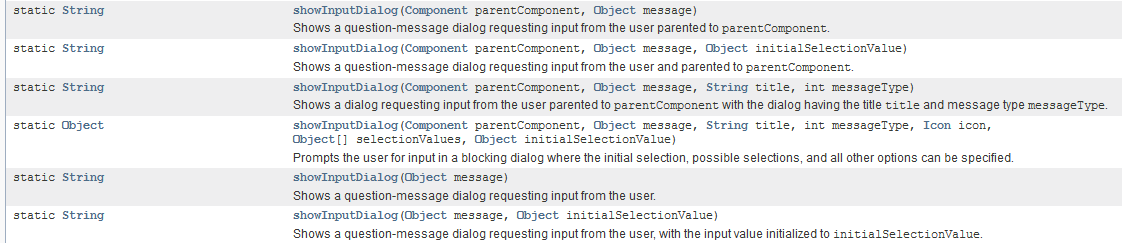
**Method Overloading**

This is another very useful OO feature. It refers to the ability to have several methods within a class that all **share exactly the same name**. The reason for this feature is to prevent the programmer from having to try to invent different names for operations that carry out almost the exact same task.

It is **present in almost all instantiable classes** because there are almost always at least two constructors defined in a class and these all share the same name. In the Computer class, each constructor has the job of initialising a new Computer object, so why not give them the same name.

The only requirement is that each overloaded method **must have a different argument list**. The **return type does not matter** in terms of method overloading – it would be illegal to have two methods in a class that share the same name and argument list, even if their return types differ.

The **Java API** is full of examples of overloaded methods e.g. the following list shows that there are 6 **showInputDialog**() methods within the JOptionPane class, even though I used just one of these throughout the semester 2 module. Can you see the one that I always used?



**Method Overriding**

Yet another important OO feature. Overriding a method just means the ability to provide an **alternative definition for an inherited method** (we will discuss **inheritance** much more in a later lab sheet).

In the last lab sheet, when we were developing the Computer class, it was mentioned that Java provides a toString() method automatically if we do not provide one in our own class. This toString() method belongs to the **Object** class which sits at the very top of the Java API hierarchy and from which all classes ultimately inherit. We discovered that this default toString() method does give us some information about the object on which it is called, the **class name** for the object and its **hashcode** (object ID), but it gives no information about the values of the object’s attributes, meaning we had to replace it with our own version.

Replacing an inherited method with our own version of it means we are **overriding** the inherited method. The method we write to replace the inherited one **must have the same name and argument list** as it, as well as the **same return type** (although a subtype of this return type is allowed).

So method overriding is something that **almost** **always takes place when creating instantiable classes** since we always need to know the state of objects created from our instantiable class and so end up overriding the toString() method from the Object class.

You will see other examples of overriding when we cover the **inheritance** topic later in the module.

**Exercise 3**

Within your **labsheet4** package, create another package called **exercise3** to store the classes needed for this exercise.

You must create an instantiable class called **Book.java** to represent a Book

A Book should have the following 4 **private** **attributes**:

* *title –* a String
* *price* – a double
* *ISBN* – a String
* *pages* – an int

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

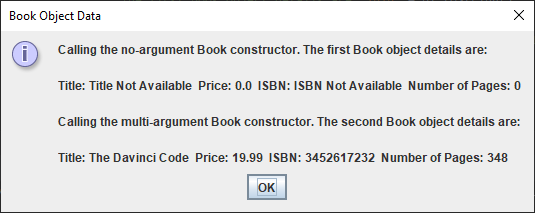
* a no-argument constructor
* a 4-argument constructor
* accessor methods for each of the attributes
* mutator methods for each of the attributes
* a toString() method which returns the state of the object’s attributes as a String

You can take it that the **no-argument constructor** will give a Book object the following initial state [“Title Not Available”, 0.00, “ISBN Not Available”, 0]

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

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

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

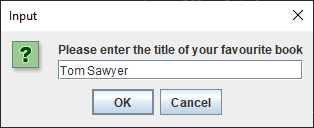
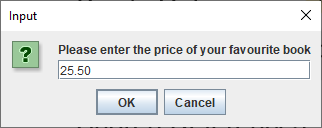


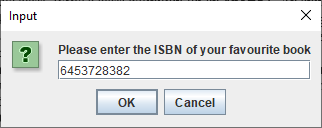
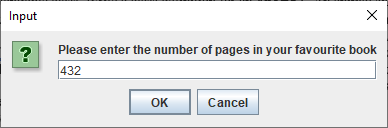
**Exercise 4**

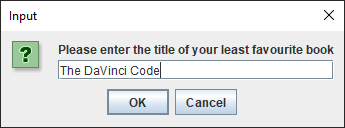
Within your **labsheet4** package, create another package called **exercise4** to store the classes needed for this exercise. **Copy** the Book and TestBook classes from the exercise3 package to this package.

Once you are satisfied that your minimalistic driver program is working from the previous exercise, alter the driver program so that it declares and creates two Book objects, which will represent the user’s favourite book and least-favourite book. Here the book details will be supplied by the user on input dialogs, rather than having just “hard-coded” values as in the original driver.

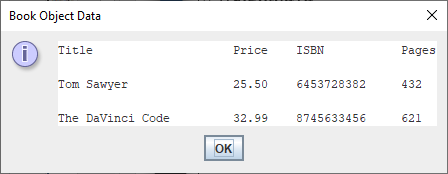
Once the two Book objects have been created, then display the details of all objects in one output dialog. You should try to display the details of the books **neatly aligned** within a **text-area** on the message dialog as indicated in the last screenshot below. Note that, in order to do this, **you can call the various accessors** in the driver to replace calling the toString() method on the object references.

 … followed by the rest of the inputs

Once the input has been supplied the Book objects get created and the output appears as follows:



**Exercise 5**

There is currently **no validation code** whatsoever in the Book class or the driver class, meaning that Book objects can be put in an “inconsistent state”. Within the **labsheet4** package, now create another package called **exercise5**

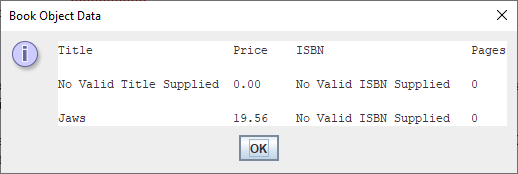
Now **copy** the **Book.java** and **TestBook.java** files from the exercise4 into the **exercise5** package.

Modify the Book class now as follows:

* change the setTitle() method so that, if the user enters no value at all for the title (enters the **empty string**) or hits the X button on the title bar of the input dialog (in which case the title attribute would be set to null), then the title attribute will be set to “No Valid Title Supplied”
* change the setPrice() method so that, if the user enters a negative value for the price, or a value over €50,000, then the price attribute will be set to zero.
* change the setISBN() method so that, if the user enters no value at all for the ISBN (enters the **empty string**) or hits the X button on the title bar of the input dialog (in which case the ISBN attribute would be set to null), then the ISBN attribute will be set to “No Valid ISBN Supplied”
* change the setPages() method so that if the user enters a negative value for the pages, or a value over 4,000, then the pages attribute will be set to zero.

With these changes made, test out the altered methods by entering some “bad” values. Your driver class can remain **completely unchanged** here.

In my case below, I entered all “bad” values for the favourite book and then 2 good and 2 bad for the least-favourite book:



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

In first year, the entire focus was on your Java coding ability and your ability get to grips with the basic Java **syntax**, to **problem-solve** and to **apply** and **adapt** what you had learned to new scenarios. There were also theoretical concepts covered of course such as variables, types, operators etc. You will still be coding and problem-solving in this module, but the focus will not be entirely on these activities. You will also **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.

**Default Constructor** – A **no-argument constructor** that is **automatically provided** by the Java runtime if we don’t provide one of our own. Such a constructor will initialize the attributes of the object to zeroes for numeric attributes, false for boolean attributes and null for attributes that are object references.

**Method Overloading** – an OO feature that enables us to create **several methods** within the same class that all have **exactly the same name**. This prevents us from having to invent different names for methods that carry out essentially the same task. The overloaded methods are **distinguished by their argument lists**.

**Method Overriding** – an OO feature that enables us to **provide an alternative definition for an inherited method**. The method we write to replace the inherited one must have the **same name and argument list** as it.