**Institute of Technology Tralee**

**Computing Department**

**Object Oriented Programming**

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

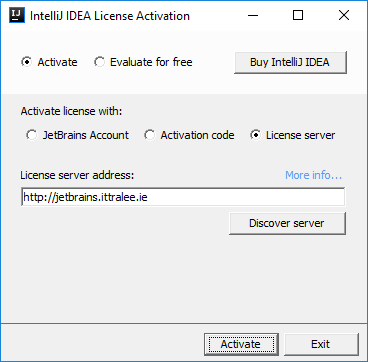
**Practical 3 – Classes & Objects**

Hopefully you got through most of the revision exercises from last week. If not, then do make sure to **spend some extra time at home** so that you don’t fall behind. Every extra hour you put in now will pay serious dividends as your Java journey will continue into semester 4, semester 5 and on into 4th year also. We begin our examination of OOP by discussing some concepts that have been mentioned several times throughout first year, **classes** and **objects**, but now we will drill down into the detail a little more.

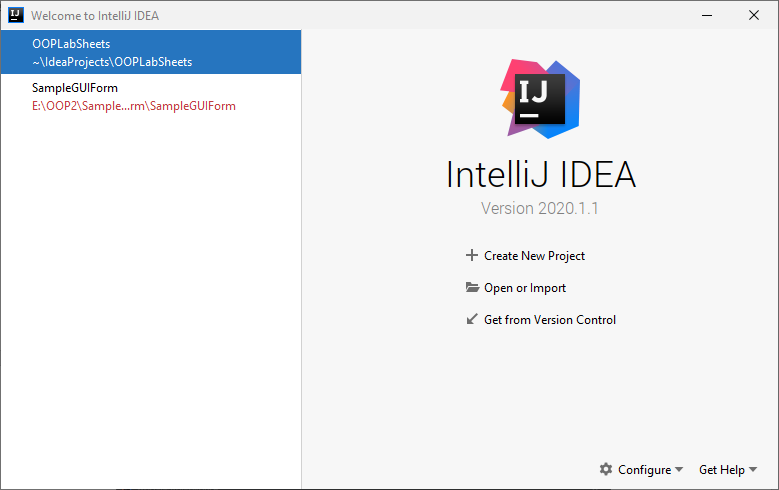
**Getting into IntelliJ**

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**



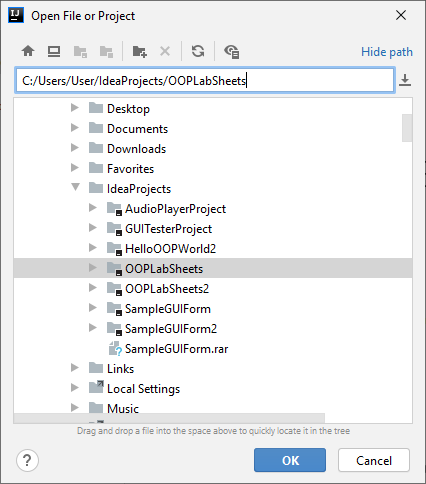
When completed, the IntelliJ **Integrated Development Environment** (**IDE**) should launch for you, after a few seconds. Once this happens, you are ready for coding!



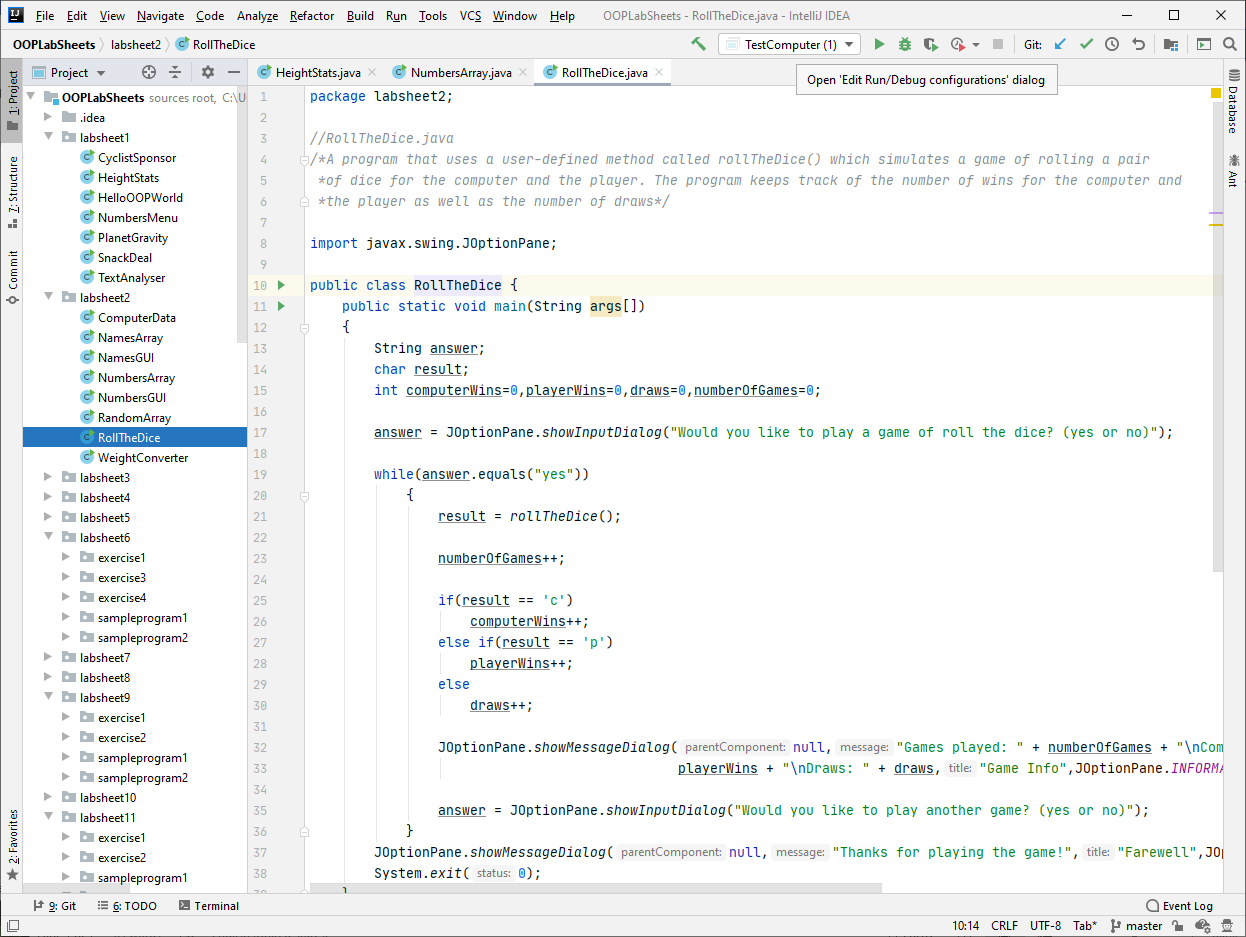
To begin with above, you will see options to “Create New Project”, “Open or Import” etc. Click “**Open or Import**” and try to navigate your way to the location where you have stored the project you created last time called **OOPLabSheets**. It may be on your X: drive or memory stick. If it is on X: 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.

**Before you leave the lab, if you have been working from the C: drive throughout the session, do make sure to copy your project to the X: drive or memory stick before you leave, otherwise your work will not be available to you.**

You can see below that 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 I select the project I want, I just click OK and then the project opens within IntelliJ as follows:



You can see that the project contains a folder called **.idea** and a file called **OOPLabSheets.iml** – these are just for **configuration** purposes and as programmers we normally have nothing to do with them. Our main focus would be the contents of the packages **labsheet1** and **labsheet2** that we created to store all the work we did for the first two lab sheets. You can see within those packages several Java files that made up the last two lab sheets and some of the ones from those packages are opened in the code editor window.

**Setting up your Folder Structure**

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 **labsheet3** 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 **labsheet3** 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 **labsheet3**. 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, later in the lab sheet, you will be completing some exercises, each of which will require you to create a **sub-package** within this labsheet3 package.

At this point you are technically ready to code! However, there are some very important **OO concepts** that need to be explained first and these will consume much of this lab sheet.

Although the modules in first year made many references to the notions of **class**, **object** and **object reference** as it went on, it was always in association with **pre-defined Java API classes** such as System, JOptionPane, String and so on. Although we will of course still be using many pre-defined API classes in this module, we will be focusing most of our attention on how **user-defined classes** are designed and coded.

**Procedural Programming versus OOP**

For the vast majority of first year, you placed all of your program code into either the main() method or within some user-defined method you created, and these were then placed within the confines of a Java class, similar to the following:

public class ProceduralClass //the class is just acting as a container for the program code

{

public static void main (String args[])

{

//a lot of your program code was typically put here (all of it in first semester!)

}

public static int anotherMethod (int a, double b, String c)

{

//and then you might have another user-defined method with its code here

}

public static void yetAnotherMethod (int a[], String b, float c, String d[])

{

//and then you might have yet another user-defined method with its code here

}

//could have defined several other user-defined methods

}

Technically, every class you created in first year (and in the first 2 lab sheets of this module) was a **user-defined class**, since you named it yourself and decided what code it would contain. However, every class you created was **simply a container** for your main() as there was **no intention to try to reuse the class** for any other purposes outside of the current application.

You were effectively writing your Java code in a so-called **procedural programming** style. This is exactly the way you would write your code in purely procedural languages such as C, Pascal and Fortran.

Purely procedural programming languages **do not support the notion of classes or objects**. The basic programming building-block in such languages is the **function/procedure** (what we call a **method** in Java) and they are often referred to as **action-oriented languages** since each function/procedure (method) describes a particular action that is to take place. However, their major drawback is that they **cannot fully support code reuse** in the same way that OOP languages can, which can affect the development time and costs of a given project.

Procedural programming languages are still hugely important in their own right and many complex systems have been developed with them e.g. the Unix/Linux operating systems are written in C.

As the name suggest, Object Oriented Programming (OOP) is all geared towards the use of **objects**. OOP languages such as Java allow us to take advantage of various OOP concepts which promote **software reuse**. These include **classes**, which arethe basic programming building-block in OOP**,** but also features such as **composition**, **inheritance** and **polymorphism** that simply are not available within procedural programming languages. Programs built with the OO style are considered **easier to maintain**, are **more extensible** and are considered **easier to manage** in group situations. These advantages can mean improved development times and costs for projects.

**Designing our First Instantiable User-Defined Class from First Principles**

We now move from creating Java classes as mere “containers” to actually using them as **modelling elements**. In everyday life, humans naturally separate things as **objects** – we have a “computer” object, a “person” object, a “chair” object etc. Each of these objects will have a particular set of **properties** associated with it e.g. the computer object might have the following properties:

* Manufacturer
* Type
* CPU Speed
* Quantity of RAM
* Price

Of course, it could have several other properties that we might be interested in modeling, but we can keep it simple for now. This key OO technique of only choosing certain properties when modelling an entity is called **abstraction**. It is **vital** to do this, so that only properties that are actually relevant to the application being developed are the only ones contained within the class structure. It means our code will be **less complex** as a result and **easier to maintain** going forward.

We could now create a **Computer** class which would define these properties for any Computer object that we might like to create in the future. In OOP terminology, the resulting object would be called an **instance** of the Computer class, making the Computer class an **instantiable class**.

public class Computer {  
 String manufacturer;  
 String type;  
 double speed;  
 int RAM;  
 double price;

These properties defined within the class are referred to as **attributes** of the class and each will be given a **type** as you might expect. As these attributes are defined outside of any method (even though I have no method defined yet), they are **globally available** within the class – whereas most of the time in year 1 when you created variables, they were just **locally available** within the method in which they were defined.

The next thing we must decide is what exactly do we want to be able to do with objects that we create from this class i.e. what will the **functionality** of the class be and what will we be able to do with **objects** **that we create from this class**? This functionality, as you might expect, will be defined by a **set of methods**.

Well one thing we definitely need to be able to do is to **create a Computer object** from the Computer class itself. This object creation will occur at **runtime**, dynamically, whereas the definition of the class itself is a **compile-time** activity and, unlike the **dynamic** nature of the object, the instantiable class is **static** in nature – this means that it is “useless” to us on its own.

Object creation will be done by creating a **special method** within the class known as a **constructor**. As the name suggests, a constructor method allows you to **construct an object** and **give its properties a set of initial values**. A constructor method will have **exactly the same name as the class** it is defined in, so here our constructor method must be called **Computer**(). Also, a constructor method will **never have a return type** (that’s what distinguishes it as a constructor versus a “regular” method).

So, when we wish to create a Computer object, we might like to give it a set of **programmer-supplied** default values to begin with, using a **no-argument constructor**. In this case, we might code the constructor as follows:

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

Now we might also like a constructor that will let us set up a Computer object’s attributes with some “real” **user-supplied** values rather than default ones. This **multi-argument constructor** might look as follows:

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;  
}

The idea here is that, at runtime, a Computer object will be created and it will be supplied with a set of user-supplied values for each of its attributes. So, the method here takes 5 arguments and each of the attributes is set up with one of these values. You won’t have seen the keyword **this** in first year. It is called the **“this” reference** and it is needed here so that we can distinguish between, for example, the manufacturer *attribute* and the manufacturer *argument*. Without this keyword, both occurrences of the word manufacturer would refer to the argument and the attribute would not be assigned a value at all (a **logical error** would occur).

Note that we could have used the this reference in the other constructor but it was not needed, as there were no arguments passed into that method. Also, it would not have been needed for this method had the names of the arguments been different to the names of the attributes, but it is considered **good OOP style** to keep them exactly the same.

Now there is lots to talk about yet in terms of modelling the Computer class and the rest of the functionality it should contain. However now is a good time to check to see if we can actually create a Computer **object** from our basic-looking class.

This will be done within the main() method, the one we always use to run the application. However, we will place this main() into a **completely separate Java file** rather than into the Computer class. There would be no issue placing the main() into the Computer class but, when coding in the “true” OO style, it is traditional to place the main() into a separate **“driver” class** of its own when testing out the functionality of a class. Also, by tradition, this “driver” is given the name TestX, where X is the name of the class being tested. So, we will call our driver class **TestComputer** here. My driver class here looks as follows:

package labsheet3;

import javax.swing.\*;  
  
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;  
  
 Computer c2 = new Computer("Dell","Laptop",3.25,16,550.99);  
  
 output += "\n\nCalling the multi-argument Computer constructor. " +  
 "The second Computer object details are: \n\n" + c2;  
  
 JOptionPane.*showMessageDialog*(null,output,"Computer Object Data",JOptionPane.*INFORMATION\_MESSAGE*);  
  
 System.*exit*(0);  
 }  
  
}

The line of code

Computer c1 = new Computer();

Creates a new Computer object using the **no-argument constructor**. You see the **new** keyword being used, it will look familiar as you seen it many times in first year. This keyword is **always associated with object creation** and is always **followed by a call to a constructor** method. So, the right-hand side of the assignment statement above is actually the Computer object that has just been created. When the constructor method is called here then the code we had earlier for the method is executed and the attributes of the newly created object are initialized with the default values you see in the method.

c1 is an **object reference**. Again, you will have seen object references many times in first year e.g. when you used the Scanner class throughout semester 1, I always used the object reference which I named input to act as a reference to the Scanner object, so that we could easily read information supplied by the user from the keyboard. The **object reference is crucial** because, for instantiable classes, it is the normal **means by which we can manipulate an object**, by calling various **instance methods** on the object reference to do so. So, going back to first year again for a second we would have had:

**int age = input.nextInt();**

when we wished to use the reference to the Scanner object (called input) to enable us to read in a whole number supplied by the user via the keyboard. This value would have been read in via the call to the method nextInt() and then (as long as the value was a valid Java integer) it would have been stored in the integer variable age.

Next I join a piece of text to the String variable output, this text will display later on the message dialog. You will see I have the object reference c1 joined on to this text.

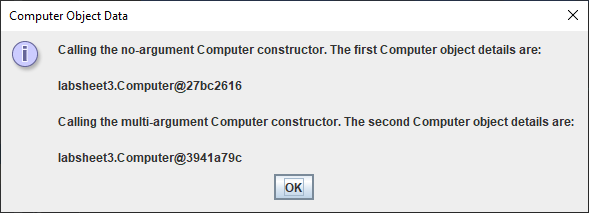
The line of code:

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

creates a new Computer object using the **multi-argument constructor**. When this constructor method is called here, then the code we had earlier for the method is executed and the attributes of the newly created object are initialized with the 5 values you see here in the method call. c2 is the object reference in this case.

Now I again join a piece of text to the String variable output, this text will display later on the message dialog. You will see I have the object reference c2 joined on to this text.

Finally, I have a **message dialog** displaying the output from the driver class. When the program runs you would get the following:



So we have some output from the driver class main(). The text is as we might expect but you see that, where I put in code to display the values associated with the references c1 and c2, we are getting some strange looking information.

The part **labsheet3.Computer@27bc2616** is output, but we might have expected the values associated with the first Computer to be displayed here instead. What is happening here, and this is not at all obvious, is that when we try to display the value of the Computer object associated with the reference c1, then automatically, at run-time, Java will call a **default toString() method** on the reference. This default toString() is defined within the Java API and, when it is called, it will display the **package/class name** which the reference is associated with, along with a so-called object **hashcode**, which is like a **unique object ID** that is supplied for every object within the system. So the general format of this is always **packagename.Classname@hashcode**

So, to be fair, this default toString() is giving some useful information alright, but not quite the information we are after 😊

We need to alter the code for the main() within the driver class to see the values associated with the Computer objects. We can do the following instead:

package labsheet3;  
  
import javax.swing.\*;  
  
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" +  
 "Manufacturer: " + c1.manufacturer + " Type: " + c1.type + " Speed: " +

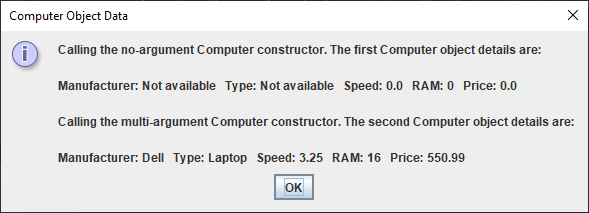
c1.speed + " RAM: " + c1.RAM + " Price: " + c1.price;  
  
 Computer c2 = new Computer("Dell","Laptop",3.25,16,550.99);  
  
 output += "\n\nCalling the multi-argument Computer constructor. " +  
 "The second Computer object details are: \n\n" +  
 "Manufacturer: " + c2.manufacturer + " Type: " + c2.type + " Speed: " +

c2.speed + " RAM: " + c2.RAM + " Price: " + c2.price;  
   
 JOptionPane.*showMessageDialog*(null,output,"Computer Object Data",

JOptionPane.*INFORMATION\_MESSAGE*);  
  
 System.*exit*(0);  
 }

}

So we **access each of the attributes directly** on the object reference as shown above. This produces the following output:



Now we are getting somewhere! We see the values of the attributes associated with each of the Computer objects created. However, there is an **issue** with this kind of approach and it isn’t very obvious at first sight.

It certainly works well to allow us to access the attributes values, but we could just as easily use a similar technique to allow us to alter the value of an attribute e.g. imagine that I now add the line of code

c2.speed = -2.75;

just before the call to showMessageDialog(). You will note that the original value of the speed attribute when the second Computer object was created was 3.25. However, I am now changing it to -2.75. We all know that the CPU speed for a computer must be a positive value and yet I am able to make it negative without any checks or balances being performed on the value. This is **major design flaw**, since any Computer object can fall into an “**inconsistent state**”, as direct access to its attributes is currently allowed from another class. So the current approach will work fine in terms of accessing attributes, but is flawed when trying to alter attribute values.

What we really want is to **be able to “hide” the attributes of a class from the outside world** so that any other class that needs to use this class cannot get at the attributes directly. This can be achieved easily though, by just making the attributes of the class **private**. This is a Java **keyword** you will have seen a few times in first year, but we never really explained it fully at the time.

At the moment the 5 attributes defined within the Computer class are said to have **package access**. This means that the attributes are directly available to any other class within the same package as the Computer class. Of course, the driver class TestComputer is also defined to be within the same package as Computer, so it can directly access those attributes, as we have seen. We now alter the code to make these attributes private as follows:

private String manufacturer;  
private String type;  
private double speed;  
private int RAM;  
private double price;

It’s as simple as sticking the keyword **private** before each of the attributes. Now, should we try to compile the driver class, we get the following string of syntax errors:

C:\Users\User\IdeaProjects\OOPLabSheets\labsheet3\TestComputer.java

Error:(13, 38) java: manufacturer has private access in labsheet3.Computer

Error:(13, 70) java: type has private access in labsheet3.Computer

Error:(13, 95) java: speed has private access in labsheet3.Computer

Error:(14, 32) java: RAM has private access in labsheet3.Computer

Error:(14, 56) java: price has private access in labsheet3.Computer

Error:(20, 38) java: manufacturer has private access in labsheet3.Computer

Error:(20, 70) java: type has private access in labsheet3.Computer

Error:(20, 95) java: speed has private access in labsheet3.Computer

Error:(21, 32) java: RAM has private access in labsheet3.Computer

Error:(21, 56) java: price has private access in labsheet3.Computer

Error:(23, 11) java: speed has private access in labsheet3.Computer

The compiler is telling us now that we cannot directly access the attributes defined within the Computer class any longer since they are all defined private within it.

So we’ve fixed one problem but created another 😊 The attributes are “hidden” now from external classes but this means we cannot display the state of the Computer objects any longer from the driver, as it won’t even compile! The process of deliberately making the attributes of a class all private is called “**information hiding**” and is considered **good programming practice** in OO.

To solve this issue, we must add code to the Computer class that will enable us to access the attributes from an external class **indirectly**, this can be done by creating appropriate **“accessor” methods**. These public methods look as follows:

public String getManufacturer() {  
 return manufacturer;  
}  
  
public String getType() {  
 return type;  
}  
  
public double getSpeed() {  
 return speed;  
}  
  
public int getRAM() {  
 return RAM;  
}  
  
public double getPrice() {  
 return price;  
}

The accessor methods (often called “**getters**”), by convention, all begin with the word “get”. This is because the methods allow us to get the value of one of the attributes of the class via the method call. Each of the methods return the appropriate attribute. So all these accessors will have a **return type and this matches exactly the type associated with the relevant attribute**.

With these accessor methods added to the Computer class, we now turn our attention to the driver class, and we must alter its code so that it now calls these accessor methods where we originally tried to access the attributes directly. The driver class now looks as follows:

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" +  
 "Manufacturer: " + c1.getManufacturer() + " Type: " +

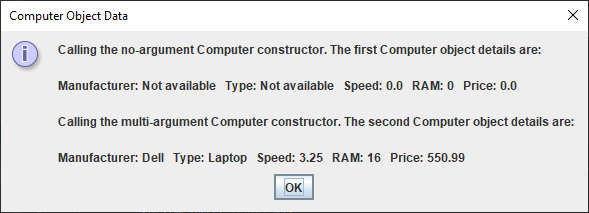
c1.getType() + " Speed: " + c1.getSpeed() +  
 " RAM: " + c1.getRAM() + " Price: " + c1.getPrice();  
  
 Computer c2 = new Computer("Dell","Laptop",3.25,16,550.99);  
  
 output += "\n\nCalling the multi-argument Computer constructor. " +  
 "The second Computer object details are: \n\n" +  
 "Manufacturer: " + c2.getManufacturer() + " Type: " +

c2.getType() + " Speed: " + c2.getSpeed() +  
 " RAM: " + c2.getRAM() + " Price: " + c2.getPrice();  
  
 JOptionPane.*showMessageDialog*(null,output,"Computer Object

Data",JOptionPane.*INFORMATION\_MESSAGE*);  
  
 System.*exit*(0);  
 }

}

Compiling and running the application now produces the following output:



So the output is as we expect. So far, so good! We have proven that we can access the values of the Computer object’s attributes. But again, what if we wish to alter the value of one of the Computer object’s attributes? With the attributes now private we know we cannot try to do this using code such as

c2.speed = -2.75;

We must add code to the Computer class that will enable us to alter the attributes from an external class **indirectly**, this can be done by creating appropriate **“mutator” methods**. These public methods look as follows:

public void setManufacturer(String manufacturer) {  
 this.manufacturer = manufacturer;  
}  
  
public void setType(String type) {  
 this.type = type;  
}  
  
public void setSpeed(double speed) {  
 this.speed = speed;  
}  
  
public void setRAM(int RAM) {  
 this.RAM = RAM;  
}  
  
public void setPrice(double price) {  
 this.price = price;  
}

The mutator methods (often called “**setters**”), by convention, all begin with the word “set”. This is because the methods allow us to set the value of one of the attributes of the class via the method call. Each of the methods alters the appropriate attribute to the value passed in as an argument, which has exactly the same type as the relevant attribute. All these mutators have a **return type of void**. Note the use of the **this** keyword again here in each of the methods.

With these mutator methods added to the Computer class, we again turn our attention to the driver class and we try to alter the value of the CPU speed for the second Computer object as follows:

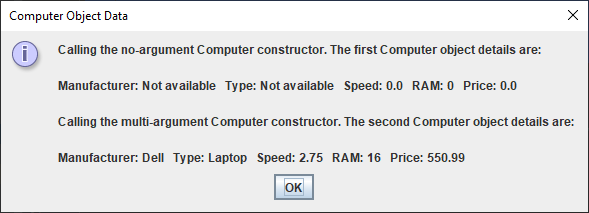
c2.setSpeed(2.75);

putting this line of code in the program in the following location:

Computer c2 = new Computer("Dell","Laptop",3.25,16,550.99);  
  
c2.setSpeed(2.75);  
  
output += "\n\nCalling the multi-argument Computer constructor. " +  
 "The second Computer object details are: \n\n" +  
 "Manufacturer: " + c2.getManufacturer() + " Type: " + c2.getType() + " Speed: " +

c2.getSpeed() + " RAM: " + c2.getRAM() + " Price: " + c2.getPrice();

The output from the program will now be



So the CPU speed of the second Computer has now been changed successfully via the call to setSpeed(). As mentioned earlier, there was a **design flaw** in the original approach of making the attributes **package access** since we could put the Computer objects into an “inconsistent state” as we had direct access to them externally.

At the moment, we can still put Computer objects into an “inconsistent state” because, even though we have mutator methods written, we have no code within those methods that perform any checks whatsoever on the values being passed in as arguments. So if we wish to change the CPU speed for the second Computer object to -3.15 in the driver we can simply say:

c2.setSpeed(-3.15);

and the speed attribute will be successfully altered to -3.15, no questions asked.

To prevent this possibility, we can add some **simple test code** to the setSpeed() method to ensure the CPU speed can never be made negative as follows:

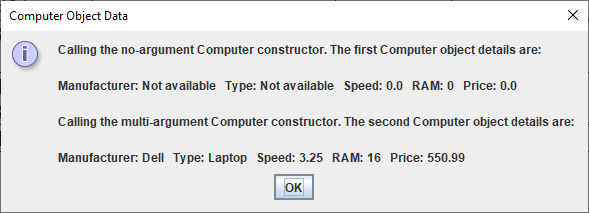
public void setSpeed(double speed) {  
 if(speed>0)  
 this.speed = speed;  
}

Now the following code in the driver class:

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" +  
 "Manufacturer: " + c2.getManufacturer() + " Type: " + c2.getType() + " Speed: " +

c2.getSpeed() + " RAM: " + c2.getRAM() + " Price: " + c2.getPrice();  
  
  
JOptionPane.*showMessageDialog*(null,output,"Computer Object Data",JOptionPane.*INFORMATION\_MESSAGE*);

produces the following output:



We have succeeded in preventing the object from becoming “inconsistent”, the speed value remains 3.25 rather than being set to -3.15. Of course, the test code added to the setSpeed() method is very primitive and an application could still accept invalid values for CPU speed, but at least we are making progress!

We are almost there now with creating a bare-bones instantiable Computer class. Because needing to know the **state of an object** is such a common requirement, for testing purposes, by convention all well-designed classes are expected to contain their own **toString**() method, which is used to return the state of an object. The state of an object just means the **values of each of its attributes**. The message dialog screenshots you have seen all prove that we can see the state of the Computer objects we have created, but we are doing so by calling the accessor methods individually, which does work. The code for the toString() method looks as follows:

public String toString() {  
 return "Manufacturer: " + getManufacturer() + " Type: " + getType() + " Speed: " + getSpeed() +  
 " RAM: " + getRAM() + " Price: " + getPrice();  
}

You can see that it is calling the various accessor methods and making a piece of text from the whole lot, before returning this text.

So now, in the driver, we can do the following instead:

package labsheet3;  
  
import javax.swing.\*;  
  
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();  
  
 JOptionPane.*showMessageDialog*(null,output,"Computer Object Data",  
 JOptionPane.*INFORMATION\_MESSAGE*);  
  
 System.*exit*(0);  
 }  
}

We simply call the toString() method on the Computer object references and it achieves the same result as before, but is a lot neater looking and is the way the complete state of an object is determined in practice.

So we have a working instantiable class at this point. You can see that we can create Computer objects and manipulate those objects only by calling the public methods (the **public “interface”**) defined within the Computer class. Also, you can see that the class’s data is now completely “hidden”, since it is private, and cannot be accessed directly from outside the class.

This OO technique of **grouping together** the attributes of a class and the methods that will act on those attributes is called **encapsulation**. **To the outside world, it provides a public interface** and **hides the data** andis a key feature of OO program design. It means that the implementation details of a class can change and, **as long as the public interface remains the same (or we just add to it)**, the end users of the class are unaffected and their **code need not change**. As mentioned earlier, it also means that the data of an object **cannot be put into an “inconsistent state”**, as long as some **error-checking** is done within the public mutators.

We will complete our instantiable Computer class in the next lab sheet.

**Exercise 1**

You won’t be typing out sample programs very often in this module, you’ll be glad to hear 😊 However, there are so many new features introduced in this lab sheet that I will ask you to type this one in for practice. You will write a lot of instantiable classes before this module is over! Zoom in on the code if you are having any trouble seeing it.

Now you should **create a new package** *within* your **labsheet3** package called **exercise1**. Do this by right-clicking on the labsheet3 package and selecting **New** 🡪 **Package**



package labsheet3.exercise1;  
  
public class Computer {  
 private String manufacturer;  
 private String type;  
 private double speed;  
 private int RAM;  
 private double price;  
  
  
 public Computer() {  
 manufacturer = "Not available";  
 type = "Not available";  
 speed = 0;  
 RAM = 0;  
 price = 0;  
 }  
  
 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;  
 }  
  
 public String getManufacturer() {  
 return manufacturer;  
 }  
  
 public String getType() {  
 return type;  
 }  
  
 public double getSpeed() {  
 return speed;  
 }  
  
 public int getRAM() {  
 return RAM;  
 }  
  
 public double getPrice() {  
 return price;  
 }  
  
 public void setManufacturer(String manufacturer) {  
 this.manufacturer = manufacturer;  
 }  
  
 public void setType(String type) {  
 this.type = type;  
 }  
  
 public void setSpeed(double speed) {  
 if(speed>0)  
 this.speed = speed;  
 }  
  
 public void setRAM(int RAM) {  
 this.RAM = RAM;  
 }  
  
 public void setPrice(double price) {  
 this.price = price;  
 }  
  
 public String toString() {  
 return "Manufacturer: " + getManufacturer() + " Type: " + getType() + " Speed: " + getSpeed()

+ " RAM: " + getRAM() + " Price: " + getPrice();  
 }  
}

You will be saving the file above as **Computer.java** and the one below as **TestComputer.java**.

package labsheet3.exercise1;  
  
import javax.swing.\*;  
  
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();  
  
 JOptionPane.*showMessageDialog*(null,output,"Computer Object Data",  
 JOptionPane.*INFORMATION\_MESSAGE*);  
  
 System.*exit*(0);  
 }  
}

Once you have the files typed in, **compile them** and then **run the driver** class.

**Exercise 2**

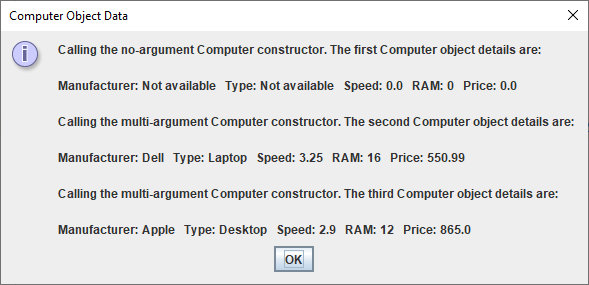
Now you should create a new package within your labsheet3 package called **exercise2**. Do this by right-clicking on the labsheet3 package and selecting **New** 🡪 **Package**

**Copy** the Computer and TestComputer classes now from the exercise1 package to this **exercise2** package.

Now perform the following additions to the files, **recompile and running** with each change

* Modify the driver program so that it creates a 3rd Computer object which should have the following initial values
  + “Apple”, “Desktop”, 2.9, 12, 865.00
* Modify the driver so that it also displays the state of this 3rd Computer object
* Modify the driver so that it calls setRAM() to change the value of the 3rd Computer object to -8 (putting it into an inconsistent state) and run it again
* Modify the Computer class so that the setRAM() and setPrice() methods also have basic testing within them to prevent a negative value being supplied for either the RAM or price attributes.

By the end of these changes, the program should produce the following output



**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.

**Class** – A class is the **main modelling element** in OOP. It is used to **model real-world entities** such as a car, a person, a computer etc. It is also the **main building-block** for application development. It is a **blueprint** for a set of entities (objects) that will all **share the same properties and functionality**. Classes are usually **static**, in that they cannot be interacted with directly at runtime, they are merely definitions and you **generally need to create (instantiate) objects** from classes in order for the classes to be of use. A class resembles a built-in data type such as int, float, char etc. but it is more complex, because it also supports functionality through its methods.

**Instantiable Class** – A class from which instances (objects) can be created. It is possible to design a class from which objects cannot be created, the Math class in the Java API is an example of such a class, it is an example of a **utility class**.

**Object** – An object is an **instance of a class**. Objects represent real-world entities, so you can have Student objects created from a Student class, Pet objects created from a Pet class etc. If two objects are instances of the same class then they will share common properties and functionality, but will typically have different values for their properties. Objects are **dynamic**, meaning that their properties can be altered, accessed or processed in various ways at runtime. Objects are **created via constructor** methods, using the new keyword.

**Abstraction** - A key OO technique which involves **only choosing certain properties** when modelling an entity. It is **vital** to do this, so that only properties that are actually relevant to the application being developed are the only ones contained within the class structure. It means our code will be **less complex** as a result and **easier to maintain** going forward.

**Object Reference** – An object reference is like a “pointer” to an object. They are crucial in OOP because it is only **through the object reference that an object can be manipulated**, via method calls defined for the object concerned.

**Constructor** – A constructor is a **special method** defined within a class, which is used to **create an instance** (object) of that class and **give its properties a set of initial values**. Constructors must always have the **same name as the class** in which they are defined and a class may have several constructors, which will have different argument lists. They have **no return type**, unlike regular methods.

**No-argument Constructor** – A constructor that takes no arguments. This type of constructor is defined in a class in order to give the objects that are created from it a set of **default initial values for their properties**, often zeroes for numerics and “not available” or “unknown” for String properties.

**Multi-argument Constructor** – A constructor that takes at least one argument. This type of constructor is defined in a class in order to give the objects that are created from it a set of **initial values supplied from outside the class**, often from a driver class but possibly from any other class that uses this class.

**this reference** – The this keyword is used as a **reference to the object under consideration**, so it is an object reference which points to “this” object. It is seen very often in class definitions because it allows us to distinguish between an attribute of a class and a method argument that has exactly the same name as the attribute, thereby preventing potential logical errors.

**Driver class** – A driver class is created to **test out the functionality** of an instantiable (or other type) of class. It contains a main(), so the driver runs the application, typically creates a number of objects and calls methods on these objects in order to ensure they are all working correctly. They are usually given a name that begins with the word “Test”.

**Information Hiding** – Information hiding is the process by which the **attributes of a class are made** **private**, thereby preventing direct access to them from outside the class, which could put any of the attribute values into an “inconsistent state”. It also means that users of a particular class **do not need to be aware of design changes** made within a class in order to use it, as long as the **public interface** of the class (the set of public methods defined for the class) remains unchanged (or is just added to). The user of the class can continue to use it without having to make any code changes to existing applications based on that class. Information hiding is facilitated by the OO technique of **encapsulation**.

**Accessor** – An accessor is a method defined within a class that allows us to **obtain (access) the value of a particular attribute** of an object created from that class. These methods are often called “getters” as, by convention, they begin with the word “get” e.g. getName(), getAge() etc. In instantiable classes, accessors are normally defined for every attribute of the class and are usually **public**.

**Mutator** – A mutator is a method defined within a class that allows us to **change (mutate) the value of a particular attribute** of an object created from that class. These methods are often called “setters” as, by convention, they begin with the word “set” e.g. setName(), setAge() etc. In instantiable classes, mutators are normally defined for every attribute of the class and are usually **public**.

**Object State** – Refers to the “state” of an object i.e. the **values of its attributes**. Because knowledge of the current state of an object is desirable for testing purposes, among other things, all instantiable classes normally define a **toString()** method which returns a stringified version of the values of the object’s attributes. This replaces the **default** **toString()** method that is **automatically provided** at runtime, which just gives the classname/hashcode of the object.

**Encapsulation** – Refers to the OO technique of **grouping together** the attributes of a class and the methods that will act on those attributes. **To the outside world, it provides a public interface** and **hides the data** andis a key feature of OO program design. It means that the implementation details of a class can change and, **as long as the public interface remains the same (or we just add to it)**, the end users of the class are unaffected and their **code need not change**. As mentioned earlier, it also means that the data of an object **cannot be put into an “inconsistent state”**, as long as some **error-checking** is done within the public mutators.