**Institute of Technology Tralee**

**Computing Department**

**Object Oriented Programming**

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

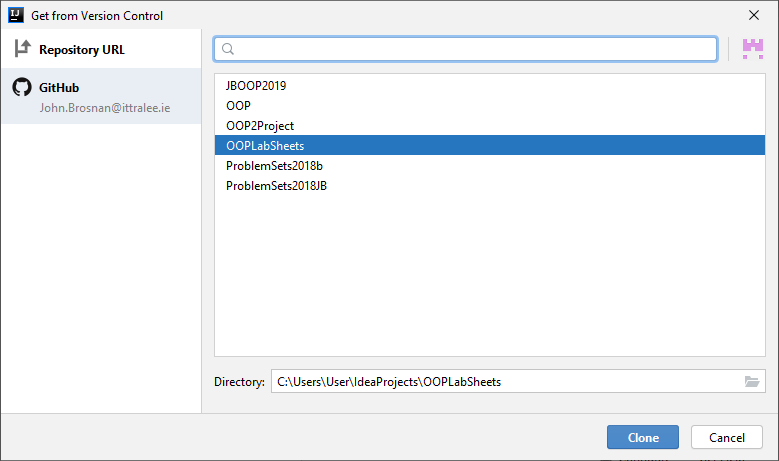
**Practical 7 – Composition & Aggregation**

Last time you were introduced to the **UML documentation standard** and the important OO concept of **composition**. In this lab sheet you will be introduced to a very similar, but slightly different concept, called **aggregation**. You will see the **UML connectors** for composition and aggregation. You will also attempt some exercises that involve both composition and aggregation. These exercises will be important preparation for your **first CA** of the module.

**Getting into IntelliJ**

Launch IntelliJ. As you were introduced to **VCS** already, 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 your X: drive to some location on C: or work directly with it from the memory stick).

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,



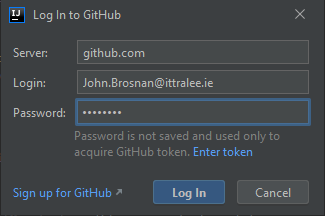
Recall that you **may need to rename** the local repository to something else if you get an error message at this point (I suggest a date e.g. OOPLabSheets08-10-20).

**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**

My own 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 **labsheet7** 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 **labsheet7** 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 **labsheet7**. 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.

**Aggregation in Java**

**Aggregation** is very similar to **composition** and is another important OO feature. It represents a **“has-a” relationship** between two entities where there is a “whole” which is made up of several “parts” but the **“parts” can exist independently of the “whole”.** A classic example in Java would be where you have a Department class and it is made up of a set of objects from a Student class. The key thing here is that the students **can exist independently** of the department. If the Department object is destroyed, the set of students that were associated with it can live on.

The following sample program **demonstrates aggregation** where there are two instantiable classes representing a LineItem and a Product. Here a given LineItem object will be able to associate with exactly one Product object.

**Instantiable class Product**

package labsheet7.sampleprogram1;  
  
*//Product.java  
/\*A simple instantiable class definition for a Product\*/*public class Product {  
 private int id;  
 private String name;  
 private String description;  
  
 public Product(int id, String name, String description) {  
 setId(id);  
 setName(name);  
 setDescription(description);  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public String getDescription() {  
 return description;  
 }  
 public void setDescription(String description) {  
 this.description = description;  
 }  
  
 public String toString() {  
 return "Product details are as follows: id: " + getId() + " name: " + getName() + " description: " + getDescription();  
 }  
}

**Analysis of the Product Class:**

● The Product class is a regular looking instantiable class containing a multi-argument constructor, a set of mutators and accessors for each of its three attributes and a toString(). Notice that I didn’t include a **no-argument constructor** here. I could have, of course, but it is good for you to see that you don’t have to write an instantiable class exactly the same way every time, and some instantiable classes can look quite different from the “conventional” ones that I have shown you so far in the notes.

● This class will allow us to create Product objects for the application. Next up will be the LineItem class, which will form a **“has-a” relationship** with the Product class. In an ordering system, there will be an Order object (which we are completely ignoring for this demo), which can contain one or more LineItem objects.

**Instantiable class LineItem**

package labsheet7.sampleprogram1;  
  
*//LineItem.java  
/\*A simple instantiable class definition for an line-item which also demonstrates  
the OO concept of aggregation\*/*public class LineItem {  
 private int id;  
 private int quantity;  
 private Product product;  
  
 public LineItem(int id, int quantity, Product product) {  
 setId(id);  
 setQuantity(quantity);  
 setProduct(product);  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public int getQuantity() {  
 return quantity;  
 }  
  
 public void setQuantity(int quantity) {  
 this.quantity = quantity;  
 }  
  
 public Product getProduct() {  
 return product;  
 }  
  
 public void setProduct(Product product) {  
 this.product = product;  
 }  
  
 public String toString() {  
 return "LineItem Details are as follows: \n\nid " + getId() + "\nquantity:" + getQuantity() + "\n" + getProduct() + "\n\n";  
 }  
}

**Analysis of the Product Class:**

● The LineItem class is also a regular looking instantiable class containing a multi-argument constructor, a set of mutators and accessors for each of its three attributes along with a toString() for testing purposes.

● However, one of the attributes of the LineItem class is a reference to a Product object. This immediately **creates an association** between the two classes. So we can say that a LineItem object “has-a” Product object “nested” within it by virtue of this association.

● Because a given Product object can still exist even if the LineItem object it is associated with it no longer exists, we say that there is an **aggregation relationship** between the two classes.

**Driver Class TestLineItem**

package labsheet7.sampleprogram1;  
  
*//TestLineItem.java  
/\*A driver (test) class that contains the main() method for  
testing the functionality of the LineItem and Product classes\*/*public class TestLineItem {  
 public static void main(String[] args) {  
  
 *//Create an array of LineItem objects* LineItem allLineItems[] = new LineItem[10];  
  
 *// Create Products* Product p1 = new Product(1, "Pen", "This is a red pen");  
 Product p2 = new Product(2, "Pencil", "This is a pencil");  
 Product p3 = new Product(3, "Ruler", "This is a ruler");  
 Product p4 = new Product(4,"Marker","This is a black permanent marker");  
  
 *// Create LineItem objects* LineItem item1 = new LineItem(1,3,p1);  
 LineItem item2 = new LineItem(2,4,p2);  
 LineItem item3 = new LineItem(3,2,p3);  
  
 allLineItems[0] = item1;  
 allLineItems[1] = item2;  
 allLineItems[2] = item3;  
  
 System.*out*.println("Displaying the state of all line-items ...\n\n");  
 for(int i=0;i<allLineItems.length;i++)  
 if(allLineItems[i]!=null)  
 System.*out*.println(allLineItems[i]);  
  
 System.*out*.println("Now changing product associated with first line-item to a marker...\n\n");  
 item1.setProduct(p4);  
  
 System.*out*.println("Displaying the state of all line-items again...\n\n");  
 for(int i=0;i<allLineItems.length;i++)  
 if(allLineItems[i]!=null)  
 System.*out*.println(allLineItems[i]);  
  
 System.*out*.println("Now destroying the second line item (the 4 pencils)...\n");  
  
 item2 = null; *//destroy the 2nd LineItem object in the array*

allLineItems[1] = null;  
  
 System.*out*.println("Displaying the state of the second product ...\n");  
 System.*out*.println(p2); *//and yet the Product associated with it still exists* System.*out*.println("Displaying the state of all line-items one last time...\n\n");  
 for(int i=0;i<allLineItems.length;i++)  
 if(allLineItems[i]!=null)  
 System.*out*.println(allLineItems[i]);

}  
}

**Analysis of the TestLineItem Class:**

● An array of LineItem is created to store up to 10 LineItem objects to begin with

● Next a set of four Product objects are created

● Next a set of three LineItem objects are created via the constructor, which are passed references to the various Product objects that exist. This **creates the link** between the LineItem objects and the Product objects associated with them.

● The three LineItem objects are added, one by one, to the array of LineItem created earlier.

● Next the state of all the LineItem objects held in the array is displayed, with the following output:

Displaying the state of all line-items ...

LineItem Details are as follows:

id 1

quantity:3

Product details are as follows: id: 1 name: Pen description: This is a red pen

LineItem Details are as follows:

id 2

quantity:4

Product details are as follows: id: 2 name: Pencil description: This is a pencil

LineItem Details are as follows:

id 3

quantity:2

Product details are as follows: id: 3 name: Ruler description: This is a ruler

So you can see that the call to the LineItem toString() method makes a call to the Product toString() method in order to display the state of the “nested” Product object. Notice the **null-checking** taking place here too, to prevent possible runtime errors.

● After this, the setProduct() mutator is called on the first LineItem object in order to change the product associated with it from a pen to a marker. The output from the program after this step is:

Now changing product associated with first line-item to a marker...

Displaying the state of all line-items again...

LineItem Details are as follows:

id 1

quantity:3

Product details are as follows: id: 4 name: Marker description: This is a black permanent marker

LineItem Details are as follows:

id 2

quantity:4

Product details are as follows: id: 2 name: Pencil description: This is a pencil

LineItem Details are as follows:

id 3

quantity:2

Product details are as follows: id: 3 name: Ruler description: This is a ruler

So you can see that the first line-item details have changed as required, to reflect the new product associated with it.

● The last part of the driver program destroys the 2nd LineItem object by setting the **item2** reference to **null** and setting the reference stored in the second slot of the array i.e. **allLineItems[1]** to **null**. You may not have seen this done before, so it is worth a mention. Any object can be destroyed as long as there are no references remaining to it. If this is the case, the so-called Java “**garbage collector**” will destroy the object and return the space it used up in RAM to the system.

Here we have set the only two references to the 2nd LineItem both to null and so the garbage collector destroys the object those references had been linked with.

The next piece of output just indicates what has happened here

Now destroying the second line item (the 4 pencils)...

Displaying the state of the second product ...

Product details are as follows: id: 2 name: Pencil description: This is a pencil

The point being proved here is simply that, even though the 2nd LineItem object no longer exists, the Product that had been associated with it i.e. the “pencil” object, still “lives” and it is meaningful for it to still exist. This is exactly what should happen with aggregation - even though the owning (parent) object may be destroyed, the owned (child) object can survive and be used elsewhere.

The very last piece of output is just to display what the LineItem array now contains. As you can see, the 2nd object no longer displays, as required.

LineItem Details are as follows:

id 1

quantity:3

Product details are as follows: id: 4 name: Marker description: This is a black permanent marker

LineItem Details are as follows:

id 3

quantity:2

Product details are as follows: id: 3 name: Ruler description: This is a ruler

This example **demonstrates aggregation** because the Product object associated with a particular LineItem object **has intrinsic meaning** **independently** of the LineItem object itself and could be used meaningfully outside of the existence of the LineItem object it is related to. So here, when a LineItem object “dies”, its nested Product object does not.

**Comparing Composition and Aggregation**

It is important not to get too hung up about the difference between composition and aggregation, as it is rather subtle. The difference is clear in terms of their definitions alright, but when designing a system, it is often not clear when a relationship may be composition or aggregation. A good example of this is the relationship between a Car class and an Engine class. It all **depends on the context** of the application being created.

We can say that a Car object **has an** Engine object and that if a Car object is destroyed then so too is the nested Engine object it “owns”. This would be a **composition relationship** because the Engine object would not live beyond the lifetime of the Car object that owns it. This is the case when we have no independent use of the Engine object as a standalone entity.

However, if you have an application which implements a Car Shop Management System, you could feasibly have an Engine class, an Exhaust class, a Wheel class for the various parts that might make up a Car class (these would be similar to the idea of a Product class in the sample program as such). In this instance, we would actually want to be able to build Car objects which “own” the various parts mentioned, but we would also want these parts to exist independently of the Car object, so that they could live on beyond the lifetime of any individual Car object that might be created and subsequently destroyed. So, in this case, the relationship between a Car class and an Engine class would be an **aggregation relationship**.

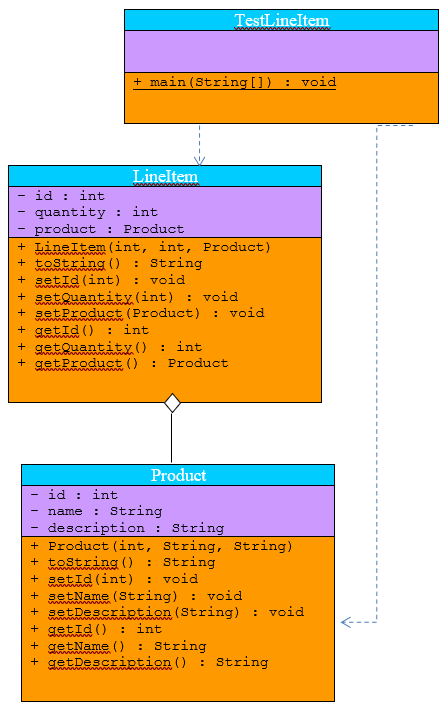
From a **Java coding perspective**, you will tend to **see the aggregation relationship much more often**, because it **provides for greater re-use** of classes compared with composition, where the relationship between classes is much “tighter” so re-use opportunities for the “owned” class are much fewer.

**View of Participating Classes (VOPC) Diagram**

You saw a VOPC diagram for the first time in the last lab sheet. In that instance, the relationship was between an instantiable class and a driver and, in these cases, the relationship is always a **dependency relationship**.

Now, we shall look at the VOPC diagrams for the situation where we have 3 classes involved in the application, as we had in our aggregation demo above:

* The **dashed lines** with the arrow-head at the end, connecting the driver class TestLineItem to the other two classes indicate a **dependency relationship** between the driver and those two classes, since it creates objects from both classes, and a change in the definition of either class could necessitate a change in the driver class which is using it. The arrow points to the classes that the other class depends on.
* The solid line with an **unfilled diamond** indicates an **aggregation relationship** between the LineItem and the Product classes. The class touched by the unfilled diamond is the “owning” class in the relationship, while the other class will be the “owned” or “nested” class.



Copy the **sampleprogram1** package to your **labsheet7** package now. This gives you your own copy of the three classes. To begin with, **compile and run** the application to see its output.

**Exercise 1**

Now create a new package called **exercise1** and copy the three files into this. **Refactor** when asked (this will automatically **change the package statements** inside the three files, so they reference the newly created package **exercise1** instead of sampleprogram1.

You will see, in the Product class, that the id attribute is set manually for an object created from the class, meaning we could set numerous Product objects to the same id value. Of course, in reality, we would like this id attribute to be **generated internally** and automatically for each Product object, so that it will be unique.

We looked at this possibility in **lab sheet 5**, where we used a **static** attribute called count for a BankAccount class in order to give each BankAccount object its own unique account number based on the value of this count attribute.

See if you can recall this now, in order to implement this functionality for the Product class. Feel free to refer back to lab sheet 5 if you are stuck with it. You will need to modify the Product constructor as part of this exercise. As a result, because the TestLineItem class creates Product objects (remember the **dependency relationship** it has with Product), it will need to be altered slightly too. Take it here that the value of the count attribute will be **initialised to 1000000**, so that the first Product object created will have an id value of 1000001 etc.

Your program will run similar to the following:

Displaying the state of all line-items ...

LineItem Details are as follows:

id 1

quantity:3

Product details are as follows: id: 1000001 name: Pen description: This is a red pen

LineItem Details are as follows:

id 2

quantity:4

Product details are as follows: id: 1000002 name: Pencil description: This is a pencil

LineItem Details are as follows:

id 3

quantity:2

Product details are as follows: id: 1000003 name: Ruler description: This is a ruler

Now changing product associated with first line-item to a marker...

Displaying the state of all line-items again...

LineItem Details are as follows:

id 1

quantity:3

Product details are as follows: id: 1000004 name: Marker description: This is a black permanent marker

LineItem Details are as follows:

id 2

quantity:4

Product details are as follows: id: 1000002 name: Pencil description: This is a pencil

LineItem Details are as follows:

id 3

quantity:2

Product details are as follows: id: 1000003 name: Ruler description: This is a ruler

Now destroying the second line item (the 4 pencils)...

Displaying the state of the second product ...

Product details are as follows: id: 1000002 name: Pencil description: This is a pencil

Displaying the state of all line-items one last time...

LineItem Details are as follows:

id 1

quantity:3

Product details are as follows: id: 1000004 name: Marker description: This is a black permanent marker

LineItem Details are as follows:

id 3

quantity:2

Product details are as follows: id: 1000003 name: Ruler description: This is a ruler

**Exercise 2**

The original VOPC diagram is given below, modify this now as necessary to reflect the changes made to the original application.

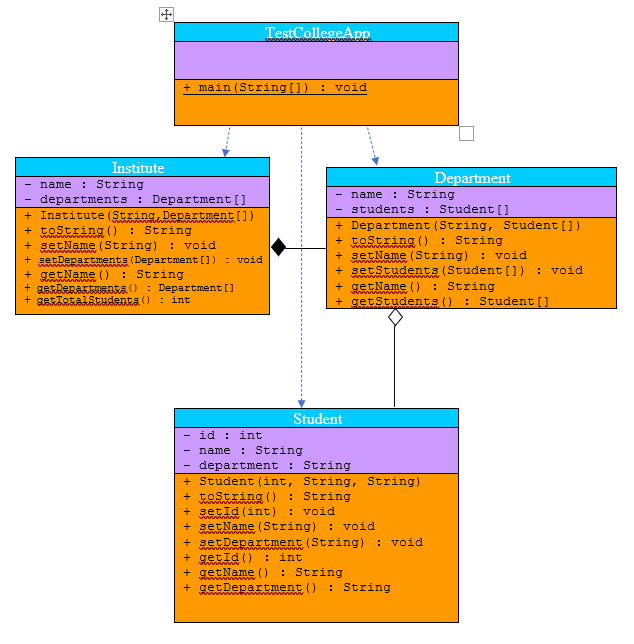
|  |
| --- |
| TestLineItem |
|  |
| + main(String[]) : void |

|  |
| --- |
| LineItem |
| - id : int  - quantity : int  - product : Product |
| + LineItem(int, int, Product)  + toString() : String  + setId(int) : void  + setQuantity(int) : void  + setProduct(Product) : void  + getId() : int  + getQuantity() : int  + getProduct() : Product |

|  |
| --- |
| Product |
| - id : int  - name : String  - description : String |
| + Product(int, String, String)  + toString() : String  + setId(int) : void  + setName(String) : void  + setDescription(String) : void  + getId() : int  + getName() : String  + getDescription() : String |

**Exercise 3**

A College Application is to be written based on the VOPC diagram you see below:



There will be 4 classes in total here as you can see. In this case, an Institute object can have many Department objects associated with it. Likewise, any given Department object can have many Student objects associated with it.

The only unusual part in the VOPC is the presence of the **getTotalStudents**() method within the Institute class diagram. The intention of this method is just to find the total number of Student objects present in the Institute. This will require looping through each of the Department objects and finding the number of Student objects each contains. Be sure to **check for null values** in both the Department and Student array objects here to prevent any NullPointerException occurring.

Your driver class **TestCollegeApp** should do the following:

* Create a set of four Student objects having the following values

|  |  |  |  |
| --- | --- | --- | --- |
| **Reference** | **Name** | **ID** | **Dept Name** |
| student1 | Jake | 154345 | Computing |
| student2 | Mary | 234532 | Creative Media |
| student3 | Tommy | 453726 | Computing |
| student4 | Peter | 623456 | Creative Media |

* Create an array of size 10 to store the Computing students and add the two Computing students listed above to that particular array
* Create an array of size 10 to store the Creative Media students and add the two Creative Media students listed above to that particular array
* Create a “Computing” Department object, passing into it a reference to the array storing the Computing students
* Create a “Creative Media” Department object, passing into it a reference to the array storing the Creative Media students
* Create an array of Department objects of size 5 and add the two Department objects to it
* Create an Institute object called “Institute of Technology, Tralee”, passing into it a reference to the array of Department objects
* Now the state of the Institute object should be displayed. In my case I get:

Name: Insitute of Technology, Tralee

Departments:

Name: Creative Media

List of Students:

Name: Mary ID: 234532 Department: Creative Media

Name: Peter ID: 623456 Department: Creative Media

Name: Computing

List of Students:

Name: Jake ID: 154345 Department: Computing

Name: Tommy ID: 453726 Department: Computing

* Next the getTotalStudents() method is called on the Institute object to report the number of students in the Institute. I get the following output here:

Total students in institute: 4

* The next part is tricky! We now wish to move Jake, who is currently in the Computing Dept, into the Creative Media dept. To do this, we first check to see if the two departments actually exist within the array of Department objects. If they do, then we note their subscript numbers. (you will have **more null testing** in this part)
* If both departments exist, then we retrieve the array of students associated with the Computing dept via the subscript linked the Computing dept noted earlier
* We then loop through this array looking to see whether the current Student object being examined has an ID that matches Jake’s ID, which is 154345.
* If we find a match, we issue a message “Found Jake!” and then set the name of the department for this Student object to “Creative Media”, then set the 3rd slot of the array containing the Creative Media students to this student object (just **hard code** the number 2 here to refer to the 3rd slot) and finally set the slot in the original array which was storing a reference to this Student object to null, to destroy the objects link to the array of Computing students.
* Even though the ID above does exist, you should put in code to issue the message “Couldn’t find Jake”, if a different ID value is used instead.
* Following this, the state of the Institute object should again be displayed, to see whether Jake is now in the Creative Media dept (also put in an invalid ID to ensure that the Institute object state remains the same in that case).

My output for this part looks as follows (with valid ID for Jake)

Jake now moving from the Computing dept to the Creative Media dept

Found Jake!

Name: Insitute of Technology, Tralee

Departments:

Name: Creative Media

List of Students:

Name: Mary ID: 234532 Department: Creative Media

Name: Peter ID: 623456 Department: Creative Media

Name: Jake ID: 154345 Department: Creative Media

Name: Computing

List of Students:

Name: Tommy ID: 453726 Department: Computing

As always code your classes to **maximise** **software re-use**

**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.

**Aggregation** – this is very similar to **composition** and is one of the most important OO features. It represents a **“has-a” relationship** between two entities where there is a “whole” which is made up of several “parts” but the **“parts” can exist independently of the “whole”.** A classic example in Java would be where you have a Department class and it is made up of a set of objects from a Student class. The key thing here is that the students **can exist independently** of the department. If the Department object is destroyed, the set of students that were associated with it can live on. In UML the aggregation relationship is denoted by a **solid line** with an **unfilled diamond** connecting the two related classes with the diamond next to the “owning” class.

**Garbage Collector** – Java has a built-in garbage collector that automatically **destroys** objects when they are no longer “pointed” to by any object references. Once destroyed, the resources used by the object are returned to the system

**null reference** – if the value of an object reference is null, then it means that the reference is currently not associated with any object. In an application, if an object is only associated with a single reference, and that reference is then set to null, then the object will be **garbage collected** and destroyed. It is important to check for the possibility of an object reference being null, to prevent a possible **NullPointerException** occurring.