**Institute of Technology Tralee**

**Computing Department**

**Object Oriented Programming**

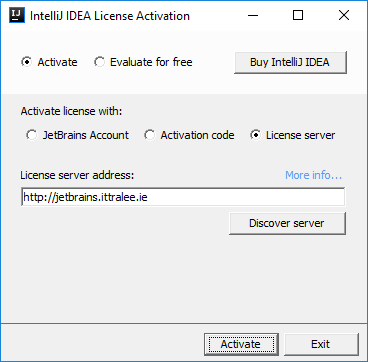
**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Practical 6 – UML, Composition and Arrays of Objects**

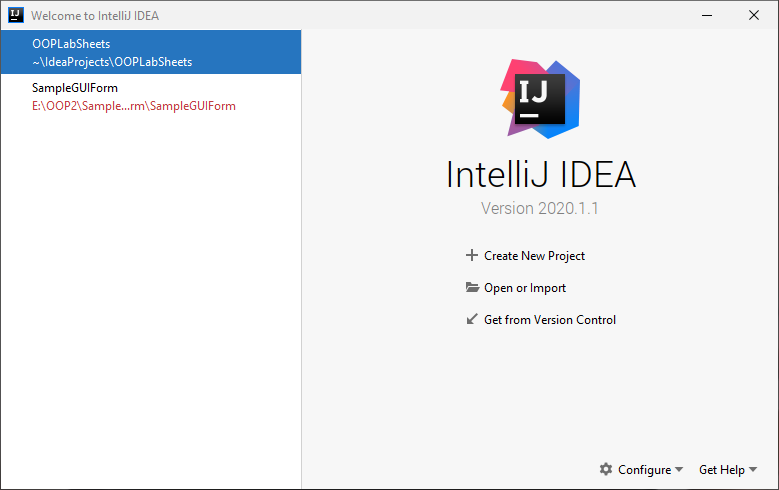
You have covered a lot of ground in the last three lab sheets and hopefully you are adapting well to this new style of programming. It does take time to become accustomed to OOP, so just keep working at it and it will come more naturally to you. You will find that many of the lab sheets refer to the same concepts over and over again, we will just be adding new bits onto it as the weeks pass. This time you will be introduced to the OO **documentation standard** called the **Unified Modeling Language** (UML). You will also discover yet another OO concept, that of **composition**.

**Getting into IntelliJ**

Launch **IntelliJ IDEA**. If you see the “License Activation” window just select **License Server** 🡪 **Discover Server** 🡪 **Activate**

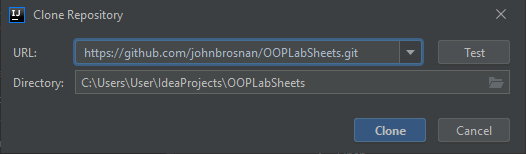


When completed, the IntelliJ **Integrated Development Environment** (**IDE**) should launch for you, after a few seconds. Once this happens, you are ready for coding!



To begin with above, you will see options to “Create New Project”, “Open or Import” etc. As you were introduced to VCS already, and since you should really have the latest version of your **OOPLabSheets** project “pushed” to GitHub, Click “**Check out 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).

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



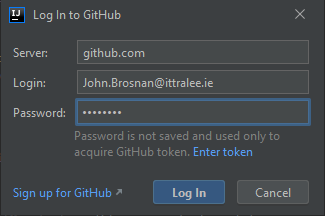
Recall that you **may need to rename** the local repository to something else if you get an error message at this point (I suggest a date e.g. OOPLabSheets08-10-20).

**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.

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



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

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

**Setting up your Folder Structure**

As you know by now, my preferred approach is to create a new folder for each lab sheet. In IntelliJ this can be done by adding a new **package** to the project. The package will be given an appropriate name, I will call it **labsheet6** 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 **labsheet6** 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 **labsheet6**. 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 new sub-packages to it as the lab goes on.

**The Unified Modelling Language (UML)**

**UML** is the **most widely used graphical representation scheme for modeling object-oriented systems**. Many software system designers use the language (in the form of various kinds of diagrams) to model their systems.

Many programmers starting off just sit down and type code. This is fine if the problem domain is very small and you can work out the logic in your head without too much trouble. This depends on how good you are at working problems out in your head, of course 😊 However, this system normally only works for programs that are relatively simple in terms of the **algorithm** used and are normally no more than 100 lines of code or so.

Then there are some problems that are **a little more complex**, either in terms of the number of tasks to be performed by the program or just in terms of the complexity of the algorithms involved. For these kinds of problems, it is very useful to write a **pseudocode** solution to the problem **before even going near the code editor**. We have applied this technique in last year’s modules in the problem-solving section, and hopefully some of you have also found it to be useful for some of the trickier problems so far this semester. Still, the pseudocode approach is normally limited to a few hundred lines of code at most, meaning it is perfectly adequate for everything we did in first year.

Then there are the **complex problems** that typically involve a large number of tasks and possibly several complex algorithms. In these cases, the best approach to take is to follow a detailed process for **analysing** your projects requirements i.e. ***what* the system is supposed to do** and then developing a **design** that satisfies them i.e. **deciding *how* the system should do it**. Ideally, this **should be done before writing a single line of code**. If this process involves analysing and designing the system from an object-oriented point of view, it is called an **Object-Oriented Analysis and Design** (**OOAD**) process.

Experience has shown that this process **can save valuable time, energy and money**, preventing projects from “falling apart” half-way through the coding stage because of a badly planned (or just no) design.

**UML is the most commonly used technique today for communicating the results of any OOAD process** and you will see and use it many times in the next few years to document programs and projects. In fact, you will take a module called **Object-Oriented Analysis and Design** in year 3!

**UML Class Diagrams**

When it comes to developing OO applications, the **approach should be to perform analysis on the system you are trying to develop and then to design it**.

Remember that in OO applications, the **most basic building-block in the system is the class**. Therefore, much of the effort in the OOAD process will involve identifying the classes required by the system and then deciding what they will contain in terms of attributes and methods (and possibly inner classes as we seen for GUIs).

A class and its contents can be described graphically in UML notation. For example, the UML **class diagram** of a very basic instantiable Person class might look as follows:

|  |
| --- |
| Person |
| - firstName : String  - lastName : String |
| + Person()  + Person(String, String)  + toString() : String  + setName(String,String) : void  + setFirstName(String) : void  + setLastName(String) : void  + getFirstName() : String  + getLastName() : String |

This diagram can be broken down as follows:

● The top box in the class diagram contains the **name** of the class.

● The middle box contains the **attributes** of the class and their types. The **minus sign** in front of the attributes indicate that they are **private**.

● The bottom box contains the **method names**, **argument types and return type** (after the **colon**). The **plus sign** in front of the methods indicate that they are all **public**.

And that is pretty much all there is to documenting a class diagram using UML! There are a few other little bits but we will introduce them as needed.

The **big advantage** of the class diagram is that, like pseudocode, it is possible to inspect it without having to worry about programming language syntax and specifics. Also, because the notation is **international**, it applies world-wide and so is completely **portable**. Indeed, tools exist that even allow you to take a class diagram as indicated above and have it translated directly into the OO language desired e.g. Java, C++ etc. in order to create “**skeleton classes**” – the programmer then fills in the missing bits, the method bodies (but also the logic for using the class and integrating it within the system as a whole, so there is still plenty to be done!)

**Your first Java CA will involve being given specifications of some instantiable classes using UML class diagrams. You will then try to convert the class diagrams to actual Java code to create your instantiable classes and then write a driver class to test out the functionality of your instantiable classes.**

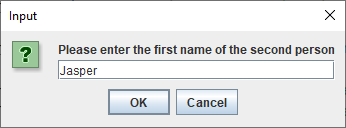
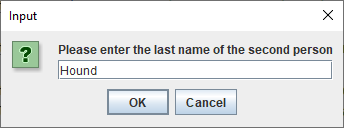
**Exercise 1**

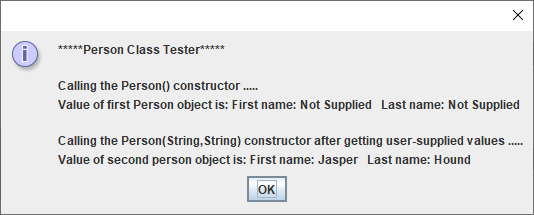
Within the **labsheet6** package create another package called **exercise1** to store the classes for this exercise.

Try to convert the UML class diagram given above to an instantiable Java class called Person. The no-argument constructor should simply set the values of the attributes both to “Not Supplied”. You should code the Person class so that it **maximises software reuse**.

Once you have the Person class written, test it out fully using a **minimalistic driver** class called TestPerson

The driver program will produce output similar to the following:



**UML Class Diagram for the TestPerson Driver Class**

As normal, you needed to create a driver in the last exercise to test out fully the functionality contained within the Person class.

This driver class TestPerson, can also be represented using a UML class diagram and it would look as follows:

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

Because there is only one method defined in the driver class, and there are no attributes at all defined, the UML diagram looks very small indeed here.

Note that, because the main() method is defined **static**, it is **underlined** in a UML diagram to distinguish it from non-static (instance) methods.

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

Another category of UML diagram is the **VOPC diagram**. This diagram is simply a graphical representation which illustrates the **set of class diagrams** making up an application or project, but the VOPC diagram shows the **relationships** between the various classes using various **UML** **connectors**. Normally only **user-defined classes** appear within such diagrams, with any reference to Java API classes omitted completely.

To illustrate an example, we look no further than our previous exercise which included the Person and TestPerson class.

The VOPC diagram in this case would look as follows:

|  |
| --- |
| Person |
| - firstName : String  - lastName : String |
| + Person()  + Person(String, String)  + toString() : String  + setName(String,String) : void  + setFirstName(String) : void  + setLastName(String) : void  + getFirstName() : String  + getLastName() : String |

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

The **nature of the connector (line and arrowhead)** here tell us something about the **relationship** between the two classes. The dotted line, with the open arrowhead here simply illustrates that, in this case, a “**uses**”/“**dependency**” relationship exists between the classes. The TestPerson class **uses** or **depends on** the Person class in order to create some Person objects (and then test them by displaying some output). This is why the **arrow is pointing towards the Person class**, as it is the one being “used” here.

A **dependency** **relationship** exists between two classes when **changes in the definition of one of the classes may necessitate changes in the other** class (but not vice versa). So if the definition of the Person class were to change, we might have to change the code in the TestPerson class as a result in order to make it work again.

There are several **other kinds of relationships** that can exist between classes, and they will be introduced as we come across them. Up until now though, the VOPC diagrams for all OO applications you have created would resemble the one illustrated above, since all the applications have just involved an **instantiable class** along with a **driver class** to test it out.

**Exercise 2**

Within IntelliJ, you should now open the files **BankAccount.java** and **TestBankAccount.java** that you created for **Exercise 5** in **labsheet 5**.

Use the definitions of the two classes now to **create a VOPC diagram** for the application.

**Composition in Java**

**Composition** is a very important OO feature. It represents a **“has-a” relationship** between two entities where there is a “whole” which is made up of several “parts” and the **“parts” cannot exist independently of the “whole”.** A classic example in Java would be where you have a House class and it is made up of several rooms, represented by a Room class. The key thing here is that the rooms cannot exist independently of the house, if the house “object” is destroyed then so too are the room “objects” it contains. Also, in such a composition relationship, the **“nested” or “owned” object** can only be part of one container at a time.

The following sample program **demonstrates composition** where one of the attributes of the class is an array of String

**Instantiable class Animal**

package labsheet6.sampleprogram1;  
  
*//Animal.java  
/\*A simple instantiable class definition for an animal which also demonstrates  
the OO concept of composition\*/*import java.util.Arrays;  
  
public class Animal {  
  
 *//true encapsulation ensured as all 4 attributes are private* private String type;  
 private String continents[]; *//a nested object within the Animal class* private double weight;  
 private int age;  
  
 *//the no-arg constructor indirectly accessing the 4 attributes via 4-arg constructor* public Animal() {  
 this("No type specified",null,0.0,0);  
 }  
  
 *//the 4-arg constructor indirectly accessing the 4 attributes via mutators* public Animal(String type, String continents[], double weight, int age) {  
 setType(type);  
 setContinents(continents);  
 setWeight(weight);  
 setAge(age);  
 }  
  
   
 *//the 4 accessor methods* public String getType() {  
 return type;  
 }  
  
 public String[] getContinents() {

*//return continents;*  
 return Arrays.*copyOf*(continents,continents.length);  
 }

public double getWeight() {  
 return weight;  
 }

public int getAge() {  
 return age;  
 }

*//the 4 mutator methods* public void setType (String type) {  
 this.type = type;  
 }  
  
 public void setContinents (String continents[]) {   
 *//this.continents=continents; //potential breaking of encapsulation* if(continents!=null)  
 this.continents = Arrays.*copyOf*(continents,continents.length);  
 }  
   
 public void setWeight(double weight) {  
 this.weight = weight;  
 }  
   
 public void setAge(int age) {  
 this.age = age;  
 }  
   
 *//displaying the state of the objects attributes indirectly via the accessors* public String toString() {  
 String str = "Animal Type: " + getType() + "\nAnimal Weight: " + getWeight() +   
 "\nAnimal Age: " + getAge() + "\nAnimal Continents: ";  
   
 if(continents==null) *//i.e. no array object exists=>no continents either* {  
 str+="No continents specified";  
 return str;  
 }  
   
 */\*if we get to here then there must have been an array object created  
 so traverse it using a loop and "join" its contents to the existing string object\*/* for(int i=0;i< getContinents().length;i++)   
 str+=continents[i] + " ";  
 return str;   
 }  
}

**Analysis of the Animal Class:**

● The Animal class has **four attributes** representing the *type* of the animal, an **array** which will hold the *continents* on which the animal is found, the *weight* of the animal and its *age*. It’s the first time we have seen an instantiable class that has an array as an attribute. You will recall that in Java **arrays are objects**, so when we create an Animal object within the driver program later, there will be a **“nested” object** within that Animal object.

● The Animal **no-argument constructor** makes a call using the this reference to the four-argument constructor, to initialise an Animal object with the values “No type specified”, **null** (for the array) and zeroes for the weight and age. The value null indicates that the continents attribute currently **“points” to nothing**. Notice that we couldn’t have put “No continents specified” here because that is a String and the argument expected is an **array of String**. However, when we display the state of an Animal object later in the driver, we want to display the text “No continents specified” for the state of the continents attribute, should it have a value of null. You will see we have to put some additional code into the **toString**() method to achieve this.

● The Animal **4-argument constructor** makes a call to the 4 mutators to initialise an Animal object with a set of user-supplied values, to be provided by the user in the driver class’ main(). It looks pretty normal and no different to any multi-argument constructor you have created to date

● The four accessors of the Animal class appear next and all are normal looking except for the **getContinents**() accessor.

public String[] getContinents() {

//return continents; //potential breaking of encapsulation  
 return Arrays.*copyOf*(continents, continents.length);  
}

What is happening here is not at all obvious and needs a little explanation. Because the continents attribute is an **array of object reference** we must deal with it carefully in order to avoid breaking the **encapsulation** of an Animal object. We **want to ensure always that the state of an object cannot be directly tampered with** outside of the class that defines it. For “regular” attributes such as int, float, char, double etc. it is enough to make them private and all will be well, since they can only be changed by using the mutators associated with those attributes.

Imagine that we did write the getContinents() method in the usual way i.e.

public String[] getContinents() {  
 return continents;  
}

this would actually compile just fine, but note that we would be now be returning a reference to the continents object nested within the Animal object.

So, within a driver program we could do the following

String lionContinents[] = {"Africa","Asia"}; *//initialise an array of strings*System.*out*.println("\n\n\ntesting the 4-argument constructor....");  
  
animal1 = new Animal("Lion",lionContinents,200,40);  
  
System.*out*.println(animal1);  
  
lionContinents = animal1.getContinents(); //this is where we would run into trouble!  
  
lionContinents[0] = "Europe"; //we can just change the first slot of the continents array for animal1

//directly now, breaking the encapsulation of the object  
  
System.*out*.println(“\n”+animal1);

Producing the following output:

testing the 4-argument constructor....

Animal Type: Lion

Animal Weight: 200.0

Animal Age: 40

Animal Continents: Africa Asia

Animal Type: Lion

Animal Weight: 200.0

Animal Age: 40

Animal Continents: Europe Asia

So you can see it would be possible to change the value of the continents attribute of the animal1 object by direct assignment outside of the class, which **violates the principle of encapsulation**.

Our solution to this problem is to do the following instead

public String[] getContinents() {  
 return Arrays.*copyOf*(continents, continents.length);  
}

This code solves the previous problem perfectly because it will **create a brand new array** object, **copy** all the values in the current continents array into this new object, and then **return a reference to this new object**. This means that the nested continents object will no longer be directly accessible from outside the Animal class and we have upheld the principle of encapsulation.

You will recognise the **Arrays** class here, as we used it last year. The method **copyOf**() will be new to you though. Is this a **static** or an **instance** method?

● The four **mutator methods** come next and, like the accessors, three of these are “normal”, but things are different for the continents attribute.

public void setContinents (String continents[]) {   
 *//this.continents=continents; //potential breaking of encapsulation* if(continents!=null)  
 this.continents = Arrays.*copyOf*(continents,continents.length);  
 }

In order to **prevent violation of the encapsulation** **principle**, we need to give the Animal object its own copy of the continents array that originates in the main() of the driver. This means that, if the continents array created in main() is changed later, then it will have no effect whatsoever on the continents array linked to the animal object.

Before this, we need to check to **make sure that the continents reference is actually pointing to some String array**. If it is not, then its value will be null and then the reference to **continents.length** would **crash the program**.

You will recall the **null reference** from last year when we were trying to manipulate arrays via GUIs and you will see it again many times in the future.

● Finally, the **toString**() method is coded as follows, as you can see it is a little more intricate that the toString() methods you have written to date and, of course, the culprit is the continents attribute!

public String toString() {  
 String str = "Animal Type: " + getType() + "\nAnimal Weight: " + getWeight() +   
 "\nAnimal Age: " + getAge() + "\nAnimal Continents: ";  
   
 if(continents==null) *//i.e. no array object exists=>no continents either* {  
 str+="No continents specified";  
 return str;  
 }  
   
 */\*if we get to here then there must have been an array object created  
 so traverse it using a loop and "join" its contents to the existing string object\*/* for(int i=0;i< getContinents().length;i++)   
 str+=continents[i] + " ";

return str;   
 }

Here a local String variable is created and values for the type, weight and age attributes are tagged onto it. Next there is a check to see whether the value of the continents attribute is null. If it is, then we just tag on the text “No continents specified” onto the String variable, otherwise we **loop** through the contents of the continents array and tag on each of the values in this array to the variable str. This is the first time you will have seen a loop used within a toString() method of a class.

**Driver Class TestAnimal**

package labsheet6.exercise3;  
  
*//TestAnimal.java  
/\*A driver (test) class that contains the main() method for   
testing the functionality of the Animal class\*/*public class TestAnimal {  
 public static void main( String args[]) {  
 System.*out*.println("testing the no-argument constructor....");  
 Animal animal1 = new Animal(); *//tests no-arg constructor* System.*out*.println(animal1); *//tests toString() and the accessor methods* String lionContinents[] = {"Africa","Asia"}; *//initialise an array of strings  
   
 //tests 4-arg constructor and the 4 mutator methods* System.*out*.println("\n\n\ntesting the 4-argument constructor....");  
 animal1 = new Animal("Lion",lionContinents,200,40);  
 System.*out*.println(animal1);  
 }  
}

**Analysis of the TestAnimal Class:**

● A first Animal object is created via a call to the **no-argument constructor**. The state of this object is then displayed on the console via the call to toString(), which also tests out the functionality of the accessors.

The output displayed to this point is:

testing the no-argument constructor....

Animal Type: No type specified

Animal Weight: 0.0

Animal Age: 0

Animal Continents: No continents specified

Since the value of the continents attribute when the no-argument constructor is called will be null, this subsequently results in the text “No continents specified” being displayed via toString().

● A String array “called” lionContinents is **initialised** with two values. Then a second Animal object is created via the 4-argument constructor, which tests the 4 mutators also. The **reference to the array** is passed as an argument to the constructor. This means that the second Animal object is now linked with an array object, meaning that **composition is taking place**, and the **array object is “nested” within the Animal object**.

● Finally, the state of the second Animal object is displayed on the console.

The output displayed in this case is:

testing the 4-argument constructor....

Animal Type: Lion

Animal Weight: 200.0

Animal Age: 40

Animal Continents: Africa Asia

This example **demonstrates composition** because the array of continents associated with a particular Animal object has no intrinsic meaning independently of the object itself and could not be used meaningfully outside of the existence of the Animal object it is related to. So here, when an Animal object “dies”, so too does its nested continents array object.

Copy the **sampleprogram1** package to your **labsheet6** package now. This gives you your own copy of the Animal and TestAnimal classes. To begin with, **compile and run** the application to see its output.

**Exercise 3**

Now create a new package called **exercise3** and copy the two files into this. **Refactor** when asked (this will automatically **change the package statements** inside the two files, so they reference the newly created package exercise3 instead of sampleprogram1.

The setContinents() method here was written as follows:

public void setContinents (String continents[]) {   
 *//this.continents=continents; //potential breaking of encapsulation* if(continents!=null)  
 this.continents = Arrays.*copyOf*(continents,continents.length);  
 }

the **copyOf**() method here creates a brand new String array, whose size will be the same as the array created in main(), which gets passed into the method as an argument. Once the array is created, then the contents of the array created in main() get copied, slot by slot, into the newly created array. Then when the copying process is completed, the copyOf() method finally assigns the newly created array to the continents attribute. So you can see that the copyOf() method is doing a fair amount of work “under the hood” and is sparing us all this effort.

However, it is good to get the hands dirty every now and again 😊The question is, can you write the code that copyOf() is sparing us, from first principles? Give it a go now by commenting out that line of code and replacing it with your own version.

You should also add the following two lines of code to the end of main() in the driver, to prove that the principle of encapsulation is still being preserved.

lionContinents[0] = "Europe";  
System.*out*.println("\n" + animal1);

Once you have it written, compile and run the application again, your output should be **exactly the same** as before.

**Arrays of Objects**

We introduced the concept of arrays in SP2 and you saw that arrays can be used as a structure to allow us to **group together a set of related items**. So we had examples where you seen arrays of int, arrays of float/double, arrays of char and even arrays of String. As you will know, in Java a **String is an object**, so technically you have seen arrays of objects being used before. But then again, a String is a lot like a “regular” variable since it actually **stores just a single piece of data** (well technically it stores a few other bits of information, but we aren’t normally too interested in those values).

What we are talking about here really is creating an array that is made up of objects from either a user-defined instantiable class, such as our Animal class, or maybe from some instantiable Java API class (other than String) such as JButton or JLabel for example.

So we will just work with the Animal class that we looked at earlier to demonstrate how we can create an array of Animal objects.

We can keep the Animal class exactly as it is right now for this demonstration (except that I am copying the original two files to a new package called **sampleprogram2**) we will just modify the driver class a little as follows:

package labsheet6.sampleprogram2;  
  
*//TestAnimal.java  
/\*A driver (test) class that contains the main() method for   
testing the functionality of the Animal class. This version of the driver is  
also to demonstrate the creation of an array of Animal objects\*/*public class TestAnimal {  
 public static void main( String args[]) {  
  
 Animal allAnimals[] = new Animal[10];  
  
 System.*out*.println("Calling the no-argument constructor to create the first animal....");  
 Animal animal1 = new Animal(); *//tests no-arg constructor* String lionContinents[] = {"Africa","Asia"}; *//initialise an array of strings  
  
 //tests 4-arg constructor and the 4 mutator methods* System.*out*.println("\nCalling the 4-argument constructor to create a lion object....");  
 Animal animal2 = new Animal("Lion",lionContinents,200,40);  
  
 System.*out*.println("\nCalling the 4-argument constructor to create a rat object....");  
 Animal animal3 = new Animal("Rat",new String[]{"Europe","America","Africa","Asia"},0.2,5);  
  
 System.*out*.println("\nCalling the 4-argument constructor to create a polar bear object....");  
 Animal animal4 = new Animal("Polar Bear",new String[]{"Antarctica"},450,15);  
  
 allAnimals[0] = animal1;  
 allAnimals[1] = animal2;  
 allAnimals[2] = animal3;  
 allAnimals[3] = animal4;  
  
 System.*out*.println("\n\nThe details of all the animals in the Animal array are:\n\n");  
  
 for(int i=0;i<allAnimals.length;i++)  
 System.*out*.println(allAnimals[i]+"\n\n");  
  
 }  
}

**Analysis of class TestAnimal**

● The first line of code within the main() creates the array of Animal objects of size 10. The syntax is exactly as you might expect. In general, to create an array from any class the syntax would be:

**Classname arrayName = new Classname[arraySize];**

At this point an array containing 10 slots exists and each slot contains the value **null**. This makes sense because none of slots are referencing any Animal objects just yet.

● The next few lines of code all just involve displaying some output to the console to inform the user what is happening and also the act of creating the individual Animal objects, one by one. In total, we end up creating 4 Animal objects here.

One thing worth mentioning, that you may not have seen before, is the following type of coding, where the **array is created and passed in to the constructor** as an argument in the one go, without using a reference, as follows:

Animal animal3 = new Animal("Rat", new String[]{"Europe","America","Africa","Asia"}, 0.2, 5);

So you can see it is **possible to just create a new String array, and initialise it, within the Animal constructor** call. In a way, it makes perfect sense to take this this approach here because the array of continents associated with a particular animal are meant to be “bound” to that animal, and they shouldn’t really exist independently of it. The composite object (the Animal object) is meant to **have complete responsibility for the existence and storage** of the composed object (the array of continents). If we take the approach of using a reference, as we did for the lion object, then the array of continents does have an existence independently of the lion object itself and can be altered via that reference (even though we modified the mutator with the call to **Arrays.copyOf()** so that the lion, and all the other Animal objects created, get their own copy of those values internally anyway so they cannot be tampered with from the outside).

● Once the Animal objects are all created, then we add them one by one to the array, using the usual syntax that you will have seen last year. The only difference now is that we are **setting each slot of the array to an object reference** rather than a specific numeric or text value.

● Once we have finished adding values to the array, a **for loop** is used to iterate through it and display its contents, in the same way we would have done last year to display the values within an array of int for example.

The program produces the following output:

Calling the no-argument constructor to create the first animal....

Calling the 4-argument constructor to create a lion object....

Calling the 4-argument constructor to create a rat object....

Calling the 4-argument constructor to create a polar bear object....

The details of all the animals in the Animal array are:

Animal Type: No type specified

Animal Weight: 0.0

Animal Age: 0

Animal Continents: No continents specified

Animal Type: Lion

Animal Weight: 200.0

Animal Age: 40

Animal Continents: Africa Asia

Animal Type: Rat

Animal Weight: 0.2

Animal Age: 5

Animal Continents: Europe America Africa Asia

Animal Type: Polar Bear

Animal Weight: 450.0

Animal Age: 15

Animal Continents: Antarctica

null

null

null

null

null

null

You can see that the values displayed for the **last six slots of the array are all null**, as expected, since no Animal objects have been assigned to those slots.

Copy the **sampleprogram2** package to your **labsheet6** package now. This gives you your own copy of the Animal and the altered TestAnimal classes. To begin with, **compile and run** the application to see its output.

**Exercise 4**

Now copy the two files to a **new package** called **exercise4**.

* Modify the driver so that it creates another Animal object with the following details:

Type: Anaconda Continents: Asia America Weight: 100 Age: 8

Once created, the new object should be added to the allAnimals array.

* At the moment, the first slot of the array contains just a “default” animal with no specifics. Alter this element of the array so that the animal object will be given the type “Gorilla” and the weight 250. The other two values can remain at their defaults. Note that you will be calling the two relevant **mutators** here.
* Currently, if the value zero is supplied for the weight or age of an Animal object then those values will appear on the output. Modify the **toString**() within the Animal class now so that, if the user should supplied a zero value for either of these attributes, then the output will be “No weight specified” and “No age specified” respectively.
* Modify the driver class now so that it will no longer display values for any slots in the allAnimals array that are null.

After all these modifications, your program should now run as follows:

Calling the no-argument constructor to create the first animal....

Details of first animal are:

Animal Type: No type specified

Animal Weight: No weight specified

Animal Age: No age specified

Animal Continents: No continents specified

Calling the 4-argument constructor to create a lion object....

Calling the 4-argument constructor to create a rat object....

Calling the 4-argument constructor to create a polar bear object....

Calling the 4-argument constructor to create an anaconda object....

The details of all the animals in the Animal array are:

Animal Type: Gorilla

Animal Weight: 250.0

Animal Age: No age specified

Animal Continents: No continents specified

Animal Type: Lion

Animal Weight: 200.0

Animal Age: 40

Animal Continents: Africa Asia

Animal Type: Rat

Animal Weight: 0.2

Animal Age: 5

Animal Continents: Europe America Africa Asia

Animal Type: Polar Bear

Animal Weight: 450.0

Animal Age: 15

Animal Continents: Antarctica

Animal Type: Anaconda

Animal Weight: 100.0

Animal Age: 8

Animal Continents: Asia America

**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.

**Unified Modeling Language (UML)** – **UML** is the **most widely used graphical representation scheme for modeling object-oriented systems**. Many software system designers use the language (in the form of various kinds of diagrams) to model their systems and communicate the results of the OOAD process. The major advantages are that it is **not programming language specific**, so diagrams can be inspected without having to get bogged down in syntax details, and also it is an **international standard**, and so is **portable**.

**OO Analysis & Design (OOAD)** – The **analysis** of an OO system means determining exactly **what** the system is meant to be capable of doing. Once the analysis phase is completed, the next step is the **design** of the system, which means figuring out **how** it will do what it is meant to do. These stages should be completed before a single line of code is written. If done correctly, it should **save time, money and effort**.

**Class Diagram** – this is an example of a **UML diagram**. It **models a class** entity, presenting it in a graphical form having three sections, the **class name** at the top, the **attributes** of the class in the middle and the **methods** of the class at the bottom.

**View of Participating Classes** (**VOPC**) **Diagram** – this is another example of a **UML diagram**. It is a graphical representation which illustrates the **set of class diagrams** making up an application or project, but the VOPC diagram shows the **relationships** between the various classes using various **connectors**. Normally only **user-defined classes** appear within such diagrams, with any reference to Java API classes omitted completely.

**Dependency** **relationship** – this type of relationship exists between two classes when **changes in the definition of one of the classes may necessitate changes in the other** class (but not vice versa). An example would be the relationship that exists between an instantiable class and its associated driver class.

**Composition** – this is a very important OO feature. It **represents a “has-a” relationship between two entities** where there is a “whole” which is made up of several “parts” and the **“parts” cannot exist independently of the “whole”.** A classic example in Java would be where you have a House class and it is made up of several rooms, represented by a Room class. The key thing here is that the rooms cannot exist independently of the house, if the house “object” is destroyed then so too are the room “objects” it contains. Also, in such a composition relationship, the **“nested” or “owned” object** can only be part of one container at a time.