Features of Object oriented Programming

Object Oriented Programming Language

Object orientation is the premise of Java. The pillars of OOP language is

1. Data encapsulation
2. Inheritance
3. Over-riding
4. Over-loading
5. Polymorphism

You will be surprised if I say you already have been introduced to first 3. Before we discuss further, you should understand packages and modules in Java. A group of classes which are similar in nature or work together are typically put in a package. For example, we have been creating a lot of examples. Let's put them all in a package called com.myob.examples. The nomenclature for the packages started with the reverse of the website. But not it has become more of a convention.

Let's now create a new package. Go to File->New->Package. Enter the package name as com.myob.examples (or anything you like). This will create a package. Copy all the java files you created into this package. It will ask if you want it to be refactored. Say yes and reorganise. You just created a package in Java. From Java 9 onwards, packages can be bundled within a module. Every module has its own module descriptor file which includes details about the packages in it and all of the dependencies. The module as a whole can be reused in one or more projects. Each module can use its own version of Java (as long as it is 9+). You can read more about module here. For this course you will just need to know about packages. You have used three packages so far. java.lang which is available by default. java.util, for the Scanner class and java.time for LocalDateTime.

Encapsulation

We saw earlier that the attributes and methods in a class can have different kind of accesses. This is an important aspect of OOP called Encapsulation. As a convention all the attributes of the class are kept as private. This is to encapsulate the attribute within the Object and allow access to it through setters and getters. Let's rewrite our Customer class.

```
public class Customer {

    private String firstName;
    private String lastName;
    private String email;
    private String accountNumber;
    private String typeOfAccount;
    private String addressForCommunication;
}
```

Now generate getters and setters to this class through IntelliJ, just we did before. Right click on the code, click on `Generate` and follow the prompt.

If these attributes were public, they could have been directly referred through the objects of Customer class. Say for example, firstName and secondName were public and we had `Customer c1 = new Customer();`, we could have accessed the firstName and secondName of the customer c1 as `c1.firstName` and `c1.lastName`. What is the point in having setters and getters? the point is to pre-process the values before accepting these. Eventually these classes will be used in full stack application to represent real objects. Like in this case, if the attributes were public they could be set to any valid String. Ideally we should validate to make sure it is all letters. **Activity:** Generate setters and getters for the attributes. And make changes in the setters to take only valid letters.

Overloading

We spoke about the constructor which constructs the object. What if I want to make it easy for the user of the class to create an object providing all the values at the outset instead of creating an empty object and setting them. In that case I will have a constructor which will take these values as parameter.

```
    public Customer(String firstName, String lastName, String email, String accountNumber, String ty
peOfAccount, String addressForCommunication) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
        this.accountNumber = accountNumber;
        this.typeOfAccount = typeOfAccount;
        this.addressForCommunication = addressForCommunication;
    }
```

But the moment you explicitly define a constructor, the constructor which Java implicitly creates is no longer available. As the creator of the class, you can decide whether the default constructor needs to be provided in addition to this or not. Let's provide the default constructor for the Customer class in addition to the other constructor above.

```
    public Customer(String firstName) {
        System.out.println("This is an empty constructor. Initialize the attributes");
    }
```

Now if you see, we have two constructors. Depending on what parameters are passed, the respective constructor is used to construct the object. What we just did is, over loaded the constructor.

We can also overload methods. We can write two methods with the same name having different parameters. Let's look at a few examples of method over loading.

```
public class BuiltInOverloadingExample {
    public static void main(String[] args) {
        String s = "Hello World";
        System.out.println(s.substring(3));
        System.out.println(s.substring(3,6));
    }
```

```
    }
}
```

If we look at this example above, substring which takes one int parameter and the substring which takes 2 int parameters are separate implementation. But it is overloaded using the same name for ease of usability. Let's now create a class with our own overloaded method.

```
public class MethodOverloadingExample {

    public void add(int i, int j) {
        System.out.println(i+j);
    }

    public void add(String i, String j) {
        System.out.println(i.concat(j));
    }

    public static void main(String s[]) {
        MethodOverloadingExample m1 = new MethodOverloadingExample();
        m1.add(3,5);
        m1.add("Hello","World");
    }
}
```

Depending on the parameters passed different add methods will be called. The idea is to help the users of the class easily comprehend expected behaviour of the methods.

Both constructor overloading and method overloading are important pillars of OOP.

Inheritance

As the name indicates something inherits something. We saw in initial stages of learning Java, that one of the ways a class can be used is through inheritance. One class can inherits attributes and methods from another class. The class from which another class inherits is called a superclass. The which inherits the super class is called subclass. The subclass inherits the methods and the attributes of the superclass depending on how they have been encapsulated and exposed in the super class. All the classes in Java implicitly inherit Object class. Let's create a superclass and two subclasses and observe the behaviour of inheritance. To create a subclass of a class we use the `extends` keyword.

SuperClass

```
package com.myob.examples;

public class SuperClass {
    //Has private access within class
    private void method1() {
        System.out.println("This is method1");
    }

    //Has default access. Available for only for subclass within same package
    // and also objects in same package
```

```
    void method2() {
        System.out.println("This is method2");
    }

    /*Has protected access. Available through subclass in same and different       package and objec
ts in same package*/

    protected void method3() {
        System.out.println("This is method3");
    }

    //Has public access. Available through all subclasses and instances
    public void method4() {
        System.out.println("This is method4");
    }
}
```

SubClass within the same package

```
package com.myob.examples;

package com.myob.examples;

public class SubClassInSamePackage extends SuperClass {

}
```

SubClass within a different package

```
package com.myob.anotherpkg;

import com.myob.examples.SuperClass;

public class SubClassInDiffPackage extends SuperClass {
}
```

Let see what methods are accessible, where. For this purpose, let's create classes within the same package and within a completely new package.

Instance of the subclasses and superclass in same package

```
package com.myob.examples;

import com.myob.anotherpkg.SubClassInDiffPackage;

public class CreatingSubclassInstance {
    public static void main(String[] args) {
        SubClassInSamePackage sc1 = new SubClassInSamePackage();
        sc1.method2();
        sc1.method3();
        sc1.method4();

        SubClassInDiffPackage sc2 = new SubClassInDiffPackage();
        sc2.method3();
        sc2.method4();
```

```
        sc2.method4();

        SuperClass superClass = new SuperClass();
        superClass.method2();
        superClass.method3();
        superClass.method4();
    }
}
```

Instance of the subclasses and superclass in different package

```
package com.myob.onemorepackage;

import com.myob.anotherpkg.SubClassInDiffPackage;
import com.myob.examples.SubClassInSamePackage;
import com.myob.examples.SuperClass;

public class CreatingSubClassInstanceInDiffPackage {
    public static void main(String[] args) {
        SuperClass superClass = new SuperClass();
        superClass.method4();

        SubClassInSamePackage sc1 = new SubClassInSamePackage();
        sc1.method4();

        SubClassInDiffPackage sc2 = new SubClassInDiffPackage();
        sc2.method4();
    }
}
```

public	Accessible in all subclasses and instances
default (when you specify nothing)	Accessible in all subclasses and instances, in same package
protected	Accessible in all subclasses but only instances, in same package
private	Accessible only within the class object

Over-riding

A subclass has the option to only keep some of the super class methods and if it wants to implement its own for the other methods. You would have noticed this when you implemented the toString method in Customer class. It automatically annotates the method implementation with a `@Overrides`. When you implement toString you are overriding the default implementation in the Object class which print the hashcode of the object.

Activity Override the methods in the subclasses you created above. 💡 Tip: Use the IntelliJ Generators. Make the method body print some distinctive message so that you know it is from the subclasses. We use these over-ridden methods for the next topic.

Polymorphism

Polymorphism is when the method takes different forms depending on the object it is invoked on. Before you understand this, you have to know that all the subclasses can be referred to with as the superclass

you understand this, you have to know that all the subclasses can be referred to with as the superclass type. I can have,

`SuperClass s1 = new SubClassInDiffPackage();` or

`SuperClass s1 = new SubClassInSamePackage()`.

Let's create different objects depending on what the user prefers and then invoke the over-ridden methods and observe the behavior.

```
package com.myob.onemorepackage;

import com.myob.examples.SuperClass;
import com.myob.examples.SubClassInSamePackage;
import com.myob.anotherpkg.SubClassInDiffPackage;

import java.util.Scanner;

public class CreatingDiffObjectsWithSuperRef {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Which object do you want to create?");
        System.out.println("Enter A for SuperClass");
        System.out.println("Enter B for SubClassInSamePackage");
        System.out.println("Enter C for SubClassInDiffPackage");
        String reply = scanner.nextLine();
        SuperClass sc = null;

        switch(reply) {
            case "A" :  sc = new SuperClass();
                        break;
            case "B" :  sc = new SubClassInSamePackage();
                        break;
            case "C" :  sc = new SubClassInDiffPackage();
                        break;
        }
        sc.method4();

    }
}
```

Depending on whether the choice was A, B or C, a different object would have been created. And the method call to method4 will be different depending on which object was created. A subclass object can be referred with super class reference. But only the methods which the subclass inherited from superclass can be invoked through the super class reference.