12 Object Oriented Programming II

The second pillar of the object oriented programming is Inheritance. Consider that we have a person class and now wish to make classes for students and teachers as well. All the students and teachers have properties like name and age which are not specific to them instead are general properties of the person class. If we include these properties in the student and teacher class as well, this would just be the repetition of code. Instead it would be better if we reuse the code of the person class and include more specific properties like marks in student class and subject to be taught in the teacher class. This reusability of code is achieved through inheritance.

Inheritance:

Inheritance is a mechanism in which one object acquires all the properties and behaviors of the parent object. The idea behind inheritance in java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.

Now let us make a person class and then use it to build our student class:

```
   public class Person {
   String name;
   int age;

   public void displayInfo() {
          System.out.println(this.name + " " + this.age);
   }
}
   public class Student extends Person {
   int marks;

   public void displayMarks() {
          System.out.println(this.marks);
   }
}
```

Here the Person class is the base class or the super class whereas the student class is the subclass and inherits the properties name, age and methods *displayInfo* from the Person class. The Student class also has its specific property of marks and a method *displayMarks*. Also note the use of the 'extends' keyword which specifies that the subclass extends or inherits the properties of the base class. Thus, we can use the data members and the methods of the Parent class and also have our own data members and methods.

Now suppose we wish that the *displayInfo* method of the Person class should not be used as such for the student class and must be changed a little to display the marks as well. This could be achieved by overriding the method of the Person class in the Student class. This is done as follows by using ***@Override*** above the method to be overridden. ***@Override*** just tells the compiler that the function is being updated and it is not compulsory to write it.

```
   public class Student extends Person {
   int marks;

   @Override
   public void displayInfo() {
          System.out.println(this.name + " " + this.age + " " + this.marks);
   }
}
   public class Client {

   public static void main(String[] args) {

          Student s = new Student();
          s.name = "Rahul";
          s.age = 15;
          s.marks = 80;
          s.displayInfo();
          // OUTPUT: Rahul 15 80
   }
}
```

Now when we call the *displayInfo* method on any student object the overridden method in the student class would be called.

Types of Inheritance:

There are various types of inheritance like:

Single inheritance: Here a single subclass inherits from a single base class. Like a Student class inherits from a Person class.

Multilevel inheritance: There is chain of classes inheriting from one another. Like a Person class inherits from a Mammal class which in turn inherits from the Animal class.

Multiple inheritance: Here a single subclass inherits from more than one parent classes. This type of inheritance is not supported in java by the use of classes. This can only be supported with the help of interfaces about which we will learn later. The reason for which it is not supported is that if there is a method by the same name present in both the parent classes and that is called on the object the JVM would not know which method to call for. So to avoid this confusion this type of inheritance is not supported in java.

We know that to refer to any instance of a class or any method of the current class, 'this' keyword is used. Now if that class extends from another class, an instance of the parent class is created implicitly whenever an instance of the subclass is created. The **instance of the parent class** is referred by the **'super'** keyword. Let us look at the uses of the super keyword:

1. The 'super' keyword can be used to refer immediate parent class instance variable.
2. The 'super' keyword can be used to invoke immediate parent class method.
3. 'super()' can be used to invoke immediate parent class constructor.

Polymorphism:

Now let us consider a little different situation from the present context. Suppose we want to write functions to add integers, decimal numbers and strings. For that we will need to write three different functions with different names. This means that for each new data type we will need to choose a different name for the function whose purpose is the same. This would quite mess up the things. Thus, to make our code maintainable, we will now learn about the third pillar of the objectoriented programming – POLYMORPHISM. Polymorphism is a way in which something behaves differently depending on its call. In other words, it is same name but different forms! In fact the ability to override the methods of the base class in the child class is also a form of polymorphism.

Let us learn about polymorphism and its types in more detail.Polymorphism in java is a concept by which we can perform a single action by different ways. There are two types of polymorphism in java:

1. Compile time polymorphism
2. Run time polymorphism

Compile Time Polymorphism:

The compile time polymorphism is implemented by **method overloading** in java.It means that there is more than one function with the same name in a class but with different signatures. Thus the functions with the same name can have different parameter types, number and even different order. Note that the return type is not the part of the function signature. Thus method cannot be overloaded by changing the return type of the function as then the compiler would not know which method to call for as the parameters are the same. Now let us look at an example to demonstrate the function overloading where the parameters can be of different type and can vary in number and order.

```
public int add(int a, int b) {
	return a + b;
}

// Change in the number of parameters
public int add(int a, int b, int c) {
	return a + b;
}

// Change in the type of parameters
public double add(double a, double b) {
	return a + b;
}
```

```
// Change in the order of parameters
public double add(double a, int b) {
	return a + b;
}
```

Here the name for all the functions is the same but the compiler chooses which function to call depending on the parameters passed.

Now suppose that we do not know how many parameters would be passed in the function call. Thus it would not be possible for us to write functions with all the possible parameters. Thus to solve this problem we can use 'varargs' (variable arguments). In the function parameters we put three dots after the data type in the function declaration. This specifies the use of varargs. Note that there can just be one vararg in a function and also this has to be the last parameter. Below we have an example to demonstrate the same.

Note that compile time polymorphism can also be implemented by the use of generics which will be discussed later. Also operator overloading is not allowed in java.

```
public static void display(String... values) {
	for (String s :values) {
		System.out.print(s + " ");
	}
}

public static void main(String[] args) {
	display(); // No output
	display("hello"); // hello
	display("hello", "world"); // hello world
}
```

Runtime Polymorphism:

Runtime polymorphism is the process in which the call to an overridden method is resolved at the run time rather than at the compile time. When an overridden method is called by a reference, java determines which version of that method to execute depending on the type of the instance it refers to. Now let us first understand method-overriding in more detail.

Method Overriding: When the derived class has a method same as that declared in the parent class, it is known as method overriding. Thus, for method overriding the function in both the base and the derived class should have the same signature. Also note that the function in the derived class cannot be more restricted than that of its parent class. This means that if we are overriding a public function from the parent class, it can't be private or protected in the derived class as this will throw an exception.

In general, all the virtual functions can be overridden in java. By default, all non-static functions are virtual functions in java. Only the ones marked with final keyword can't be over ridden and the private functions which can't be inherited are non-virtual.

Now after having learnt about method overriding let us learn how runtime polymorphism is implemented.

Let us consider a context where we have a parent class **P** and a child class derived from P called **C**.

Now while making an object there can be four combinations depending on the type of the instance and its reference. Let us understand each in detail:

'P obj = new P()' : Here the reference is of P type and its instance is also of type P. Thus we can call any method of the class P on 'obj'.

'P obj = new C()' : here the reference is of type P but its instance is of type C. Now let C class override a function 'fun()' of class P. Now if we call the method fun() on the object 'obj' of class P, since it refers tothe object of the sub-class C and the subclass overrides the parent class method, the subclass method is invoked at the run time. Now as the method invocation is decided by the JVM at the run time, this is known as runtime polymorphism.

'C obj = new P()' : Here we are making the reference of C class which is the subclass of P and making it point toan instance of type P. This is not allowed in java and it throws a compile time error. This is because there may be some data members and methods in the subclass which are absent in the parent class. Thus, if we make the reference of class C refer to instance of class P, we won't be able to access those data members and methods. Thus a compile time error is raised whenever we try to do so.

'C obj = new C()' : Here the reference is of C type and its instance is also of type C. Thus, we can call any method of the class C on 'obj'.

Thus we can summarize the above points with a rule of thumb:

The Compiler has its eyes on the LHS or the reference and it will compile code congruent to the LHS.Whereas JVM has its eyes on the RHS or the instance that got created and it will invoke functionality congruent to the RHS.

Now there are two exceptions to this rule because run time polymorphism cannot be achieved by datamembers and static functions. This is because data members and static functions are not overridden in the sub-class. Instead they are the properties of the class and not the object! Thus whatever the case be data members and static functions that are accessed will always be that of the parent class.

In the last module we talked about abstraction in java. **Abstraction** is the process of hiding the implementation details from the end user and only showing the necessary functionality. There are two ways to achieve abstraction. It can be achieved by the use of Abstract Class or Interfaces. We will talk about in Interfaces in the later modules.

Abstract Methods & Classes : The literal meaning of the word abstract is that something exists in idea but not physically defined. Similar are the abstract methods and classes in java.A method that doesn't have any implementation and is marked as abstract is an abstract method.Similarly, a class marked abstract, which needs to be extended and all its abstract methods to be implemented, is known as an abstract class.

An abstract class cannot be instantiated that is its object can't be made. An abstract class can have data members, abstract and non-abstract methods and also constructors. An important point to note here is that any non-abstract class cannot have abstract methods. Thus if any class has abstract methods, it has to marked abstract. Also, while extending an abstract class all the abstract methods have to be provided with the implementations or the subclass should also be marked abstract. Let us understand this by making an abstract person class and extend it to make our student class.

```
abstract class Person {
abstract void display();
}

class Student extends Person {

// Display method of the person class is implemented
void display() {
        System.out.println("this is a student");
}

public static void main(String[] args) {
        Student s = new Student();
        s.display();
        // Output: this is a student
}
}
```

Now having learnt thoroughly about the three pillars of OOPs, let us discuss various modifiers that can be used with the data members, methods and classes. There are two types of modifiers: Access modifiers and Non-Access modifiers. The access modifiers change the scope of the properties, methods and that of the class itself whereas the non-access modifiers achieve other functionality. We have already discussed about the use of modifiers with data members, methods and classes. Let us now summarize it all together.

Modifiers & Data Members:

Access Modifiers:

Public: they are accessible everywhere.

Protected: they are available only within the package and outside the package using inheritance.

Private: they are available only within the class.

Default: they are available only within the package.

Non-Access Modifiers:

Static: it creates data members that are common to all the objects of the class. They belong to the class and can be accessed by the class name followed by a dot and their name.

Final: it creates data members whose value cannot be changed. Thus their value is defined either at the time of declaration or in the constructor.

Modifiers & Functions:

Access Modifiers:

Public: the method is available anywhere, even outside the package.

Protected: the method is available within the package and outside the package using inheritance.

Private: the method is available only in the class.

Default: the method is available only within the package.

Non-Access Modifiers:

Static: it creates methods that are independent of the instances of the class. They cannot use the non-static data members of the class and can be accessed by the class name followed by a dot and their name.

Final: it creates methods which cannot be overridden and hence can't be changed in the subclasses.

Abstract: it creates a method declared without any implementation which has to be provided in the subclass. Thus an abstract method can never be final.

Modifiers & Classes:

Access Modifiers:

Public: this class can be accessed anywhere.

Private: we can have a private class only within a public or a default class and its access is limited to only those classes.

Default: it can be accessed anywhere within the package but not outside of it.

Non-Access Modifiers:

Final: a final class is the one that cannot be inherited.

Abstract: a class which is marked abstract and has to be extended and its method implemented is known as an abstract class.