14 Objective Oriented Programming III

In the last module we have learnt about the various types of inheritance. Among them one was the multiple inheritance. Multiple inheritance was where the subclass had more than one parent classes. But this is not allowed in java as if there was a common method that existed in both the base classes, the compiler would not know which method to call. Thus this gives a compile time error and hence is not allowed. However multiple inheritance can be achieved by the use of interfaces in java. Let us learn about them in detail.

Interfaces:

An interface is a reference type in java. It is similar to classes but is purely abstract. Interface is a collection of abstract methods. In an interface, all the **methods are public and abstract** and all the data members are public static and final. Well this is because an interface cannot be instantiated that is an instance cannot be made. Thus an interface doesn't even have a constructor. All the methods don't have their body as all of them are abstract.

For declaring an interface we need to use the keyword 'interface' followed by the name of the interface.Interface can contain as many abstract methods that we need. An interface is by default abstract so we do not need to specify it explicitly. Also, all the methods in the interface are also abstract and public, thus we don't need to specify the 'abstract' keyword.

Let us write our person interface to understand clearly.

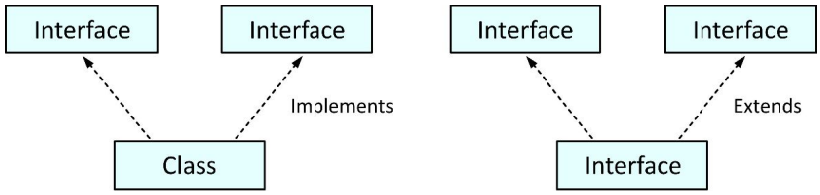
```
interface Person {

	// public abstract methods
	public void name();
	public void age();
	public void gender();
}
```

Here the person interface has three abstract methods that don't have their body. Let us now learn how to use an interface. As a class can extend another class, similarly interfaces can also be 'implemented' by the classes. This is done by using the keyword ***implements***followed by the name of the interface. The class which implements an interface has to implement all the methods of that interface. Thus, an interface can be thought of as a contract signed by the class to implement all the functions of that interface, if the class that implements an interface doesn't implement all the abstract methods then that class must be marked abstract.

While overriding the methods of the interface, the signature and the return type of the method cannot be changed. Another point to note here is that while a class can extend only one class at a time, it can implement more than one interface at a time. Now let us make a Student class to implement our Person interface that we just made.

```
class Student1 implements Person {

	// providing the implementation of all methods
	public void name() {
		System.out.println(“the name method”);
	}

	public void age() {
		System.out.println(“the age method”);
	}

	public void gender() {
		System.out.println(“the gender method”);
	}

	public stati cvoid main(String[] args) {
		Student1 s = new Student1();
		s.name(); // The name method
		s.age(); // The age method
		s.gender(); // The gender method
	}
}

abstract class Student2 implements Person {

	// Abstract class as all methods are not implemented
	// Providing the implementation of name method only
	public void name() {
		System.out.println(“the name method”);
	}

	public static void main(String[] args) {
		Student2 s = new Student2(); // error
		// abstract class can’t be instantiated
	}
}
```

Note here that the Student1 class that implements the person interface implements all the methods of the interface whereas the Student2 class doesn't implement all the methods of the interface and hence is marked abstract.



Multiple Inheritance in Java

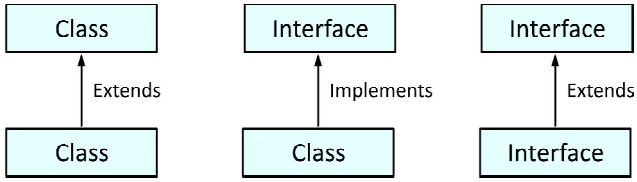
We know that multiple inheritance cannot be achieved by classes in java but interfaces can implement multiple inheritance. We just saw that a class can implement multiple interfaces. This can be thought of as multiple inheritance as the class is implementing methods of multiple interfaces. Also as a class can extend another class, interfaces can also extend other interfaces by using the 'extends' keyword. Also, as interfaces showcase multiple inheritance, an interface can extend multiple interfaces. Thus, when a class implements an interface that has extended another interface, it will have to implement the methods of both the interfaces failing which it would be marked as abstract. Now let us write an interface that extends from another interface and our Student class implements these interfaces.

```
interface male {
	public void gender();
}
interface female {
	public void gender();
}
	// Interface extending 2 interfaces
interface person extends male, female {
	public void name();
}
class student implements person {
	public void name() {
		System.out.println("the name method");
	}
	public void gender() {
		System.out.println("the gender method");
	}
	public static void main(String[] args) {
		student s = new student();
		s.name(); // The name method
		s.gender(); // The gender method
	}
}
```

```
package Demo;

public class person {
	public void display() {
		System.out.println("hello");
	}

}

// For accessing outside package using package name
import Demo.*;

// Using package name and class name
import Demo.person;

// Using full name
public static void main(String args[]){
	Demo.person obj = new Demo.person();
}
```

In a class, multiple inheritance is not supported as there is ambiguity but in the case of interfaces, there is no ambiguity as the implementation is provided within the implementation class only.

All the relations between interfaces and class are summarized below.



Packages:

A java package is a group of similar type of classes, interfaces and sub-packages.Packages can be of two forms:

1. Built-in Packages
2. User-defined Packages

There are two main uses for creating packages. Firstly, packages help us to organize our code better as similar classes and interfaces can be enclosed in one package which helps us to maintain them properly. The second usefulness of packages is that it **prevents name collisions**. Thus, if there were no packages, no two classes would be able to have the same name. Thus, if we have packages we can have two classes by the same name in two different packages. A package within a package is called a **sub-package.** It is created to categorize a package further.

There are already many built-in packages like java, lang, swing, util, io, etc. We can also make our own user defined packages and further make classes and interfaces in them.

Now if we want to use a class or all the classes of a particular package in another package, we need to import that package in our package. Let us learn how we can do that.

There are three ways to access the package from outside of it.

- Using the ***packagename.****By using this, all the classes and the interfaces can be accessed outside of the package. However, the sub-packages cannot be accessed.The **import** keyword is used for the same.
- Using***packagename.classname***By using this, only the specified class will be made available.
- Using ***full name***We write the packagename followed by a dot and the class/interface name wherever we want to access.

```
package Demo;

public class person {
	public void display() {
		System.out.println(“hello”);
	}

}

// For accessing outside package using package name
import Demo.*;

// Using package name and class name
import Demo.person;

// Using full name
public static void main(String args[]){
	Demo.person obj = new Demo.person();
}
```

15 Objective Oriented Programming IV

In the previous module we learnt about the linked list data structure. But in that our node class could store only integer values and thus our linked list was restricted to only integers. Thus, if we wanted to make a character or a string linked list, we would need to make another linked list. Thus for each new data type or object we will need to make separate linked lists. To solve this problem we use Generics.

```
public static<T> T add(T a, T b) {
	return a;
}
```

Here <T> represents the type parameter and 'T' is an identifier that specifies a generic type name, it could have been any other letter like K, S etc.

Creating a Generic Class: Suppose we make a pair class to store two integers. Now if we want to have a pair of two chars, strings or double then we will have to create separate pair classes for each of them. Generics allow us to create a single Pair class that will work for different types.

```
public class Pair<T> {
	T first;
	T second;
}

// Pair of two Integers
Pair<Integer> pInts = new Pair<Integer>();

// Pair of two Strings
Pair<String> pStrings = new Pair<String>();
```

So this class creates a pair which has two variables first and second of types specified in angle brackets (<>). The type parameters can represent only reference types, not primitive types. So, for the primitive data types like int, char, etcJava has corresponding Wrapper classes Integer, Character etc. A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety.

There are mainly 3 advantages of generics. They are as follows:

1. **Type-Safety:** We can hold only a single type of objects in generics. It doesn't allow storing other objects.
2. **Type casting is not required:** There is no need to typecast the object.
3. **Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime.

The good programming strategy says it is far better to handle the problem at compile time than at runtime.

Multiple Type Parameters: We can have multiple type parameters as well i.e. we can create a pair class where *first* and *second* can be of different types unlike the Pair class defined above where both have to be of same type.

```
public class Pair<T, S> {
	T first;
	S second;
}
```

Its instance can be created as follows:

```
Pair<Integer, String> pair = new Pair<Integer, String>();
```

So here pair.first is an Integer and pair.second is a String.

Multilayer Generic Parameters: The generic parameters can be multilayered.

```
Pair<Pair<Integer>> pLayered = new Pair<>();
```

Here pLayered.first and pLayered.second are themselves pair of Integers. Similarly, we can add multiple layers to the parameters.

Bounded Type Parameters: Many a times when you might want to restrict the kinds of types that are allowed to be passed to a type parameter. Say we want to create a generic sort function. In order to sort elements we will have to compare them. The "<" or ">" are not defined for non-primitives. So instead we will have to use *compareTo()* method (in *Comparable* interface) which compares two objects and returns an int based on result.

Now in our sort function we should allow only those non-primitives who have *compareTo()* method defined for them or in other terms who have implemented the *Comparable* interface (as interface serves as a contract, so if a non-abstract class has implemented *Comparable* method then we can be sure that it has *compareTo()* method).

We can do this as shown below:

```
public static <T extends Comparable<T>> void sort(T[] input) {
	for (int i = 0; i<input.length; i++) {
		for (int j = 0; j<input.length - i - 1; j++) {
			if (input[j].compareTo(input[j + 1]) == 1) {
				T temp = input[j + 1];
				input[j + 1] = input[j];
				input[j] = temp;
			}
		}
	}
}
```

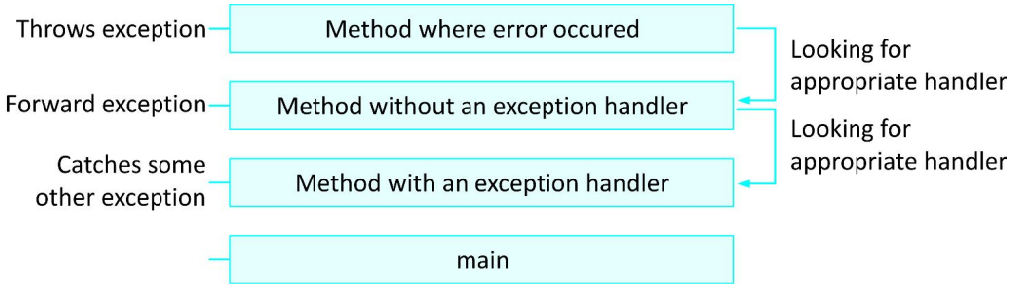
When we write <T extends Comparable<T>> this means that only those parameters are allowed those who have implemented Comparable Interface.

Exception Handling:

An exception is an event, which occurs during the execution of a program, and that disrupts the normal flow of the program's instructions. The **exception handling in java** is one of the powerful mechanisms to handle the runtime errors so that normal flow of the application can be maintained.

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called throwing an exception.

After a method throws an exception, the runtime system attempts to find something to handle it. The block of code that handles an exception is called exception handler. When an exception occurs at the run time, system first tries to find an exception handler in the method where the exception occurred and then searches the methods in the reverse order in which they were called for the exception handler. The list of methods is known as the call stack (shown below). If no method handles the exception then exception appears on the console (like we see ArrayIndexOutOfBoundsExceptionetc).



Types of Exception:

Checked Exceptions: These are exceptional conditions that we can anticipate when user makes mistake.

For example, computing factorial of a negative number. A well-written program should catch this exception and notify the user of the mistake, possibly prompting for a correct input. Checked exceptions *are subject* to the Catch or Specify Requirement i.e. either the function where exception can occur should handle or specify that it can throw an exception (We will look into it in detail later).

Error: These are exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from. For example, suppose that an application successfully opens a file for input, but is unable to read the file because of a hardware or system malfunction.

Unchecked Exception: These are exceptional conditions that might occur at runtime but we don't expect them to occur while writing code. These usually indicate programming bugs, such as logic errors or improper use of an API. For example:StackOverflowException.

Exception Handling: Exception handling is achieved by using try catch and/or finally block.

Try block - The code which can cause an exception is enclosed within try block.

Catch block - The action to be taken when an exception has occurred is done in catch block. It must be used after the try block only.

Finally block - Java finally block is a block that is used to execute important code such as closing connection, stream etc. Java finally block is always executed whether exception is handled or not.

Here is a sample code to explain the same.

```
Scanner s = new Scanner(System.in);

	System.out.println("Enter dividend");
	int dividend = s.nextInt();

	System.out.println("Enter divisor");
	int divisor = s.nextInt();

	try {
		int data = dividend / divisor;
		System.out.println(data);
	} catch (ArithmeticExceptione) {
		System.out.println("Divide by zero error");
	} finally {
		System.out.println("Finally block is always executed");
	}

	System.out.println("Rest of the code...");
```

Note **:**

1. Whenever an exception occurs statements in the try block after the statement in which exception occurred are not executed
2. For each try block there can be zero or more catch blocks, but only one finally block.

Creating an Exception/User Defined Exceptions: A user defined exception is a sub class of the exception class. For creating an exception you simply need to extend Exception class as shown below:

```
public class InvalidInputException extends Exception {
	private static final long serialVersionUID = 1L;
}
```

Throwing an Exception: Sometimes, it's appropriate for code to catch exceptions that can occur within it. In other cases, however, it's better to let a method further up the call stack handle the exception. For example if input to the factorial method is a negative number, then it makes more sense for the factorial to throw an exception and the method that has called factorial method to handle the exception.

Here is the code for the factorial method:

```
public static int fact(int n) throws InvalidInputException {
            if (n< 0) {
                  InvalidInputException e = new InvalidInputException();
                  throw e;
            }
            if (n == 0) {
                  return 1;
            }
            return n * fact(n - 1);
      }
```

The fact method throws an InvalidInputException that we created above and we will handle the exception in main.

```
public static void main(String[] args) {
            Scanner scn = new Scanner(System.in);
            System.out.println("Enter number");
            int n = scn.nextInt();
            int a = 10;
            try {
                  System.out.println(fact(n));
                  a++;
            } catch (InvalidInputException e) {
                  System.out.println("Invalid input. Try again.");
                  return;
            }
      }
```