UNIT-3

- **Abstract classes and interfaces**
- **Packages**

Abstract classes and interfaces:

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the obje does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

Abstract class

A class which is declared with the abstract keyword is known as an abstract class in java. It can have abstract and non-abstract methods (method with the body).It needs to be extended and its method implemented. It cannot be instantiated.

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Example of abstract class

```
abstract class A{

}
```

Abstract Method

A method which is declared as abstract and does not have implementation is known as an abstract method.

Example of abstract method

```
abstract void printStatus();//no method body and abstract
```

Example of Abstract classes and Method

```
abstract class Shape {
   abstract void draw();
}

// Concrete subclass implementing draw()
class Rectangle extends Shape {
   void draw() {
      System.out.println("drawing rectangle");
   }
}

// Concrete subclass implementing draw()
class Circle1 extends Shape {
   void draw() {
      System.out.println("drawing circle");
   }
}

// Test class
public class Abstract {
   public static void main(String args[]) {
      // Polymorphic behavior: Shape reference to a Circle1 object
      Shape s = new Circle1(); // In a real scenario, object is provided through method, e.g.,
getShape() method
      s.draw();
   }
}
```

An abstract class can have a data member, abstract method, concrete method (non-abstract method), constructor, and even main() method.

```
abstract class Bike {
   Bike() {                                    // Constructor
      System.out.println("bike is created");
   }

   abstract void run();                        // Abstract method


   void changeGear() {                         // Concrete method
      System.out.println("gear changed");
   }
}
```

Interface in Java

The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple multiple inheritance in java.interfaces can have abstract methods and variables. It cannot have a method body.

- Java Interface also **represents the IS-A relationship**.
- It cannot be instantiated just like the abstract class.
- we can have **default and static methods** in an interface.

There are mainly two reasons to use interface.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.

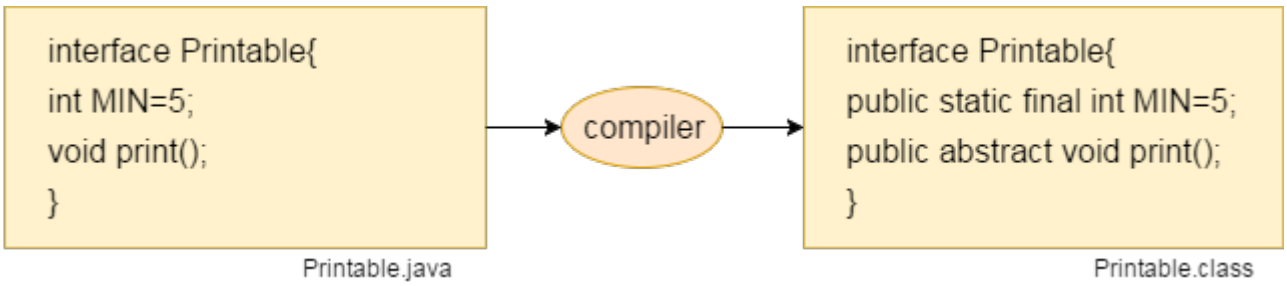
Declaration of interface

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

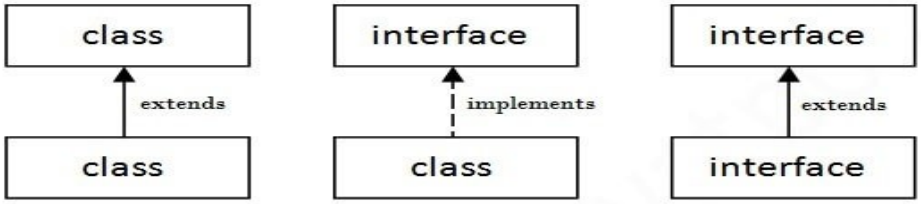
Syntax:

```
interface  <interface_name>
{
// declare constant fields
 // declare methods that abstract
 // by default.
 }
```

The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.



A class extends another class, an interface extends another interface, but a **class implements an interface**.



```
interface Bank {
   float rateOfInterest();
}

class SBI implements Bank {
   public float rateOfInterest() {
      return 9.15f;
   }
}

class PNB implements Bank {
   public float rateOfInterest() {
      return 9.7f;
   }
}
```

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.

Example:

```
interface A{

void print();

}

interface B {

void show();

}

class multiple implements A, B {

public void print() {

System.out.println("Hello");

}

public void show() {

System.out.println("Welcome");

}

public static void main(String args[]) {

multiple obj = new multiple();

obj.print();

obj.show();

}
```

```
}
```

Multiple inheritance is not supported through class in java, but it is possible by an interface.

multiple inheritance is not supported in the case of class because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class.

Interface inheritance

A class implements an interface, but one interface extends another interface.

```
interface Animal {
    void eat();
    void sleep();
}

// Child interface inheriting from Animal
interface Dog extends Animal {
    void bark();
}

// Child interface inheriting from Animal
interface Cat extends Animal {
    void meow();
}

// Concrete class implementing Dog and Cat interfaces
class Pet implements Dog, Cat {
    public void eat() {
        System.out.println("Pet is eating");
    }
    public void sleep() {
        System.out.println("Pet is sleeping");
    }
    public void bark() {
        System.out.println("Pet is barking");
    }
    public void meow() {
        System.out.println("Pet is meowing");
    }
}

// Main class
public class Interface {
    public static void main(String[] args) {
        Pet myPet = new Pet();
        myPet.eat();
        myPet.sleep();
        myPet.bark();
        myPet.meow();
    }
}
```

Nested Interface

An interface can have another interface which is known as a nested interface.

```
interface printable {
   void print();
   interface MessagePrintable {  // Nested interface
      void msg();
   }
}
```

Difference between abstract class and interface

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Inner Classes (Nested Classes)

Java inner class or nested class is a class that is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place to be more readable and maintainable.

Additionally, it can access all the members of the outer class, including private data members and methods.

Syntax of Inner class

```
class Java_Outer_class {
   // Outer class code


   class Java_Inner_class {
      // Inner class code
   }
}
```

Advantage of Java inner classes

1. Nested classes represent a particular type of relationship that is **it can access all the members (data members and methods) of the outer class,** including private.
2. Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.
3. **Code Optimization**: It requires less code to write.

Local inner classes

Local inner classes in Java are defined within a block of code, typically within a method, constructor, or a code block. Here's the syntax:

```
class OuterClass {
   // Outer class code

   void outerMethod() {
      // Method code

      class LocalInner {
         // Local inner class code
      }

      // Code within the method
   }

   // More outer class code
}
```

Example:

```
public class Outer {

   private int i = 10;
   public void outerMethod() {
         int j = 20;
      // Local inner class defined within a method
      class LocalInnerClass {
         private int k = 30;
         public void display() {
```

```
            System.out.println("Sum of i and k: " + (i+k));
            System.out.println("Sum of i,j and k: " + (i+j+k));
        }
    }
    // Creating an instance of the local inner class
    LocalInnerClass innerObj = new LocalInnerClass();
    innerObj.display();
  }

  public static void main(String[] args) {
    Outer o = new Outer();
    o.outerMethod();
  }
}
```

Anonymous inner class

Java anonymous inner class is an inner class without a name and for which only a single object is created. An anonymous inner class can be useful when making an instance of an object with certain "extras" such as overloading methods of a class or interface, without having to actually subclass a class.

In simple words, a class that has no name is known as an anonymous inner class in Java. It should be used if you have to override a method of class or interface. Java Anonymous inner class can be created in two ways:

1. Class (may be abstract or concrete).
2. Interface

- A normal class can implement any number of interfaces but the anonymous inner class can implement only one interface at a time.
- A regular class can extend a class and implement any number of interfaces simultaneously. But anonymous Inner class can extend a class or can implement an interface but not both at a time.
- For regular/normal class, we can write any number of constructors but we can't write any constructor for anonymous Inner class because the anonymous class does not have any name and while defining constructor class name and constructor name must be same.

Example:

```
interface Anonymous {
  void run();
}
class BikeTest{
 public static void main(String args[]){
  Anonymous obj;
  obj = new Anonymous(){
```

```
        public void run(){
System.out.println("Bike is running");
}
        }
    obj.run();
   }
}
```

static nested (inner)class

A static class is a class that is created inside a class, is called a static nested class in Java. It cannot access non-static data members and methods. It can be accessed by outer class name.

- It can access static data members of the outer class, including private.
- The static nested class cannot access non-static (instance) data members

Example:

```
public class Outer {
private int i=10;
static int j=20;
static class StsticInner{
int k=30;
void add(){
//System.out.println(i+k);
System.out.println(j+k);
}
}
public static void main(String[] args) {
Outer.StsticInner obj = new Outer.StsticInner();
obj.add();
}
}
```

Packages

A **java package** is a group of similar types of classes, interfaces and sub-packages.

A package is a container of a group of related classes where some of the classes are accessible are exposed and others are kept for internal purpose. We can reuse existing classes from the packages as many time as we need it in our program.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

Making searching/locating and usage of classes, interfaces, enumerations and annotations easier.

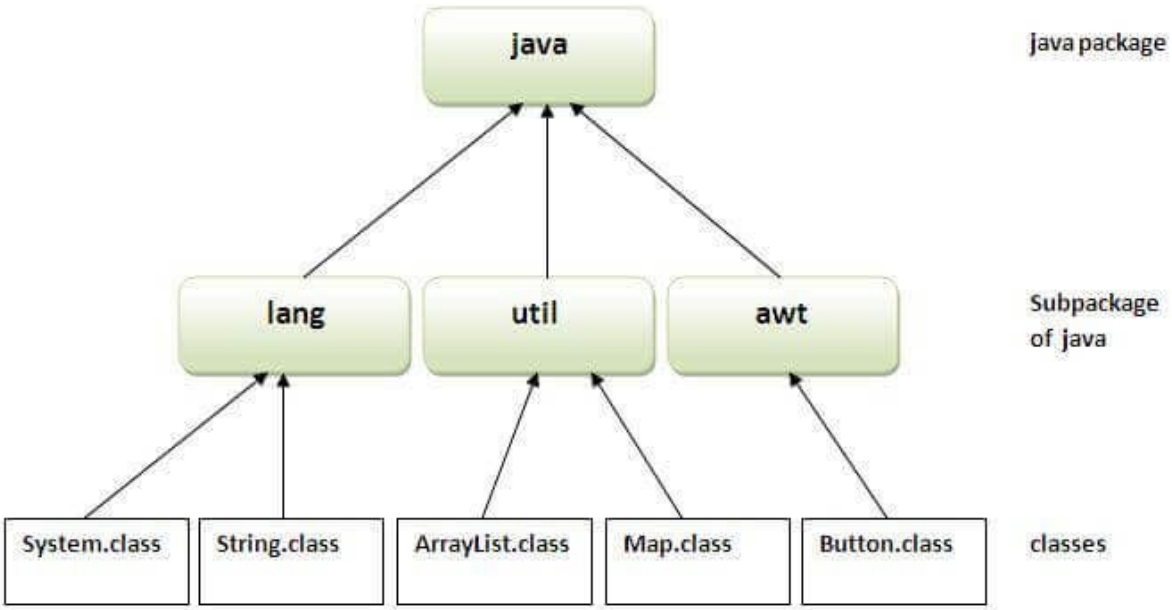
Packages can be considered as data encapsulation (or data-hiding).

2) Java package provides access protection.

protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.

3) Java package removes naming collision.

For example there can be two classes with name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee



Simple example of java package

The **package keyword** is used to create a package in java.

```
// Save as Simple.java
package mypack;

public class Simple {
   public static void main(String[] args) {
      System.out.println("Welcome to package");
   }
}
```

How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

javac -d directory . javafilename

javac -d . Simple.java

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java
To Run: java mypack.Simple

How to access package from another package?

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

1. import package.*;
2. import package.classname;
3. fully qualified name.

Package naming conventions : Packages are named in reverse order of domain names, For example, in a college, the recommended convention is college.tech.cse, college.tech.ee, college.art.history, etc.

Adding a class to a Package : We can add more classes to a created package by using package name at the top of the program and saving it in the package directory. We need a new **java** file to define a public class, otherwise we can add the new class to an existing **.java** file and recompile it.

Subpackages: Packages that are inside another package are the **subpackages**. These are not imported by default, they have to imported explicitly. Also, members of a subpackage have no access privileges, i.e., they are considered as different package for protected and default access specifiers.

Example :

import java.util.*;

util is a subpackage created inside **java** package.

Accessing classes inside a package

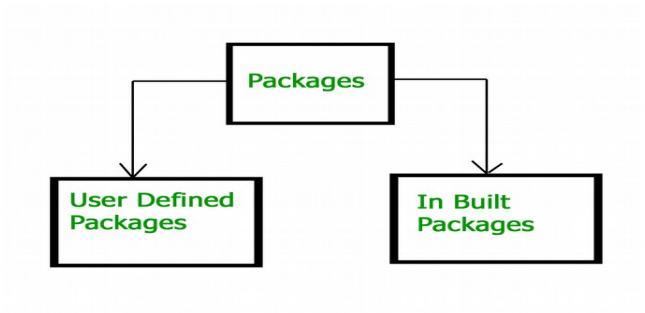
```
// import the Vector class from util package.
import java.util.vector;

// import all the classes from util package
import java.util.*;
```

First Statement is used to import **Vector** class from **util** package which is contained inside **java**.

- Second statement imports all the classes from **util** package.

Types of packages:



Built-in Packages

These packages consist of a large number of classes which are a part of Java **API**.Some of the commonly used built-in packages are:

1. **java.lang:** Contains language support classes(e.g classes which defines primitive data types, math operations). This package is automatically imported.
2. **java.io:** Contains classes for supporting input / output operations.
3. **java.util:** Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.
4. **java.applet:** Contains classes for creating Applets.
5. **java.awt:** Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc). 6)
6. **java.net:** Contain classes for supporting networking operations.

User-defined packages: These are the packages that are defined by the user. First we create a directory **myPackage** (name should be same as the name of the package). Then create the **MyClass** inside the directory with the first statement being the **package names.**

Creating Packages: Java also allows developers to create their own packages. By organizing classes into custom packages, developers can structure their codebase in a way that reflects its logical components and promotes reusability.

To create your own package in Java, follow these steps, keeping in mind that Java utilizes a file system directory to organize packages:

1. **Organize Directory Structure**: Organize your directories on the file system according to the package structure you want.

2. For instance:

└── OOP

└── package1

└── MyPackageClass.java

2. **Define Package**: In your Java source file (MyPackageClass.java in this case), use the package keyword to define the package to which the class belongs. For example:

Syntax:

```
package package1;

public class MyPackageClass {

    // Your class code here

}
```

3. **Compilation**: Ensure that the file is located in the correct directory structure matching the package name.

For example, if your source file MyPackageClass.java is located at OOP/package1, navigate to the OOP directory and compile it:

Syntax:

javac -d directory javafilename

Example:

javac -d . MyPackageClass.java

If you want to keep a package within the same directory you're working in, you would reference it using the (.)dot notation.

4. **Usage**: Once compiled, you can use your package in other Java files by importing it using the import statement:

Example:

```
package Calculator;


public class ADD {
int i,j;
public ADD(int i,int j){
        this.i=i;
        this.j=j;
}
public void display()
{
        System.out.println("addition of i,j is:  "+(i+j));
}
}
/*--------------------------------------------------------------------*/

package Calculator;

public class MUL {
        int i,j;
        public MUL(int i,int j){
                this.i=i;
                this.j=j;
        }
        public void display()
        {
                System.out.println("multiplication of i,j is:  "+(i*j));
        }
}

/*-----------------------------------------------------------*/

package Calculator;

public class SUB {
```

```
        int i,j;
        public SUB(int i,int j){
                this.i=i;
                this.j=j;
        }
        public void display()
        {
                System.out.println("subtraction of i,j is:  "+(i-j));
        }
}
/*-------------------------------------------------------*/
import Calculator.*;                    /*import package*/
public class Calcy {
 public static void main(String args[])
 {
        ADD a=new ADD(10,20);
        a.display();
        SUB s=new SUB(30,10);
        s.display();
        MUL m=new MUL(2,5);
        m.display();

 }
}
```