UNIT-III

Interfaces:

Interface is used to achieve full abstraction in JAVA

By using interface, we can specify what a class must do, but not how it does it.

Interfaces are syntactically similar to classes but they lack instance variables and their methods are declared without any body.

Defining an interface:

An interface is defined much like a class.

To define an interface, the following syntax is used.

Syntax:

```
access-specifier interface interface_name
{
  datatype variable_name1=value1;
  datatype variable_name2=value2;
 .
 .
  .
  datatype variable_numn=valuen;
 returntype method_name1(parameter_list);
 returntype method_name2(parameter_list);
  .
  .
  Returntype method_namen(parameter_list);
}
```

Here in the place of access-specifier, we use public or no modifier.

When no access modifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as **public**, the interface can be used by any other code.
interface_name is any valid identifier.
All the variables declared inside interface are considered as final and static. That means they cannot be changed further. They must be initialized.
All the methods declared in interface are implicitly abstract methods.

Example:

```
interface A
{
  void show();
}
```

Implementing interfaces:

Once an **interface** has been defined, one or more classes can implement that interface. To implement an interface, we use the keyword **implements** in a class definition.

Syntax:

```
class class_name [implements interface1[,interface2.....]]
{
   //class body
}
```

If class implements more than one interface, interface names are separated by comma operator.
In implementation class, the methods defined by interface are implemented using the keyword public.

Example:

```
interface A
{
   void show();
}
class B implements A
{
  public void show()
  {
    System.out.println("interface method");
   }
}
```

Write a program to declare an interface and implement an interface.

```
//Program to illustrate the interface
interface A
{
    void show();
 }
class B implements A
{
   public void show()
   {
     System.out.println("interface method");
    }
}
class InterfaceDemo
{
  public static void main(String args[])
   {
       B b1=new B();
        A a;
```

```
      a=b1;
       a.show();
   }
}
```

An interface reference variable has knowledge only of the methods declared by its **interface** declaration.

```
interface A
{
  void show();
 }
Class B implements A
{
   public void show()
   {
     System.out.println("interface method");
    }
  void display()
  {
    System.out.println("non interface method");
   }
}
class InterfaceDemo
{
   public static void main(String args[])
   {
      B b1=new B();
      A a;
      a=b1;
      a.show();
      b1.display();
    }
}
```

Interface implemented by multiple classes:

```
interface A
{
   void show();
 }
class B implements A
{
   public void show()
    {
       System.out.println("interface method in B");
     }
}
class C implements A
{
    public void show()
```

```
   {
     System.out.println("interface method in C");
    }
}
class InterfaceDemo
{
   public static void main(String args[])
   {
     B b1=new B();
    C c1=new C();
     A a;
     a=b1;
     a.show();
     a=c1;
     a.show();
   }
}
```

Write a program to find the areas of different shapes using interface.

```
//Program to find the areas of different shapes using interface
interface Shape
{
  double area();
 }
class Triangle implements Shape
{
   double b,h;
    Triangle(double x,double y)
    {
      b=x;
      h=y;
     }
  public double area()
  {
     return 0.5*b*h;
   }
}
class Rectangle implements Shape
{
   double l,b;
    Rectangle(double x,double y)
   {
       l=x;
       b=y;
    }
   public double area()
  {
     return l*b;
   }
}
class ShapeDemo
```

```
{
   public static void main(String args[])
   {
     Triangle t=new Triangle(1,2);
     Rectangle r=new Rectangle(3,4);
      Shape s;
    s=t;
    System.out.println("Area of triangle is "+s.area());
    s=r;
    System.out.println("Area of rectangle is "+s.area());
 }
}
```

Nested interfaces:

An interface can be declared a member of a class or another interface. Such an interface is called a member interface or a nested interface.

A nested interface declared within an interface must be public.

A nested interface declared within a class can have any access modifier.

When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member.

Example1: Interface inside another interface

```
interface OuterInterface
{
   void show();
   interface InnerInterface
   {
      void display();
   }
}
class A implements OuterInterface
{
   public void show()
   {
      System.out.println("OuterInterface method");
   }
}
class B implements OuterInterface.InnerInterface
{
   public void display()
   {
      System.out.println("InnerInterface method");
   }
}
 class NestedInterfaceDemo
{
        public static void main(String[] args) {
                A a1=new A();
                B b1=new B();
```

```
                a1.show();
                b1.display();
        }
}
```

Example2: Interface inside a class

```
class VehicleTypes
{
   interface Vehicle
   {
      int getNoOfWheels();
   }
}
class Bike implements VehicleTypes.Vehicle
{
   public int getNoOfWheels()
   {
      return 2;
   }
}
class Car implements VehicleTypes.Vehicle
{
   public int getNoOfWheels()
   {
      return 4;
   }
}
class Bus implements VehicleTypes.Vehicle
{
   public int getNoOfWheels()
   {
      return 6;
   }
}
public class NestedInterfaceDemo
{
        public static void main(String[] args) {
                Bike b=new Bike();
                Car c=new Car();
                Bus b1=new Bus();
                System.out.println("Wheels in Bike are "+b.getNoOfWheels());
                System.out.println("Wheels in Car are "+c.getNoOfWheels());
                System.out.println("Wheels in Bus are "+b1.getNoOfWheels());
        }
}
```

Multiple interfaces:

One class can implement multiple interfaces.

In JAVA language, multiple inheritance is achieved by implementing multiple interfaces in single class.

Example:

```
interface Eatable
{
    void eat();
}
interface Flyable
{
    void fly();
}
class Bird implements Eatable,Flyable
{
   public void eat()
   {
      System.out.println("Bird eats");
    }
 public void fly()
 {
  System.out.println("Bird flying");
 }
}
class MultipleInterfaceDemo
{
   public static void main(String args[])
   {
      Bird b=new Bird();
       b.eat();
       b.fly();
    }
}
```

Extending Interfaces:
In JAVA language, like classes interfaces can also be extended by using the keyword **extends**.
When a class implements an interface that inherits another interface, it must provide implementations for all methods required by the interface inheritance chain.

Write a program to illustrate how to extend interfaces.

```
//Program to illustrate extending interfaces
interface A
{
   void method1();
   void method2();
}
interface B extends A
{
   void method3();
}
class C implements B
{
    public void method1()
```

```
  {
     System.out.println("Method one");
  }
public void method2()
{
   System.out.println("Method Two");
}
public void method3()
{
    System.out.println("Method Three");
}
}
class ExtendDemo
{
 public static void main(String args[])
 {
    C c1=new C();
    c1.method1();
    c1.method2();
    c3.method3();
  }
}
```

Default methods in interfaces:
From JAVA 8 onwards, it is possible to add default methods in an interface.
Default methods are declared with the keyword **default** and they are implemented in interface.

Example:

```
interface A
{
   void show();
   default void display()
  {
     System.out.println("Default method");l
    }
}
class B implements A
{
    public void show()
    {
       System.out.println("show method");
    }
}
class DefaultDemo
{
   public static void main(String args[])
   {
       B b1=new B();
       b1.show();
       b1.display();
```

```
    }
}
```

Note: If implementation class overrides the default method then always the method available in implementation class gets executed.

Example:

```
interface A
{
   void show();
   default void display()
   {
      System.out.println("Display method in interface A");
    }
}
class B implements A
{
   public void show()
   {
    System.out.println("show method");
    }
  public void display()
  {
    System.out.println("Display method in class B");
   }
}
class DefaultDemo
{
   public static void main(String args[])
   {
      B b1=new B();
      b1.show();
      b1.display();
    }
}
```

Note: In cases in which one interface inherits another, with both defining a common default method, the inheriting interface's version of the method takes precedence. Therefore,
It is possible to explicitly refer to a default implementation in an inherited interface by using a new form of **super**.

Syntax:

```
InterfaceName.super.methodName( )
```

Example:

```
interface A
{
  void show();
  default void display()
```

```
 {
    System.out.println("Default method in A");
  }
}
interface B extends A
{
   default void display()
  {
      A.super.display();
      System.out.println("Default method in B");
   }
}
class C implements A,B
{
  public void show()
  {
   System.out.println("show method");
   }
}
class DefaultDemo
{
  public static void main(String args[])
  {
   C c1=new C();
   c1.show();
   c1.display();
   }
}
```

Example:

```
interface A
{
  default void display()
 {
    System.out.println("Default method in A");
  }
}
interface B
{
   default void display()
  {
      System.out.println("Default method in B");
   }
}
class C implements A,B
{
  public void display()
  {
     A.super.display();
```

```
    B.super.display();
    System.out.println("Default method in C");
   }
}
class DefaultDemo
{
  public static void main(String args[])
  {
   C c1=new C();
   c1.display();
  }
}
```

Static methods in interfaces:

JDK 8 added another new capability to **interface**: the ability to define one or more **static** methods.
Like **static** methods in a class, a **static** method defined by an interface can be called independently of any object. Thus, no implementation of the interface is necessary, and no instance of the interface is required, in order to call a **static** method.
Instead, a **static** method is called by specifying the interface name, followed by a period, followed by the method name.

Here is the general form:

InterfaceName.staticMethodName

Example:

```
interface A
{
  static void display()
  {
    System.out.println("Display method");
   }
 void show();
}
class B implements A
{
  public void show()
  {
    System.out.println("show method");
   }
}
class StaticDemo
{
  public static void main(String args[])
  {
   B b1=new B();
   b1.show();
   A.display();
```

```
    }
}
```

Functional Interfaces:

An interface which contains only one abstract method is called as functional interface.
Functional interface is also called as Single Abstract Method interface.
In functional interfaces, in addition to the single abstract method we can write any number of default methods and static methods.
Functional interfaces can be implemented by using Lambda expressions.

Example1:

```
interface A
{
  void show();
}
class FunctionalDemo
{
   public static void main(String args[])
   {
     A a=()->System.out.println("Show method");
     a.show();
    }
}
```

Example2:

```
interface A
{
   int square(int x);
}
class FunctionalDemo
{
   public static void main(String args[])
   {
     A a=(x)->x*x;
    System.out.println("Square of 5 is "+a.square(5));
   }
}
```

Differences 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) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
5) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
6) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

7)**Example:**
abstract class Shape{
 	
abstract void draw();
}
	Example:
	interface A
	{
	void show();
	}

Packages:
Packages are containers for classes.
Package is a collection of classes, interfaces and sub packages.
Packages are used to keep the name space compartmentalized.

Generally, there are two types of packages.
They are

1) Built-in packages
2) User defined packages

1) Built-in packages: These packages are already defined in JAVA API
Example: java.lang, java.io, java.util, java.awt etc.

2) User defined packages: Java package created by user to categorize their classes and interfaces known as user defined package.

Creating a package:

To create a package, we use **package** command as first statement in JAVA source file. Any classes declared within that file will belong to the specified package.
The **package** statement defines a name space in which classes are stored.
If you omit the **package** statement, the class names are put into the default package, which has no name.

Syntax:

```
package pkg;
```

Here pkg is the name of the package.

Example:

```
package p1;
```

We can create hierarchy of packages. To do that, simply separate each package name from the one above it by use of period.

Syntax:

```
package  pkg1[.pkg2[.pkg3]];
```

Example:

```
package p1.p2;
```

Write a program to illustrate the use of packages.

```
//Program to illustrate the use of packages
package p1;
class A
{
   int i,j;
   A(int x,int y)
   {
      i=x;
      j=y;
   }
 void display()
 {
    System.out.println(i+" "+j);
  }
}
class Sample
{
   public static void main(String args[])
   {
     A a1=new A(10,20);
     a1.display();
  }
```

```
}
```

For compiling the above program , we use the following command

```
javac –d . Sample.java
```

Here –d represents creating a directory in which generated class files are stored.
. represents the current directory. That means, a package is created within the current directory.

After compilation, for execution we use the following command

```
java p1.Sample
```

Importing Packages:

In JAVA, import key word is used to import built-in and user defined packages.
When a package is imported, we can refer all the classes of that package using their name directly.
import statement must be after the package statement(if exists) and before any class definitions.

Syntax:

```
import pkg1[.pkg2].(classname|*);
```

Here pkg1 is the name of the top-level package and pkg2 is the sub ordinate package inside the outer package separated by period.
If we want to import a specific class from the package, we use specific class name.
If we use * then entire package is imported.

Note: when a package is imported, only those items within the package declared as **public** will be available to non-subclasses in the importing code.

Example:

```
import java.util.Date;
import java.util.*;
```

Write a program to illustrate how to import a specific class from a package.

A.java

```
package p1;
public class A
{
  int i,j;
  public A(int x,int y)
 {
     i=x;
     j=y;
```

```
  }
 public void display()
{
    System.out.println(i+" "+j);
 }
}
```

Sample.java

```
import p1.A;
class Sample
{
  public static void main(String args[])
  {
    A a1=new A(10,20);
    a1.display();
  }
}
```

Write a program to illustrate how to import multiple classes from a package.

Add.java

```
package Arithmetic;
public class Add
{
   public int add(int x,int y)
  {
    return x+y;
   }
}
```

Sub.java

```
package Arithmetic;
public class Sub
{
   public int sub(int x,int y)
   {
      return x-y;
    }
}
```

Sample.java

```
import Arithmetic.*;
class Sample
{
  public static void main(String args[])
  {
    Add a=new Add();
```

```
    Sub s=new Sub();
    System.out.println("Addition is "+a.add(10,20));
   System.out.println("Subtraction is "+s.sub(5,4));
  }
}
```

Access Control:

Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:

- Subclasses in the same package
- Non-subclasses in the same package
- Subclasses in different packages
- Classes that are neither in the same package nor subclasses

1) Anything declared **public** can be accessed from anywhere.
2) Anything declared **private** cannot be seen outside of its class.
3) When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access.
4) If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element **protected**.

	private	**No modifier**	**protected**	**public**
Same class	Yes	Yes	Yes	Yes
Same Package sub class	No	Yes	Yes	Yes
Same Package non-sub class	No	Yes	Yes	Yes
Different package sub class	No	No	Yes	Yes
Different package non-sub class	No	No	No	Yes

Example:

A.java

```
package p1;

public class A

{

 int a=1;
```

```
 private int b=2;
 protected int c=3;
 public int d=4;
 public A()
 {
   System.out.println(a+" "+b+" "+c+" "+d);
 }
}
```

B.java

```
package p1;
class B extends A
{
 B()
 {
  System.out.println(a+" "+c+" "+d);
 }
}
```

C.java

```
package p1;
class C
{
 C()
 {
  A a1=new A();
  System.out.println(a1.a+" "+a1.c+" "+a1.d);
  }
```

```
}
```

Demo.java

```
package p1;
class Demo
{
  public static void main(String args[])
  {
    A a1=new A();
    B b1=new B();
    C c1=new C();
  }
}
```

D.java

```
package p2;
class D extends p1.A
{
  D()
  {
   System.out.println(c+" "+d);
  }
}
```

E.java

```
package p2;
class E
{
  E()
  {
```

```
   p1.A a1=new p1.A();
   System.out.println(a1.d);
  }
}
```

Demo1.java

```
package p2;
class Demo1
{
 public static void main(String args[])
 {
   D d1=new D();
   E e1=new E();
  }
}
```

java.lang package:

This package provides classes that are fundamental to the design of the Java programming language.

Following are the important classes in java.lang package

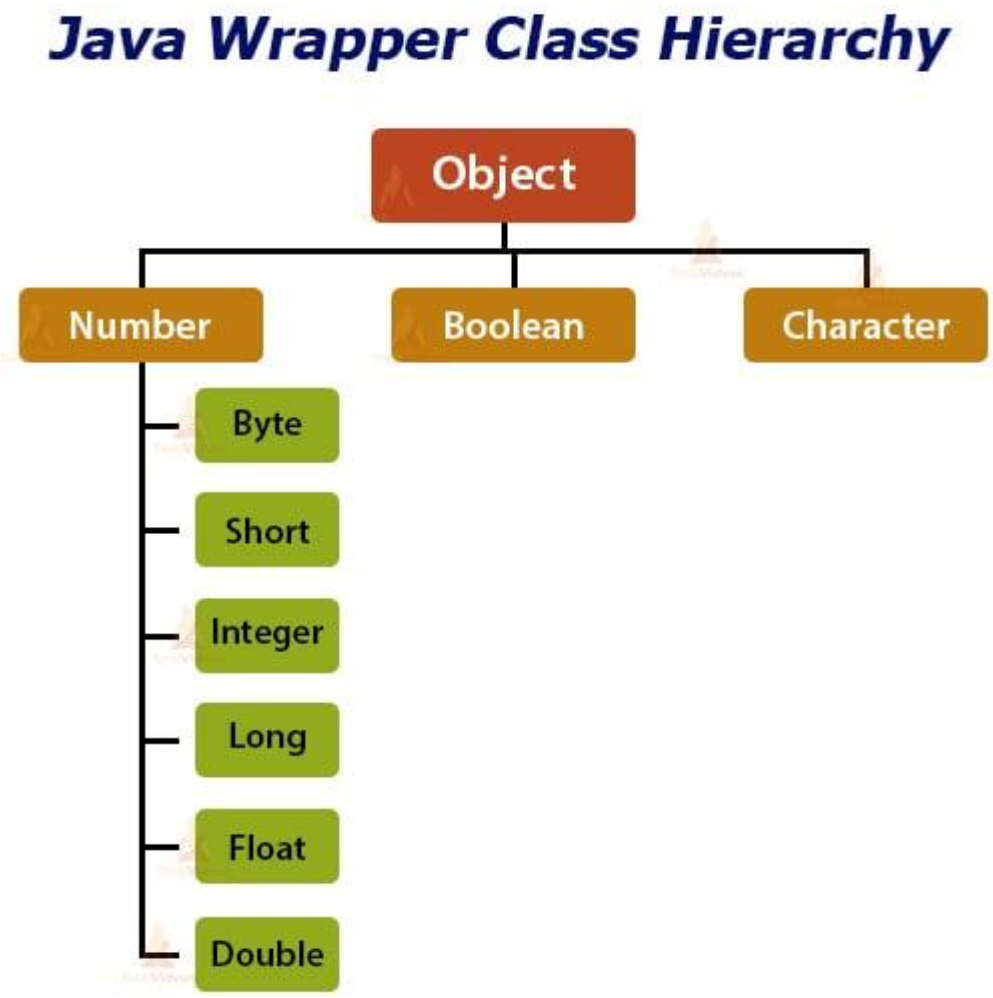
1) **Boolean:** The Boolean class wraps a value of the primitive type boolean in an object.
2) **Byte**: The Byte class wraps a value of primitive type byte in an object.
3) **Character**: The Character class wraps a value of the primitive type char in an object.
4) **Double**: The Double class wraps a value of the primitive type double in an object.
5) **Float**: The Float class wraps a value of primitive type float in an object.
6) **Integer**: The Integer class wraps a value of the primitive type int in an object.
7) **Long**: The Long class wraps a value of the primitive type long in an object.
8) **Math**: The class Math contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.
9) **Object:** Class Object is the root of the class hierarchy.

10)**Package:** Package objects contain version information about the implementation and specification of a Java package.
11)**Short:** The Short class wraps a value of primitive type short in an object.
12)**String**: The String class represents character strings.
13)**StringBuffer**: A thread-safe, mutable sequence of characters.
14)**StringBuilder**: A mutable sequence of characters.
15)**System**: The System class contains several useful class fields and methods.
16)**Thread:** A thread is a thread of execution in a program.
17)**ThreadGroup**: A thread group represents a set of threads.
18)**Throwable**: The Throwable class is the super class of all errors and exceptions in the Java language.

Wrapper classes:

In JAVA language, sometimes we need to use objects instead of primitive data types. To achieve this, we use Wrapper classes.

A wrapper class provides a mechanism to convert the primitive data types into objects and objects into primitive data types.



All the wrapper classes Byte, Short, Integer, Long, Double and, Float, are subclasses of the abstract class **Number.** While Character and Boolean wrapper classes are the subclasses of class **Object.**

Primitive Data Type	Wrapper Class
boolean	**Boolean**
byte	**Byte**
char	**Character**
int	**Integer**
float	**Float**
double	**Double**
long	**Long**
short	**Short**

Write a program to illustrate wrapper classes.

```
//Program to illustrate wrapper classes

class  WrapperDemo

{

    public static void main(String[] args)

 {

        byte a = 1;
        Byte byteobj = new Byte(a);
        int b = 10;
        Integer intobj = new Integer(b);
        float c = 18.6f;
        Float floatobj = new Float(c);
        double d = 250.5;
        Double doubleobj = new Double(d);
        char e='a';
        Character charobj=e;
        System.out.println("Values of Wrapper objects (printing as objects)");
        System.out.println("Byte object byteobj:  " + byteobj);
        System.out.println("Integer object intobj:  " + intobj);
        System.out.println("Float object floatobj:  " + floatobj);
        System.out.println("Double object doubleobj:  " + doubleobj);
        System.out.println("Character object charobj:  " + charobj);
        byte bv = byteobj;
        int iv = intobj;
        float fv = floatobj;
        double dv = doubleobj;
       char cv = charobj;
```

```
        System.out.println("Unwrapped values (printing as data types)");
        System.out.println("byte value, bv: " + bv);
        System.out.println("int value, iv: " + iv);
        System.out.println("float value, fv: " + fv);
        System.out.println("double value, dv: " + dv);
        System.out.println("char value, cv: " + cv);
            }

}
```

Auto Boxing and unboxing:

In **autoboxing**, the Java compiler automatically converts primitive types into their corresponding wrapper class objects.

Example:

```
int a = 56;

Integer a1 = a;   // autoboxing
```

Autoboxing has a great advantage while working with Java collections.

```
import java.util.ArrayList;

class Main {

  public static void main(String[] args) {

    ArrayList<Integer> list = new ArrayList<Integer>();

    list.add(5);

    list.add(6);

    System.out.println("ArrayList: " + list);

  }

}
```

In the above example, we have created an array list of Integer type. Hence the array list can only hold objects of Integer type.

In line list.add(5), we are passing primitive type value. However, due to **autoboxing**, the primitive value is automatically converted into an Integer object and stored in the array list.

In **unboxing**, the Java compiler automatically converts wrapper class objects into their corresponding primitive types.

Example:

```
Integer a1 = 56;  // autoboxing

int a = a1;         // unboxing


import java.util.ArrayList;

class Main {

  public static void main(String[] args) {

    ArrayList<Integer> list = new ArrayList<Integer>();

    list.add(5);

    list.add(6);

    System.out.println("ArrayList: " + list);

    int a = list.get(0);

    System.out.println("Value at index 0: " + a);

  }

}
```

Here, the get() method returns the object at index 0. However, due to **unboxing**, the object is automatically converted into the primitive type int and assigned to the variable a.

java.util package:

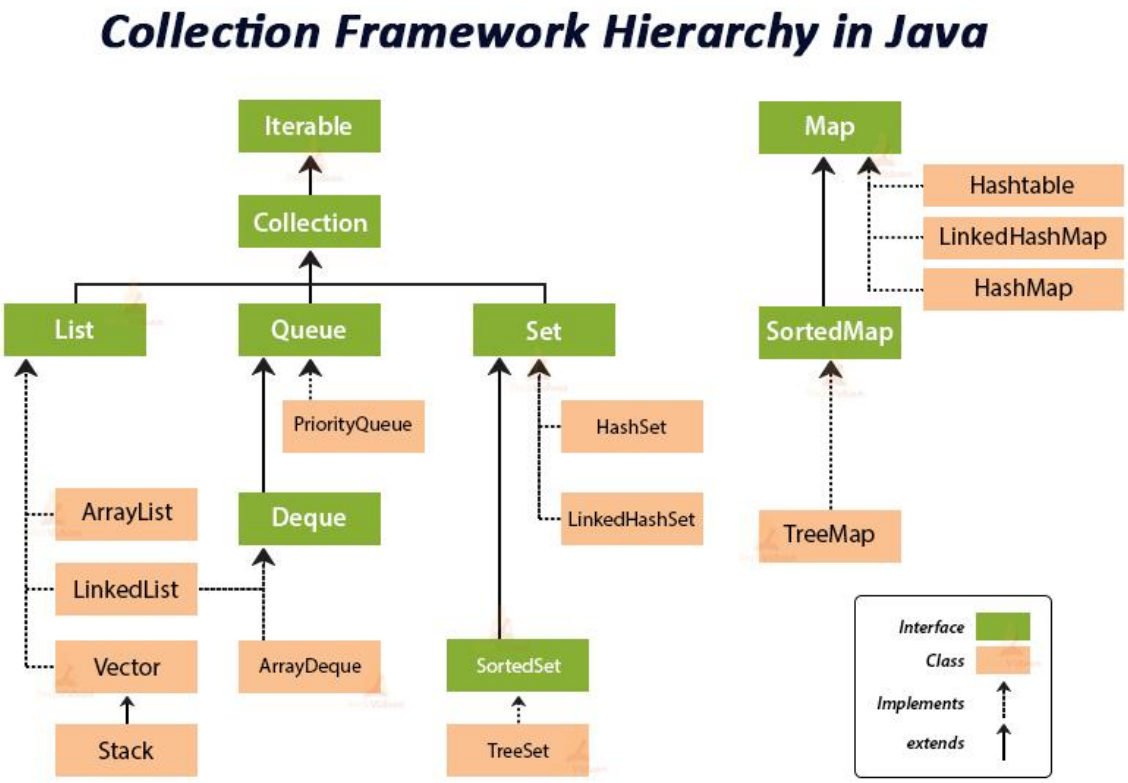
java.util package contains number of classes and interfaces with different functionalities.

java.util package contains important subsystem i.e., Collection Framework.

Collections Framework:

Collection Framework provides unified architecture for storing and manipulating group of objects.

The java collections framework contains List, Queue, Set, and Map as top-level interfaces. The List, Queue, and Set stores single value as its element, whereas Map stores a pair of a key and value as its element.



Collection Interface:

The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

Collection is a generic interface that has this declaration:

interface Collection<E>

Here, **E** specifies the type of objects that the collection will hold.

Method	Description
boolean add(E obj)	Adds *obj* to the invoking collection. Returns **true** if *obj* was added to the collection. Returns **false** if *obj* is already a member of the collection and the collection does not allow duplicates.
boolean addAll(Collection c)	Adds all the elements of *c* to the invoking collection. Returns **true** if the collection changed (i.e., the elements were added). Otherwise, returns **false**.
void clear()	Removes all elements from the invoking collection
boolean contains(Object obj)	Returns **true** if *obj* is an element of the invoking collection. Otherwise, returns **false**.
boolean containsAll(Collection c)	Returns **true** if the invoking collection contains all elements of *c*. Otherwise, returns **false**.
boolean equals(Object obj)	Returns **true** if the invoking collection and *obj* are

	equal. Otherwise, returns **false**.
boolean isEmpty()	Returns **true** if the invoking collection is empty. Otherwise, returns **false**.
Iterator<E> iterator()	Returns an iterator for the invoking collection.
boolean remove(Object obj)	Removes one instance of *obj* from the invoking collection. Returns **true** if the element was removed. Otherwise, returns **false**.
boolean removeAll(Collection c)	Removes all elements of *c* from the invoking collection. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
int size()	Returns the number of elements held in the invoking collection.
Object[] toArray()	Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.

List Interface:

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

List is a generic interface that has this declaration:

interface List<E>

Here, **E** specifies the type of objects that the list will hold.

In addition to the methods defined by **Collection**, **List** defines some of its own, which

are summarized in the following table.

Method	Description
void add(int index,E obj)	Inserts *obj* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
boolean addAll(int index, Collection c)	Inserts all elements of *c* into the invoking list at the index passed in *index*. Any pre existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns **true** if the invoking list changes and returns **false** otherwise.
E get(int index)	Returns the object stored at the specified index within the invoking collection.
int indexOf(Object obj)	Returns the index of the first instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.

int lastIndexOf(Object obj)	Returns the index of the last instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.
E remove(int index)	Removes the element at position *index* from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.
E set(int index, E obj)	Assigns *obj* to the location specified by *index* within the invoking list. Returns the old value.

ArrayList Class:

The **ArrayList** class extends **AbstractList** and implements the **List** interface. **ArrayList** is a generic class that has this declaration:

class ArrayList<E>

Here, **E** specifies the type of objects that the list will hold.

- **ArrayList** supports dynamic arrays that can grow as needed.
- In Java, standard arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that we must know in advance how many elements an array will hold. But, sometimes, you may not know until run time precisely how large an array you need. To handle this situation, the Collections Framework defines **ArrayList**.
- In essence, an **ArrayList** is a variable-length array of object references. That is, an **ArrayList** can dynamically increase or decrease in size.
- Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array can be shrunk.

ArrayList has the constructors shown here:
1)ArrayList( )
2)ArrayList(Collection *c*)
3)ArrayList(int *capacity*)
The first constructor builds an empty array list.
The second constructor builds an array list that is initialized with the elements of the collection c.
The third constructor builds an array list that has the specified initial *capacity*.

Example:

```
import java.util.*;
class ArrayListDemo
{
public static void main(String args[])
{
ArrayList<String> al = new ArrayList<String>();
System.out.println("Initial size of al: " +al.size());
```

```
al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D");
al.add("F");
al.add(1, "A2");
System.out.println("Size of al after additions: " +al.size());
System.out.println("Contents of al: " + al);
al.remove("F");
al.remove(2);
System.out.println("Size of al after deletions: " +al.size());
System.out.println("Contents of al: " + al);
}
}
```

LinkedList class:

The **LinkedList** class extends **AbstractSequentialList** and implements the **List**, **Deque**, and **Queue** interfaces. It provides a linked-list data structure.

LinkedList is a generic class that has this declaration:

```
class LinkedList<E>
```

Here, **E** specifies the type of objects that the list will hold.

LinkedList has the two constructors shown here:
1)LinkedList( )
2)LinkedList(Collection *c*)

The first constructor builds an empty linked list.

The second constructor builds a linked list that is initialized with the elements of the collection *c*.

Example:

```
import java.util.*;
class LinkedListDemo {
public static void main(String args[]) {

LinkedList<String> ll = new LinkedList<String>();
ll.add("F");
ll.add("B");
ll.add("D");
ll.add("E");
ll.add("C");
ll.addLast("Z");
ll.addFirst("A");
```

```
ll.add(1, "A2");
System.out.println("Original contents of ll: " + ll);
ll.remove("F");
ll.remove(2);
System.out.println("Contents of ll after deletion: "+ ll);
ll.removeFirst();
ll.removeLast();
System.out.println("ll after deleting first and last: "+ ll);
ll.set(2, “G”);
System.out.println("ll after change: " + ll);
}
}
```

Queue interface:

The **Queue** interface extends **Collection** and declares the behaviour of a queue, which is often a first-in, first-out list.

Queue is a generic interface that has this declaration:

interface Queue<E>

Here, **E** specifies the type of objects that the queue will hold.

Method	Description
E element()	Returns the element at the head of the queue. The element is not removed. It throws **NoSuchElementException** if the queue is empty.
boolean offer(E obj)	Attempts to add *obj* to the queue. Returns **true** if *obj* was added and **false** otherwise.
E peek()	Returns the element at the head of the queue. It returns **null** if the queue is empty. The element is not removed.
E poll()	Returns the element at the head of the queue, removing the element in the process. It returns **null** if the queue is empty.
E remove()	Removes the element at the head of the queue, returning the element in the process. It throws **NoSuchElementException** if the queue is empty.

Set interface:

The **Set** interface defines a set. It extends **Collection** and specifies the behavior of a collection that does not allow duplicate elements.

Set is a generic interface that has this declaration:

interface Set<E>

Here, **E** specifies the type of objects that the set will hold.

HashSet class:

HashSet extends **AbstractSet** and implements the **Set** interface. It creates a collection that uses a hash table for storage.

HashSet is a generic class that has this declaration:

class HashSet<E>

Here, **E** specifies the type of objects that the set will hold.

A hash table stores information by using a mechanism called hashing. In *hashing*, the informational content of a key is used to determine a unique value, called its *hash code*. The hash code is then used as the index at which the data associated with the key is stored.
The following constructors are defined:
HashSet( )
HashSet(Collection c)
HashSet(int *capacity*)
The first form constructs a default hash set. The second form initializes the hash set by using the elements of *c*. The third form initializes the capacity of the hash set to *capacity*. (The default capacity is 16.)

Example:

```
// Demonstrate HashSet.
import java.util.*;
class HashSetDemo {
public static void main(String args[]) {
HashSet<String> hs = new HashSet<String>();
hs.add("Beta");
hs.add("Alpha");
hs.add("Eta");
hs.add("Gamma");
hs.add("Epsilon");
hs.add("Omega");
System.out.println(hs);
}
}
```

Map interface:
The **Map** interface maps unique keys to values. A *key* is an object that you use to retrieve a value at a later date. Given a key and a value, you can store the value in a **Map** object. After the value is stored, you can retrieve it by using its key.

Map is generic and is declared as shown here:

```
interface Map<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values.

Method	Description
V put(Object key, Object value)	It is used to insert an entry in the map.
void putAll(Map map)	It is used to insert the specified map in the map.
V remove(Object key)	It is used to delete an entry for the specified key.

boolean remove(Object key, Object value)	It removes the specified values with the associated specified keys from the map.
Set keySet()	It returns the Set view containing all the keys.
void clear()	It is used to reset the map.
boolean containsValue(Object value)	This method returns true if some value equal to the value exists within the map, else return false.
boolean containsKey(Object key)	This method returns true if some key equal to the key exists within the map, else return false.
boolean equals(Object o)	It is used to compare the specified Object with the Map.
V get(Object key)	This method returns the object that contains the value associated with the key.
boolean isEmpty()	This method returns true if the map is empty; returns false if it contains at least one key.
V replace(K key, V value)	It replaces the specified value for a specified key.
boolean replace(K key, V oldValue, V newValue)	It replaces the old value with the new value for a specified key.
Collection values()	It returns a collection view of the values contained in the map.
int size()	This method returns the number of entries in the map.

HashMap class:

The **HashMap** class extends **AbstractMap** and implements the **Map** interface. It uses a hash table to store the map.
HashMap is a generic class that has this declaration:

class HashMap<K, V>

Here, **K** specifies the type of keys, and **V** specifies the type of values.

The following constructors are defined:
HashMap( )
HashMap(Map *m*)
HashMap(int *capacity*)

The first form constructs a default hash map. The second form initializes the hash map by using the elements of *m*. The third form initializes the capacity of the hash map to *capacity*.

Example:

```
import java.util.*;

class HashMapExample1{

 public static void main(String args[]){

   HashMap<Integer,String> map=new HashMap<Integer,String>();//Creating HashMap
     map.put(1,"Mango");

   map.put(2,"Apple");

   map.put(3,"Banana");

   map.put(4,"Grapes");

     System.out.println("Iterating Hashmap...");

   for(Map.Entry m : map.entrySet()){

    System.out.println(m.getKey()+" "+m.getValue());

   }

}

}
```