Inheritance in Java

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

The syntax of Java Inheritance

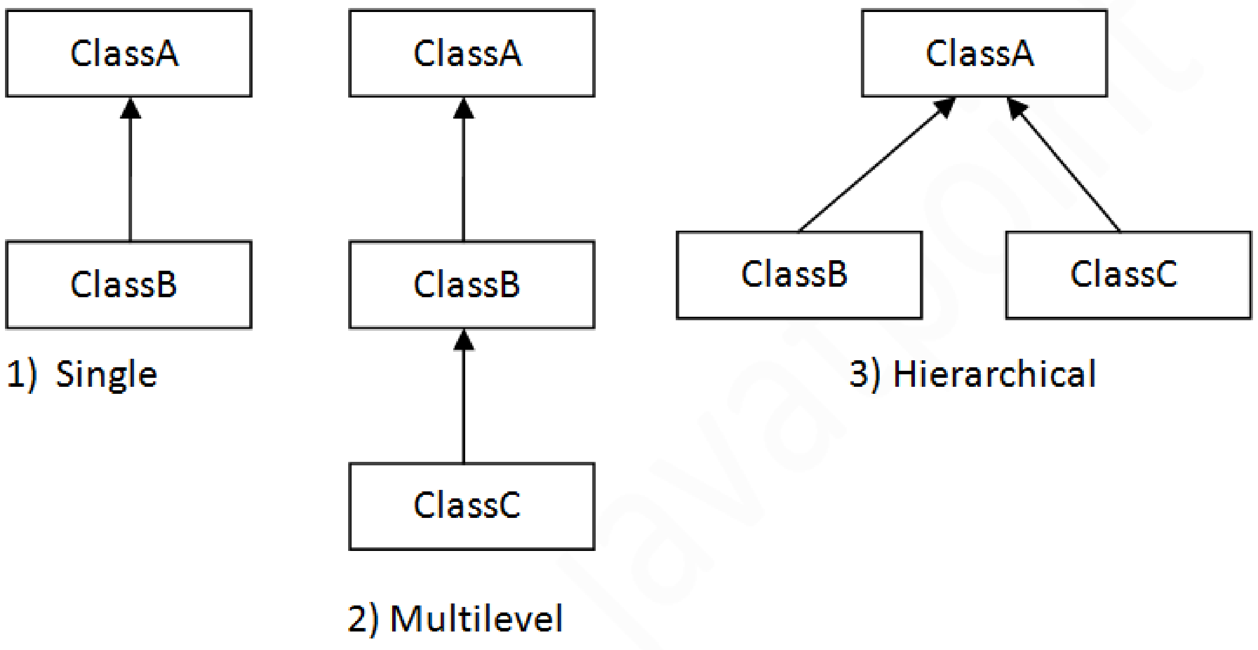
```
class Subclass-name extends Superclass-name
{
   //methods and fields
}
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, **multiple and hybrid inheritance** is supported through interface only.



Note: Multiple inheritance is not supported in Java through class.

Single level inheritance in Java

Single Level inheritance - A class inherits properties from a single class. For example, Class B inherits Class A.

```
class Shape {
  public void display() {
    System.out.println("Inside display");
  }
}
class Rectangle extends Shape {
  public void area() {
    System.out.println("Inside area");
  }
}
public class Tester {
  public static void main(String[] arguments) {
    Rectangle rect = new Rectangle();
    rect.display();
    rect.area();
  }
}
```

Output

```
Inside display
Inside area
```

Multilevel inheritance in Java

When there is a chain of inheritance, it is known as *multilevel inheritance*.

```
class Shape {
  public void display() {
    System.out.println("Inside display");
  }
}
class Rectangle extends Shape {
  public void area() {
    System.out.println("Inside area");
  }
}
class Cube extends Rectangle {
  public void volume() {
    System.out.println("Inside volume");
  }
}
public class Tester {
  public static void main(String[] arguments) {
    Cube cube = new Cube();
```

```
        cube.display();
        cube.area();
        cube.volume();
    }
}
```

```
Inside display
Inside area
Inside volume
```

Super keyword in Java

There are several reasons why we might need to use the **super** keyword in Java:

- Access the members of the superclass from within the subclass:
- Invoke the superclass constructor from within the subclass constructor.
- Resolve ambiguity between a field or method in the subclass and a field or method in the superclass.
- Access the static members of the superclass from within the subclass.

1).Use of super with constructors

- The super keyword can also be used to access the parent class constructor.
- One more important thing is that 'super' can call both parametric as well as non-parametric constructors depending on the situation.

```
class Person {
    Person()
    {
        System.out.println("Person class Constructor");
    }
}

class Student extends Person {
    Student()
    {
        // invoke or call parent class constructor
        super();

        System.out.println("Student class Constructor");
    }
}

class Test {
    public static void main(String[] args)
    {
        Student s = new Student();
    }
}
```

Output

```
Person class Constructor

Student class Constructor
```

2.Use of super with Methods

- This is used when we want to call the parent class method.
- So whenever a parent and child class have the same-named methods then to resolve ambiguity we use the super keyword.

```
// superclass Person
class Person {
    void message()
    {
        System.out.println("This is person class\n");
    }
}
// Subclass Student
class Student extends Person {
    void message()
    {
        super.message();
        System.out.println("This is student class");
    }


}
// Driver Program
class Test {
    public static void main(String args[])
    {
        Student s = new Student();
        s.message();
    }
    }
```

Output

```
This is person class


This is student class
```

3.Use of super with Variables

- This scenario occurs when a derived class and base class has the same data members.
- In that case, there is a possibility of ambiguity for the JVM.

```
// Base class vehicle

class Vehicle {

        int maxSpeed = 120;
```

```
}

// sub class Car extending vehicle
class Car extends Vehicle {
        int maxSpeed = 180;

        void display()
        {
                // print maxSpeed of base class (vehicle)
                System.out.println("Maximum Speed: "
                                                + super.maxSpeed);
        }
}

// Driver Program
class Test {
        public static void main(String[] args)
        {
                Car small = new Car();
                small.display();
        }
```

```
}
```

Method Overriding in Java

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.
- In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Rules for Java Method Overriding

1. The method must have the <u>same name as in the parent class</u>
2. The method must have the <u>same parameter as in the parent class.</u>
3. There must be an <u>IS-A relationship (inheritance).</u>

```
class Bank{
int getRateOfInterest(){return 0;}
}
//Creating child classes.
class SBI extends Bank{
int getRateOfInterest(){return 8;}
}

class ICICI extends Bank{
int getRateOfInterest(){return 7;}
}
class AXIS extends Bank{
int getRateOfInterest(){return 9;}
}
//Test class to create objects and call the methods
class Test2{
public static void main(String args[]){
SBI s=new SBI();
ICICI i=new ICICI();
AXIS a=new AXIS();
System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
}
}
```

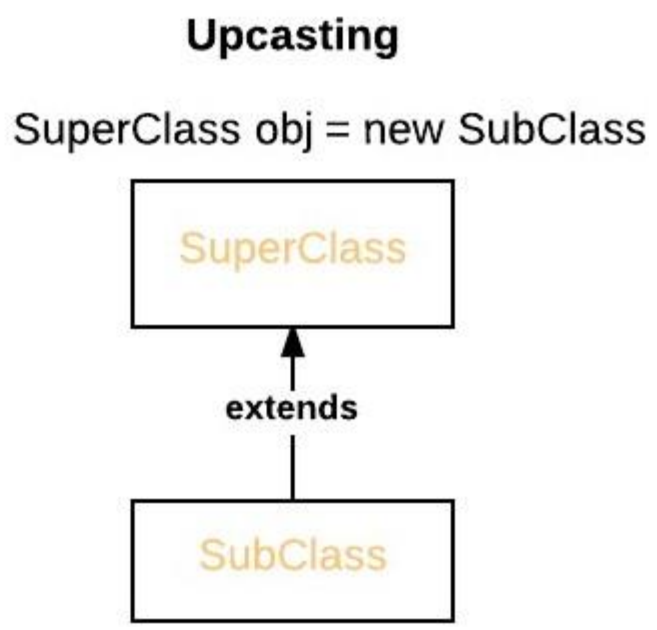
Test it Now

```
Output:
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

Dynamic Method Dispatch or Runtime Polymorphism in Java

- Method overriding is one of the ways in which Java supports Runtime Polymorphism.
- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

- When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.
- At run-time, it depends on the type of the object being referred to that determines which version of an overridden method will be executed
- A superclass reference variable can refer to a subclass object. This is also known as upcasting. Java uses this fact to resolve calls to overridden methods at run time.



Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed. Here is an example that illustrates dynamic method dispatch:

```
// A Java program to illustrate Dynamic Method

// Dispatch using hierarchical inheritance

class A

{

        void m1()

        {

                System.out.println("Inside A's m1 method");

        }

}


class B extends A

{

        // overriding m1()
        void m1()
        {
                System.out.println("Inside B's m1 method");
        }
}

class C extends A
{
        // overriding m1()
        void m1()
        {
                System.out.println("Inside C's m1 method");
        }
}

// Driver class
class Dispatch
{
        public static void main(String args[])
        {
                // object of type A
                A a = new A();
       a.m1();
        A ref;
        ref=new B();
        ref.m1();
        ref=new C();
        ref.m1();


}
}
```

Output:

```
Inside A's m1 method
Inside B's m1 method
Inside C's m1 method
```

Abstract class in Java

A class which is declared with the abstract keyword is known as an abstract class in Java.

- 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.
- **Syntax:**
- **abstract type method_name (parameters);**

  The class is declared as abstract as follows
- **Syntax:**
- **abstract class class_name**
- **{**
- **- - -**

- **abstract type method_name1 (parameter); // abstract Method**

type method_name2 (parameter);// Other Concrete Method

}

Example:

```
abstract class Shape{
    abstract void draw();
    }
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
    }
    class Circle1 extends Shape{
    void draw(){System.out.println("drawing circle");}
    }
    class TestAbstraction1{
    public static void main(String args[]){
    Circle1 s=new Circle1();
    s.draw();
     Rectangle r1=new Rectangle();
     r1.draw();
    }
}
```

Interface in Java

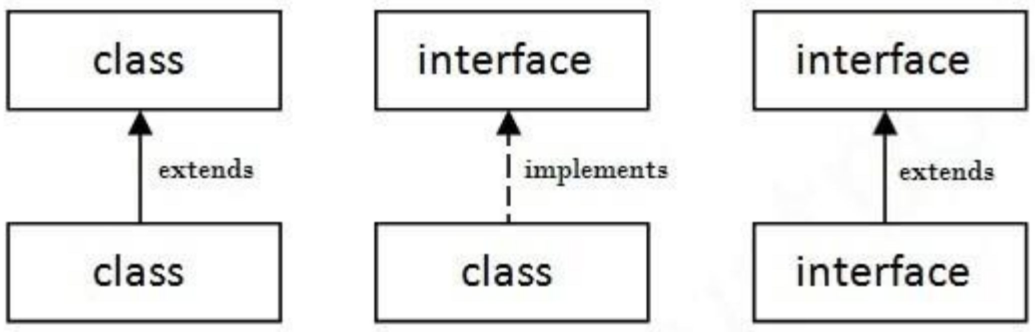
- An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.
- The interface in Java is *a mechanism to achieve abstraction*.
- There can be only abstract methods in the Java interface, not method body.
- It is used to achieve abstraction and multiple inheritance in Java.
- Java Interface also **represents the IS-A relationship**.

Syntax:

```
interface <interface_name>{

    // declare constant fields
    // declare methods that abstract
    // by default.
}
```

The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



Java Interface Example

In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.

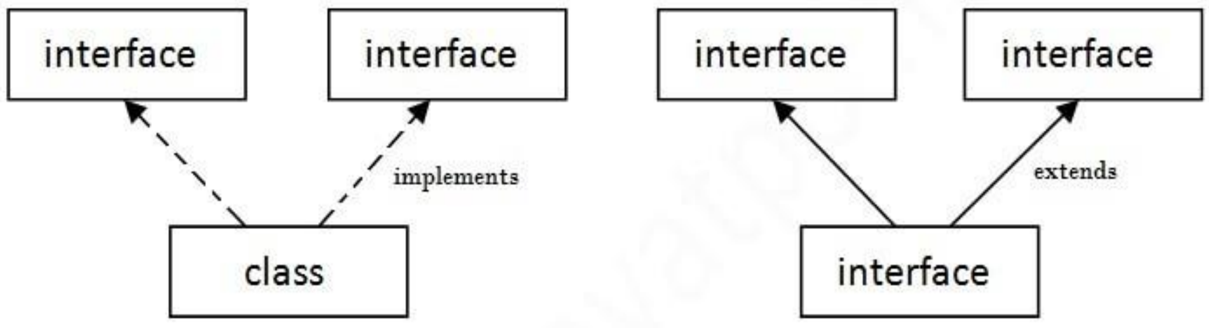
```
interface printable{
void print();
}
class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
        }
       }
```

Output:

```
Hello
```

Multiple inheritance in Java by interface

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



Multiple Inheritance in Java

```
interface Printable{
void print();
}
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

    public static void main(String args[]){
    A7 obj = new A7();
    obj.print();
    obj.show();
     }
    }
```

```
Output:Hello
       Welcome
```

Interface inheritance

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

```
interface Printable{
```

```
void print();
}
interface Showable extends Printable{
void show();
}
class TestInterface4 implements Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

        public static void main(String args[]){
        TestInterface4 obj = new TestInterface4();
        obj.print();
        obj.show();
         }
        }
```

Output:

```
Hello
Welcome
```

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.

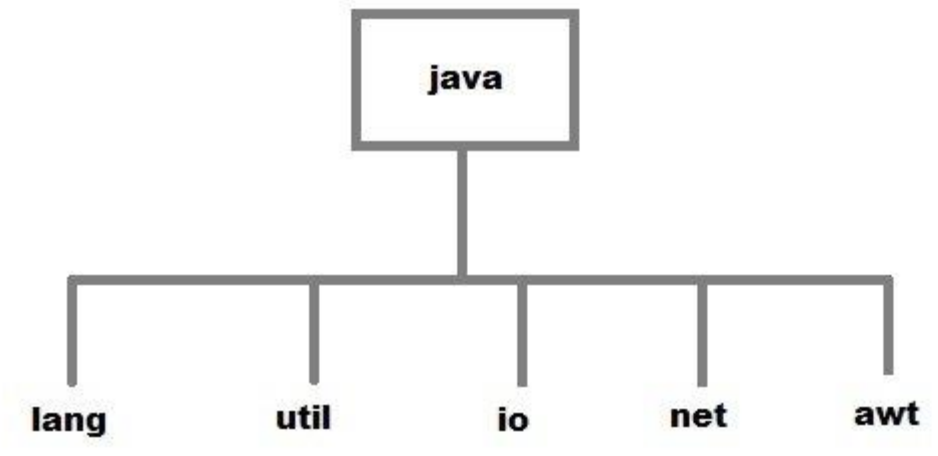
9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Java Package

- Package is a collection of related classes.
- Java uses package to group related classes, interfaces and sub-packages in any Java project.
- We can assume package as a folder or a directory that is used to store similar files.

Types Of Java Package

- **Built-in Package:** math, util, lang, i/o etc are the example of built-in packages.
- **User-defined-package:** Java package created by user to categorize their project's classes and interface are known as user-defined packages.



Additional points about package:

- Package statement must be first statement in the program even before the import statement.
- A package is always defined as a separate folder having the same name as the package name.
- Store all the classes in that package folder.
- All classes of the package which we wish to access outside the package must be declared public.
- All classes within the package must have the package statement as its first line.
- All classes of the package must be compiled before use.

Simple example of java package

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

```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:

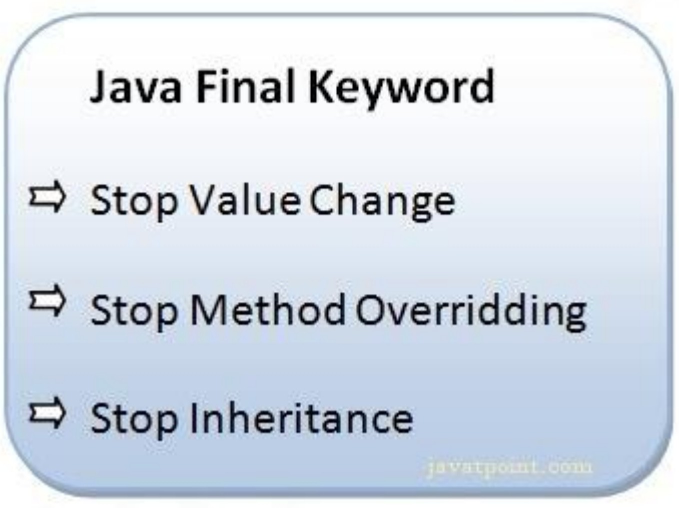
```
javac -d directory javafilename
Example
javac -d . Simple.java
```

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

Final Keyword In Java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class



1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example of final variable

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
}/
Output:Compile Time Error
```

2) Java final method

If you make any method as final, you cannot override it.

Example of final method

```
class Bike{
  final void run(){System.out.println("running");}
}

class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}

   public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
   }
}
```

```
Output:Compile Time Error
```

3) Java final class

If you make any class as final, you cannot extend it.

Example of final class

```
final class Bike{}

class Honda1 extends Bike{
```

```
void run(){System.out.println("running safely with 100kmph");}

public static void main(String args[]){
Honda1 honda= new Honda1();
honda.run();
}
}
```

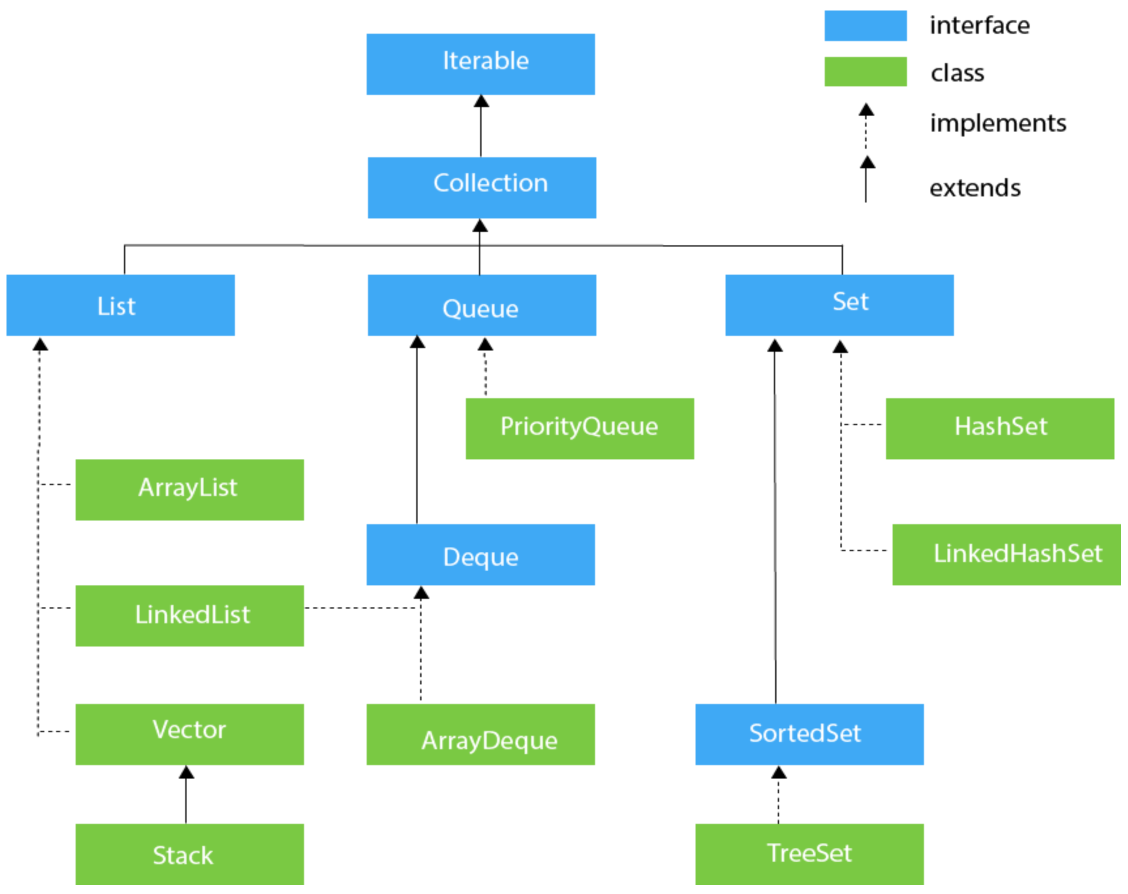
```
Output:Compile Time Error
```

Collections in Java

- The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.
- Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.
- Java Collection means a single unit of objects.
- Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

Hierarchy of Collection Framework

The **java.util** package contains all the classes and interfaces for the Collection framework.



Collection interface

- The Collection interface is the foundation upon which the collections framework is built.
- It declares the core methods that all collections will have.

Sr.No.	Method & Description
1	**boolean add(Object 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, or if the collection does not allow duplicates.
2	**boolean addAll(Collection c)** Adds all the elements of c to the invoking collection. Returns true if the operation succeeds (i.e., the elements were added).

	Otherwise, returns false.
3	**void clear( )** Removes all elements from the invoking collection.
4	**boolean contains(Object obj)** Returns true if obj is an element of the invoking collection. Otherwise, returns false.
5	**boolean containsAll(Collection c)** Returns true if the invoking collection contains all elements of **c**. Otherwise, returns false.
6	**boolean equals(Object obj)** Returns true if the invoking collection and obj are equal. Otherwise, returns false.
7	**int hashCode( )** Returns the hash code for the invoking collection.
8	**boolean isEmpty( )** Returns true if the invoking collection is empty. Otherwise, returns false.
9	**Iterator iterator( )** Returns an iterator for the invoking collection.
10	**boolean remove(Object obj)** Removes one instance of obj from the invoking collection. Returns true if the element was removed. Otherwise, returns false.

11	**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.
12	**boolean retainAll(Collection c)** Removes all elements from the invoking collection except those in c. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false.
13	**int size( )** Returns the number of elements held in the invoking collection.

List interface

The List interface extends **Collection** and declares the behavior of a collection that stores a sequence of elements.

- Elements can be inserted or accessed by their position in the list, using a zero-based index.
- A list may contain duplicate elements.

Sr.No.	Method & Description
1	**void add(int index, Object obj)** Inserts obj into the invoking list at the index passed in the index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
2	**boolean addAll(int index, Collection c)** Inserts all elements of **c** into the invoking list at the index passed in

	the 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.
3	**Object get(int index)** Returns the object stored at the specified index within the invoking collection.
4	**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.
5	**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.
6	**ListIterator listIterator( )** Returns an iterator to the start of the invoking list.
7	**ListIterator listIterator(int index)** Returns an iterator to the invoking list that begins at the specified index.
8	**Object 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.
9	**Object set(int index, Object obj)** Assigns obj to the location specified by index within the invoking list.

10	**List subList(int start, int end)** Returns a list that includes elements from start to end.1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

Set interface

- A Set is a Collection that cannot contain duplicate elements.
- It models the mathematical set abstraction.
- The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited.
- Set also adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ.

Sr.No.	Method & Description
1	**add( )** Adds an object to the collection.
2	**clear( )** Removes all objects from the collection.
3	**contains( )** Returns true if a specified object is an element within the collection.
4	**isEmpty( )**

	Returns true if the collection has no elements.
5	**iterator( )** Returns an Iterator object for the collection, which may be used to retrieve an object.
6	**remove( )** Removes a specified object from the collection.
7	**size( )** Returns the number of elements in the collection.

Navigation:Enumeration,Iterator,ListIterator

- All Iterator, ListIterator, and Enumeration represents cursors in java which are used to iterate over collection elements. But they have some differences also.

Iterator

- Iterator interface can be applied to all collection classes but it can traverse only in single direction that is Forward direction.
- The iterator() method of Collection interface is used to get the Iterator interface.

The Methods Declared by Iterator

Sr.No.	Method & Description
1	**boolean hasNext( )**Returns true if there are more elements. Otherwise, returns false.
2	**Object next( )**Returns the next element..
3	**void remove( )**Removes the current element

Iterator iterator = collection.iterator();

Example

```
import java.util.ArrayList;
import java.util.Iterator;

public class Main {

    public static void main(String[] args) {

        // Creating ArrayList
        ArrayList subjects = new ArrayList();

        // Adding elements to ArrayList
        subjects.add("Java");
        subjects.add("Servlet");
        subjects.add("JSP");
        subjects.add("Spring");
        subjects.add("Hibernet");

        // Get Iterator
        Iterator iterator = subjects.iterator();

        // Print ArrayList elements
        while (iterator.hasNext()){
            System.out.println(iterator.next());
        }
    }
}
```

Output

```
Java
Servlet
JSP
Spring
Hibernet
```

ListIterator

- Iterator interface can be applicable only on List objects like ArrayList or LinkedList but it can traverse in both directions that is Forward as well as Backward direction.
- The listIterator() method of List interface is used to get the ListIterator object.

The Methods Declared by ListIterator

Sr.No.	Method & Description
1	**void add(Object obj)**Inserts obj into the list in front of the element that will be returned by the next call to next( ).
2	**boolean hasNext( )**Returns true if there is a next element. Otherwise, returns false.
3	**boolean hasPrevious( )**Returns true if there is a previous element. Otherwise, returns false.

Sr.No.	Method & Description
4	**Object next( )**Returns the next element.
5	**int nextIndex( )**Returns the index of the next element. If there is not a next element, returns the size of the list.
6	**Object previous( )**Returns the previous element
7	**int previousIndex( )**Returns the index of the previous element. If there is not a previous element, returns -1.
8	**void remove( )**Removes the current element from the list
9	**void set(Object obj)**Assigns obj to the current element.

ListIterator listIterator = list.listIterator();

Example

```
import java.util.ArrayList;
import java.util.ListIterator;

public class Main {

    public static void main(String[] args) {

        // Creating ArrayList
        ArrayList subjects = new ArrayList();

        // Adding elements to ArrayList
        subjects.add("Java");
        subjects.add("Servlet");
        subjects.add("JSP");
        subjects.add("Spring");
        subjects.add("Hibernet");

        // Get ListIterator
        ListIterator listIterator = subjects.listIterator();

        // Print ArrayList elements in Forward Direction
        System.out.println("ArrayList elements in Forward Direction");
        while (listIterator.hasNext()){
            System.out.println(listIterator.next());
        }

        // Print ArrayList elements in Backward Direction
        System.out.println("ArrayList elements in Backward Direction");
        while (listIterator.hasPrevious()){
            System.out.println(listIterator.previous());
        }
    }
}
```

Output

```
ArrayList elements in Forward Direction
Java
Servlet
JSP
Spring
Hibernet
ArrayList elements in Backward Direction
Hibernet
Spring
JSP
Servlet
Java
```

Enumeration

- Iterator interface can be applied to Vector, Hashtable, Stack, Properties and Dictionary abstract class and it can traverse only in Forward direction.
- We can not perform any operation other then read or get.

Vector vector = new Vector();

Enumeration enumeration = vector.elements();

Example

```
import java.util.Enumeration;
import java.util.Vector;

public class Main {

    public static void main(String[] args) {

        // Creating Vector
        Vector subjects = new Vector();

        // Adding elements to Vector
        subjects.addElement("Java");
        subjects.addElement("Servlet");
        subjects.addElement("JSP");
        subjects.addElement("Spring");
        subjects.addElement("Hibernet");

        // Get Enumeration
        Enumeration enumeration = subjects.elements();

        // Print Vector elements
        while (enumeration.hasMoreElements()){
            System.out.println(enumeration.nextElement());
```

```
      }

   }
}
```

Output

```
Java
Servlet
JSP
Spring
Hibernet
```

Classes: Linked List ,Array List ,Vector, HashSet

Linked List

- Java LinkedList class uses a doubly linked list to store the elements.
- It provides a linked-list data structure.
- It inherits the AbstractList class and implements List and Deque interfaces.

The important points about Java LinkedList are:

- Java LinkedList class can contain duplicate elements.
- Java LinkedList class maintains insertion order.
- Java LinkedList class is non synchronized.
- In Java LinkedList class, manipulation is fast because no shifting needs to occur.
- Java LinkedList class can be used as a list, stack or queue.

Following are the constructors supported by the LinkedList class.

Sr.No.	Constructor & Description
1	**LinkedList( )** This constructor builds an empty linked list.
2	**LinkedList(Collection c)** This constructor builds a linked list that is initialized with the elements of the collection **c**.

Apart from the methods inherited from its parent classes, LinkedList defines following methods −

Sr.No.	Method & Description
1	**void add(int index, Object element)** Inserts the specified element at the specified position index in this list.
2	**boolean add(Object o)** Appends the specified element to the end of this list.
3	**boolean addAll(Collection c)** Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator
4	**boolean addAll(int index, Collection c)** Inserts all of the elements in the specified collection into this list, starting at the specified position.
5	**void addFirst(Object o)**

	Inserts the given element at the beginning of this list.
6	**void addLast(Object o)** Appends the given element to the end of this list.
10	**Object get(int index)** Returns the element at the specified position in this list.
11	**Object getFirst()** Returns the first element in this list.
12	**Object getLast()** Returns the last element in this list.
16	**Object remove(int index)** Removes the element at the specified position in this list
17	**boolean remove(Object o)** Removes the first occurrence of the specified element in this list.
18	**Object removeFirst()** Removes and returns the first element from this list.
19	**Object removeLast()** Removes and returns the last element from this list.

Array List

- Java **ArrayList** class uses a *dynamic* *array* for storing the elements.
- It is like an array, but there is *no size limit*. We can add or remove elements anytime.
- So, it is much more flexible than the traditional array. It is found in the *java.util* package. It is like the Vector in C++.
- The ArrayList in Java can have the duplicate elements also.
- It implements the List interface so we can use all the methods of the List interface here. The ArrayList maintains the insertion order internally.
- It inherits the AbstractList class and implements List interface.

The important points about the Java ArrayList class are:

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because the array works on an index basis.
- In ArrayList, manipulation is a little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.
- We can not create an array list of the primitive types, such as int, float, char, etc. It is required to use the required wrapper class in such cases. For example:

is the list of the constructors provided by the ArrayList class.

Sr.No.	Constructor & Description
1	**ArrayList( )** This constructor builds an empty array list.

2	**ArrayList(Collection c)** This constructor builds an array list that is initialized with the elements of the collection **c**.
3	**ArrayList(int capacity)** This constructor builds an array list that has the specified initial capacity. The capacity grows automatically as elements are added to an array list.

Example

The following program illustrates several of the methods supported by ArrayList −

```
import java.util.*;
public class ArrayListDemo {

   public static void main(String args[]) {
      // create an array list
      ArrayList al = new ArrayList();
      System.out.println("Initial size of al: " + al.size());

      // add elements to the array list
      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());

      // display the array list
      System.out.println("Contents of al: " + al);

      // Remove elements from the array list
      al.remove("F");
      al.remove(2);
      System.out.println("Size of al after deletions: " + al.size());
      System.out.println("Contents of al: " + al);
   }
}
```

This will produce the following result −

Output

```
Initial size of al: 0
Size of al after additions: 7
Contents of al: [C, A2, A, E, B, D, F]
Size of al after deletions: 5
Contents of al: [C, A2, E, B, D]
```

HashSet

Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

The important points about Java HashSet class are:

- HashSet stores the elements by using a mechanism called **hashing.**
- HashSet contains unique elements only.
- HashSet allows null value.
- HashSet class is non synchronized.
- HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
- HashSet is the best approach for search operations.

Following is the list of constructors provided by the HashSet class.

Sr.No.	Constructor & Description

1	**HashSet( )** This constructor constructs a default HashSet.
2	**HashSet(Collection c)** This constructor initializes the hash set by using the elements of the collection **c**.
3	**HashSet(int capacity)** This constructor initializes the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet.
4	**HashSet(int capacity, float fillRatio)** This constructor initializes both the capacity and the fill ratio (also called load capacity) of the hash set from its arguments.

Apart from the methods inherited from its parent classes, HashSet defines following methods −

Sr.No.	Method & Description
1	**boolean add(Object o)** Adds the specified element to this set if it is not already present.
2	**void clear()** Removes all of the elements from this set.
3	**Object clone()** Returns a shallow copy of this HashSet instance: the elements

	themselves are not cloned.
4	**boolean contains(Object o)** Returns true if this set contains the specified element.
5	**boolean isEmpty()** Returns true if this set contains no elements.
6	**Iterator iterator()** Returns an iterator over the elements in this set.
7	**boolean remove(Object o)** Removes the specified element from this set if it is present.
8	**int size()** Returns the number of elements in this set (its cardinality).

Example

The following program illustrates several of the methods supported by HashSet −

```
import java.util.*;
public class HashSetDemo {

   public static void main(String args[]) {
      // create a hash set
      HashSet hs = new HashSet();

      // add elements to the hash set
      hs.add("B");
      hs.add("A");
      hs.add("D");
      hs.add("E");
      hs.add("C");
      hs.add("F");
      System.out.println(hs);
   }
}
```

This will produce the following result −

Output

```
[A, B, C, D, E, F]
```

Vector

Vector implements a dynamic array. It is similar to ArrayList, but with two differences −

- Vector is synchronized.
- Vector contains many legacy methods that are not part of the collections framework.

Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program.

Following is the list of constructors provided by the vector class.

Sr.No.	Constructor & Description
1	**Vector( )** This constructor creates a default vector, which has an initial size of 10.
2	**Vector(int size)** This constructor accepts an argument that equals to the required size, and creates a vector whose initial capacity is specified by size.

3	**Vector(int size, int incr)** This constructor creates a vector whose initial capacity is specified by size and whose increment is specified by incr.
4	**Vector(Collection c)** This constructor creates a vector that contains the elements of collection c.

Apart from the methods inherited from its parent classes, Vector defines the following methods −

Sr.No.	Method & Description
1	**void addElement(Object obj)** Adds the specified component to the end of this vector, increasing its size by one.
2	**int capacity()** Returns the current capacity of this vector.
3	**boolean contains(Object elem)** Tests if the specified object is a component in this vector.
4	**boolean containsAll(Collection c)** Returns true if this vector contains all of the elements in the specified Collection.
5	**void copyInto(Object[] anArray)** Copies the components of this vector into the specified array.

6	**Object elementAt(int index)** Returns the component at the specified index.
7	**Enumeration elements()** Returns an enumeration of the components of this vector.
8	**void ensureCapacity(int minCapacity)** Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.
11	**Object get(int index)** Returns the element at the specified position in this vector.
12	**void removeElementAt(int index)** removeElementAt(int index).
13	**Object set(int index, Object element)** Replaces the element at the specified position in this vector with the specified element.
14	**void setElementAt(Object obj, int index)** Sets the component at the specified index of this vector to be the specified object.
15	**void setSize(int newSize)** Sets the size of this vector.
16	**int size()**

	Returns the number of components in this vector.

Example

The following program illustrates several of the methods supported by this collection

```
import java.util.*;
public class VectorDemo {

   public static void main(String args[]) {
      // initial size is 3, increment is 2
      Vector v = new Vector(3, 2);
      System.out.println("Initial size: " + v.size());
      System.out.println("Initial capacity: " + v.capacity());

      v.addElement(new Integer(1));
      v.addElement(new Integer(2));
      v.addElement(new Integer(3));
      v.addElement(new Integer(4));
      System.out.println("Capacity after four additions: " +
v.capacity());

      v.addElement(new Double(5.45));
      System.out.println("Current capacity: " + v.capacity());

      v.addElement(new Double(6.08));
      v.addElement(new Integer(7));
      System.out.println("Current capacity: " + v.capacity());

      v.addElement(new Float(9.4));
      v.addElement(new Integer(10));
      System.out.println("Current capacity: " + v.capacity());

      v.addElement(new Integer(11));
      v.addElement(new Integer(12));
      System.out.println("First element: " + (Integer)v.firstElement());
      System.out.println("Last element: " + (Integer)v.lastElement());
      if(v.contains(new Integer(3)))
         System.out.println("Vector contains 3.");

      // enumerate the elements in the vector.
      Enumeration vEnum = v.elements();
      System.out.println("\nElements in vector:");

      while(vEnum.hasMoreElements())
         System.out.print(vEnum.nextElement() + " ");
      System.out.println();
   }
}
```

This will produce the following result −

Output

```
Initial size: 0
Initial capacity: 3
Capacity after four additions: 5
Current capacity: 5
Current capacity: 7
Current capacity: 9
First element: 1
Last element: 12
Vector contains 3.

Elements in vector:
1 2 3 4 5.45 6.08 7 9.4 10 11 12
```