Collections in Java

- The Collection in Java is a framework that provides an architecture to store and manipulate

  the group of objects.

- All the operations that you perform on a data such as searching, sorting, insertion,

  manipulation, deletion, etc. can be achieved by Java Collections.

- Java Collection means a single unit of objects. Java Collection framework provides many

  interfaces (Set, List, Queue, Deque, etc.) and classes (ArrayList, Vector, LinkedList,

PriorityQueue, HashSet, LinkedHashSet, TreeSet, etc.).

What is Collection in java:

A Collection represents a single unit of objects, i.e., a group.

What is a framework in Java:

- It provides readymade architecture.
- It represents a set of classes and interface.
- It is optional.

What is Collection framework:

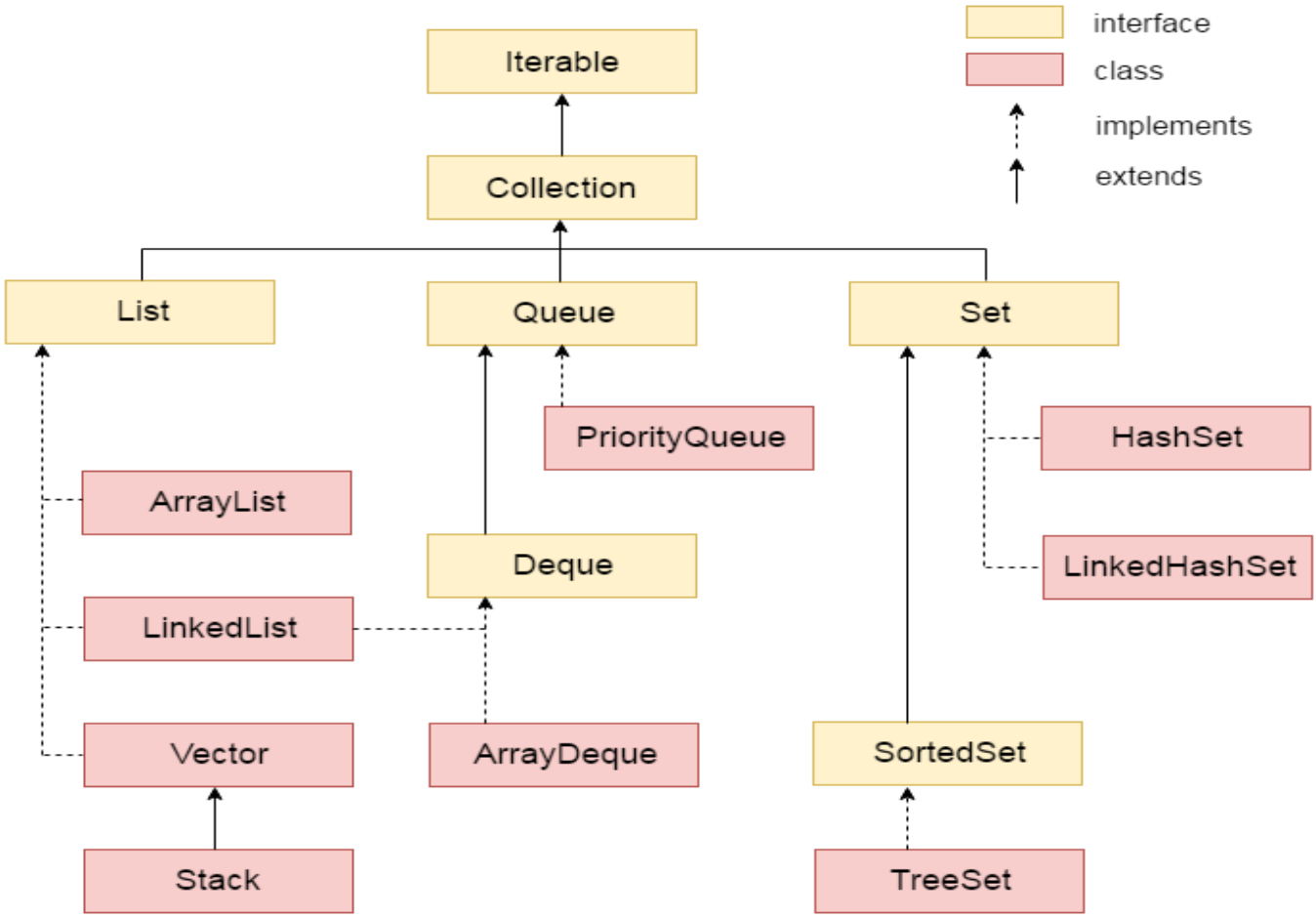
- The Collection framework represents a unified architecture for storing and manipulating

  a group of objects. It has:

- Interfaces and its implementations, i.e., classes
- Algorithm

Hierarchy of Collection Framework:

- Let us see the hierarchy of Collection framework. The java.util package contains all the classes and interfaces for Collection framework.



Methods of Collection interface:

- There are many methods declared in the Collection interface. They are as follows:

No.	Method	Description
1	public boolean add(Object element)	is used to insert an element in this collection.
2	public booleanaddAll(Collection c)	is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	is used to delete an element from this collection.
No.	Method	Description
4	public booleanremoveAll(Collection c)	is used to delete all the elements of specified collection from the invoking collection.
5	public booleanretainAll(Collection c)	is used to delete all the elements of invoking collection except the specified collection.
6	public int size()	return the total number of elements in the collection.
7	public void clear()	removes the total no. of

		elements from the collection.
8	**public boolean contains(Object element)**	**is used to search an element.**
9	**public booleancontainsAll(Collection c)**	**is used to search the specified collection in this collection.**
10	**public Iterator iterator()**	**returns an iterator.**
11	**public Object[] toArray()**	**converts collection into array.**
12	**public booleanisEmpty()**	**checks if collection is empty.**
13	**public boolean equals(Object element)**	**matches two collections.**
14	**public int hashCode()**	**returns the hash code number of the collection.**

Methods of Iterator interface:

- There are only three methods in the Iterator interface. They are:

No.	Method	Description
1	**public Boolean hasNext()**	**It returns true if the iterator has more elements otherwise it returns false.**

2	**public Object next()**	**It returns the element and moves the cursor pointer to the next element.**
3	**public void remove()**	**It removes the last elements returned by the iterator. It is less used.**

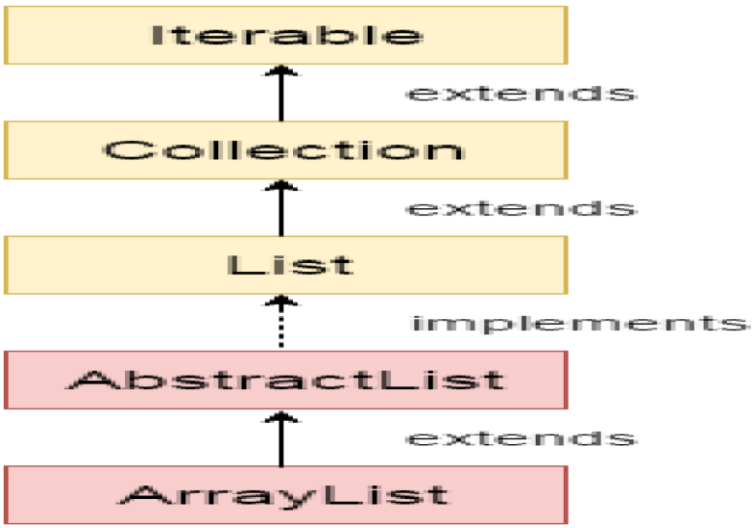
ArrayList class

Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.

The important points about 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 array works at the index basis.
- In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

Hierarchy of ArrayList



As shown in above diagram, Java ArrayList class extends AbstractList class which implements List interface. The List interface extends Collection and Iterable interfaces in hierarchical order.

ArrayList class declaration

Let's see the declaration for java.util.ArrayList class.

Syntax

publicclass ArrayList<E>**extends** AbstractList<E>**implements** List<E>, RandomAcce
ss, Cloneable, Serializable

Constructors of ArrayList

Constructor	**Description**
ArrayList()	It is used to build an empty array list.
ArrayList(Collection c)	It is used to build an array list that is initialized with the elements of the collection c.
ArrayList(int capacity)	It is used to build an array list that has the specified initial capacity.

Methods of ArrayList

Method	**Description**
void add(int index, Object element)	It is used to insert the specified element at the specified position index in a list.

boolean addAll(Collection c)	It is used to append 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.
void clear()	It is used to remove all of the elements from this list.
int lastIndexOf(Object o)	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
Object[] toArray()	It is used to return an array containing all of the elements in this list in the correct order.
Object[] toArray(Object[] a)	It is used to return an array containing all of the elements in this list in the correct order.
boolean add(Object o)	It is used to append the specified element to the end of a list.
boolean addAll(int index, Collection c)	It is used to insert all of the elements in the specified collection into this list, starting at the specified position.
Object clone()	It is used to return a shallow copy of an ArrayList.
int indexOf(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.

void trimToSize()	It is used to trim the capacity of this ArrayList instance to be the list's current size.

Java Non-generic vs Generic Collection

- Java collection framework was non-generic before JDK 1.5. Since 1.5, it is generic.
- Java new generic collection allows you to have only one type of object in collection. Now it is type safe so typecasting is not required at run time.

Non-generic collection

Example of creating java collection

```
ArrayList al=new ArrayList();
```

Generic collection

Example of creating java collection

```
ArrayList<String> al=new ArrayList<String>();
```

- In generic collection, we specify the type in angular braces.
- Now ArrayList is forced to have only specified type of objects in it.
- If you try to add another type of object, it gives *compile time error*.

Two ways to iterate the elements of collection in java

There are two ways to traverse collection elements:

1. By Iterator interface.
2. By for-each loop.

Example using iterator interface

```
import java.util.*;

class TestCollection1

{
    public static void main(String args[])
    {
        ArrayList<String> list=new
        ArrayList<String>();//Creating arraylist
        list.add("Ravi");//Adding object in arraylist
        list.add("Vijay");
        list.add("Ravi");
        list.add("Ajay");
```

```
        //Traversing list through Iterator
        Iterator itr=list.iterator();
        while(itr.hasNext())
            {
                System.out.println(itr.next());
            }
      }
}
```

Output

```
Ravi
Vijay
Ravi
Ajay
```

Example using for-each loop

```
import java.util.*;
classTestCollectio
n2
{
      public static void main(String args[])
     {
```

```
 ArrayList<String>  al=new
ArrayList<String>();
al.add("Ravi");
al.add("Vijay");
al.add("Ravi");
al.add("Ajay");
for(String obj:al)
        System.out.println(obj);
} }
```

Output

Ravi

Vijay

Ravi

Ajay

User-defined class objects in Java

ArrayList

```
class Student

{
 int rollno;

String name;

int age;

        Student(int rollno,String name,int age)
        {
                this.rollno=rollno;
        this.name=name;
        this.age=age;

        }
}


import java.util.*;

public class
TestCollection3
```

```
{
	public static void main(String args[])
	{
		//Creating user-defined class objects
	  Student s1=new Student(101,"Sonoo",23);
		Student s2=new Student(102,"Ravi",21);  Student s2=new Student(103,"Hanumat",25);
		//creating arraylist
		ArrayList<Student> al=new ArrayList<Student>();
		al.add(s1);//adding Student class object
		al.add(s2);
			al.add(s3);
		//Getting Iterator
		Iterator itr=al.iterator();
		//traversing elements of ArrayList object
				while(itr.hasNext())
		{
			Student st=(Student)itr.next();
			  System.out.println(st.rollno+" "+st.name+" "+st.age);
		}
	}
}
```

Output

101 Sonoo 23

102 Ravi 21

103 Hanumat 25

Example of addAll(Collection c)

```
import java.util.*;

class

TestCollection4

{

        public static void main(String args[])

        {

                ArrayList<String> al=new ArrayList<String>();

al.add("Ravi");

al.add("Vijay");

al.add("Ajay");

  ArrayList<String> al2=new ArrayList<String>();
```

```
        al2.add("Sonoo");

al2.add("Hanumat");

        al.addAll(al2);//adding second list
in first list            Iterator

itr=al.iterator();

while(itr.hasNext())

        {
            System.out.println(itr.next());
        }
    }
}
```

Output:

```
Ravi
Vijay
Ajay
Sonoo
Hanumat
```

Example of

removeAll()

```
import java.util.*;
class
TestCollection5
{
        public static void main(String args[])
        {
          ArrayList<String> al=new ArrayList<String>();
        al2.add("Ravi");
        al2.add("Hanumat");
        al.removeAll(al2);

        System.out.println("iterating  the elements  after removing
the elements of al2...");               Iterator itr=al.iterator();
        while(itr.hasNext())
                {
```

```
                System.out.println(itr.next());
            }
        }
}
```

Output

```
iterating the elements after removing the elements of al2...
Vijay
Ajay
```

Example of retainAll()

```
import java.util.*;

class TestCollection6
{
    public static void main(String args[])
    {
        ArrayList<String> al=new ArrayList<String>();
```

```
        al2.add("Ravi");
            al2.add("Hanumat");

    al.retainAll(al2);
        System.out.println("iterating the elements after retaining the elements of al2...");
        Iterator itr=al.iterator();
        while(itr.hasNext())
        {
            System.out.println(itr.next());
        }
    }
}
```

Output

```
iterating the elements after removing the elements of al2...
Vijay
Ajay
```

Java ArrayList

Example: Book import

java.util.*;

class Book

{

int id;

String name,author,publisher;

int quantity;

public Book(int id, String name, String author, String publisher, int quantity)

```
        {
                this.id = id;
        this.name = name;
        this.author = author;
                this.publisher =
publisher;
        this.quantity = quantity;

        }
}
public class ArrayListExample
{
        public static void main(String[] args)
        {
         //Creating list of Books
         List<Book> list=new ArrayList<Book>();
         //Creating Books
         Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);

        Book b2=new Book(102,"Data Communications &
Networking","Forouzan","Mc Graw H ill",4);
         Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
```

```
        //Adding Books to
list            list.add(b1);
        list.add(b2);
        list.add(b3);
        //Traversing list
        for(Book b:list)
        {
       System.out.println(b.id+" "+b.name+" "+b.author+"
"+b.publisher+" "+b.quantity);              }
        }
}
```

Output

```
101 Let us C Yashwant Kanetkar BPB 8
102 Data Communications & Networking Forouzan Mc Graw Hill 4
103 Operating System Galvin Wiley 6
```

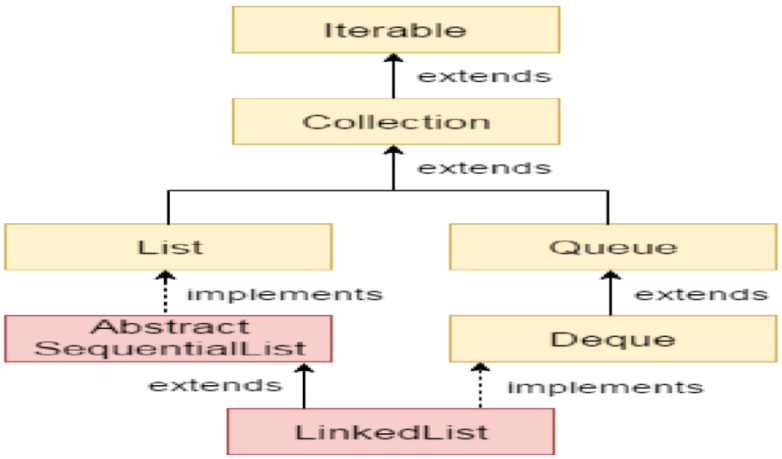
LinkedList class

Java LinkedList class uses 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 be occurred.
- Java LinkedList class can be used as list, stack or queue

Hierarchy of LinkedList class



Java LinkedList class extends AbstractSequentialList class and implements List and Deque interfaces. The Deque interface extends Queue interface. A list and Queue interface extends Collection and Iterable interfaces.

Doubly Linked List

In case of doubly linked list, we can add or remove elements from both sides.

fig- doubly linked list

LinkedList class declaration

Let's see the declaration for java.util.LinkedList class.

Syntax

publicclass LinkedList<E>**extends**

AbstractSequentialList<E>**implements** List<E>, Deque<E>, Cloneable,

Serializable

Constructors

Constructor	Description
LinkedList()	It is used to construct an empty list.
LinkedList(Collection c)	It is used to construct a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Methods

Method	Description
void add(int index, Object element)	It is used to insert the specified element at the specified position index in a list.

void addFirst(Object o)	It is used to insert the given element at the beginning of a list.
void addLast(Object o)	It is used to append the given element to the end of a list.
int size()	It is used to return the number of elements in a list.
boolean add(Object o)	It is used to append the specified element to the end of a list.
boolean contains(Object o)	It is used to return true if the list contains a specified element.
boolean remove(Object o)	It is used to remove the first occurence of the specified element in a list.
Object getFirst()	It is used to return the first element in a list.
Object getLast()	It is used to return the last element in a list.
int indexOf(Object o)	It is used to return the index in a list of the first occurrence of the specified element, or -1 if the list does not contain any element.
int lastIndexOf(Object o)	It is used to return the index in a list of the last occurrence of the specified element, or -1 if the list does not contain any element.

LinkedList Example

```
import java.util.*;
public class
TestCollection7
{
	public static void main(String args[])
	{
		LinkedList<String> al=new
LinkedList<String>();
	al.add("Ravi");
		al.add("Vijay");
		al.add("Ravi");
		al.add("Ajay");
		Iterator<String> itr=al.iterator();
		while(itr.hasNext())
		{
			System.out.println(itr.next());
		}
	}
```

```
}
```

Output

```
Ravi
Vijay
Ravi
Ajay
```

LinkedList Example: Book

```
import java.util.*;

class Book

{
        int id;
         String
name,author,publisher;

        int quantity;

 public Book(int id, String name, String author, String publisher, int
quantity)
        {
                this.id = id;
                this.name = name;
                this.author = author;
```

```
        this.publisher =
publisher;
        this.quantity = quantity;
        }
}
public class LinkedListExample
{
        public static void main(String[] args)
        {
         //Creating list of Books
         List<Book> list=new LinkedList<Book>();
         //Creating Books
         Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);

 Book b2=new Book(102,"Data Communications &
Networking","Forouzan","Mc  Graw
Hill",4);
         Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
        //Adding Books to
        list
        list.add(b1);
```

```
                list.add(b2);


                list.add(b3);
    //Traversing list
       for(Book b:list)
        {
         System.out.println(b.id+" "+b.name+" "+b.author+"
"+b.publisher+" "+b.quantity);              }
       }
}
```

Output

```
101 Let us C Yashwant Kanetkar BPB 8
102 Data Communications & Networking Forouzan Mc Graw Hill 4
103 Operating System Galvin Wiley 6
```

Difference between ArrayList and LinkedList

ArrayList	LinkedList
1) ArrayList internally uses dynamic array to store the elements.	LinkedList internally uses doubly linked list to store the elements.

2) Manipulation with ArrayList is slow because it internally uses array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is faster than ArrayList because it uses doubly linked list so no bit shifting is required in memory.
3) ArrayList class can act as a list only because it implements List only.	LinkedList class can act as a list and queue both because it implements List and Deque interfaces.
4) ArrayList is better for storing and accessing data.	LinkedList is better for manipulating data.

Java HashSet class

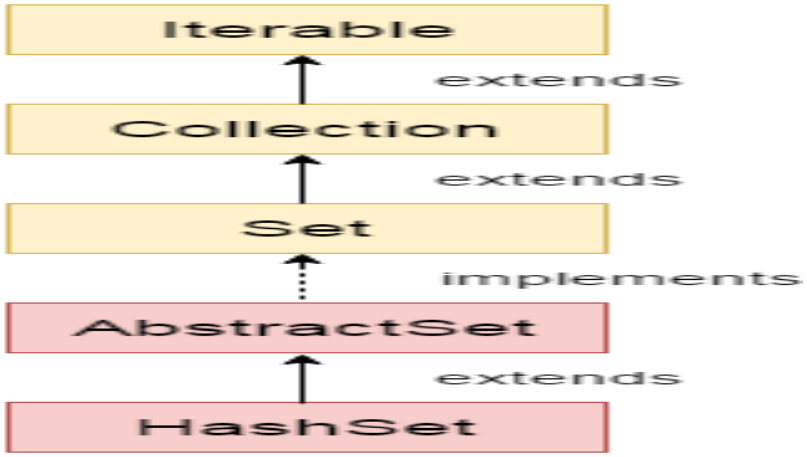
- Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.
- The important points about Java HashSet class are:
- HashSet stores the elements by using a mechanism called **hashing.**
- HashSet contains unique elements only.

Difference between List and Set

- List can contain duplicate elements whereas Set contains unique elements only.

Hierarchy of HashSet class

- The HashSet class extends AbstractSet class which implements Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.



HashSet class declaration

- Let's see the declaration for java.util.HashSet class.
- **public class** HashSet<E> **extends** AbstractSet<E> **implements** Set<E>, Cloneable, Serializable

- **Constructors of Java HashSet class**

SN	Constructor	Description
1)	HashSet()	It is used to construct a default HashSet.

2)	HashSet(int capacity)	It is used to initialize the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet.
3)	HashSet(int capacity, float loadFactor)	It is used to initialize the capacity of the hash set to the given integer value capacity and the specified load factor.
4)	HashSet(Collection c)	It is used to initialize the hash set by using the elements of the collection c.

Methods of Java HashSet class

SN	Modifier & Type	Method	Description
1)	boolean	add(Object o)	It is used to adds the specified element to this set if it is not already present.
2)	Void	clear()	It is used to remove all of the elements from this set.

3)	Object	clone()	It is used to return a shallow copy of this HashSet instance: the elements themselves are not cloned.
4)	boolean	contains(Object o)	It is used to return true if this set contains the specified element.
5)	boolean	isEmpty()	It is used to return true if this set contains no elements.
6)	Iterator<E>	iterator()	It is used to return an iterator over the elements in this set.

Methods of Java HashSet class

SN	Modifier & Type	Method	Description
1)	boolean	add(Object o)	It is used to adds the specified element to this set if it is not already present.
2)	Void	clear()	It is used to remove all of the elements from this set.
3)	Object	clone()	It is used to return a shallow copy of this HashSet instance: the elements

			themselves are not cloned.
4)	boolean	contains(Object o)	It is used to return true if this set contains the specified element.
5)	boolean	isEmpty()	It is used to return true if this set contains no elements.
6)	Iterator<E>	iterator()	It is used to return an iterator over the elements in this set.
7)	boolean	remove(Object o)	It is used to remove the specified element from this set if it is present.
8)	Int	size()	It is used to return the number of elements in this set.
9)	Spliterator<E>	spliterator()	It is used to create a late-binding and fail-fast Spliterator over the elements in this set.

Java HashSet Example

```
import java.util.*;
class TestCollection9{
 public static void main(String args[]){
  //Creating HashSet and adding elements
  HashSet<String> set=new HashSet<String>();
  set.add("Ravi");
  set.add("Vijay");
  set.add("Ravi1");
  set.add("Ajay");
  //Traversing elements
  Iterator<String> itr=set.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

Output:

```
Ajay
Vijay
Ravi1
Ajay
```

Java HashSet Example: Book

```
import java.util.*;
class Book {
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publisher, in
t quantity) {
    this.id = id;
    this.name = name;
```

```
    this.author = author;
    this.publisher = publisher;
    this.quantity = quantity;
  }
  }
  public class HashSetExample {
  public static void main(String[] args) {
    HashSet<Book> set=new HashSet<Book>();
    //Creating Books
    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB
  ",8);
    Book b2=new Book(102,"Data Communications & Networking
  ","Forouzan","Mc Graw Hill",4);
    Book b3=new Book(103,"Operating System","Galvin","Wiley",
  6);
    //Adding Books to HashSet
    set.add(b1);
    set.add(b2);
    set.add(b3);
    //Traversing HashSet
    for(Book b:set){
    System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publish
  er+" "+b.quantity);
    }
  }
  }
```

Output:

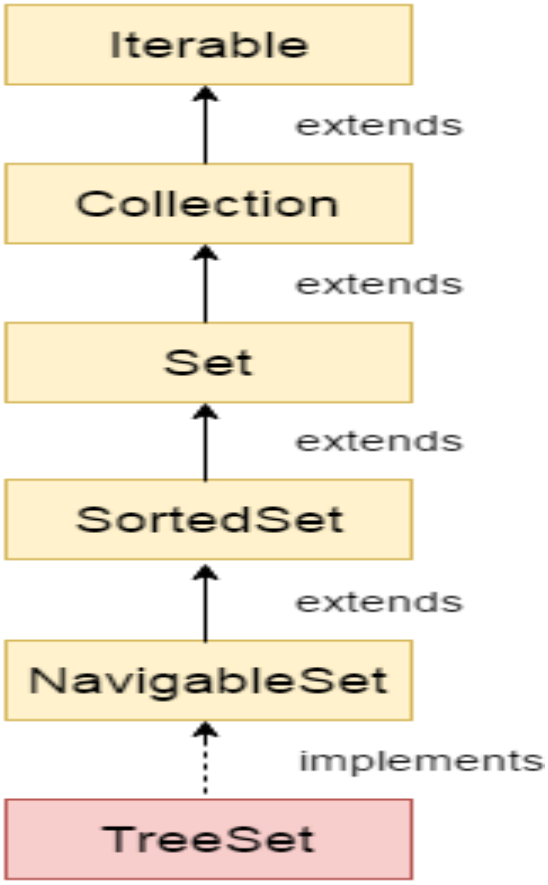
```
101 Let us C Yashwant Kanetkar BPB 8
102 Data Communications & Networking Forouzan Mc
Graw Hill 4
103 Operating System Galvin Wiley 6
```

Java TreeSet class

- Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements NavigableSet interface. The objects of TreeSet class are stored in ascending order.
- The important points about Java TreeSet class are:
- Contains unique elements only like HashSet.
- Access and retrieval times are quiet fast.
- Maintains ascending order.

Hierarchy of TreeSet class

- As shown in above diagram, Java TreeSet class implements NavigableSet interface. The NavigableSet interface extends SortedSet, Set, Collection and Iterable interfaces in hierarchical order.



TreeSet class declaration

- Let's see the declaration for java.util.TreeSet class.
- **public class** TreeSet<E> **extends** AbstractSet<E> **implements** NavigableSet<E>, Cloneable, Serializable

Constructors of Java TreeSet class

Constructor	Description
TreeSet()	It is used to construct an empty tree set that will be sorted in an ascending order according to the natural order of the tree set.

TreeSet(Collection c)	It is used to build a new tree set that contains the elements of the collection c.
TreeSet(Comparator comp)	It is used to construct an empty tree set that will be sorted according to given comparator.
TreeSet(SortedSet ss)	It is used to build a TreeSet that contains the elements of the given SortedSet.

Methods of Java TreeSet class

Method	Description
booleanaddAll(Collection c)	It is used to add all of the elements in the specified collection to this set.
booleancontains(Object o)	It is used to return true if this set contains the specified element.
booleanisEmpty()	It is used to return true if this set contains no elements.
booleanremove(Object o)	It is used to remove the specified element from this set if it is present.
void add(Object o)	It is used to add the specified element to this set if it is not already present.

void clear()	It is used to remove all of the elements from this set.
Object clone()	It is used to return a shallow copy of this TreeSet instance.
Object first()	It is used to return the first (lowest) element currently in this sorted set.
Object last()	It is used to return the last (highest) element currently in this sorted set.
int size()	It is used to return the number of elements in this set.

Java TreeSet Example

```
import java.util.*;
class TestCollection11{
 public static void main(String args[]){
  //Creating and adding elements
  TreeSet<String> al=new TreeSet<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ravi1");
  al.add("Ajay");
  //Traversing elements
  Iterator<String> itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
```

```
}
```

Output:

```
Ajay
Ravi
Vijay
```

Java TreeSet Example: Book

- Let's see a TreeSet example where we are adding books to set and printing all the books. The elements in TreeSet must be of Comparable type. String and Wrapper classes are Comparable by default. To add user-defined objects in TreeSet, you need to implement Comparable interface.

```
import java.util.*;
class Book implements Comparable<Book>{
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publisher, int quantity) {
   this.id = id;
   this.name = name;
   this.author = author;
   this.publisher = publisher;
   this.quantity = quantity;
}
public int compareTo(Book b) {
   if(id>b.id){
      return 1;
   }else if(id<b.id){
      return -1;
   }else{
   return 0;
```

```
    }
}
}
public class TreeSetExample {
public static void main(String[] args) {
    Set<Book> set=new TreeSet<Book>();
    //Creating Books
    Book b1=new Book(121,"Let us C","Yashwant Kanetkar","BPB
",8);
    Book b2=new Book(233,"Operating System","Galvin","Wiley",
6);
    Book b3=new Book(101,"Data Communications & Networking
","Forouzan","Mc Graw Hill",4);
    //Adding Books to TreeSet
    set.add(b1);
    set.add(b2);
    set.add(b3);
    //Traversing TreeSet
    for(Book b:set){
    System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publish
er+" "+b.quantity);
    }
}
}
```

Output:

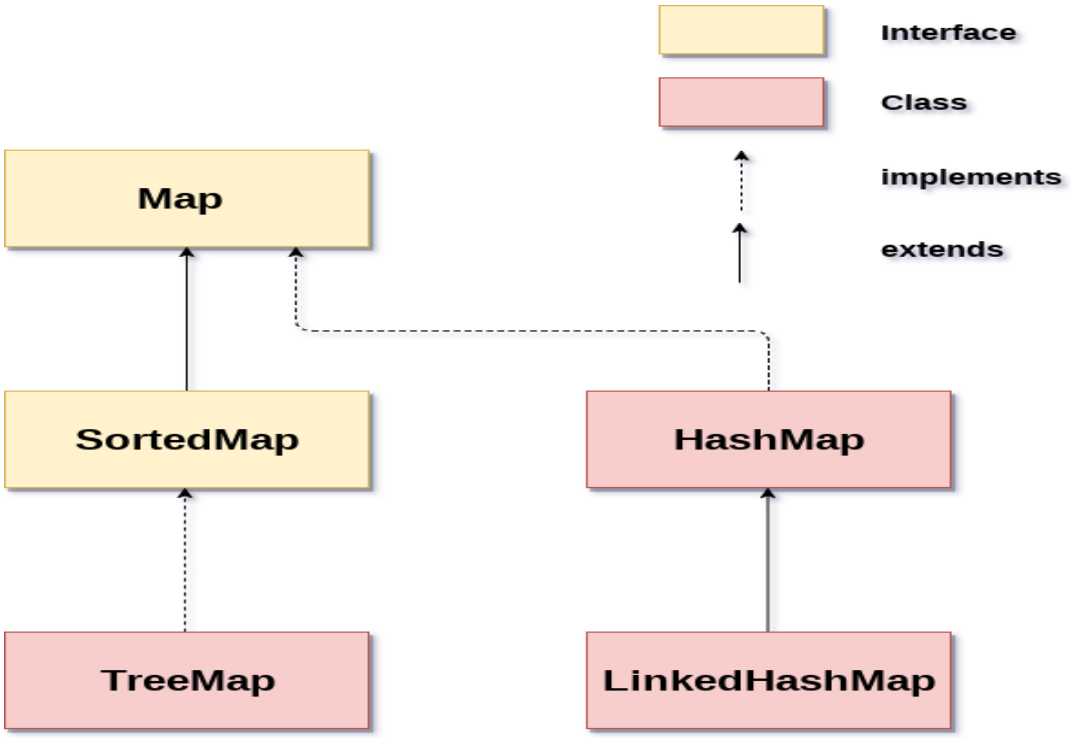
```
101 Data Communications & Networking Forouzan Mc Graw Hill
4 121 Let us C Yashwant Kanetkar BPB 8
233 Operating System Galvin Wiley 6
```

Java Map Interface

- Java Map Interface
- A map contains values on the basis of key i.e. key and value pair. Each key and value pair is known as an entry. Map contains only unique keys.
- Map is useful if you have to search, update or delete elements on the basis of key.

Java Map Hierarchy

- There are two interfaces for implementing Map in java: Map and SortedMap, and three classes : HashMap, LinkedHashMap and TreeMap. The hierarchy of Java Map is given below:



- Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allows null keys and values but TreeMap doesn't allow any null key or value.
- Map can't be traversed so you need to convert it into Set using *keySet()* or *entrySet()* method

Class	Description
HashMap	HashMap is the implementation of Map but it doesn't maintain any order.
LinkedHashMap	LinkedHashMap is the implementation of Map, it inherits HashMap class. It maintains insertion order.
TreeMap	TreeMap is the implementation of Map and SortedMap, it maintains ascending order.

Useful methods of Map interface

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

boolean containsKey(Object key)	It is used to search the specified key from this map.
Set keySet()	It is used to return the Set view containing all the ke
Set entrySet()	It is used to return the Set view containing all the ke

Map.Entry Interface

- Entry is the sub interface of Map. So we will be accessed it by Map.Entry name. It provides methods to get key and value.

Methods of Map.Entry interface

Method	Description
Object getKey()	It is used to obtain key.
Object getValue()	It is used to obtain value.

Java Map Example: Generic (New Style)

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

```
Map<Integer,String> map=new HashMap<Integer,String>();
map.put(100,"Amit");
map.put(101,"Vijay");
map.put(102,"Rahul");
for(Map.Entry m:map.entrySet()){
System.out.println(m.getKey()+" "+m.getValue());
}
}
```

output:

102 Ragul

100 Amit

101 Vijay

Java Map Example: Non-Generic (Old Style)

```
//Non-generic
import java.util.*;
public class MapExample1 {
public static void main(String[] args) {
Map map=new HashMap();
//Adding elements to map
map.put(1,"Amit");
map.put(5,"Rahul");
```

```
    map.put(2,"Jai");
    map.put(6,"Amit");
    //Traversing Map
    Set set=map.entrySet();//Converting to Set so that we can traverse
    Iterator itr=set.iterator();
    while(itr.hasNext()){
    //Converting to Map.Entry so that we can get key and value separately
     Map.Entry entry=(Map.Entry)itr.next();
     System.out.println(entry.getKey()+" "+entry.getValue());
    }
}
}
```

Output

```
1 Amit
2 Jai
5 Rahul
6 Amit
```

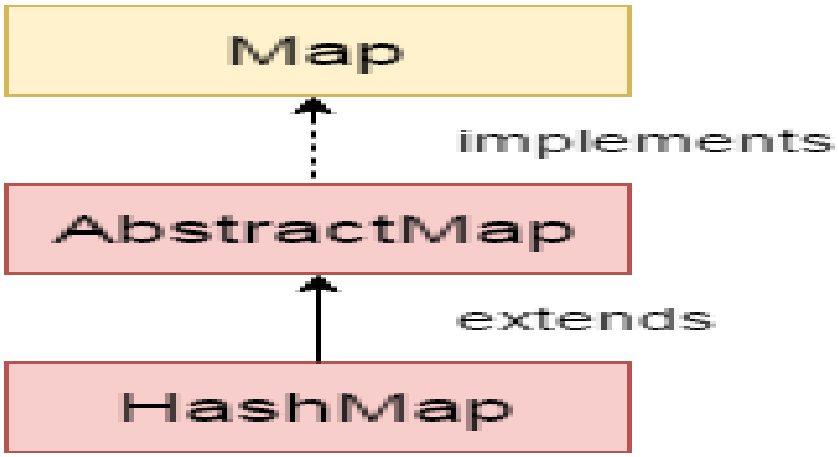
Java HashMap class

- Java HashMap class implements the map interface by using a hashtable. It inherits AbstractMap class and implements Map interface.

- The important points about Java HashMap class are:
- A HashMap contains values based on the key.
- It contains only unique elements.
- It may have one null key and multiple null values.
- It maintains no order

Hierarchy of HashMap class

- As shown in the above figure, HashMap class extends AbstractMap class and implements Map interface.



HashMap class declaration

- Let's see the declaration for java.util.HashMap class.
- public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Cloneable, Serializable

HashMap class Parameters

- Let's see the Parameters for java.util.HashMap class.
- K: It is the type of keys maintained by this map.
- V: It is the type of mapped value

Constructors of Java HashMap class

Constructor	Description
HashMap()	It is used to construct a default HashMap.
HashMap(Map m)	It is used to initializes the hash map by using the elem of the given Map object m.
HashMap(int capacity)	It is used to initializes the capacity of the hash map to the given integer value, capacity.
HashMap(int capacity, float fillRatio)	It is used to initialize both the capacity and fill ratio of the hash map by using its arguments.

Methods of Java HashMap class

Method	Description
void clear()	It is used to remove all of the mappings from this map
boolean containsKey(Object key)	It is used to return true if this map contains a mapping the specified key.
boolean containsValue(Object value)	It is used to return true if this map maps one or more k to the specified value.
boolean isEmpty()	It is used to return true if this map contains no key-val
Object clone()	It is used to return a shallow copy of this HashMap ins and values themselves are not cloned.
Set entrySet()	It is used to return a collection view of the mappings c this map.

Set keySet()	It is used to return a set view of the keys contained in
Object put(Object key, Object value)	It is used to associate the specified value with the spec in this map.
int size()	It is used to return the number of key-value mappings
Collection values()	It is used to return a collection view of the values cont in this map.

Java HashMap Example

```
import java.util.*;
class TestCollection13{
 public static void main(String args[]){
  HashMap<Integer,String> hm=new HashMap<Integer,String>();

  hm.put(100,"Amit");
  hm.put(101,"Vijay");
  hm.put(102,"Rahul");
  for(Map.Entry m:hm.entrySet()){
   System.out.println(m.getKey()+"  "+m.getValue());
  }
 }
}
```

Output:

- 102 Rahul
- 100 Amit
- 101 Vijay

Java HashMap Example: remove()

```
import java.util.*;
public class HashMapExample {
 public static void main(String args[]) {
  // create and populate hash map
  HashMap<Integer, String> map = new HashMap<Integer, String >();  map.put(101,"Let us C");
  map.put(102, "Operating System");
  map.put(103, "Data Communication and Networking");
  System.out.println("Values before remove: "+ map);
  // Remove value for key 102
  map.remove(102);
  System.out.println("Values after remove: "+ map);
  }
}
```

Output

- Values before remove: {102=Operating System, 103=Data Communication and Networking, 101=Let us C}

- Values after remove: {103=Data Communication and Networking, 101=Let us C}

Difference between HashSet and HashMap

- HashSet contains only values whereas HashMap contains entry(key and value).

Java HashMap Example: Book

- import java.util.*;
- class Book {
- int id;
- String name,author,publisher;
- int quantity;
- public Book(int id, String name, String author, String publisher, int quantity) {
- this.id = id;
- this.name = name;
- this.author = author;
- this.publisher = publisher;
- this.quantity = quantity;
- }
- }
- public class MapExample {
- public static void main(String[] args) {

```
    //Creating map of Books
    Map<Integer,Book> map=new HashMap<Integer,Book>();
    //Creating Books
    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
    Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
    Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
    //Adding Books to map
    map.put(1,b1);
    map.put(2,b2);
    map.put(3,b3);
   //Traversing map
    for(Map.Entry<Integer, Book> entry:map.entrySet()){
        int key=entry.getKey();
        Book b=entry.getValue();
        System.out.println(key+" Details:");
System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
    }
}
```

- }

Output

- 1 Details:
- 101 Let us C Yashwant Kanetkar BPB 8
- 2 Details:
- 102 Data Communications & Networking Forouzan Mc Graw Hill 4
- 3 Details:
- 103 Operating System Galvin Wiley 6

Java Comparable interface

- Java Comparable interface is used to order the objects of user-defined class.This interface is found in java.lang package and contains only one method named compareTo(Object). It provide single sorting sequence only i.e. you can sort the elements on based on single data member only. For example it may be rollno, name, age or anything else.

Java Comparable interface

- Java Comparable interface is used to order the objects of user-defined class.This interface is found in java.lang package and contains only one method named compareTo(Object). It provide single sorting sequence only i.e. you can sort the elements on based on single data member only. For example it may be rollno, name, age or anything else.

compareTo(Object obj) method

- public int compareTo(Object obj): is used to compare the current object with the specified object.
- We can sort the elements of:
- String objects
- Wrapper class objects
- User-defined class objects

Collections class

- **Collections** class provides static methods for sorting the elements of collections. If collection elements are of Set or Map, we can use TreeSet or TreeMap. But We cannot sort the elements of List.

Collections class provides methods for sorting the elements of List type elements.

Method of Collections class for sorting List elements

- **public void sort(List list):** is used to sort the elements of List. List elements must be of Comparable type.

Java Comparable Example

```
class Student implements Comparable<Student>{
int rollno;
String name;
int age;
Student(int rollno,String name,int age){
this.rollno=rollno;
this.name=name;
this.age=age;
}
public int compareTo(Student st){
if(age==st.age)
return 0;
else if(age>st.age)
return 1;
else
return -1;
}
}
```

```
import java.util.*;
import java.io.*;
public class TestSort3{
public static void main(String args[]){
ArrayList<Student> al=new ArrayList<Student>();
al.add(new Student(101,"Vijay",23));
al.add(new Student(106,"Ajay",27));
al.add(new Student(105,"Jai",21));
Collections.sort(al);
for(Student st:al){
System.out.println(st.rollno+" "+st.name+" "+st.age);
}
}
}
```

Java Comparator interface

- **Java Comparator interface** is used to order the objects of user-defined class.
- This interface is found in java.util package and contains 2 methods compare(Object obj1,Object obj2) and equals(Object element).
- It provides multiple sorting sequence i.e. you can sort the elements on the basis of any data member, for example rollno, name, age or anything else.

compare() method

- **public int compare(Object obj1,Object obj2):** compares the first object with second object.

Collections class

- **Collections** class provides static methods for sorting the elements of collection. If collection elements are of Set or Map, we can use TreeSet or TreeMap. But we cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements also.

Method of Collections class for sorting List elements

- public void sort(List list, Comparator c): is used to sort the elements of List by the given Comparator.

Java Comparator Example (Non-generic Old Style)

- Let's see the example of sorting the elements of List on the basis of age and name. In this example, we have created 4 java classes:
- Student.java
- AgeComparator.java
- NameComparator.java
- Simple.java

Student.java

- class Student{
- int rollno;
- String name;
- int age;
- Student(int rollno,String name,int age){
- this.rollno=rollno;

- this.name=name;
- this.age=age;
- } }

AgeComparator.java

- This class defines comparison logic based on the age. If age of first object is greater than the second, we are returning positive value, it can be any one such as 1, 2, 10 etc. If age of first object is less than the second object, we are returning negative value, it can be any negative value and if age of both objects are equal, we are returning 0.
- **import** java.util.*;
- **class** AgeComparator **implements** Comparator{
- **public int** compare(Object o1,Object o2){
- Student s1=(Student)o1;
- Student s2=(Student)o2;
-
- **if**(s1.age==s2.age)
- **return** 0;
- **else if**(s1.age>s2.age)
- **return** 1;
- **else**
- **return** -1;
- }
- }

NameComparator.java

- This class provides comparison logic based on the name. In such case, we are using the compareTo() method of String class, which internally provides the comparison logic.
- **import** java.util.*;
- **class** NameComparator **implements** Comparator{
- **public int** compare(Object o1,Object o2){
- Student s1=(Student)o1;
- Student s2=(Student)o2;
-
- **return** s1.name.compareTo(s2.name);
- }
- }

Simple.java

- In this class, we are printing the objects values by sorting on the basis of name and age.
- **import** java.util.*;
- **import** java.io.*;
- **class** Simple{
- **public static void** main(String args[]){
- ArrayList al=**new** ArrayList();
- al.add(**new** Student(101,"Vijay",23));
- al.add(**new** Student(106,"Ajay",27));
- al.add(**new** Student(105,"Jai",21));

```
System.out.println("Sorting by Name...");
Collections.sort(al,new NameComparator());
Iterator itr=al.iterator();
while(itr.hasNext()){
Student st=(Student)itr.next();
System.out.println(st.rollno+" "+st.name+" "+st.age);
}
System.out.println("sorting by age...");
Collections.sort(al,new AgeComparator());
Iterator itr2=al.iterator();
while(itr2.hasNext()){
Student st=(Student)itr2.next();
System.out.println(st.rollno+" "+st.name+" "+st.age);
}
}
}
```

OUTPUT

```
Sorting by Name...
106 Ajay 27
105 Jai 21
101 Vijay 23
Sorting by age...
105 Jai 21
```

- 101 Vijay 23
- 106 Ajay 27

Java Comparator Example (Generic)
Student.java

```
class Student{
int rollno;
String name;
int age;
Student(int rollno,String name,int age){
this.rollno=rollno;
this.name=name;
this.age=age;  }
}
```

AgeComparator.java

```
import java.util.*;
class AgeComparator implements Comparator<Student>{
public int compare(Student s1,Student s2){
if(s1.age==s2.age)
return 0;
else if(s1.age>s2.age)
return 1;
else
return -1;  }
```

- }

NameComparator.java

- This class provides comparison logic based on the name. In such case, we are using the compareTo() method of String class, which internally provides the comparison logic.
- **import** java.util.*;
- **class** NameComparator **implements** Comparator<Student>{
- **public int** compare(Student s1,Student s2){
- **return** s1.name.compareTo(s2.name);
- }
- }

Simple.java

- In this class, we are printing the objects values by sorting on the basis of name and age.
- **import** java.util.*;
- **import** java.io.*;
- **class** Simple{
- **public static void** main(String args[]){
- ArrayList<Student> al=**new** ArrayList<Student>();
- al.add(**new** Student(101,"Vijay",23));
- al.add(**new** Student(106,"Ajay",27));
- al.add(**new** Student(105,"Jai",21));
- System.out.println("Sorting by Name...");
- Collections.sort(al,**new** NameComparator());

```
for(Student st: al){
System.out.println(st.rollno+" "+st.name+" "+st.age);
}
System.out.println("sorting by age...");
Collections.sort(al,new AgeComparator());
for(Student st: al){
System.out.println(st.rollno+" "+st.name+" "+st.age);
}
}
}
```

Output:

```
Sorting by Name...
106 Ajay 27
105 Jai 21
101 Vijay 23
Sorting by age...
105 Jai 21
101 Vijay 23
106 Ajay 27
```

Comparable	Comparator

Difference between Comparable and Comparator

1) Comparable provides **single sorting sequence**. In other words, we can sort the collection on the basis of single element such as id or name or price etc.	Comparator provides **multiple sorting sequence**. In other words, we can sort the collection on the basis of multiple elements such as id, name and price etc.
2) Comparable **affects the original class** i.e. actual class is modified.	Comparator **doesn't affect the original class** i.e. actual class is not modified.
3) Comparable provides **compareTo() method** to sort elements.	Comparator provides **compare() method** to sort elements.
4) Comparable is found in **java.lang** package.	Comparator is found in **java.util** package.
5) We can sort the list elements of Comparable type by **Collections.sort(List)** method.	We can sort the list elements of Comparator type by **Collections.sort(List,Comparator)** method.

Java Annotations:

Java Annotations:

- **Java annotation**. In the **Java** computer programming language, an **annotation** is a form of syntactic metadata that can be added to **Java** source code. Classes, methods, variables, parameters and packages may be **annotated**. Like Javadoc tags,**Java annotations** can be read from source files.
- Java **Annotation** is a tag that represents the *metadata* i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.
- Annotations in java are used to provide additional information, so it is an alternative option for XML and java marker interfaces.
- First, we will learn some built-in annotations then we will move on creating and using custom annotations.

Built-In Java Annotations:

- There are several built-in annotations in java. Some annotations are applied to java code and some to other annotations.

Built-In Java Annotations used in java code:

- @Override
- @SuppressWarnings
- @Deprecated

Built-In Java Annotations used in other annotations:

- @Target
- @Retention
- @Inherited

- @Documented

Understanding Built-In Annotations in java:

- @Override
- @Override annotation assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.
- Sometimes, we does the silly mistake such as spelling mistakes etc. So, it is better to mark @Override annotation that provides assurity that method is overridden.

Program:

```
class Animal{
void eatSomething(){System.out.println("eating something");}
}
  class Dog extends Animal{
@Override
void eatsomething(){System.out.println("eating foods");}//should be eatSomething
}
  class TestAnnotation1{
public static void main(String args[]){
Animal a=new Dog();
a.eatSomething();
}}
```

Output:

Comple Time Error

@SuppressWarnings:

- @SuppressWarnings annotation: is used to suppress warnings issued by the compiler.

Program:

```
import java.util.*;
class TestAnnotation2{
@SuppressWarnings("unchecked")
public static void main(String args[]){
ArrayList list=new ArrayList();
list.add("sonoo");
list.add("vimal");
list.add("ratan");

for(Object obj:list)
System.out.println(obj);
 }}
```

Output:

Now no warning at compile time.

Note:

If you remove the @SuppressWarnings("unchecked") annotation, it will show warning at compile time because we are using non-generic collection.

@Deprecated:

- @Deprecated annoation marks that this method is deprecated so compiler prints warning.
- It informs user that it may be removed in the future versions. So, it is better not to use such methods.

Program:

```
class A{
void m(){System.out.println("hello  m");}
@Deprecated
void n(){System.out.println("hello  n");}
}
class TestAnnotation3{
public  static void main(String args[]){
A a=new A();
a.n();
}}
```

Output:

At Compile Time:

Note: Test.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

At Runtime:

hello n

Java Custom Annotation:

- **Java Custom annotations** or Java User-defined annotations are easy to create and use. The *@interface* element is used to declare an annotation. For example:
- **@interface** MyAnnotation{}
- Here, MyAnnotation is the custom annotation name.

Points to remember for java custom annotation signature

- There are few points that should be remembered by the programmer.
- Method should not have any throws clauses
- Method should return one of the following: primitive data types, String, Class, enum or array of these data types.
- Method should not have any parameter.
- We should attach @ just before interface keyword to define annotation.
- It may assign a default value to the method.

Types of Annotation:

- There are three types of annotations.
- Marker Annotation
- Single-Value Annotation
- Multi-Value Annotation

1) Marker Annotation:

- An annotation that has no method, is called marker annotation. For example:
- **@interface** MyAnnotation{}
- The @Override and @Deprecated are marker annotations.

2) Single-Value Annotation:

- An annotation that has one method, is called single-value annotation. For example:

```
@interface MyAnnotation{
int value();
}
```

We can provide the default value also. For example:

```
@interface MyAnnotation{
int value() default 0;
}
```

How to apply Single-Value Annotation:

- Let's see the code to apply the single value annotation.
- @MyAnnotation(value=10)
- The value can be anything.

3) Multi-Value Annotation:

- An annotation that has more than one method, is called Multi-Value annotation.

For example:

```
@interface MyAnnotation{
int value1();
String value2();
String value3();
}
}
```

We can provide the default value also. For example:

```
@interface MyAnnotation{
int value1() default 1;
String value2() default "";
String value3() default "xyz";
}
```

How to apply Multi-Value Annotation:

- Let's see the code to apply the multi-value annotation.
- @MyAnnotation(value1=10,value2="Arun Kumar",value3="Ghaziabad")

Built-in Annotations used in custom annotations in java

- @Target
- @Retention
- @Inherited

- @Documented

@Target:

- **@Target** tag is used to specify at which type, the annotation is used.
- The java.lang.annotation.**ElementType** enum declares many constants to specify the type of element where annotation is to be applied such as TYPE, METHOD, FIELD etc. Let's see the constants of ElementType enum:

Element Types	Where the annotation can be
TYPE	class, interface or enumeration
FIELD	Fields
METHOD	Methods
CONSTRUCTOR	Constructors
LOCAL_VARIABLE	local variables
ANNOTATION_TYPE	annotation type
PARAMETER	Parameter

Example to specify annoation for a class:

@Target(ElementType.TYPE)

```
@interface MyAnnotation{
int value1();
String value2();
}
```

Example to specify annotation for a class, methods or fields:

```
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.METHOD})
@interface MyAnnotation{
int value1();
String value2();
}
```

@Retention:

- **@Retention** annotation is used to specify to what level annotation will be available.

RetentionPolicy	Availability
RetentionPolicy.SOURCE	refers to the source code, discarded during compilation. It will not be available in the compiled class.
RetentionPolicy.CLASS	refers to the .class file, available to java compiler but not to JVM . It is included in the class file.

RetentionPolicy.RUNTIME	refers to the runtime, available to java compiler and JVM .
RetentionPolicy	Availability

Example to specify the RetentionPolicy:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface MyAnnotation{
int value1();
String value2();
}
```

Example of custom annotation: creating, applying and accessing annotation:

- Let's see the simple example of creating, applying and accessing annotation.

Program:

```
//Creating annotation
import java.lang.annotation.*;
import java.lang.reflect.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
```

```
@interface MyAnnotation{
int value();
}

//Applying annotation
class Hello{
@MyAnnotation(value=10)
public void sayHello(){System.out.println("hello annotation");}
}
//Accessing annotation
class TestCustomAnnotation1{
public static void main(String args[])throws Exception{
Hello h=new Hello();
Method m=h.getClass().getMethod("sayHello");
MyAnnotation manno=m.getAnnotation(MyAnnotation.class);
System.out.println("value is: "+manno.value());
}}
```

How built-in annotaions are used in real scenario?:

- In real scenario, java programmer only need to apply annotation. He/She doesn't need to create and access annotation. Creating and Accessing annotation is performed by the implementation provider. On behalf of the annotation, java compiler or JVM performs some additional operations.

@Inherited:

- By default, annotations are not inherited to subclasses. The @Inherited annotation marks the annotation to be inherited to subclasses.

```
@Inherited
@interface ForEveryone { }//Now it will be available to subclass als
o
@interface ForEveryone { }
class Superclass{}
class Subclass extends Superclass{}
```

@Documented:

- The @Documented Marks the annotation for inclusion in the documentation.

Generics in Java

- Generics in Java
- The Java Generics programming is introduced in J2SE 5 to deal with type-safe objects.
- Before generics, we can store any type of objects in collection i.e. non-generic. Now generics, forces the java programmer to store specific type of objects.

Advantage of Java Generics

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

- 1) Type-safety : We can hold only a single type of objects in generics. It doesn't allow to store other objects.
- 2) Type casting is not required: There is no need to typecast the object.

Before Generics, we need to type cast

- List list = new ArrayList();
- list.add("hello");
- String s = (String) list.get(0);//typecasting
- After Generics, we don't need to typecast the object.
- List<String> list = new ArrayList<String>();
- list.add("hello");
- String s = list.get(0);

3) Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

- List<String> list = new ArrayList<String>();
- list.add("hello");
- list.add(32);//Compile Time Error
- Syntax to use generic collection
- ClassOrInterface<Type>
- Example to use Generics in java
- ArrayList<String>

Full Example of Generics in Java

Here, we are using the ArrayList class, but you can use any collection class such as ArrayList, LinkedList, HashSet, TreeSet, HashMap, Comparator etc

```
import java.util.*;
class TestGenerics1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();
list.add("rahul");
list.add("jai");
//list.add(32);//compile time error
  String s=list.get(1);//type casting is not required
System.out.println("element is: "+s);
  Iterator<String> itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
element is:
jai
rahul
```

- jai

Example of Java Generics using Map

- Now we are going to use map elements using generics. Here, we need to pass key and value. Let us understand it by a simple example:

```
import java.util.*;
class TestGenerics2{
public static void main(String args[]){
Map<Integer,String> map=new HashMap<Integer,String>();
map.put(1,"vijay");
map.put(4,"umesh");
map.put(2,"ankit");
 //Now use Map.Entry for Set and Iterator
Set<Map.Entry<Integer,String>> set=map.entrySet();
 Iterator<Map.Entry<Integer,String>> itr=set.iterator();
while(itr.hasNext()){
Map.Entry e=itr.next();//no need to typecast
System.out.println(e.getKey()+" "+e.getValue());
}
}}
```

Output:

- 1 vijay
- 2 ankit

- 4 umesh

Generic class

- A class that can refer to any type is known as generic class. Here, we are using **T** type parameter to create the generic class of specific type.
- Let's see the simple example to create and use the generic class.
- **Creating generic class:**
- **class** MyGen<T>{
- T obj;
- **void** add(T obj){**this**.obj=obj;}
- T get(){**return** obj;}
- }
- The T type indicates that it can refer to any type (like String, Integer, Employee etc.). The type you specify for the class, will be used to store and retrieve the data.

Using generic class

- **class** TestGenerics3{
- **public static void** main(String args[]){
- MyGen<Integer> m=**new** MyGen<Integer>();
- m.add(2);
- //m.add("vivek");//Compile time error
- System.out.println(m.get());

- }}
- Output:2

Type Parameters

- The type parameters naming conventions are important to learn generics thoroughly. The commonly type parameters are as follows:
- T - Type
- E - Element
- K - Key
- N - Number
- V - Value

Generic Method

- Like generic class, we can create generic method that can accept any type of argument.
- Let's see a simple example of java generic method to print array elements. We are using here E to denote the element.
- public class TestGenerics4{
- public static < E > void printArray(E[] elements) {
- for (E element : elements){
- System.out.println(element);
- }
- System.out.println();
- }

```
    public static void main( String args[] ) {
        Integer[] intArray = { 10, 20, 30, 40, 50 };
        Character[] charArray = { ‘G', ’T', ’E', ’C', ’-',’M',’C',’A',’-',’2' };
         System.out.println( "Printing Integer Array" );
        printArray( intArray  );
         System.out.println( "Printing Character Array" );
        printArray( charArray );
    }
}
```

Output:

```
Printing Integer Array
10
20
30
40
50
Printing Character Array
G
T
E
C
–
M
C
A
-
2
```

Wildcard in Java Generics

- The ? (question mark) symbol represents wildcard element. It means any type. If we write <? extends Number>, it means any child class of Number e.g. Integer, Float, double etc. Now we can call the method of Number class through any child class object.
- Let's understand it by the example given below:
- import java.util.*;
- abstract class Shape{
- abstract void draw();
- }
- class Rectangle extends Shape{
- void draw(){System.out.println("drawing rectangle");}
- }
- class Circle extends Shape{
- void draw(){System.out.println("drawing circle");}

}

- **class** GenericTest{
- //creating a method that accepts only child class of Shape
- **public static void** drawShapes(List<? **extends** Shape> lists){
- **for**(Shape s:lists){
- s.draw();//calling method of Shape class by child class instance

```
}
}
public static void main(String args[]){
List<Rectangle> list1=new ArrayList<Rectangle>();
list1.add(new Rectangle());

List<Circle> list2=new ArrayList<Circle>();
list2.add(new Circle());
list2.add(new Circle());

drawShapes(list1);
drawShapes(list2);
}}
```

Output:

drawing rectangle

drawing circle

drawing circle