UNIT-V

Collections, Iteration, Collection Interface, Set and SortedSet, List, Map and SortedMap, Wrapped Collections and Collections Class, Wrapper classes and loading classes.

Wrapper Classes

Wrapper classes in Java

- A Wrapper class is a class whose object wraps or contains primitive data types.
- When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types.
- Wrapper class in java provides the mechanism to convert primitive data type into object is called **boxing** and object into primitive data type is called **unboxing**.
- Since J2SE 5.0, auto boxing and unboxing feature converts primitive data type into object and object into primitive data type automatically. The automatic conversion of primitive data type into object is known as **auto-boxing** and vice-versa **auto-unboxing**.

Wrapper Classes

- The eight classes of the *java.lang* package are known as wrapper classes in Java.

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

Example

// Wrapper class Example: **Primitive to Wrapper**

```
import java.lang.*;

public class WrapperExample1
{
    public static void main(String args[])
    {
        //Converting int into Integer
        int a=20;
        Integer i=Integer.valueOf(a);//converting int into Integer
        Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
        System.out.println(a+" "+i+" "+j);
    }
```

Example

// Wrapper class Example: **Wrapper to Primitive**

```
import java.lang.*;

public class WrapperExample2
{
    public static void main(String args[])
    {
        //Converting Integer to int
        Integer a=new Integer(3);
        int i=a.intValue();//unboxing i.e converting Integer to int
        int j=a;//auto unboxing, now compiler will write a.intValue() internally

        System.out.println(a+" "+i+" "+j);
    }
}
```

Example

```
public class WrapperClassExample {
    public static void main(String[] args) {
        // Integer wrapper class
        Integer intObj1 = new Integer(42);
        Integer intObj2 = Integer.valueOf("123"); // Using valueOf to create Integer

        int intPrimitive = intObj1.intValue(); // Converting back to a primitive int

        // Double wrapper class
        Double doubleObj1 = new Double(3.14159);
        Double doubleObj2 = Double.valueOf("2.71828");

        double doublePrimitive = doubleObj1.doubleValue(); // Converting back to a primitive double

        // Boolean wrapper class
        Boolean boolObj1 = new Boolean(true);
        Boolean boolObj2 = Boolean.valueOf("false");

        boolean boolPrimitive = boolObj1.booleanValue(); // Converting back to a primitive boolean

        // Character wrapper class
        Character charObj = new Character('A');

        char charPrimitive = charObj.charValue(); // Converting back to a primitive char

        // Autoboxing and unboxing (Java 5 and later)
        Integer x = 5; // Autoboxing (int to Integer)
        int y = x;     // Unboxing (Integer to int)

        System.out.println("Integer Example: " + intObj1 + ", " + intObj2 + ", " + intPrimitive);
        System.out.println("Double Example: " + doubleObj1 + ", " + doubleObj2 + ", " + doublePrimitive);
        System.out.println("Boolean Example: " + boolObj1 + ", " + boolObj2 + ", " + boolPrimitive);
        System.out.println("Character Example: " + charObj + ", " + charPrimitive);
        System.out.println("Autoboxing and Unboxing Example: " + x + ", " + y);
    } }
```

Need of Wrapper Classes

- They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
- The classes in java.util package handles only objects and hence wrapper classes help in this case.
- Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
- An object is needed to support synchronization in multithreading.

Methods Supported by the Wrapper Class

Method	Method Description
typeValue()	Converts the value of this Number object to the specified primitive data type returned
compareTo()	Compares this Number object to the argument
equals()	Determines whether this Number object is equal to the argument
valueOf()	Returns an Integer object holding the value of the specified primitive data type value
toString()	Returns a String object representing the value of specified Integer type argument
parseInt()	Returns an Integer type value of a specified String representation
decode()	Decodes a String into an integer
min()	Returns the smaller value after comparison of the two arguments
max()	Returns the larger value after comparison of the two arguments
round()	Returns the closest round off long or int value as per the method return type

Example

Integer Wrapper Class Methods

```
int num = Integer.parseInt("123"); // Converts String to int
String str = Integer.toString(456); // Converts int to String
Integer value = Integer.valueOf(789); // Converts int to Integer object
```

Boolean Wrapper Class Methods

```
Boolean bool = Boolean.valueOf("true"); // Converts String to Boolean
boolean b = Boolean.parseBoolean("false"); // Converts String to boolean
```

Points to Remember

- Wrapper classes in Java wraps the primitive data type in its class object.
- Java wrapper classes are provided in the java.lang package.
- Autoboxing and unboxing converts the primitive into objects and objects into primitives automatically.

Performance Consideration

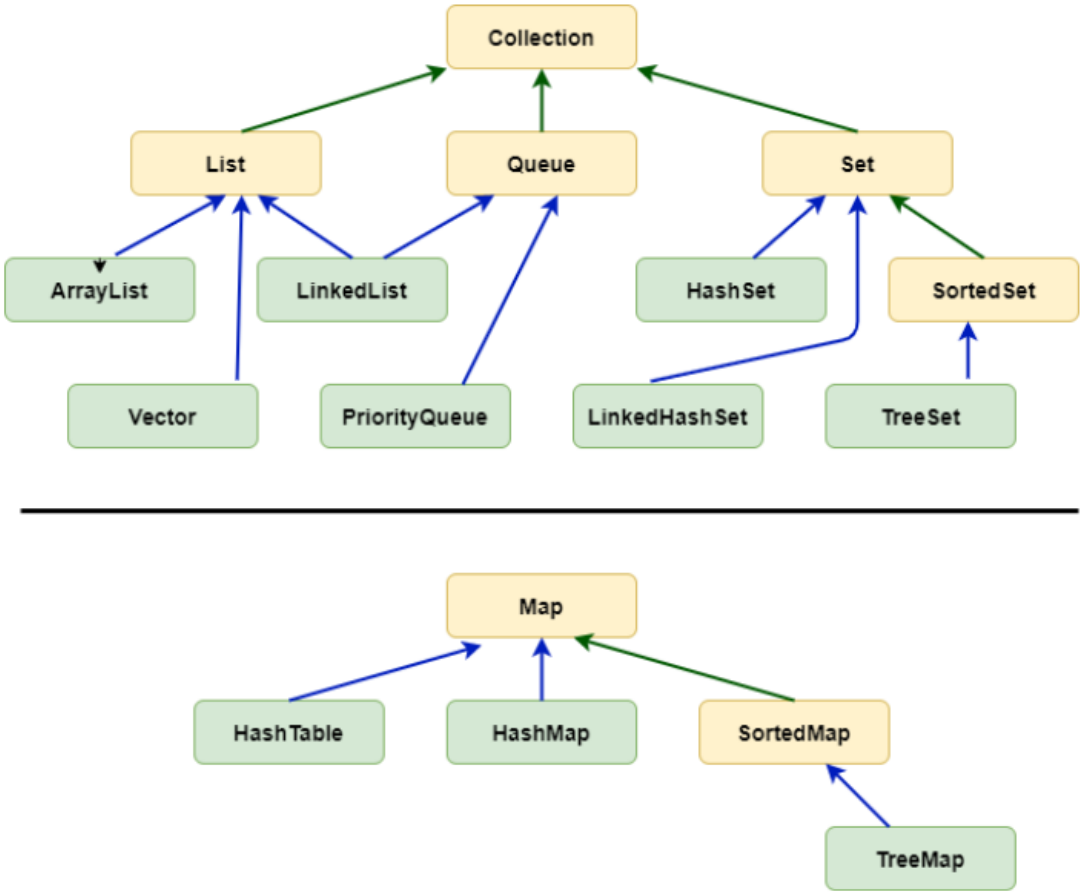
- Wrapper classes introduce additional overhead due to object creation.
- When dealing with large-scale computations, prefer primitive types for performance reasons.

Collections

- The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.
- Java Collection means a single unit of objects.
- Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.
- The Collection interface is at the root of the Java Collections Framework hierarchy.
- 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.
- It's an interface that defines the basic methods for working with collections.
- Key methods include add, remove, size, isEmpty, and contains.
- Some of the Inetrfaces are: List,Queue, Set …



Methods of Collection interface

1	public boolean add(E e)	It is used to insert an element in this collection.
2	public boolean addAll(Collection<? extends E> c)	It is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	It is used to delete an element from the collection.
4	public boolean removeAll(Collection<?> c)	It is used to delete all the elements of the specified collection from the invoking collection.
5	default boolean removeIf(Predicate<? super E> filter)	It is used to delete all the elements of the collection that satisfy the specified predicate.
6	public boolean retainAll(Collection<?> c)	It is used to delete all the elements of invoking collection except the specified collection.
7	public int size()	It returns the total number of elements in the collection.
8	public void clear()	It removes the total number of elements from the collection.
9	public boolean contains(Object element)	It is used to search an element.

10	public boolean containsAll(Collection<?> c)	It is used to search the specified collection in the collection.
11	public Iterator iterator()	It returns an iterator.
12	public Object[] toArray()	It converts collection into array.
13	public <T> T[] toArray(T[] a)	It converts collection into array. Here, the runtime type of the returned array is that of the specified array.
14	public boolean isEmpty()	It checks if collection is empty.
15	default Stream<E> parallelStream()	It returns a possibly parallel Stream with the collection as its source.
16	default Stream<E> stream()	It returns a sequential Stream with the collection as its source.
17	default Spliterator<E> spliterator()	It generates a Spliterator over the specified elements in the collection.
18	public boolean equals(Object element)	It matches two collections.
19	public int hashCode()	It returns the hash code number of the collection.

Iterator interface

- Iterator interface provides the facility of iterating the elements in a forward direction only.
- allows us to access elements of the collection and is used to iterate over the elements in the collection(Map, List or Set).
- There are only three methods in the Iterator interface.

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.

Iterable Interface

- The Iterable interface is the root interface for all the collection classes.
- The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.
- It contains only one abstract method. i.e.,

Iterator<T> iterator()

It returns the iterator over the elements of type T.

```
public interface Iterable<T> {
    Iterator<T> iterator();
}
```

Collection Interface

- The Collection interface is the interface which is implemented by all the classes in the collection framework.
- Some of the methods of Collection interface implemented by all the subclasses of Collection interface are :
  - Boolean add (Object obj),
  - Boolean addAll (Collection c),
  - void clear(), etc.

List Interface

- List interface is the child interface of Collection interface.
- It inhibits a list type data structure in which we can store the ordered collection of objects.
- It can have duplicate values.
- List interface is implemented by the classes :
  - ArrayList, LinkedList, Vector, and Stack

To instantiate the List interface, we must use :

```
List <data-type> list1= new ArrayList();
List <data-type> list2 = new LinkedList();
List <data-type> list3 = new Vector();
List <data-type> list4 = new Stack();
```

ArrayList

- The ArrayList class implements the List interface.
- It uses a dynamic array to store the duplicate element of different data types.
- The ArrayList class maintains the insertion order and is non-synchronized*.
- The elements stored in the ArrayList class can be randomly accessed.

```
import java.util.*;
class TestJavaCollection1{
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
```

***ArrayList is not thread-safe**—it does not handle concurrent modifications (add, remove, update) by multiple threads safely

LinkedList

- LinkedList implements the Collection interface.
- It uses a doubly linked list internally to store the elements.
- It can store the duplicate elements.
- It maintains the insertion order and is not synchronized.
- In LinkedList, the manipulation is fast because no shifting is required.

```
import java.util.*;
public class TestJavaCollection2{
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());
}
}
}
```

Vector

- Vector uses a dynamic array to store the data elements.
- It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

```
import java.util.*;
public class TestJavaCollection3{
public static void main(String args[]){
Vector<String> v=new Vector<String>();
v.add("Ayush");
v.add("Amit");
v.add("Ashish");
v.add("Garima");
Iterator<String> itr=v.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ayush
Amit
Ashish
Garima
```

Stack

- The stack is the subclass of Vector.
- It implements the last-in-first-out data structure, i.e., Stack.
- The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

```
import java.util.*;
public class TestJavaCollection4{
public static void main(String args[]){
Stack<String> stack = new Stack<String>();
stack.push("Ayush");
stack.push("Garvit");
stack.push("Amit");
stack.push("Ashish");
stack.push("Garima");
stack.pop();
Iterator<String> itr=stack.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ayush
Garvit
Amit
Ashish
```

Queue Interface

- Queue interface maintains the first-in-first-out order.
- It can be defined as an ordered list that is used to hold the elements which are about to be processed.
- There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated as:

```
Queue<String> q1 = new PriorityQueue();

Queue<String> q2 = new ArrayDeque();
```

PriorityQueue

- It holds the elements or objects which are to be processed by their priorities.
- PriorityQueue doesn't allow null values to be stored in the queue.

```
import java.util.*;
public class TestJavaCollection5{
public static void main(String args[]){
PriorityQueue<String> queue=new PriorityQueue<String>();
queue.add("Amit Sharma");
queue.add("Vijay Raj");
queue.add("JaiShankar");
queue.add("Raj");
System.out.println("head:"+queue.element());
System.out.println("head:"+queue.peek());
System.out.println("iterating the queue elements:");
Iterator itr=queue.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
queue.remove();
queue.poll();//returns and removes the element at the front end
System.out.println("after removing two elements:");
Iterator<String> itr2=queue.iterator();
while(itr2.hasNext()){
System.out.println(itr2.next());
}
}
}
```

Output:

```
head:Amit Sharma
head:Amit Sharma
iterating the queue elements:
Amit Sharma
Raj
JaiShankar
Vijay Raj
after removing two elements:
Raj
Vijay Raj
```

Deque Interface

- Deque interface extends the Queue interface.
- Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

Deque can be instantiated as:

```
Deque d = new ArrayDeque();
```

ArrayDeque

- ArrayDeque class implements the Deque interface.
- It facilitates us to use the Deque.
- Unlike queue, we can add or delete the elements from both the ends.
- ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

```
import java.util.*;
public class TestJavaCollection6{
public static void main(String[] args) {
//Creating Deque and adding elements
Deque<String> deque = new ArrayDeque<String>();
deque.add("Gautam");
deque.add("Karan");
deque.add("Ajay");
//Traversing elements
for (String str : deque) {
System.out.println(str);
}
}
}
```

Output:

```
Gautam
Karan
Ajay
```

Set Interface

- Set Interface in Java is present in java.util package.
- It extends the Collection interface.
- It represents the unordered set of elements which doesn't allow us to store the duplicate items.
- We can store at most one null value in Set.
- Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

```
Set<data-type> s1 = new HashSet<data-type>();
Set<data-type> s2 = new LinkedHashSet<data-type>();
Set<data-type> s3 = new TreeSet<data-type>();
```

HashSet

- HashSet class implements Set Interface.
- It represents the collection that uses a hash table for storage.
- Hashing is used to store the elements in the HashSet.
- It contains unique items.

```
import java.util.*;
public class TestJavaCollection7{
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("Ravi");
set.add("Ajay");
//Traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Vijay
Ravi
Ajay
```

LinkedHashSet

- LinkedHashSet class represents the LinkedList implementation of Set Interface.
- It extends the HashSet class and implements Set interface.
- Like HashSet, it also contains unique elements.
- It maintains the insertion order and permits null elements.

```
import java.util.*;
public class TestJavaCollection8{
public static void main(String args[]){
LinkedHashSet<String> set=new LinkedHashSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ravi
Vijay
Ajay
```

SortedSet Interface

- SortedSet is the alternate of Set interface that provides a total ordering on its elements.
- The elements of the SortedSet are arranged in the increasing (ascending) order.
- The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

The SortedSet can be instantiated as:

```
SortedSet<data-type> set = new TreeSet();
```

TreeSet

- Java TreeSet class implements the Set interface that uses a tree for storage.
- Like HashSet, TreeSet also contains unique elements.
- However, the access and retrieval time of TreeSet is quite fast.
- The elements in TreeSet stored in ascending order.

```
import java.util.*;
public class TestJavaCollection9{
public static void main(String args[]){
//Creating and adding elements
TreeSet<String> set=new TreeSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ajay
Ravi
Vijay
```

Java List

Create a List

- You create a List instance by creating an instance of one of the classes that implements the List interface.
- Here are a few examples of how to create a `List` instance:

```
List listA = new ArrayList();
List listB = new LinkedList();
List listC = new Vector();
List listD = new Stack();
```

- Remember, most often you will use the ArrayList class, but there can be cases where using one of the other implementations might make sense.

Java ArrayList

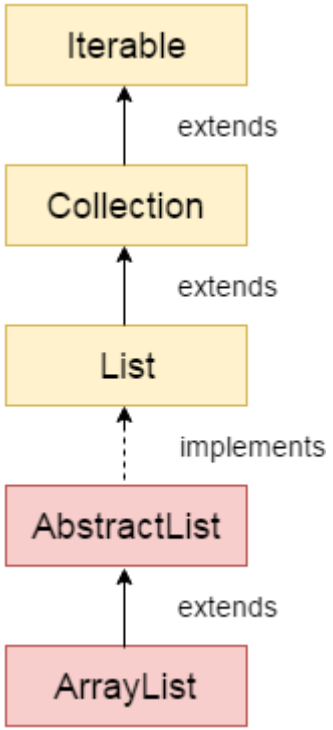
- Java **ArrayList** class uses a *dynamic array* for storing the elements.
- It is like an array, but there is *no size limit*.
- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows 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.

Java ArrayList

- 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.

```
ArrayList<int> al = ArrayList<int>(); // does not work

ArrayList<Integer> al = new ArrayList<Integer>(); // works fine
```



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 a collection. Now it is type-safe, so typecasting is not required at runtime.

Non-generic example of creating a Java collection:

```
ArrayList list=new ArrayList();//creating old non-generic arraylist
```

generic example of creating java collection:

```
ArrayList<String> list=new ArrayList<String>();//creating new generic arraylist
```

- In a generic collection, we specify the type in angular braces.
- Now ArrayList is forced to have the only specified type of object in it.

ArrayList Example

```
import java.util.*;
 public class ArrayListExample1{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();//Creating arraylist
      list.add("Mango");//Adding object in arraylist
      list.add("Apple");
      list.add("Banana");
      list.add("Grapes");
      //Printing the arraylist object
      System.out.println(list);
 }
}
```

Output:

[Mango, Apple, Banana, Grapes]

Insert Elements in a Java List

```
List<String> listA = new ArrayList<>();


listA.add("element 1");
listA.add("element 2");
listA.add("element 3");
```

- The first three **add()** calls add a String instance to the end of the list.

Insert null Values

```
Object element = null;

List<Object> list = new ArrayList<>();

list.add(element);
```

Insert Elements at Specific Index

- inserting an element at index 0 into a Java List

```
list.add(0, "element 4");
```

Insert All Elements From One List Into Another

- List addAll() method
- example adds all elements from listSource into listDest

```
List<String> listSource = new ArrayList<>();
listSource.add("123");
listSource.add("456");
List<String> listDest   = new ArrayList<>();
listDest.addAll(listSource);
```

Note: can add all elements from a List or Set into a List with **addAll()**

Get Elements From a Java List

- get(int index) method

```
List<String> listA = new
ArrayList<>();

listA.add("element 0");
listA.add("element 1");
listA.add("element 2");

//access via index
String element0 = listA.get(0);
String element1 = listA.get(1);
String element3 = listA.get(2);
```

Find Elements in a List

- indexOf()
- lastIndexOf()

```
List<String> list = new ArrayList<>();

String element1 = "element 1";
String element2 = "element 2";

list.add(element1);
list.add(element2);

int index1 = list.indexOf(element1);
int index2 = list.indexOf(element2);

System.out.println("index1 = " + index1);
System.out.println("index2 = " + index2);
```

output:

```
index1 = 0
index2 = 1
```

Find Last Occurrence of Element in a List

- lastIndexOf() method

```
List<String> list = new ArrayList<>();


String element1 = "element 1";
String element2 = "element 2";

list.add(element1);
list.add(element2);
list.add(element1);


int lastIndex = list.lastIndexOf(element1);
System.out.println("lastIndex = " + lastIndex);
```

O/P:

lastIndex = 2

Checking if List Contains Element

- contains() method

```
List<String> list = new ArrayList<>();

String element1 = "element 1";

list.add(element1);
boolean containsElement = list.contains("element 1");

System.out.println(containsElement);
```

O/P:

true

Remove Elements From a Java List

2 Methods:

- remove(Object element)
- remove(int index)

- Removes that element in the list, if it is present.
- All subsequent elements in the list are then moved up in the list.

```
List<String> list = new ArrayList<>();
String element = "first element";
list.add(element);
list.remove(element);
```

```
List<String> list = new ArrayList<>();
list.add("element 0");
list.add("element 1");
list.add("element 2");
list.remove(0);//remove(int index)
```

Remove All Elements From a Java List

- clear() method which removes all elements from the list.

```
List<String> list = new ArrayList<>();
list.add("object 1");
list.add("object 2");
//etc.

list.clear(); //will make list empty
```

Retain All Elements From One List in Another

retainAll()

- removes all the elements from the target List which are not found in the other List.
- The resulting List is the ***intersection*** of the two lists

```
List<String> list      = new ArrayList<>();
List<String> otherList = new ArrayList<>();
String element1 = "element 1";
String element2 = "element 2";
String element3 = "element 3";
String element4 = "element 4";
list.add(element1);
list.add(element2);
list.add(element3);
otherList.add(element1);
otherList.add(element3);
otherList.add(element4);
list.retainAll(otherList);
```

O/P:
List will have element1 and element3

List Size

- **size() method:** number of elements

```
List<String> list = new ArrayList<>();
list.add("object 1");
list.add("object 2");

int size = list.size();
```

Convert List to Set

- Convert list to set using addall() method.
- Will remove all duplicate elements

```
List<String> list       = new ArrayList<>();

list.add("element 1");
list.add("element 2");
list.add("element 3");
list.add("element 3");

Set<String> set = new HashSet<>();
set.addAll(list); //
```

Convert List to Array

- toArray() method

```
List<String> list       = new ArrayList<>();

list.add("element 1");
list.add("element 2");
list.add("element 3");
list.add("element 3");

Object[] objects = list.toArray();
```

Convert Array to List

- **Arrays.asList()** method : converts the array to a List

```
String[] values = new String[]{ "one", "two", "three" };

List<String> list = (List<String>) Arrays.asList(values);
```

Sort List

- Collections sort() method

```
List<String> list = new ArrayList<>();

list.add("c");
list.add("b");
list.add("a");

Collections.sort(list);
```

Iterate List

- Using an Iterator
- Using a for-each loop
- Using a for loop
- ……

Iterate List Using Iterator

```
List<String> list = new ArrayList<>();

list.add("first");
list.add("second");
list.add("third");

Iterator<String> iterator = list.iterator();
while(iterator.hasNext()) {
    String next = iterator.next();
}
```

Iterate List Using For-Each Loop

```
List<String> list = new ArrayList<String>();

//add elements to list

for(String element : list) {
    System.out.println(element);
}
```

Iterate List Using For Loop

```
List list = new ArrayList();

list.add("first");
list.add("second");
list.add("third");

for(int i=0; i < list.size(); i++) {
    Object element = list.get(i);
}
```

```
List<String> list = new ArrayList<String>();

list.add("first");
list.add("second");
list.add("third");

for(int i=0; i < list.size(); i++) {
    String element = list.get(i);
}
```

Query:
which one to prefer for each/for or iterator?

Trying to remove items using a **for loop** or a **for-each loop** would not work correctly because the collection is changing size at the same time that the code is trying to loop.

```
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> myList = new ArrayList<>();
        myList.add("A");
        myList.add("B");
        myList.add("C");
        myList.add("D");

        // This is an incorrect way to remove elements from the list
        for (String item : myList) {
            if (item.equals("B")) {
                myList.remove(item); // This will cause a ConcurrentModificationException
            }
        }
    }
}
```

Removing elements Using iterator is safe

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

public class ArrayListExample {
    public static void main(String[] args) {
        List<String> myList = new ArrayList<>();
        myList.add("A");
        myList.add("B");
        myList.add("C");
        myList.add("B");
        myList.add("D");

        Iterator<String> iterator = myList.iterator();
        while (iterator.hasNext()) {
            String item = iterator.next();
            if (item.equals("B")) {
                iterator.remove(); // Safely removes "B" from the list
            }
        }

        System.out.println("ArrayList after removing 'B' with Iterator: " + myList);

}
}
```

Output:
ArrayList after removing 'B' with Iterator: [A, C, D]

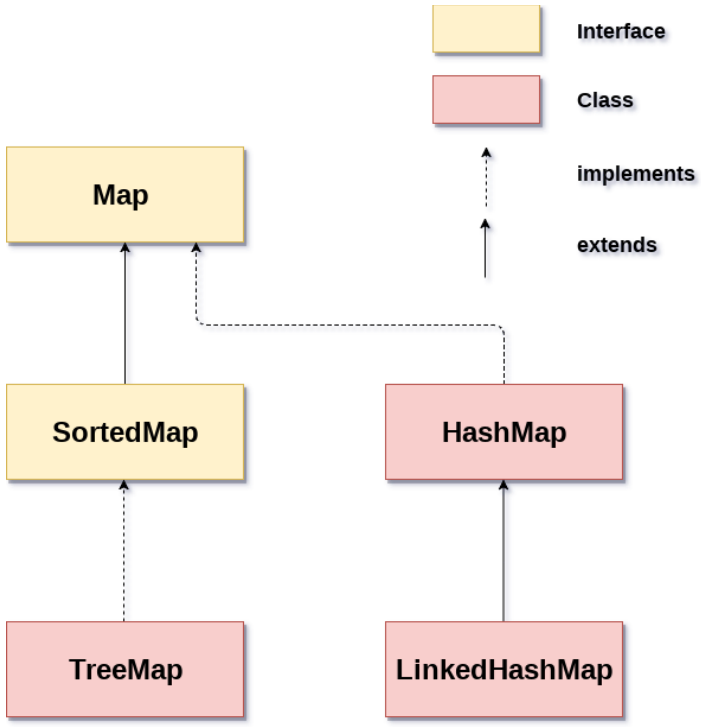
Java Map

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. A Map contains unique keys.
- A Map is useful if you have to search, update or delete elements on the basis of a key.

Java Map Hierarchy

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



Useful methods of Map interface

Method	Description
V put(Object key, Object value)	It is used to insert an entry in the map.
void putAll(Map map)	It is used to insert the specified map in the map.
V putIfAbsent(K key, V value)	It inserts the specified value with the specified key in the map only if it is not already specified.
V remove(Object key)	It is used to delete an entry for the specified key.
boolean remove(Object key, Object value)	It removes the specified values with the associated specified keys from the map.
Set keySet()	It returns the Set view containing all the keys.
Set<Map.Entry<K,V>> entrySet()	It returns the Set view containing all the keys and values.
void clear()	It is used to reset the map.

Useful methods of Map interface

boolean containsValue(Object value)	This method returns true if some value equal to the value exists within the map, else return false.
boolean containsKey(Object key)	This method returns true if some key equal to the key exists within the map, else return false.
boolean equals(Object o)	It is used to compare the specified Object with the Map.
void forEach(BiConsumer<? super K,? super V> action)	It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.
V get(Object key)	This method returns the object that contains the value associated with the key.
V getOrDefault(Object key, V defaultValue)	It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key.
int hashCode()	It returns the hash code value for the Map
boolean isEmpty()	This method returns true if the map is empty; returns false if it contains at least one key.

Map.Entry Interface

- The Map.Entry<K, V> interface is a nested static interface inside the Map interface in Java.
- It represents a key-value pair stored in a Map and provides methods to access and manipulate the entry

```
public interface Map<K, V> {
    interface Entry<K, V> {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

Methods of Map.Entry interface

Methods

- K getKey() : Returns the key of the entry
- V getValue() : Returns the value associated with the key
- V setValue(V value) : Updates the value associated with the key

Java Map Example: Generic

```
import java.util.*;
class MapExample2{
 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");
  //Elements can traverse in any order

for(Map.Entry m : map.entrySet()){
   System.out.println(m.getKey()+" "+m.getValue());
  }

 }
}
```

Output:

102 Rahul
100 Amit
101 Vijay

Using setValue() to Modify Values

```
import java.util.*;

public class MapModifyExample {
    public static void main(String[] args) {
        Map<String, Integer> prices = new HashMap<>();
        prices.put("Apple", 100);
        prices.put("Banana", 50);
        prices.put("Cherry", 150);

        // Increase all prices by 10
        for (Map.Entry<String, Integer> entry : prices.entrySet()) {
            entry.setValue(entry.getValue() + 10);
        }

        System.out.println(prices);
    }
}
```

O/P:
{Apple=110, Banana=60, Cherry=160}

Java HashMap

- Java HashMap contains values based on the key.
- Java HashMap contains only unique keys.
- Java HashMap may have one null key and multiple null values.
- Java HashMap is non synchronized.
- Java HashMap maintains no order.
- The initial default capacity of Java HashMap class is 16

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 values.

Constructor	Description
HashMap()	It is used to construct a default HashMap.
HashMap(Map<? extends K,? extends V> m)	It is used to initialize the hash map by using the elements 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 loadFactor)	It is used to initialize both the capacity and load factor 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 isEmpty()	It is used to return true if this map contains no key-value mappings.
Object clone()	It is used to return a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
Set entrySet()	It is used to return a collection view of the mappings contained in this map.
Set keySet()	It is used to return a set view of the keys contained in this map.
V put(Object key, Object value)	It is used to insert an entry in the map.
void putAll(Map map)	It is used to insert the specified map in the map.
V putIfAbsent(K key, V value)	It inserts the specified value with the specified key in the map only if it is not already specified.
V remove(Object key)	It is used to delete an entry for the specified key.
boolean remove(Object key, Object value)	It removes the specified values with the associated specified keys from the map.
V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)	It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null.
V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null.

boolean containsValue(Object value)	This method returns true if some value equal to the value exists within the map, else return false.
boolean containsKey(Object key)	This method returns true if some key equal to the key exists within the map, else return false.
boolean equals(Object o)	It is used to compare the specified Object with the Map.
void forEach(BiConsumer<? super K,? super V> action)	It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.
V get(Object key)	This method returns the object that contains the value associated with the key.
V getOrDefault(Object key, V defaultValue)	It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key.
boolean isEmpty()	This method returns true if the map is empty; returns false if it contains at least one key.
V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)	If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
V replace(K key, V value)	It replaces the specified value for a specified key.
boolean replace(K key, V oldValue, V newValue)	It replaces the old value with the new value for a specified key.
void replaceAll(BiFunction<? super K,? super V,? extends V> function)	It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
Collection<V> values()	It returns a collection view of the values contained in the map.
int size()	This method returns the number of entries in the map.

Java HashMap Example

```
import java.util.*;
public class HashMapExample1{
 public static void main(String args[]){
   HashMap<Integer,String> map=new HashMap<Integer,String>();//Creating HashMap
   map.put(1,"Mango");  //Put elements in Map
   map.put(2,"Apple");
   map.put(3,"Banana");
   map.put(4,"Grapes");

   System.out.println("Iterating Hashmap...");
   for(Map.Entry m : map.entrySet()){
    System.out.println(m.getKey()+" "+m.getValue());
   }
}
}
```

Iterating Hashmap...
1 Mango
2 Apple
3 Banana
4 Grapes

Note: You cannot store duplicate keys in HashMap.
However, if you try to store duplicate key with another value, it will replace the value.

Java HashMap example to add() elements

```
import java.util.*;
class HashMap1{
 public static void main(String args[]){
  HashMap<Integer,String> hm=new HashMap<Integer,String>();
   System.out.println("Initial list of elements: "+hm);
    hm.put(100,"Amit");
    hm.put(101,"Vijay");
    hm.put(102,"Rahul");
    System.out.println("After invoking put() method ");
    for(Map.Entry m:hm.entrySet()){
     System.out.println(m.getKey()+" "+m.getValue());
    }

    hm.putIfAbsent(103, "Gaurav");
    System.out.println("After invoking putIfAbsent() method ");
    for(Map.Entry m:hm.entrySet()){
        System.out.println(m.getKey()+" "+m.getValue());
       }
    HashMap<Integer,String> map=new HashMap<Integer,String>();
    map.put(104,"Ravi");
    map.putAll(hm);
    System.out.println("After invoking putAll() method ");
    for(Map.Entry m:map.entrySet()){
        System.out.println(m.getKey()+" "+m.getValue());
       }
 }
}
```

```
Initial list of elements: {}
After invoking put() method
100 Amit
101 Vijay
102 Rahul
After invoking putIfAbsent() method
100 Amit
101 Vijay
102 Rahul
103 Gaurav
After invoking putAll() method
100 Amit
101 Vijay
102 Rahul
103 Gaurav
104 Ravi
```

Java HashMap example to remove() elements

```
import java.util.*;
public class HashMap2 {
  public static void main(String args[]) {
   HashMap<Integer,String> map=new HashMap<Integer,String>();
     map.put(100,"Amit");
     map.put(101,"Vijay");
     map.put(102,"Rahul");
     map.put(103, "Gaurav");
   System.out.println("Initial list of elements: "+map);
   //key-based removal
   map.remove(100);
   System.out.println("Updated list of elements: "+map);

  //key-value pair based removal
   map.remove(102, "Rahul");
   System.out.println("Updated list of elements: "+map);
  }
}
```

Hashmap: Example

```
import java.util.*;

public class HashMapOperations {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();

        // Adding key-value pairs
        map.put("A", 10);
        map.put("B", 20);
        map.put("C", 30);

        // Retrieving value
        System.out.println("Value for key A: " + map.get("A"));  // Output: 10

        // Checking key existence
        System.out.println("Contains key B? " + map.containsKey("B"));  // Output: true

        // Checking value existence
        System.out.println("Contains value 40? " + map.containsValue(40));  // Output: false

        // Removing a key-value pair
        map.remove("C");

        // Size of HashMap
        System.out.println("Size: " + map.size());  // Output: 2
    }
}
```

Difference between HashSet and HashMap

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

HashSet

- Java HashSet class is used to create a collection that uses a hash table for storage.
- 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.
- The initial default capacity of HashSet is 16

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<? extends E> 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(E e)	It is used to add 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 the 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 the set.
9)	Spliterator<E>	spliterator()	It is used to create a late-binding and fail-fast Spliterator over the elements in the set.

Java HashSet Example

```
import java.util.*;
class HashSet1{
 public static void main(String args[]){
  //Creating HashSet and adding elements
    HashSet<String> set=new HashSet();
          set.add("One");
          set.add("Two");
          set.add("Three");
          set.add("Four");
          set.add("Five");
          Iterator<String> i=set.iterator();
          while(i.hasNext())
          {
          System.out.println(i.next());
          }
 }
}
```

```
Five
One
Four
Two
Three
```

Java HashSet example ignoring duplicate elements

```
import java.util.*;
class HashSet2{
 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("Ravi");
  set.add("Ajay");
  //Traversing elements
  Iterator<String> itr=set.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
```

```
Ajay
Vijay
Ravi
```

Java HashSet from another Collection

```
import java.util.*;
class HashSet4{
 public static void main(String args[]){
   ArrayList<String> list=new ArrayList<String>();
        list.add("Ravi");
        list.add("Vijay");
        list.add("Ajay");

        HashSet<String> set=new HashSet(list);
        set.add("Gaurav");
        Iterator<String> i=set.iterator();
        while(i.hasNext())
        {
        System.out.println(i.next());
        }
 }
}
```

```
Vijay
Ravi
Gaurav
Ajay
```

Practice Question: List Operations & Iteration

Q1. WAP that

- Creates an ArrayList of strings ("Apple", "Banana", "Cherry", "Date").
- Uses an iterator to remove elements containing the letter 'a' (case-insensitive).
- Prints the modified list.

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

public class ListIteration {
    public static void main(String[] args) {
        ArrayList<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");
        fruits.add("Date");

        Iterator<String> iterator = fruits.iterator();
        while (iterator.hasNext()) {
            String fruit = iterator.next();
            if (fruit.toLowerCase().contains("a")) {
                iterator.remove(); // Safely remove using iterator
            }
        }

        System.out.println(fruits); // Output: [Cherry]
    }
}
```

Practice Question: Set & SortedSet (TreeSet)

Q2. WAP that

- Creates a HashSet with integers (5, 3, 8, 1, 2).
- Converts it to a TreeSet (sorted order).
- Prints the smallest and largest elements.

```
import java.util.HashSet;
import java.util.TreeSet;

public class SetOperations {
    public static void main(String[] args) {
        HashSet<Integer> numbers = new HashSet<>();
        numbers.add(5);
        numbers.add(3);
        numbers.add(8);
        numbers.add(1);
        numbers.add(2);

        TreeSet<Integer> sortedNumbers = new TreeSet<>(numbers);
        System.out.println("Smallest: " + sortedNumbers.first()); // 1
        System.out.println("Largest: " + sortedNumbers.last());   // 8
    }
}
```

Practice Question: Map & SortedMap (TreeMap)

Q3. WAP that

- Creates a HashMap with country-capital pairs ("India" → "Delhi", "Japan" → "Tokyo", "USA" → "Washington").
- Converts it to a TreeMap (sorted by keys).
- Prints capitals in alphabetical order of countries.

```
import java.util.*;

public class MapSorting {
    public static void main(String[] args) {
        HashMap<String, String> countryCapitals = new HashMap<>();
        countryCapitals.put("India", "Delhi");
        countryCapitals.put("Japan", "Tokyo");
        countryCapitals.put("USA", "Washington");

        TreeMap<String, String> sortedMap = new TreeMap<>(countryCapitals);
        System.out.println("Capitals: " + sortedMap.values());
        // Output: [Delhi, Tokyo, Washington]
    }
}
```

Practice Question: Collections Class Utilities

Q4. WAP that

- Creates a List of integers (10, 5, 20, 15, 25).
- Uses Collections.sort() to sort it in descending order.
- Reverses the list using Collections.reverse().
- Prints the final list.

```
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class CollectionsDemo {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(10, 5, 20, 15, 25);
        Collections.sort(numbers);          // Ascending sort
        Collections.reverse(numbers);      // Reverse to get descending
        System.out.println(numbers);       // Output: [25, 20, 15, 10, 5]
    }
}
```

Practice Question: Wrapper classes & Autoboxing

Q5. WAP that

- Creates an ArrayList<Integer> and adds values using autoboxing (1, 2, 3).
- Computes the sum by unboxing the values.
- Prints the sum.

```
import java.util.ArrayList;

public class WrapperExample {
    public static void main(String[] args) {
        ArrayList<Integer> numbers = new ArrayList<>();
        numbers.add(1);  // Autoboxing (int -> Integer)
        numbers.add(2);
        numbers.add(3);

        int sum = 0;
        for (int num : numbers) {  // Unboxing (Integer -> int)
            sum += num;
        }
        System.out.println("Sum: " + sum); // Output: 6
    }
}
```

Checkpoint

Create a Student class with attributes:

- name (String)
- cgpa (double)
- sapid (long)
- Define a display() method in the Student class to print all details.

In the main program:

- Create an ArrayList of 3 Student objects.
- Add students with names, random CGPA (5.0 to 10.0), and SAP IDs.
- Print details of all students using the display() method.

Expected Output:

Student Name: Aarav | CGPA: 8.5 | SAP ID: 5000123456
Student Name: Priya | CGPA: 9.2 | SAP ID: 5000234567
Student Name: Rahul | CGPA: 7.8 | SAP ID: 5000345678

Solution

```
class Student {
    String name;
    double cgpa;
    long sapid;

    // Constructor
    public Student(String name, double cgpa, long sapid) {
        this.name = name;
        this.cgpa = cgpa;
        this.sapid = sapid;
    }

    // Display method
    public void display() {
        System.out.printf("Student Name: %s | CGPA: %.1f | SAP ID: %d%n", name, cgpa, sapid);
    }
}
```

```
import java.util.ArrayList;

public class StudentList {
    public static void main(String[] args) {
        ArrayList<Student> students = new ArrayList<>();

        // Add students with Indian names
        students.add(new Student("Aarav", 8.5, 5000123456L));
        students.add(new Student("Priya", 9.2, 5000234567L));
        students.add(new Student("Rahul", 7.8, 5000345678L));

        // Display all students
        for (Student s : students) {
            s.display();
        }
    }
}
```