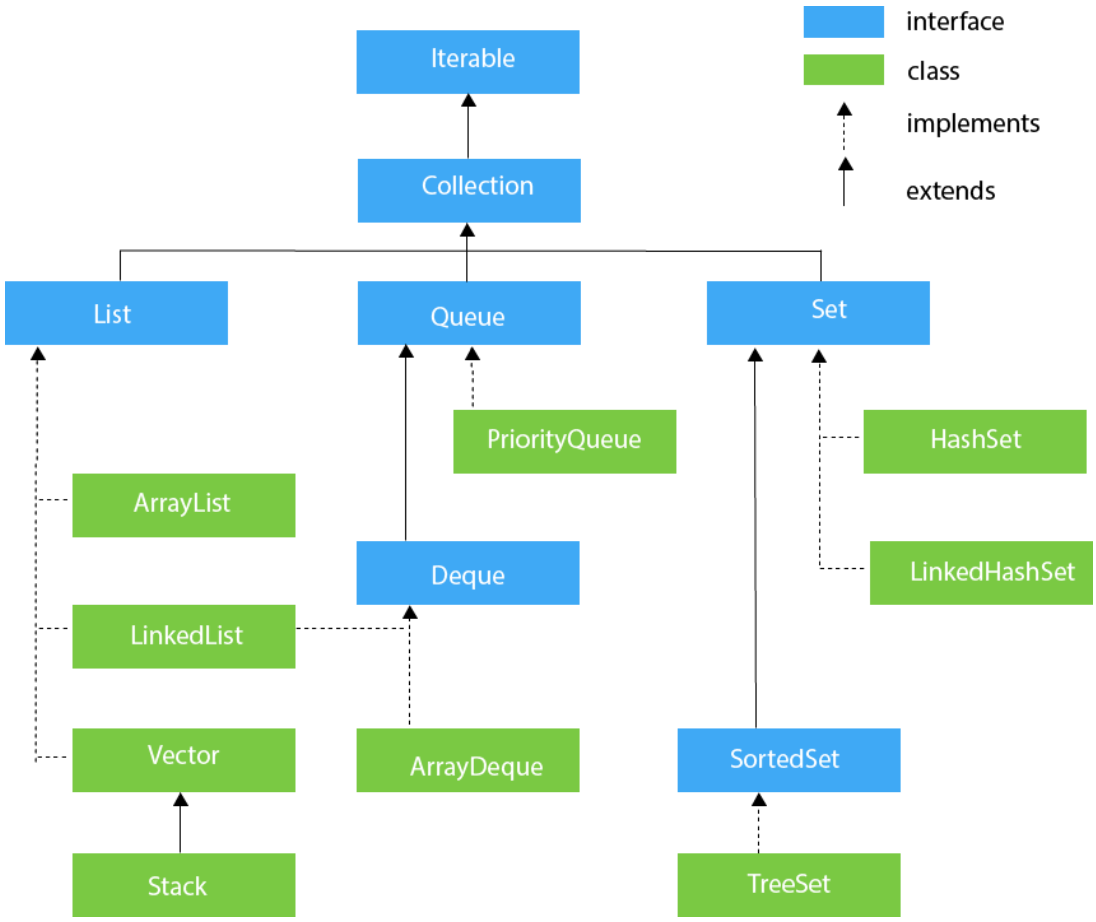
Collections in Java

The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

Hierarchy of Collection Framework



Methods of Collection interface

No.	Method	Description
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.

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.

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.

Collection Interface

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

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();
```

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

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.

Consider the following example.

```
import java.util.*;

class TestJavaCollection1{

public static void main(String args[]){

ArrayList<String> list=new ArrayList<String>();//Creating arraylist

list.add("Ben");//Adding object in arraylist

list.add("Buttler");

list.add("Broad");

list.add("Bopara");
```

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

Output

Ben

Buttler

Broad

Bopara

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.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection2{
public static void main(String args[]){
```

```
LinkedList<String> al=new LinkedList<String>();
al.add("Roy");
al.add("Bairstow");
al.add("Root");
al.add("Morgan");
Iterator<String> itr=al.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

Roy

Bairstow

Root

Morgan

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.

Consider the following example.

```
import java.util.*;

public class TestJavaCollection3{

public static void main(String args[]){

Vector<String> v=new Vector<String>();

v.add("Sam");

v.add("Tom");

v.add("Woakes");

v.add("Jofra");

Iterator<String> itr=v.iterator();

while(itr.hasNext()){

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

}

}

}
```

Output:

Sam

Tom

Woakes

Jofra

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.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection4{
public static void main(String args[]){
Stack<String> stack = new Stack<String>();
stack.push("Plunket");
stack.push("Rashid");
stack.push("Moeen");
stack.push("Wood");
stack.push("Anderson");
stack.pop();
Iterator<String> itr=stack.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

Plunket

Rashid

Moeen

Wood

Anderson

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();
```

There are various classes that implement the Queue interface, some of them are given below.

PriorityQueue

The PriorityQueue class implements the Queue interface. 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.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection5{
public static void main(String args[]){
PriorityQueue<String> queue=new PriorityQueue<String>();
queue.add("Graeme Swann");
queue.add("Tim Bresnan");
queue.add("Chris Jordan");
queue.add("Chris Tremlett");
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();
System.out.println("after removing two elements:");
Iterator<String> itr2=queue.iterator();
while(itr2.hasNext()){
System.out.println(itr2.next());
}
}
```

```
}
```

Output:

```
head:Graeme Swann

head:Graeme Swann

iterating the queue elements:

Graeme Swann

Tim Bresnan

Chris Jordan

Chris Tremlett

after removing two elements:

Tim Bresnan

Chris Tremlett
```

Deque Interface

Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. 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.

Consider the following example.

```
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("Collingwood");
deque.add("Flintoff");
deque.add("Strauss");
//Traversing elements
for (String str : deque) {
System.out.println(str);
}
}
}
```

Output:

Collingwood

Flintoff

Strauss

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.

Consider the following example.

```
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("Cook");
```

```
set.add("Pietersen");

set.add("Trott");

set.add("Bell");

//Traversing elements

Iterator<String> itr=set.iterator();

while(itr.hasNext()){

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

}

}

}
```

Output:

Cook

Pietersen

Trott

Bell

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.

Consider the following example.

```
import java.util.*;

public class TestJavaCollection8{

public static void main(String args[]){

LinkedHashSet<String> set=new LinkedHashSet<String>();

set.add("Compton");

set.add("Carberry");

set.add("Billings”);

set.add("Prior");

Iterator<String> itr=set.iterator();

while(itr.hasNext()){

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

}

}

}
```

Output:

Compton

Carberry

Billings

Prior

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.

Consider the following example:

```
import java.util.*;
public class TestJavaCollection9{
public static void main(String args[]){
//Creating and adding elements
TreeSet<String> set=new TreeSet<String>();
set.add("Giles");
set.add("Jimmy");
set.add("Hoggard");
set.add("Nixon");
```

```
//traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output

Giles

Jimmy

Hoggard

Nixon

Java ArrayList

Java ArrayList class uses a dynamic array for storing the elements. It is like an array, but there is no size limit. We can add or remove elements anytime. So, it is much more flexible than the traditional array. It is found in the java.util package. It is like the Vector in C++.

The ArrayList in Java can have the duplicate elements also. It implements the List interface so we can use all the methods of List interface here. The ArrayList maintains the insertion order internally.

It inherits the AbstractList class and implements List interface.

ArrayList class declaration

public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, Serializable

Constructors of ArrayList

Constructor	Description
ArrayList()	It is used to build an empty array list.
ArrayList(Collection<? extends E> 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, E element)	It is used to insert the specified element at the specified position in a list.
boolean add(E e)	It is used to append the specified element at the end of a list.
boolean addAll(Collection<? extends E> 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.
boolean addAll(int index, Collection<? extends E> c)	It is used to append all the elements in the specified collection, starting at the specified position of the list.
void clear()	It is used to remove all of the elements from this list.
void ensureCapacity(int requiredCapacity)	It is used to enhance the capacity of an ArrayList instance.
E get(int index)	It is used to fetch the element from the particular position of the list.
boolean isEmpty()	It returns true if the list is empty, otherwise false.
Iterator()	
listIterator()	
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.
<T> T[] toArray(T[] a)	It is used to return an array containing all of the elements in this list in the correct order.

Object clone()	It is used to return a shallow copy of an ArrayList.
boolean contains(Object o)	It returns true if the list contains the specified element
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.
E remove(int index)	It is used to remove the element present at the specified position in the list.
boolean remove(Object o)	It is used to remove the first occurrence of the specified element.
boolean removeAll(Collection<?> c)	It is used to remove all the elements from the list.
boolean removeIf(Predicate<? super E> filter)	It is used to remove all the elements from the list that satisfies the given predicate.
protected void removeRange(int fromIndex, int toIndex)	It is used to remove all the elements lies within the given range.
void replaceAll(UnaryOperator<E> operator)	It is used to replace all the elements from the list with the specified element.
void retainAll(Collection<?> c)	It is used to retain all the elements in the list that are present in the specified collection.
E set(int index, E element)	It is used to replace the specified element in the list, present at the specified position.
void sort(Comparator<? super E> c)	It is used to sort the elements of the list on the basis of specified comparator.
Spliterator<E> spliterator()	It is used to create spliterator over the elements in a list.
List<E> subList(int fromIndex, int toIndex)	It is used to fetch all the elements lies within the given range.
int size()	It is used to return the number of elements present in the list.
void trimToSize()	It is used to trim the capacity of this ArrayList instance to be the list's current size.

Java 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]

Iterating ArrayList using Iterator

Let's see an example to traverse ArrayList elements using the Iterator interface.

```
import java.util.*;
public class ArrayListExample2{
 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");
  //Traversing list through Iterator
  Iterator itr=list.iterator();//getting the Iterator
  while(itr.hasNext()){//check if iterator has the elements
   System.out.println(itr.next());//printing the element and move to next
  }
 }
}
```

Output:

Mango

Apple

Banana

Grapes

Get and Set ArrayList

The get() method returns the element at the specified index, whereas the set() method changes the element.

```
import java.util.*;
public class ArrayListExample4{
 public static void main(String args[]){
  ArrayList<String> al=new ArrayList<String>();
```

```
  al.add("Mango");
  al.add("Apple");
  al.add("Banana");
  al.add("Grapes");
  //accessing the element
  System.out.println("Returning element: "+al.get(1));//it will return the 2nd element, because index starts from 0
  //changing the element
  al.set(1,"Dates");
  //Traversing list
  for(String fruit:al)
    System.out.println(fruit);

 }
}
```

Output:

Returning element: Apple

Mango

Dates

Banana

Grapes

How to Sort ArrayList

The java.util package provides a utility class Collections which has the static method sort(). Using the Collections.sort() method, we can easily sort the ArrayList.

```
import java.util.*;
class SortArrayList{
 public static void main(String args[]){
  //Creating a list of fruits
  List<String> list1=new ArrayList<String>();
  list1.add("Mango");
  list1.add("Apple");
  list1.add("Banana");
  list1.add("Grapes");
  //Sorting the list
  Collections.sort(list1);
   //Traversing list through the for-each loop
  for(String fruit:list1)
    System.out.println(fruit);

 System.out.println("Sorting numbers...");
  //Creating a list of numbers
  List<Integer> list2=new ArrayList<Integer>();
  list2.add(21);
```

```
  list2.add(11);
  list2.add(51);
  list2.add(1);
  //Sorting the list
  Collections.sort(list2);
   //Traversing list through the for-each loop
  for(Integer number:list2)
    System.out.println(number);
 }
}
```

Output:

Apple

Banana

Grapes

Mango

Sorting numbers...

1

11

21

51

Java ArrayList example to add elements

Here, we see different ways to add an element.

```
import java.util.*;
 class ArrayList7{
 public static void main(String args[]){
  ArrayList<String> al=new ArrayList<String>();
        System.out.println("Initial list of elements: "+al);
        //Adding elements to the end of the list
        al.add("Ravi");
        al.add("Vijay");
        al.add("Ajay");
        System.out.println("After invoking add(E e) method: "+al);
        //Adding an element at the specific position
        al.add(1, "Gaurav");
        System.out.println("After invoking add(int index, E element) method: "+al);
        ArrayList<String> al2=new ArrayList<String>();
        al2.add("Sonoo");
        al2.add("Hanumat");
        //Adding second list elements to the first list
        al.addAll(al2);
        System.out.println("After invoking addAll(Collection<? extends E> c) method: "+al);
        ArrayList<String> al3=new ArrayList<String>();
```

```
        al3.add("John");
        al3.add("Rahul");
        //Adding second list elements to the first list at specific position
        al.addAll(1, al3);
        System.out.println("After invoking addAll(int index, Collection<? extends E> c) method: "+al);
 }
}
```

Output:

Initial list of elements: []

After invoking add(E e) method: [Ravi, Vijay, Ajay]

After invoking add(int index, E element) method: [Ravi, Gaurav, Vijay, Ajay]

After invoking addAll(Collection<? extends E> c) method:

[Ravi, Gaurav, Vijay, Ajay, Sonoo, Hanumat]

After invoking addAll(int index, Collection<? extends E> c) method:

[Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat]

Java ArrayList example to remove elements

Here, we see different ways to remove an element.

```
import java.util.*;
 class ArrayList8 {
```

```
public static void main(String [] args)
{
  ArrayList<String> al=new ArrayList<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ajay");
  al.add("Anuj");
  al.add("Gaurav");
  System.out.println("An initial list of elements: "+al);
  //Removing specific element from arraylist
  al.remove("Vijay");
  System.out.println("After invoking remove(object) method: "+al);
  //Removing element on the basis of specific position
  al.remove(0);
  System.out.println("After invoking remove(index) method: "+al);

  //Creating another arraylist
  ArrayList<String> al2=new ArrayList<String>();
  al2.add("Ravi");
  al2.add("Hanumat");
  //Adding new elements to arraylist
  al.addAll(al2);
```

```
        System.out.println("Updated list : "+al);
        //Removing all the new elements from arraylist
        al.removeAll(al2);
        System.out.println("After invoking removeAll() method: "+al);
        //Removing elements on the basis of specified condition
        al.removeIf(str -> str.contains("Ajay"));   //Here, we are using Lambda expression
        System.out.println("After invoking removeIf() method: "+al);
        //Removing all the elements available in the list
        al.clear();
        System.out.println("After invoking clear() method: "+al);
    }
}
```

Output:

An initial list of elements: [Ravi, Vijay, Ajay, Anuj, Gaurav]

After invoking remove(object) method: [Ravi, Ajay, Anuj, Gaurav]

After invoking remove(index) method: [Ajay, Anuj, Gaurav]

Updated list : [Ajay, Anuj, Gaurav, Ravi, Hanumat]

After invoking removeAll() method: [Ajay, Anuj, Gaurav]

After invoking removeIf() method: [Anuj, Gaurav]

After invoking clear() method: []

Java ArrayList example of isEmpty() method

```
import java.util.*;
 class ArrayList10{

       public static void main(String [] args)
       {
         ArrayList<String> al=new ArrayList<String>();
         System.out.println("Is ArrayList Empty: "+al.isEmpty());
         al.add("Ravi");
         al.add("Vijay");
         al.add("Ajay");
         System.out.println("After Insertion");
         System.out.println("Is ArrayList Empty: "+al.isEmpty());
      }
   }
```

Output:

Is ArrayList Empty: true

After Insertion

Is ArrayList Empty: false

How to reverse ArrayList in Java?

The reverse method of Collections class can be used to reverse any collection. It is a static method. Let's see the signature of reverse method:

```
public static void reverse(Collection c)
```

Let's see a simple example to reverse ArrayList in Java:

```
public class ReverseArrayList {
    public static void main(String[] args) {
        List<String> l = new ArrayList<String>();
        l.add("Mango");
        l.add("Banana");
        l.add("Mango");
        l.add("Apple");
        System.out.println("Before Reversing");
        System.out.println(l.toString());

        Collections.reverse(l);
        System.out.println("After Reversing");
        System.out.println(l);
    }
}
```

Output:

Before Reversing

[Mango, Banana, Mango, Apple]

After Reversing

[Apple, Mango, Banana, Mango]

How to remove duplicates from ArrayList in Java?

To remove dupliates from ArrayList, we can convert it into Set. Since Set doesn't contain duplicate elements, it will have only unique elements.

Let's see an example to remove duplicates from ArrayList:

```
public class RemoveDuplicateArrayList {
    public static void main(String[] args) {
        List<String> l = new ArrayList<String>();
        l.add("Mango");
        l.add("Banana");
        l.add("Mango");
        l.add("Apple");
        System.out.println(l.toString());
        Set<String> s = new LinkedHashSet<String>(l);
        System.out.println(s);
    }
}
```

Output:

Before converting to set

[Mango, Banana, Mango, Apple]

After converting to set

[Mango, Banana, Apple]

Java LinkedList class

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

The important points about Java LinkedList are:

- Java LinkedList class can contain duplicate elements.
- Java LinkedList class maintains insertion order.

LinkedList class declaration

public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>, Deque<E>, Cloneable, Serializable

Constructors of Java LinkedList

Constructor	Description
LinkedList()	It is used to construct an empty list.
LinkedList(Collection<? extends E> 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 of Java LinkedList

Method	Description
boolean add(E e)	It is used to append the specified element to the end of a list.
void add(int index, E element)	It is used to insert the specified element at the specified position index in a list.
boolean addAll(Collection<? extends E> 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.
boolean addAll(Collection<? extends E> 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.
boolean addAll(int index, Collection<? extends E> c)	It is used to append all the elements in the specified collection, starting at the specified position of the list.
void addFirst(E e)	It is used to insert the given element at the beginning of a list.
void addLast(E e)	It is used to append the given element to the end of a list.
void clear()	It is used to remove all the elements from a list.
Object clone()	It is used to return a shallow copy of an ArrayList.
boolean contains(Object o)	It is used to return true if a list contains a specified element.
Iterator<E> descendingIterator()	It is used to return an iterator over the elements in a deque in reverse sequential order.
E element()	It is used to retrieve the first element of a list.
E get(int index)	It is used to return the element at the specified position in a list.

E getFirst()	It is used to return the first element in a list.
E 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.
ListIterator<E> listIterator(int index)	It is used to return a list-iterator of the elements in proper sequence, starting at the specified position in the list.
boolean offer(E e)	It adds the specified element as the last element of a list.
boolean offerFirst(E e)	It inserts the specified element at the front of a list.
boolean offerLast(E e)	It inserts the specified element at the end of a list.
E peek()	It retrieves the first element of a list
E peekFirst()	It retrieves the first element of a list or returns null if a list is empty.
E peekLast()	It retrieves the last element of a list or returns null if a list is empty.
E poll()	It retrieves and removes the first element of a list.
E pollFirst()	It retrieves and removes the first element of a list, or returns null if a list is empty.
E pollLast()	It retrieves and removes the last element of a list, or returns null if a list is empty.
E pop()	It pops an element from the stack represented by a list.
void push(E e)	It pushes an element onto the stack represented by a list.
E remove()	It is used to retrieve and removes the first element of a list.
E remove(int index)	It is used to remove the element at the specified position in a list.
boolean remove(Object o)	It is used to remove the first occurrence of the specified element in a list.
E removeFirst()	It removes and returns the first element from a list.
boolean removeFirstOccurrence(Object o)	It is used to remove the first occurrence of the specified element in a list (when traversing the list from head to tail).
E removeLast()	It removes and returns the last element from a list.
boolean removeLastOccurrence(Object o)	It removes the last occurrence of the specified element in a list (when traversing the list from head to tail).
E set(int index, E element)	It replaces the element at the specified position in a list with the specified element.
Object[] toArray()	It is used to return an array containing all the elements in a list in proper sequence (from first to the last element).
<T> T[] toArray(T[] a)	It returns an array containing all the elements in the proper sequence (from first to the last element); the runtime type of the returned array is that of the specified array.
int size()	It is used to return the number of elements in a list.

Java LinkedList Example

```
import java.util.*;
public class LinkedList1{
 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

Java LinkedList example to add elements

Here, we see different ways to add elements.

```
import java.util.*;
public class LinkedList2{
 public static void main(String args[]){
 LinkedList<String> ll=new LinkedList<String>();
        System.out.println("Initial list of elements: "+ll);
        ll.add("Ravi");
        ll.add("Vijay");
        ll.add("Ajay");
        System.out.println("After invoking add(E e) method: "+ll);
        //Adding an element at the specific position
        ll.add(1, "Gaurav");
        System.out.println("After invoking add(int index, E element) method: "+ll);
        LinkedList<String> ll2=new LinkedList<String>();
        ll2.add("Sonoo");
        ll2.add("Hanumat");
        //Adding second list elements to the first list
        ll.addAll(ll2);
        System.out.println("After invoking addAll(Collection<? extends E> c) method: "+ll);
        LinkedList<String> ll3=new LinkedList<String>();
```

```
        ll3.add("John");
        ll3.add("Rahul");
        //Adding second list elements to the first list at specific position
        ll.addAll(1, ll3);
        System.out.println("After invoking addAll(int index, Collection<? extends E> c) method:
"+ll);
        //Adding an element at the first position
        ll.addFirst("Lokesh");
        System.out.println("After invoking addFirst(E e) method: "+ll);
        //Adding an element at the last position
        ll.addLast("Harsh");
        System.out.println("After invoking addLast(E e) method: "+ll);

 }
}
```

Output:

Initial list of elements: []

After invoking add(E e) method: [Ravi, Vijay, Ajay]

After invoking add(int index, E element) method: [Ravi, Gaurav, Vijay, Ajay]

After invoking addAll(Collection<? extends E> c) method:

[Ravi, Gaurav, Vijay, Ajay, Sonoo, Hanumat]

After invoking addAll(int index, Collection<? extends E> c) method:

[Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat]

After invoking addFirst(E e) method:

[Lokesh, Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat]

After invoking addLast(E e) method:

[Lokesh, Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat, Harsh]

Java LinkedList example to remove elements

Here, we see different ways to remove an element.

```
import java.util.*;
public class LinkedList3 {

    public static void main(String [] args)
    {
      LinkedList<String> ll=new LinkedList<String>();
      ll.add("Ravi");
      ll.add("Vijay");
      ll.add("Ajay");
      ll.add("Anuj");
      ll.add("Gaurav");
      ll.add("Harsh");
```

```
  ll.add("Virat");
  ll.add("Gaurav");
  ll.add("Harsh");
  ll.add("Amit");
  System.out.println("Initial list of elements: "+ll);
//Removing specific element from arraylist
    ll.remove("Vijay");
    System.out.println("After invoking remove(object) method: "+ll);
//Removing element on the basis of specific position
    ll.remove(0);
    System.out.println("After invoking remove(index) method: "+ll);
    LinkedList<String> ll2=new LinkedList<String>();
    ll2.add("Ravi");
    ll2.add("Hanumat");
// Adding new elements to arraylist
    ll.addAll(ll2);
    System.out.println("Updated list : "+ll);
//Removing all the new elements from arraylist
    ll.removeAll(ll2);
    System.out.println("After invoking removeAll() method: "+ll);
//Removing first element from the list
    ll.removeFirst();
    System.out.println("After invoking removeFirst() method: "+ll);
```

```
        //Removing first element from the list
            ll.removeLast();
            System.out.println("After invoking removeLast() method: "+ll);
        //Removing first occurrence of element from the list
            ll.removeFirstOccurrence("Gaurav");
            System.out.println("After invoking removeFirstOccurrence() method: "+ll);
        //Removing last occurrence of element from the list
            ll.removeLastOccurrence("Harsh");
            System.out.println("After invoking removeLastOccurrence() method: "+ll);

            //Removing all the elements available in the list
            ll.clear();
            System.out.println("After invoking clear() method: "+ll);
    }
}
```

Output:

Initial list of elements: [Ravi, Vijay, Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]

After invoking remove(object) method: [Ravi, Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]

After invoking remove(index) method: [Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]

Updated list : [Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit, Ravi, Hanumat]

After invoking removeAll() method: [Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]

After invoking removeFirst() method: [Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]

After invoking removeLast() method: [Gaurav, Harsh, Virat, Gaurav, Harsh]

After invoking removeFirstOccurrence() method: [Harsh, Virat, Gaurav, Harsh]

After invoking removeLastOccurrence() method: [Harsh, Virat, Gaurav]

After invoking clear() method: []

Java LinkedList Example to reverse a list of elements

```
import java.util.*;
public class LinkedList4{
 public static void main(String args[]){

  LinkedList<String> ll=new LinkedList<String>();
        ll.add("Ravi");
        ll.add("Vijay");
        ll.add("Ajay");
        //Traversing the list of elements in reverse order
        Iterator i=ll.descendingIterator();
        while(i.hasNext())
        {
           System.out.println(i.next());
        }
```

```
 }
}
```

Output: Ajay

Vijay

Ravi

Difference between ArrayList and LinkedList

ArrayList	LinkedList
1) ArrayList internally uses a **dynamic array** to store the elements.	LinkedList internally uses a **doubly linked list** to store the elements.
2) Manipulation with ArrayList is **slow** because it internally uses an 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 a doubly linked list, so no bit shifting is required in memory.
3) An 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 List

List in Java provides the facility to maintain the ordered collection. It contains the index-based methods to insert, update, delete and search the elements. It can have the duplicate elements also. We can also store the null elements in the list.

The List interface is found in the java.util package and inherits the Collection interface. It is a factory of ListIterator interface. Through the ListIterator, we can iterate the list in forward and backward directions. The implementation classes of List interface are ArrayList, LinkedList, Stack and Vector. The ArrayList and LinkedList are widely used in Java programming.

List Interface declaration

public interface List<E> extends Collection<E>

Java List Methods

Method	Description
void add(int index, E element)	It is used to insert the specified element at the specified position in a list.
boolean add(E e)	It is used to append the specified element at the end of a list.
boolean addAll(Collection<? extends E> c)	It is used to append all of the elements in the specified collection to the end of a list.
boolean addAll(int index, Collection<? extends E> c)	It is used to append all the elements in the specified collection, starting at the specified position of the list.
void clear()	It is used to remove all of the elements from this list.
boolean equals(Object o)	It is used to compare the specified object with the elements of a list.
int hashcode()	It is used to return the hash code value for a list.
E get(int index)	It is used to fetch the element from the particular position of the list.
boolean isEmpty()	It returns true if the list is empty, otherwise false.
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.
<T> T[] toArray(T[] a)	It is used to return an array containing all of the elements in this list in the correct order.
boolean contains(Object o)	It returns true if the list contains the specified element

boolean containsAll(Collection<?> c)	It returns true if the list contains all the specified element
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.
E remove(int index)	It is used to remove the element present at the specified position in the list.
boolean remove(Object o)	It is used to remove the first occurrence of the specified element.
boolean removeAll(Collection<?> c)	It is used to remove all the elements from the list.
void replaceAll(UnaryOperator<E> operator)	It is used to replace all the elements from the list with the specified element.
void retainAll(Collection<?> c)	It is used to retain all the elements in the list that are present in the specified collection.
E set(int index, E element)	It is used to replace the specified element in the list, present at the specified position.
void sort(Comparator<? super E> c)	It is used to sort the elements of the list on the basis of specified comparator.
Spliterator<E> spliterator()	It is used to create spliterator over the elements in a list.
List<E> subList(int fromIndex, int toIndex)	It is used to fetch all the elements lies within the given range.
int size()	It is used to return the number of elements present in the list.

How to create List

The ArrayList and LinkedList classes provide the implementation of List interface. Let's see the examples to create the List:

```
//Creating a List of type String using ArrayList

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


//Creating a List of type Integer using ArrayList
```

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

//Creating a List of type Book using ArrayList
List<Book> list=new ArrayList<Book>();

//Creating a List of type String using LinkedList
List<String> list=new LinkedList<String>();
```

Java List Example

Let's see a simple example of List where we are using the ArrayList class as the implementation.

```
import java.util.*;
public class ListExample1{
public static void main(String args[]){
 //Creating a List
 List<String> list=new ArrayList<String>();
 //Adding elements in the List
 list.add("Mango");
 list.add("Apple");
 list.add("Banana");
 list.add("Grapes");
 //Iterating the List element using for-each loop
 for(String fruit:list)
```

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


}

}
```

Output:

Mango

Apple

Banana

Grapes

How to convert Array to List

We can convert the Array to List by traversing the array and adding the element in list one by one using list.add() method. Let's see a simple example to convert array elements into List.

```
import java.util.*;

public class ArrayToListExample{

public static void main(String args[]){

//Creating Array

String[] array={"Java","Python","PHP","C++"};

System.out.println("Printing Array: "+Arrays.toString(array));

//Converting Array to List
```

```
List<String> list=new ArrayList<String>();
for(String lang:array){
list.add(lang);
}
System.out.println("Printing List: "+list);

}
}
```

Output:

Printing Array: [Java, Python, PHP, C++]

Printing List: [Java, Python, PHP, C++]

How to convert List to Array

We can convert the List to Array by calling the list.toArray() method. Let's see a simple example to convert list elements into array.

```
import java.util.*;
public class ListToArrayExample{
public static void main(String args[]){
 List<String> fruitList = new ArrayList<>();
 fruitList.add("Mango");
 fruitList.add("Banana");
```

```
 fruitList.add("Apple");
 fruitList.add("Strawberry");
 //Converting ArrayList to Array
 String[] array = fruitList.toArray(new String[fruitList.size()]);
 System.out.println("Printing Array: "+Arrays.toString(array));
 System.out.println("Printing List: "+fruitList);
}
}
```

Output:

Printing Array: [Mango, Banana, Apple, Strawberry]

Printing List: [Mango, Banana, Apple, Strawberry]

Get and Set Element in List

The get() method returns the element at the given index, whereas the set() method changes or replaces the element.

```
import java.util.*;
public class ListExample2{
 public static void main(String args[]){
 //Creating a List
 List<String> list=new ArrayList<String>();
```

```
 //Adding elements in the List
 list.add("Mango");
 list.add("Apple");
 list.add("Banana");
 list.add("Grapes");
 //accessing the element
 System.out.println("Returning element: "+list.get(1));//it will return the 2nd element, because index starts from 0
 //changing the element
 list.set(1,"Dates");
 //Iterating the List element using for-each loop
 for(String fruit:list)
  System.out.println(fruit);

 }
}
```

Output:

```
Returning element: Apple
Mango
Dates
Banana
Grapes
```

How to Sort List

There are various ways to sort the List, here we are going to use Collections.sort() method to sort the list element. The java.util package provides a utility class Collections which has the static method sort(). Using the Collections.sort() method, we can easily sort any List.

```
import java.util.*;
class SortArrayList{
 public static void main(String args[]){
  //Creating a list of fruits
  List<String> list1=new ArrayList<String>();
  list1.add("Mango");
  list1.add("Apple");
  list1.add("Banana");
  list1.add("Grapes");
  //Sorting the list
  Collections.sort(list1);
   //Traversing list through the for-each loop
  for(String fruit:list1)
    System.out.println(fruit);

 System.out.println("Sorting numbers...");
  //Creating a list of numbers
  List<Integer> list2=new ArrayList<Integer>();
```

```
  list2.add(21);
  list2.add(11);
  list2.add(51);
  list2.add(1);
  //Sorting the list
  Collections.sort(list2);
   //Traversing list through the for-each loop
  for(Integer number:list2)
    System.out.println(number);
 }
}
```

Output:

Apple

Banana

Grapes

Mango

Sorting numbers...

1

11

21

51

Java Comparable interface

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

compareTo(Object obj) method

public int compareTo(Object obj): It is used to compare the current object with the specified object.

It returns

positive integer, if the current object is greater than the specified object.

negative integer, if the current object is less than the specified object.

zero, if the current object is equal to 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. However, 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): It is used to sort the elements of List. List elements must be of the Comparable type.

Java Comparable Example

Let's see the example of the Comparable interface that sorts the list elements on the basis of age.

File: Student.java

```
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;
}
}
```

File: TestSort1.java

```
import java.util.*;
public class TestSort1{
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);
}
}
}
```

Output:

105 Jai 21

101 Vijay 23

106 Ajay 27

Java Comparable Example: reverse order

Let's see the same example of the Comparable interface that sorts the list elements on the basis of age in reverse order.

File: Student.java

```
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;
 }
 }
```

File: TestSort2.java

```
import java.util.*;
public class TestSort2{
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);
}
}
```

```
}
```

Output:

106 Ajay 27

101 Vijay 23

105 Jai 21

Java Comparator interface

Java Comparator interface is used to order the objects of a 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 sequences, i.e., you can sort the elements on the basis of any data member, for example, rollno, name, age or anything else.

Methods of Java Comparator Interface

Method	Description
public int compare(Object obj1, Object obj2)	It compares the first object with the second object.
public boolean equals(Object obj)	It is used to compare the current object with the specified object.
public boolean equals(Object obj)	It is used to compare the current object with the specified object.

Collections class

Collections class provides static methods for sorting the elements of a collection. If collection elements are of Set or Map, we can use TreeSet or TreeMap. However, 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.

Comparator Example

Let's see the example of sorting the elements of List on the basis of age and name.

File: 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;
   }

   public int getRollno() {
       return rollno;
   }
```

```
public void setRollno(int rollno) {
    this.rollno = rollno;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}

}
```

File: TestSort1.java

```
import java.util.*;
public class TestSort1{
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));
/Sorting elements on the basis of name
Comparator<Student> cm1=Comparator.comparing(Student::getName);
 Collections.sort(al,cm1);
 System.out.println("Sorting by Name");
 for(Student st: al){
  System.out.println(st.rollno+" "+st.name+" "+st.age);
  }
 //Sorting elements on the basis of age
  Comparator<Student> cm2=Comparator.comparing(Student::getAge);
Collections.sort(al,cm2);
 System.out.println("Sorting by Age");
 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

Difference between Comparable and Comparator

Comparable	Comparator
1) Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price.	The Comparator provides **multiple sorting sequences**. 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., the actual class is modified.	Comparator **doesn't affect the original class**, i.e., the actual class is not modified.
3) Comparable provides **compareTo() method** to sort elements.	Comparator provides **compare() method** to sort elements.
4) Comparable is present in **java.lang** package.	A Comparator is present in the **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.

Generics in Java

The Java Generics programming is introduced in J2SE 5 to deal with type-safe objects. It makes the code stable by detecting the bugs at compile time.

Before generics, we can store any type of objects in the collection, i.e., non-generic. Now generics force the java programmer to store a 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.

Without Generics, we can store any type of objects.

```
List list = new ArrayList();

list.add(10);

list.add("10");
```

With Generics, it is required to specify the type of object we need to store.

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

list.add(10);

list.add("10");// compile-time error
```

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());
}
}
}
```

```
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
```

Generic class

A class that can refer to any type is known as a generic class. Here, we are using the T type parameter to create the generic class of specific type.

Let's see a simple example to create and use the generic class.

Creating a 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, and Employee). The type you specify for the class will be used to store and retrieve the data.

Using generic class:

Let's see the code to use the 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 common type parameters are as follows:

T - Type

E - Element

K - Key

N - Number

V – Value

Generic Method

Like the generic class, we can create a generic method that can accept any type of arguments. Here, the scope of arguments is limited to the method where it is declared. It allows static as well as non-static methods.

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 = { 'J', 'A', 'V', 'A', 'T','P','O','I','N','T' };

        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

J

A

V

A

T

P

O

I

N

T