# **Collections in Java**

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 Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

### **Hierarchy of Collection Framework**

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



Java ArrayList

The ArrayList class is a resizable [array](https://www.w3schools.com/java/java_arrays.asp), which can be found in the java.util package.

Create an ArrayList object called **cars** that will store strings:

import java.util.ArrayList; // import the ArrayList class

ArrayList<String> cars = new ArrayList<String>(); // Create an ArrayList object

## Add Items

The ArrayList class has many useful methods. For example, to add elements to the ArrayList, use the add() method:

### **Example**

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

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

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

System.out.println(cars);

}

}

To access an element in the ArrayList, use the get() method and refer to the index number:

cars.get(0);

To modify an element, use the set() method and refer to the index number:

cars.set(0, "Opel");

To remove an element, use the remove() method and refer to the index number:

cars.remove(0);

To remove all the elements in the ArrayList, use the clear() method:

cars.clear();

cars.size();

Cars.isEmpty();

Cars.search();

## Loop Through an ArrayList

Loop through the elements of an ArrayList with a for loop, and use the size() method to specify how many times the loop should run:

loop through an ArrayList with the **for-each** loop:

### **Example**

public class Main {

public static void main(String[] args) {

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

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

for (String i : cars) {

System.out.println(i);

}

}

}

## Sort an ArrayList

Another useful class in the java.util package is the Collections class, which include the sort() method for sorting lists alphabetically or numerically:

### **Example**

Sort an ArrayList of Strings:

import java.util.ArrayList;

import java.util.Collections; // Import the Collections class

public class Main {

public static void main(String[] args) {

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

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

Collections.sort(cars); // Sort cars

for (String i : cars) {

System.out.println(i);

}

}

}

### **When To Use**

Use an ArrayList for storing and accessing data, and LinkedList to manipulate data.

LinkedList Methods

For many cases, the ArrayList is more efficient as it is common to need access to random items in the list, but the LinkedList provides several methods to do certain operations more efficiently:

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Try it** |
| addFirst() | Adds an item to the beginning of the list. | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_linkedlist_addfirst" \t "_blank) |
| addLast() | Add an item to the end of the list | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_linkedlist_addlast" \t "_blank) |
| removeFirst() | Remove an item from the beginning of the list. | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_linkedlist_removefirst" \t "_blank) |
| removeLast() | Remove an item from the end of the list | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_linkedlist_removelast" \t "_blank) |
| getFirst() | Get the item at the beginning of the list | [Try it »](https://www.w3schools.com/java/tryjava.asp?filename=demo_linkedlist_getfirst" \t "_blank) |
| getLast() | Get the item at the end of the list |  |

### **How the ArrayList works**

The ArrayList class has a regular array inside it. When an element is added, it is placed into the array. If the array is not big enough, a new, larger array is created to replace the old one and the old one is removed.

### **How the LinkedList works**

The LinkedList stores its items in "containers." The list has a link to the first container and each container has a link to the next container in the list. To add an element to the list, the element is placed into a new container and that container is linked to one of the other containers in the list.

Rest of all is same

// Import the LinkedList class

import java.util.LinkedList;

public class Main {

public static void main(String[] args) {

LinkedList<String> cars = new LinkedList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

System.out.println(cars);

}

}

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("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.

The peek() method returns an object from the top of the stack

Consider the following example.

**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:

1. Queue<String> q1 = **new** PriorityQueue();
2. 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("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();

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. 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:

1. 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.

addFirst(); removeFirst();

addLast(); removeLast();

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("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:

1. Set<data-type> s1 = **new** HashSet<data-type>();
2. Set<data-type> s2 = **new** LinkedHashSet<data-type>();
3. 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.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection7{
3. **public** **static** **void** main(String args[]){
4. //Creating HashSet and adding elements
5. HashSet<String> set=**new** HashSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //Traversing elements
11. Iterator<String> itr=set.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

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.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection8{
3. **public** **static** **void** main(String args[]){
4. LinkedHashSet<String> set=**new** LinkedHashSet<String>();
5. set.add("Ravi");
6. set.add("Vijay");
7. set.add("Ravi");
8. set.add("Ajay");
9. Iterator<String> itr=set.iterator();
10. **while**(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }

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:

1. 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:

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection9{
3. **public** **static** **void** main(String args[]){
4. //Creating and adding elements
5. TreeSet<String> set=**new** TreeSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //traversing elements
11. Iterator<String> itr=set.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

Output:

*Ajay*

*Ravi*

*Vijay*

## Java HashMap

In the [ArrayList](https://www.w3schools.com/java/java_arraylist.asp) chapter, you learned that Arrays store items as an ordered collection, and you have to access them with an index number (int type). A HashMap however, store items in "**key**/**value**" pairs, and you can access them by an index of another type (e.g. a String).

Create a HashMap object called **capitalCities** that will store String **keys** and String **values**:

import java.util.HashMap; // import the HashMap class

HashMap<String, String> capitalCities = new HashMap<String, String>();

## Add Items

The HashMap class has many useful methods. For example, to add items to it, use the put() method:

// Import the HashMap class

import java.util.HashMap;

public class Main {

public static void main(String[] args) {

// Create a HashMap object called capitalCities

HashMap<String, String> capitalCities = new HashMap<String, String>();

// Add keys and values (Country, City)

capitalCities.put("England", "London");

capitalCities.put("Germany", "Berlin");

capitalCities.put("Norway", "Oslo");

capitalCities.put("USA", "Washington DC");

System.out.println(capitalCities);

}

}

{USA=Washington DC, Norway=Oslo, England=London, Germany=Berlin}

To access a value in the HashMap, use the get() method and refer to its key:

capitalCities.get("England");

To remove an item, use the remove() method and refer to the key:

capitalCities.remove("England");

To remove all items, use the clear() method:

capitalCities.clear();

To find out how many items there are, use the size() method:

capitalCities.size();

Loop through the items of a HashMap with a **for-each** loop.

**Note: Use the keySet() method if you only want the keys, and use the values() method if you only want the values:**

### **Example**

// Print keys

for (String i : capitalCities.keySet()) {

System.out.println(i);

}

// Print values

for (String i : capitalCities.values()) {

System.out.println(i);

}

// Print keys and values

for (String i : capitalCities.keySet()) {

System.out.println("key: " + i + " value: " + capitalCities.get(i));

}

## Java HashSet

A HashSet is a collection of items where every item is unique, and it is found in the java.util package:

Create a HashSet object called **cars** that will store strings:

import java.util.HashSet; // Import the HashSet class

HashSet<String> cars = new HashSet<String>();

The HashSet class has many useful methods. For example, to add items to it, use the add() method:

To check whether an item exists in a HashSet, use the contains() method:

cars.contains("Mazda");

remove()

clear()

size()

## Loop Through a HashSet

Loop through the items of an HashSet with a **for-each** loop:

### **Example**

for (String i : cars) {

System.out.println(i);

}

## Java Iterator

An Iterator is an object that can be used to loop through collections, like ArrayList and HashSet. It is called an "iterator" because "iterating" is the technical term for looping.

To use an Iterator, you must import it from the java.util package.

## Getting an Iterator

The iterator() method can be used to get an Iterator for any collection:

// Import the ArrayList class and the Iterator class

import java.util.ArrayList;

import java.util.Iterator;

public class Main {

public static void main(String[] args) {

// Make a collection

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

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

// Get the iterator

Iterator<String> it = cars.iterator();

// Print the first item

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

}

}

## Looping Through a Collection

To loop through a collection, use the hasNext() and next() methods of the Iterator:

### **Example**

while(it.hasNext()) {

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

}

## Removing Items from a Collection

Iterators are designed to easily change the collections that they loop through. The remove() method can remove items from a collection while looping.

### **Example**

Use an iterator to remove numbers less than 10 from a collection:

import java.util.ArrayList;

import java.util.Iterator;

public class Main {

public static void main(String[] args) {

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

numbers.add(12);

numbers.add(8);

numbers.add(2);

numbers.add(23);

Iterator<Integer> it = numbers.iterator();

while(it.hasNext()) {

Integer i = it.next();

if(i < 10) {

it.remove();

}

}

System.out.println(numbers);

}

}

## Java Wrapper Classes

Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects.

Sometimes you must use wrapper classes, for example when working with Collection objects, such as ArrayList, where primitive types cannot be used (the list can only store objects):

ArrayList<int> myNumbers = new ArrayList<int>(); // Invalid

ArrayList<Integer> myNumbers = new ArrayList<Integer>(); // Valid

public class Main {

public static void main(String[] args) {

Integer myInt = 5;

Double myDouble = 5.99;

Character myChar = 'A';

System.out.println(myInt.intValue());

System.out.println(myDouble.doubleValue());

System.out.println(myChar.charValue());

}

}

1. What is java collection and its hirearchy?
2. Explain all collections and code it
3. Wrapper class and and Iterables