HowToDoInJava

# Collections in Java

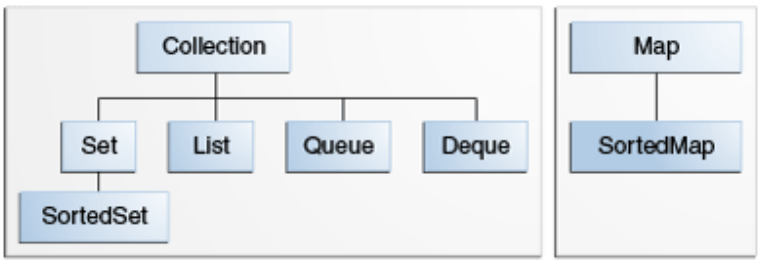
Last Updated: December 26, 2020

A collection, as name implies, is group of objects. **Java Collections framework** is consist of the interfaces and classes which helps in working with different types of collections such as **lists, sets, maps, stacks and queues** etc.

These ready-to-use collection classes solve lots of very common problems where we need to deal with group of homogeneous as well as heterogeneous objects. The common operations in involve **add, remove, update, sort, search** and more complex algorithms. These collection classes provide very transparent support for all such operations using **Collections APIs**.

## 1. Java Collections Hierarchy

The Collections framework is better understood with the help of **core interfaces**. The collections classes implement these interfaces and provide concrete functionalities.




Java Collections Hierarchy

### 1.1. Collection

**Collection interface** is at the root of the hierarchy. Collection interface provides all general purpose methods which all collections classes must support (or throw `UnsupportedOperationException`). It **extends Iterable** interface which adds support for iterating over collection elements using the "for-each loop" statement.

All other collection interfaces and classes (except Map) either extend or implement this interface. For example, List *(indexed, ordered)* and Set *(sorted)* interfaces implement this collection.

### 1.2. List

**Lists** represents an **ordered** collection of elements. Using lists, we can access elements by their integer index (position in the list), and search for elements in the list. index start with `0`, just like an array.

Some useful classes which implement `List` interface are – **ArrayList**, **CopyOnWriteArrayList**, **LinkedList**, **Stack** and **Vector**.

### 1.3. Set

**Sets** represents a collection of **sorted** elements. Sets do not allow the duplicate elements. Set interface does not provides no guarantee to return the elements in any predictable order; though some Set implementations store elements in their natural ordering and guarantee this order.

Some useful classes which implement `Set` interface are – **ConcurrentSkipListSet**, **CopyOnWriteArraySet**, **EnumSet**, **HashSet**, **LinkedHashSet** and **TreeSet**.

### 1.4. Map

The **Map** interface enable us to store data in *key-value pairs* (keys should be immutable). A map cannot contain duplicate keys; each key can map to at most one value.

The Map interface provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. Some map implementations, like the TreeMap class, make specific guarantees as to their order; others, like the HashMap class, do not.

Some useful classes which implement `Map` interface are – **ConcurrentHashMap**, **ConcurrentSkipListMap**, **EnumMap**, **HashMap**, **Hashtable**, **IdentityHashMap**,

**LinkedHashMap**, **Properties**, **TreeMap** and **WeakHashMap**.

### 1.5. Stack

The Java **Stack** interface represents a classical stack data structure, where elements can be pushed to last-in-first-out (LIFO) stack of objects. In Stack we push an element to the top of the stack, and popped off from the top of the stack again later.

### 1.6. Queue

A queue data structure is intended to hold the elements (put by producer threads) prior to processing by consumer thread(s). Besides basic Collection operations, queues provide additional insertion, extraction, and inspection operations.

Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner. One such exception is priority queue which order elements according to a supplied Comparator, or the elements' natural ordering.

In general, queues do not support blocking insertion or retrieval operations. Blocking queue implementations classes implement **BlockingQueue** interface.

Some useful classes which implement `Map` interface are – ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue and SynchronousQueue.

### 1.7. Deque

A double ended queue (pronounced "***deck***") that supports element insertion and removal at both ends. When a deque is used as a queue, FIFO (First-In-First-Out) behavior results. When a deque is used as a stack, LIFO (Last-In-First-Out) behavior results.

This interface should be used in preference to the legacy Stack class. When a deque is used as a stack, elements are pushed and popped from the beginning of the deque.

Some common known classes implementing this interface are ArrayDeque, ConcurrentLinkedDeque, LinkedBlockingDeque and LinkedList.

## 2. Java Collections and Generics

By purpose, generics provide type safety. It detects the incompatible types (in method arguments) and prevent **ClassCastException** in runtime. In Java collections as well, we can define a collection class to store only a certain type of objects. All other types should be disallowed. This is done via generics.

In given example, first two add() methods are allowed. Third one will not compile and will give error – "The method put(Integer, String) in the type **`HashMap<Integer,String>`** is not applicable for the arguments (String, String)". It helps in detecting incompatible types early to prevent unpredictable behavior in runtime.

```
Generic HashMap

HashMap<Integer, String> map = new HashMap<>();

map.put(1, "A");  //allowed
map.put(2, "B");  //allowed

map.put("3", "C");  //NOT allowed - Key is string
```

## 3. equals() and hashCode() Methods

Many collection classes provide specific functionalities such as sorted elements, no duplicate elements etc. To implement this behavior, the added elements (objects) must implement the equals() and hashCode() methods correctly.

All Java wrapper classes and String class override these functions with their specific implementation so they behave correctly in such collections. we need to make sure that these functions are overridden correctly in our user defined custom classes as well.

```
SortedSet HashMap
```

```
SortedSet<Integer> sortedSet = new TreeSet<>();

sortedSet.add(2);

sortedSet.add(1);
sortedSet.add(1);

sortedSet.add(3);

System.out.println(sortedSet);  //[1,2,3]
```

## 4. Java 8 Collections

Java 8 was a major release which introduced lambda style of programming in Java. Collections classes was also improved as a result. For example, we can iterate over collections in single line and perform an action on all elements of collection using **forEach** statement.

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

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

list.forEach(System.out::print);
```

## 5. Benefits of Java Collections

- **Consistent and reusable APIs** – This is any framework does. It provides a consistent set of classes methods which can be used to solve a similar set of problems over and over, without getting unpredictable results. Java collections framework also helps in solving common problems related to a group of objects – in consistent manner.
  All collection classes have consistent implementation and provide some common methods like add, get, put, remove etc. No matter what kind of data structure you are dealing with, these methods work according to underlying implementation and perform actions transparently.

- **Less development time** – A common and predictable framework always decreases the development time and helps in writing application program in speedy manner. Java collection also helps in performing some most repeated common tasks with objects and collections and thus improve time factor.
- **Performance** – The Java collection APIs are written by some most brilliant minds of industry and their performance is top notch in most of the scenarios. Ongoing development work by Oracle and very enthusiastic Java developer community helps in making it better.
- **Clean code** – These APIs have been written with all good coding practices and documented very well. They follow a certain standard across whole Java collection framework. It makes the programmer code look good and clean. The code is easier to read as well because of consistent class and method names.

## 6. Java collection examples

- Array
- ArrayList
- LinkedList
- HashMap
- Hashtable
- LinkedHashMap
- TreeMap
- HashSet
- LinkedHashSet
- TreeSet
- Comparable
- Comparator
- Iterator

- ListIterator
- Spliterator
- PriorityQueue
- PriorityBlockingQueue
- ArrayBlockingQueue
- LinkedTransferQueue
- CopyOnWriteArrayList
- CopyOnWriteArraySet
- Collection Sorting
- Interview Questions

Read More:

Wikepedia Link
Java Docs

**Email Address**

Subscribe

** We do not spam !!*

Search ...

## HowToDoInJava

A blog about Java and related technologies, the best practices, algorithms, and interview questions.

**Meta Links**

- About Me
- Contact Us
- Privacy policy
- Advertise
- Guest Posts

## Blogs

REST API Tutorial