**Collections**

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

A **collection** is an object that groups multiple elements into a single unit, like a **List**, **Set**, **Map**.

|  |  |  |
| --- | --- | --- |
| **Interface** | **Description** | **Example Implementations** |
| List | Ordered, allows duplicates | ArrayList, LinkedList |
| Set | Unordered, no duplicates | HashSet, TreeSet, LinkedHashSet |
| Queue | FIFO data structure | LinkedList, PriorityQueue |
| Deque | Double-ended queue | ArrayDeque |
| Map | Key-value pairs | HashMap, TreeMap, LinkedHashMap |

**Advantages of the Java Collection Framework**

Reusability**,** Quality, Performance, Reduces Programming Effort, Thread-Safe Utilities etc

## **Iterators in Java**

**Iterators** are used to retrieve the elements one by one from a collection object.

## **Types of Iterators in Java**

There are four types of iterators or cursors available in Java. They are as follows:

* Enumeration
* Iterator
* ListIterator
* Spilterator

## **Enumeration:** Enumeration is read-only. You cannot add/remove elements. It is less powerful than Iterator or ListIterator. It traverses only in the forward direction. It is a legacy.

# **Iterator (Cursor or Universal**

# The Iterator in Java is an interface used to iterate over collection elements. It provides a universal and safe way to access elements one at a time.

## **Limitations of Iterator in Java**

* By using Enumeration and Iterator, we can move only in the forward direction. We cannot move in the backward direction. Hence, these are called single-direction cursors.
* We can perform either a read operation or a remove operation.
* We cannot perform the replacement of new objects.
* For example, suppose there are five mangoes in a box. Out of five, two mangoes are not good, but we cannot replace those damaged mangoes with new mangoes.

**ListIterator (Bi-Directional)**

In simple words, it is an object by which we can iterate the elements of a list in both forward and backward directions.

By using ListIterator, we can perform different kinds of operations such as read, remove, replacement (current object), and the addition of new elements.

**Spliterator:** It is **similar to Iterator**, but can **split** itself into multiple parts to support **parallelism**. Works with **Streams**, **Collections**, and **arrays**.

|  |  |
| --- | --- |
| **Method** | **Description** |
| tryAdvance(Consumer action) | Performs an action on the next element if available |
| trySplit() | Splits the iterator and returns a new Spliterator for parallel traversal |
| estimateSize() | Returns an estimate of remaining elements |
| characteristics() | Returns properties (e.g., ORDERED, SORTED, DISTINCT etc.) as bit flags |

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature / Property** | **Iterator** | **ListIterator** | **Spliterator** |
| **Traversal Direction** | **Forward only** | **Forward and backward** | **Forward only**, but can **split** |
| **Modify list** | Remove | Add, Remove, Set, Replace | No |
| **Access by Index** | No | Yes (nextIndex(), previousIndex()) | No |
| **Works On** | All Collection types | Only List implementations | All sequential & parallel data sources |
| **Iterator Type** | **Cursor** | **Bidirectional cursor** | **Split iterator** |
| **Thread Safety** | No (depends on collection) | No | No (depends on source) |
| **Use Case** | forward iteration, safe removal | Modify list during iteration | Efficient parallelism and stream traversal |

## **List in Java**

➲ A list in Java is a collection for storing elements in sequential order.

➲ It is used to store a collection of elements where duplicate elements are allowed.

## **Features of List Interface in Java**

1. The list allows storing duplicate elements in Java.

2. In the list, we can add an element at any position.

3. It maintains insertion order.

4. It allows for storing many null elements.

5. Java list uses a resizable array for its implementation. Resizable means we can increase or decrease the size of the array.

6. Except for LinkedList, ArrayList, and Vector is an indexed-based structure.

7. It provides a special Iterator called a ListIterator that allows accessing the elements in the forward direction using hasNext() and next() methods.

**7. int indexOf(Object o):** It is used to return the index of a particular element of the first occurrence in the list. If the element is not present in the list then it will return -1.

## **When to use List?**

1. List can be used when we want to allow or store duplicate elements.

2. It can be used when we want to store null elements.

3. When we want to preserve our insertion order, we should go for a list.

**Serializable Interface:** The **Serializable** interface in Java is used to **enable object serialization**

**ArrayList:** ArrayList in Java is a **resizable array-based** implementation of the List. It stores elements in a dynamic array, automatically growing/shrinking as needed.

**Internal working:**

**1. Creation:** ArrayList<String> list = new ArrayList<>();

Internally create a array with the capacity 10 and size 0.

2. **Adding Elements (add()):** if size>= capacity. Capacity become 1.5 times.

3. **Removing Elements (remove(index)):** No shrinking of array unless done manually (e.g., trimToSize())

## **Features of ArrayList**

**1. Resizable-array:** ArrayList is a resizable array

**2. Index-based structure:** It uses an index-based structure in java.

**3. Duplicate elements:** Duplicate elements are allowed in the array list.

**4. Null elements:** Any number of null elements can be added to ArrayList.

**5. Insertion order:** It maintains the insertion order. i.e.insertion order is preserved.

**6. Heterogeneous objects:** Heterogeneous objects are allowed everywhere except TreeSet and TreeMap.

**7. Synchronized:** ArrayList is not synchronized. It is not thread safe.

**8. Random Access:** ArrayList implements random access because it uses an index-based structure. Therefore, we can get, set, insert, and remove elements of the array list from any arbitrary position.

**9. Performance:** In ArrayList, manipulation is slow because if any element is removed from the ArrayList, a lot of shifting takes place.

10. we can not add or remove an element in the ArrayList during Iteration, else, it will throw a **ConcurrentModificationException**

ArrayList with a default initial capacity of 10.In this array list, we can store only 10 elements.

**New capacity** = (current capacity\*3/2) + 1 = 10\*3/2 + 1 = 16

### **How do we manually increase or decrease the current capacity of an ArrayList?**

**1. ensureCapacity():** This method is used to increase the current capacity of the ArrayList Manually

**2. trimTosize():** The trimTosize() method is used to trim the capacity of the ArrayList to the current size of the ArrayList.

# **How to Synchronize an ArrayList in Java**

**1. using the Collections.synchronizedList() method:** Iteration of the list must be manually synchronized.

**2. using CopyOnWriteArrayList class:-** It is recommended for Read-Heavy Scenarios

1**. It is thread-safe for concurrent access to the ArrayList. When an ArrayList is modified, it will create a fresh copy of the underlying array.**

2. CopyOnWriteArrayList does not lock the whole list. When a thread writes into the list, It simply replaces the list with a fresh copy of the underlying array.

In this way, it provides concurrent access to the ArrayList for multiple threads without locking. Since the read operation is thread-safe, two threads cannot write into the list simultaneously.

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Thread-Safe** | **Suitable For** | **Notes** |
| Collections.synchronizedList | ✅ Yes | Balanced read-write | Manual synchronization needed during iteration |
| CopyOnWriteArrayList | ✅ Yes | Read-heavy, few writes | Safe for iteration, writes are expensive |
| synchronized block | ✅ Yes | Custom locking scenarios | Tedious and error-prone |

**Limitations of ArrayList:**

|  |  |
| --- | --- |
| **Limitation** | **Description** |
| **Not thread-safe** | Needs external synchronization |
| **Slow insertion/deletion** | Especially in the middle (O(n)) |
| **Resizing overhead** | Array copying when full |
| **Allows duplicates** | No uniqueness enforced |
| **Object-only storage** | Primitives are boxed |
| **Not optimized for searching** | Uses linear search |
| **Manual trimming required** | trimToSize() must be called to free unused memory |

**LinkedList:** Java LinkedList class uses a doubly linked list to store elements**.**

**Offer():** add the element in last

|  |  |
| --- | --- |
| **Feature** | **Description** |
| **Data Structure** | Doubly Linked List |
| **Implements** | List, Deque, Queue |
| **Allows Duplicates** | ✅ Yes |
| **Allows Null Elements** | ✅ Yes |
| **Insertion/Deletion** | ✅ Fast at both ends |
| **Random Access** | ❌ Slow (O(n)) |
| **Thread-safe** | ❌ No (not synchronized) |

**Internal Structure of LinkedList:**

LinkedList uses a **doubly-linked list**, where each node stores:

* The **data** (element)
* A **reference to the next node**
* A **reference to the previous node**

**Adding Elements: add(E e) → Adds to end**

* Creates a new node.
* Links last to the new node.
* Updates the last reference.

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **poll()** | **peek()** | **remove()** |
| **Belongs To** | Queue interface | Queue interface | Queue interface |
| **Purpose** | Retrieves and removes head | Retrieves head without removing | Retrieves and removes head |
| **Returns** | Head element or null if empty | Head element or null if empty | Head element or throws NoSuchElementException if empty |
| **Removes Element** | ✅ Yes | ❌ No | ✅ Yes |

**HashSet:** HashSet is a class in Java used to store a collection of unique elements.

|  |  |
| --- | --- |
| **Feature** | **Description** |
| **Duplicates Allowed** | ❌ No (stores only unique elements) |
| **Maintains Order?** | ❌ No (unordered) |
| **Allows null?** | ✅ Yes (only one null element) |
| **Thread-safe?** | ❌ No (use Collections.synchronizedSet() if needed) |
| **Underlying Structure** | **HashMap** (each element is stored as key) |

**How HashSet Works Internally**

* Internally uses a **HashMap<E, Object>** where each value is a constant dummy object.
* When you add an element, it becomes a **key** in the map.
* Hashing is done using the element’s hashCode() and equals() methods.

**LinkedHashSet:** LinkedHashSet is a class that combines the features of both:

* **HashSet →** for storing unique elements
* **Linked list →** to maintain insertion order

**How It Works Internally:**

* Inherits from HashSet, and uses a linked list to maintain insertion order.
* Internally uses a **LinkedHashMap** where:
  + Keys are the set elements.
  + Values are a constant dummy object.
* Every element remembers its insertion position.

**TreeSet:** It is used to store unique elements in sorted (ascending) order. Custom class must implement the Comparable interface. We can provide our custom Comparator in the treeset constructor for sorting.

|  |  |
| --- | --- |
| **Feature** | **Description** |
| **Duplicates allowed?** | ❌ No (stores only unique elements) |
| **Ordering** | ✅ Yes (natural or custom comparator) |
| **Null allowed?** | ❌ No (throws NullPointerException if null added) |
| **Thread-safe?** | ❌ No (must synchronize externally if needed) |
| **Backed by** | Red-Black Tree |

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **HashSet** | **LinkedHashSet** | **TreeSet** |
| **Order** | ❌ Unordered | ✅ Insertion order | ✅ Sorted order |
| **Null allowed** | ✅ One | ✅ One | ❌ Not allowed |
| **Performance** | ✅ Fastest | ✅ Fast | ❌ Slower (O log n) |
| **Backed by** | HashMap | LinkedHashMap | TreeMap (Red-Black Tree) |

**Map:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Implementation** | **Ordered?** | **Sorted?** | **Null Keys/Values** | **Thread-Safe** |
| **HashMap** | No order | No | 1 null key, many null values | No |
| **LinkedHashMap** | Insertion order | No | Yes | No |
| **TreeMap** | Sorted by keys | Yes | No null key | No |
| **Hashtable** | No | No | No null allowed | Yes |
| **ConcurrentHashMap** | No | No | No nulls | Yes (High performance) |

**Internal Working**

* TreeSet internally uses a TreeMap, where:
  + Each element is stored as a key.
  + Value is a dummy object (PRESENT).
* Elements are sorted using:
  + Their natural ordering (**Comparable**)
  + Or a **custom comparator** passed to the constructor

# **Internal Working of HashMap**

**HashMap** is an array of buckets where each bucket uses a linked list to hold elements. A linked list is a list of nodes where each node contains a key-value pair.

**HashMap works on the principle of a hashing data structure that uses an object’s hash code to place that object inside the map.**

**Hashing**: Hashing is the process of converting a **key** into an **integer (hash code)**, which determines **where** the value will be stored in a data structure.

**Initial Capacity** is the size of the bucket array internally by HashMap at the time of the creation of the HashMap.

The default initial capacity of HashMap is 16.

**Load Factor** is a factor that is internally used by HashMap to determine when the size of a Bucket array requires to be increased. By default, it is 0.75.

**Bucket**: A **bucket** is a container (array index) where all **entries with the same hash index** are stored.

**Bucket Table:** The **bucket table** is the **underlying array** that holds **buckets**, where each bucket stores **entries (key-value pairs)** that map to **same hash index**.

**Node:** stores a **key-value pair**, along with its **hash** and a reference to the **next node** in case of **hash collisions**.

**hashCode():** The hashCode() method returns an **integer hash code** value for an object, used to determine its **bucket index** in hash-based data structures.

If the key is null then the hash value returned by the hashCode() will be 0.

**equals():**

HashMap uses equals() method to compare Keys whether they are equal or not.

The equals() method of Object class can be overridden. If we override the equals() method, it is mandatory to override the hashCode() method.

## **Put Operation:**

When we call the put() method to add an entry to the HashMap, the HashMap calculates the hash code of the key by calling its hashCode() method.

HashMap uses that code to calculate the bucket index in which key/value pair will be placed.

Index = hash(key.hashCode()) & (n-1); where n is the size of the array.

* **Check if a node already exists** at that index (bucket):
  + If **no node**, insert a new node.
  + If **a node already exists**, check:
    - If the **same key** → **overwrite** value.
    - If **different key with same hash** → handle **collision** by:
      * Adding to the **end of the linked list** (or tree node in Java 8+ if threshold crossed).

## **get() method**

When the get(K Key) method takes a Key, it calculates the index of the bucket. Then that bucket’s List is searched for the given key using equals() method and the final result is returned.

**Collision Handling**

When multiple keys generate the **same index**, it’s a **hash collision**.

**Java handles collisions by:**

* **Chaining**: Store entries in a **linked list** at the same bucket.
* **Treeing (Java 8+)**: If one bucket gets **> 8 entries** and map size ≥ 64, switch to a **Red-Black Tree** for faster lookup.

## **Concept of Rehashing**

**Rehashing** is a process that occurs automatically by HashMap when the number of keys in the map reaches the threshold value. The threshold value is calculated as **threshold** = capacity \* (load factor of 0.75).

In this case, a new size of bucket array is created with its double capacity and all the existing contents are copied over to it.

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **HashMap** | **LinkedHashMap** | **TreeMap** |
| **Ordering** | No order | Maintains **insertion** order | Maintains **sorted** order (by key) |
| **Underlying Data Structure** | Hash Table | Hash Table + Doubly Linked List | Red-Black Tree (Self-balancing BST) |
| **Null Keys** | One null key allowed | One null key allowed | No null keys allowed |
| **Null Values** | Allowed | Allowed | Allowed |
| **Key Uniqueness** | Required | Required | Required |
| **Thread-Safe?** | No | No | No |
| **Use Case** | Fast access, unordered data | Fast access, ordered by insertion | Sorted data (ascending by key) |

**Key points:**

1. [HashTable](https://www.scientecheasy.com/2020/12/hashtable-in-java.html/) is suitable when you are not working in a multithreading environment.

2. HashMap is slightly better than HashTable but it is not thread-safe. It is suitable if the order of elements is not an issue.

3. TreeMap is slower than HashMap but it is suitable when you need the map in sorted ascending order.

4. LinkedHashMap is also slower than HashMap, but it is preferred if more insertions and deletions happen on the map.

|  |  |  |
| --- | --- | --- |
| **Feature** | **HashMap** | **Hashtable** |
| **Thread-Safe** | No (not synchronized) | Yes (synchronized) |
| **Performance** | Faster (no locking) | Slower (due to synchronization) |
| **Null Keys Allowed?** | One null key | Not allowed (throws NullPointerException) |
| **Null Values Allowed?** | Multiple null values | Not allowed |
| **Fail-Fast Iterator?** | ✅Yes | No (uses Enumerator, not Iterator) |
| **Extends** | AbstractMap | Dictionary (an obsolete abstract class) |

# **WeakHashMap in Java**

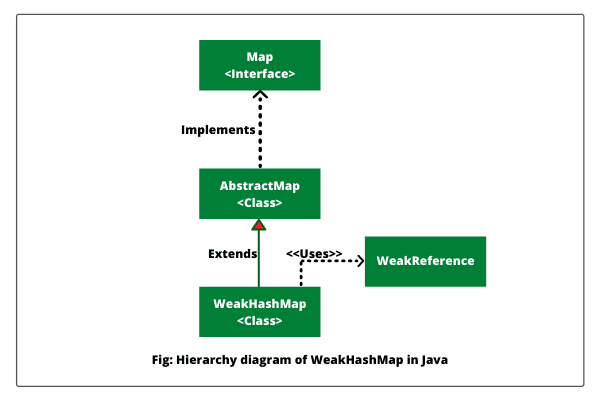
The key of WeakHashMap has a weak reference that allows an entry in a map to be garbage-collected when its key is no longer in use.

an entry in a WeakHashMap will be automatically removed from the map by the garbage-collector when its key is unused.

**Garbage-collector clears all weak references to the keys(inside and outside of the map) when keys are no longer in ordinary use.**

## **Hierarchy of WeakHashMap class in Java**

WeakHashMap class extends AbstractMap class that implements Map interface. The hierarchy diagram of the WeakHashMap class is shown in the below figure.

****

## **Features of WeakHashMap**

1. Garbage collectors may remove keys and their associated values from the WeakHashMap at any time in java.

2. It enables us to store data in the form of key-value pairs in java.

3. Java WeakHashMap does not allow storing duplicate keys.

4. It allows inserting only one null key in the map but several null values can be added.

5. WeakHashMap does not maintain insertion order in java.

6. WeakHashMap in Java is not synchronized. So, it is not thread-safe. Two or more threads on the same WeakHashMap object can access at the same time in java.

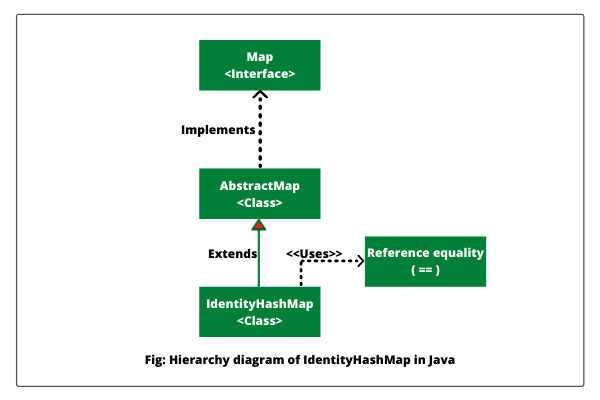
We can synchronize weakHashMap by using Collections’ class synchronizedMap() method.

# **IdentityHashMap in Java**

**IdentityHashMap in Java** is a concrete class that is similar to [HashMap](https://www.scientecheasy.com/2020/11/hashmap-in-java.html/) except that it uses reference equality (==) when comparing entries (keys and values).

In simple words, Java IdentityHashMap uses reference equality (==) instead of object equality (equals() method) when comparing keys and values.

While comparing keys, HashMap uses the equals() method whereas IdentityHashMap uses reference equality (==).

.

## **Features of IdentityHashMap**

1. IdentityHashMap allows us to store data in the form of key-value pairs.

2. In IdentityHashMap, only one null key is allowed to be added but many null values can be inserted.

3. Like HashMap, IdentityHashMap does not maintain insertion order. There is no guarantee that elements will be retrieved in the same order as we have inserted.

4. It is not synchronized. That means it is not thread-safe. So, two or more threads on the same IdentityHashMap object can access it at the same time.

## **When to use IdentityHashMap in Java?**

* We want to perform deep-copying.
* We want to maintain proxy objects.
* We want to store only one null key on the map.
* We do not want the insertion order of entries.
* We are not working in a multithreading environment.

# **EnumMap in Java**

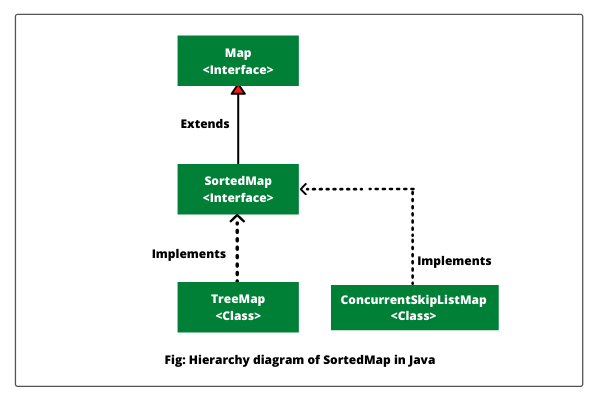
Enum maps are represented internally as an array to hold the corresponding values that are extremely compact and efficient.

# **SortedMap in Java**

**SortedMap in Java** is an interface that is a subinterface of [Map interface](https://www.scientecheasy.com/2020/10/map-in-java.html/). The entries in the map are maintained in sorted ascending order based on their keys.

In simple words, it guarantees that the entries are maintained in ascending key order. It allows very efficient manipulations of subsets of a map.

SortedMap interface extends Map interface. [TreeMap](https://www.scientecheasy.com/2020/11/treemap-in-java.html/) and ConcurrentSkipListMap classes implement the SortedMap interface.



**6. SortedMap<K,V> headMap(K toKey):** This method returns a portion of the map whose keys are strictly less than toKey.

**7. SortedMap<K,V> tailMap(K fromKey):** This method returns a portion of the map whose keys are greater than or equal to fromKey.

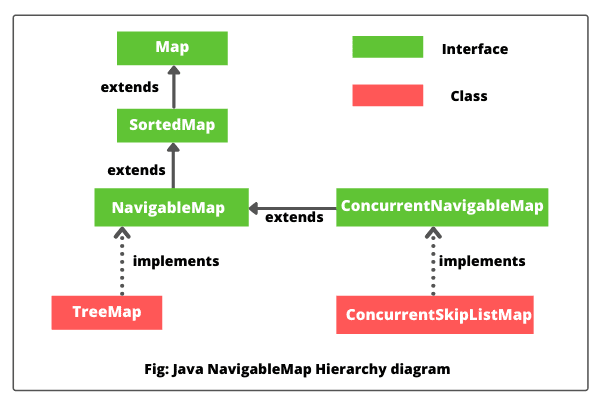
# 

# **NavigableMap in Java**

**NavigableMap in Java** is an interface that is a subinterface of [SortedMap interface](https://www.scientecheasy.com/2020/12/sortedmap-in-java.html/).

It extends the SortedMap interface to handle the retrieval of entries based on the closest match to a given key or keys.

In simple words, we can navigate the Map easily with the NavigationMap interface. We can retrieve the nearest value matching with the specified key, all the values less than the specified key, all values greater than the given key, and so on.



**1. Map.Entry<K,V> ceilingEntry(K obj):**

**2. K ceilingKey(K obj):**

**3. NavigableSet<K> descendingKeySet():**

**6. Map.Entry<K,V> floorEntry(K obj):**

Concurrent hash map applies locks only at bucket level called fragment while adding or updating the map. So, a concurrent hash map allows concurrent read and write operation to the map.

Synchronized hashmap(Collection.syncronizedHashMap()) is a method of Collection framework. This method applies a lock on the entire collection. So, if one thread is accessing the map then no other thread can access the same map.

|  |  |  |  |
| --- | --- | --- | --- |
| **S** | **Key** | **Concurrent hash map** | **Synchronized hashmap** |
| **1** | **Implementation** | **It is a class that implements a Concurrent hash map and serializable interface.** | **It is a method in Collection class.** |
| **2** | **Lock mechanism** | **Locks the portion** | **Locks the whole map.** |
| **3** | **Performance** | **Concurrent hashmap allows concurrent read and write. So performance is relatively better than a synchronized map.** | **Multiple threads can't access the map concurrently. So, performance is relatively less than a concurrent hash map.** |
| **4** | **Null key** | **It doesn't allow null as a key or value.** | **It allows null as a key.** |
| **5** | **Concurrent modification exception** | **It doesn't throw concurrent modification exceptions.** | **Iterator return by synchronized map throws concurrent modification exception** |

[**JVM**](https://www.scientecheasy.com/2021/03/what-is-jvm.html/) **(Java Virtual Machine)** stores the reference of other objects into a collection object. It never stores physical copies of other objects because other objects are already available in the memory and storing another copy of objects into a collection object would be a waste of memory.

## **Difference between Arrays & Collections in Java**

1. Arrays are fixed in size, but collections are growable. We can increase or decrease the size.

2. Arrays can store only homogeneous data elements (similar type of data), but collections can hold both homogeneous and heterogeneous elements.

3. Arrays do not support any methods, but collections support various kinds of methods.

4. Arrays can hold both primitive and object types, but collections can hold only objects but not primitive.

## **Advantages of the Collections Framework in Java**

1. The collections framework reduces the development time and the burden of designers, programmers, and users.
2. Your code is easier to maintain because it provides useful data structures and interfaces that reduce programming efforts.
3. The size of the container is growable in nature.
4. It implements high-performance, useful data structures and algorithms that increase the performance.
5. It enables software reuse.

## **Limitation of the Collections Framework in Java**

There are two limitations of the collections framework in Java. They are as follows:

1. Care must be taken to use the appropriate cast operation.
2. Compile-time checking is not possible.

**Q. Does a collection object store copies of other objects?**

**A:** No, a collection object works with reference types. It stores references of other objects, not copies of other objects.

**Q**. **Can we store a primitive data type into a collection?**

**A:** No, collections store only objects.

**Queue Interface**

➲ A queue is an ordered of the homogeneous group of elements in which new elements are added at one end(rear) and elements are removed from the other end(front). Just like a queue in a supermarket or any shop.

➲ This interface represents a special type of list whose elements are removed only from the head.

➲ LinkedList, Priority queue, ArrayQueue, Priority Blocking Queue, and Linked Blocking Queue are the concrete subclasses that implement the queue interface.

**Deque Interface**

➲ It is a linear collection of elements in which elements can be inserted and removed from either end. i.e, it supports insertion and removal at both ends of an object of a class that implements it.

➲ LinkedList and ArrayDeque classes implement the Deque interface.

# Generics in Java

The **Java Generics** programming is used to deal with type-safe objects.

## Advantage of Java Generics

**1) Type-safety:** We can hold only a single type of objects in generics. It doesn?it allows me to store other objects.

**2) Type casting is not required:** There is no need to typecast the object.

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

## Java Stream.flatMap() Method

In [Java](https://www.javatpoint.com/java-tutorial) 8 Streams, the flatMap() method applies operation as a mapper function and provides a stream of element values. It means that in each iteration of each element the map() method creates a separate new stream. **By using the flattening mechanism, it merges all streams into a single resultant stream**. **In short, it is used to convert a Stream of Stream into a list of values.**

### What is flattening?

**Flattening is the process of converting several lists of lists and merging all those lists to create a single list containing all the elements from all the lists.**