**Java Collections**

**─**

Contents

[**1. What are Java Collections?** 2](#_Toc183876407)

[# Java Collections Framework Architecture 2](#_Toc183876408)

[# Iterable Interface 4](#_Toc183876409)

[# Collection Interface 5](#_Toc183876410)

[**2. List Interface and its Implementations** 6](#_Toc183876411)

[# ArrayList 7](#_Toc183876412)

[# Stack 8](#_Toc183876413)

[**3. Set Interface and its Implementations** 9](#_Toc183876414)

[# HashSet 10](#_Toc183876415)

[# Union of Sets 10](#_Toc183876416)

[**4. Queue Interface and its Implementations** 11](#_Toc183876417)

[# LinkedList 12](#_Toc183876418)

[# PriorityQueue 12](#_Toc183876419)

[**5. Deque Interface and its Implementations** 13](#_Toc183876420)

[# ArrayDeque 13](#_Toc183876421)

[**6. Map Interface and its Implementations** 14](#_Toc183876422)

[# HashMap 15](#_Toc183876423)

[**7. Collection utility class** 16](#_Toc183876424)

[**8. Generics** 18](#_Toc183876425)

[# WildCards 21](#_Toc183876426)

## **1. What are Java Collections?**

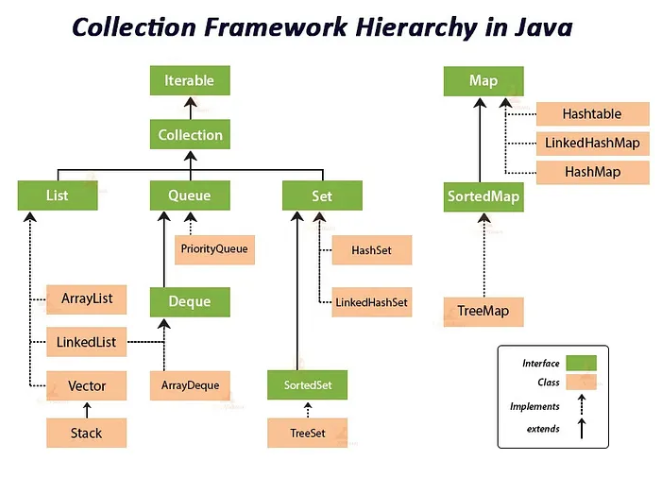
The Java Collections Framework (JCF) is a set of classes and interfaces that provide data structures to store and manipulate groups of objects efficiently. It serves as the foundation for handling collections of data in Java, simplifying many common data manipulation tasks and improving code reusability and performance.

Why Use Collections Instead of Arrays?

Arrays are fixed in size, making them rigid for dynamic data manipulation. Collections provide flexible and efficient ways to manage data that grows or shrinks dynamically. With collections, you get:

* Dynamic Size: Collections grow or shrink in size automatically, while arrays have a fixed size.
* Data Structure Variety: Collections provide a wide range of data structures (lists, sets, maps, queues, etc.).
* Built-in Utility Methods: Collections come with methods to add, remove, search, and manipulate data efficiently.

### # Java Collections Framework Architecture



The JCF has a well-defined architecture of interfaces and classes, making it easy to understand the relationships among different collection types.

Here’s a high-level view of the JCF architecture:

**Core Interfaces:** Define the behavior of collections.

* Iterable
* Collection
* List, Set, Queue, Map, and their variants

**Implementations:** Classes implementing these interfaces.

ArrayList, LinkedList, HashSet, TreeSet, HashMap, etc.

**List:-**

This stores data in ordered collection and can perform index based operations.

|  |  |
| --- | --- |
| ArrayList | Dynamic array, non-synchronized  Example of insertion and deletion |
| Vector | Synchronized version of ArrayList, legacy class  Are thread-safe (should only be use in multithreading environment) |
| LinkedList | Doubly linked list, best for frequent insertions and deletions. |
| Stack | Supports last-in-first-out access to elements. |

**Set:-**

Unordered collection of object in which no duplicates are allowed.

|  |  |
| --- | --- |
| HashSet | Hash table, no duplicates, allows null |
| LinkedHashSet | Maintains insertion order |
| TreeSet | Sorted order |

**Queue:-**

Maintain order of data in FIFO structure.

|  |  |
| --- | --- |
| PriorityQueue | FIFO structure, custom priority with comparator |

**Deque:-**

Double-ended queue, supports insertion/removal from both ends

|  |  |
| --- | --- |
| ArrayDeque | Resizable array implementation of the Deque |

**Map:-**

Stores data in key-value pair. Does not extends the Collection Interface.

|  |  |
| --- | --- |
| HashMap | Key-value pairs, hash table, allows one null key,non-synchronized |
| LinkedHashMap | Maintains insertion order |
| TreeMap | Sorted map based on natural ordering or comparator |
| HashTable | Synchronized, no null keys/values, legacy class |

Note:-

* All class starting with Linked, means they will maintain insertion order.
* All class starting with Tree, means they are sorted.

### # Iterable Interface

The Iterable interface is the root of the Java Collections Framework. It was introduced in Java 5 to allow collections to be iterated with the enhanced for loop (also known as the for-each loop). It belongs to java.lang package.

**Ways of Iterating**

There are three ways in which objects of **Iterable** can be iterated.

1. Using [enhanced for loop](https://www.geeksforgeeks.org/for-each-loop-in-java/)(for-each loop)
2. Using Iterable [forEach](https://www.geeksforgeeks.org/iterable-foreach-method-in-java-with-examples/) loop
3. Using **Iterator<T>** interface

The methods used while traversing the collections using Iterator to perform the operations are:

* hasNext(): It returns false if we have reached the end of the collection, otherwise returns true.
* next(): Returns the next element in a collection.

import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.Iterator;  
import java.util.List;  
  
public class LearnIterator {  
 public static void main(String[] args) {  
 ArrayList<String> arr = new ArrayList<>(Arrays.*asList*("Apple", "Ball", "Cat"));  
  
 //Iterate using enhanced for-each loop  
 for(String ele : arr){  
 System.*out*.println(ele);  
 }  
  
 //Iterate using ForEach Iterator  
 arr.forEach((ele) -> { System.*out*.println(ele); });  
  
 //Iterate using an Iterator  
 Iterator<String> iterator = arr.iterator();  
 while (iterator.hasNext()) {  
 String element = iterator.next();  
 System.*out*.println(element);  
 }  
 }  
}

### # Collection Interface

The Collection interface extends Iterable and represents a group of objects, known as elements. Collection is the root interface for many other collection types, such as List, Set, and Queue.

Key methods:-

| **Method Signature** | **Description** |
| --- | --- |
| boolean add(E e) | Adds an element to the collection. |
| boolean remove(index) | Removes a specified element from the collection. |
| boolean contains(Object o) | Checks if the collection contains the specified element. |
| int size() | Returns the number of elements in the collection. |
| boolean isEmpty() | Checks if the collection is empty. |
| void clear() | Removes all elements from the collection. |
| boolean addAll(Collection<? extends E> c) | Adds all elements from a specified collection to the current collection. |
| boolean containsAll(Collection<?> c) | Checks if the current collection contains all elements from another. |
| boolean removeAll(Collection<?> c) | Removes all elements in the collection that are also in another collection. |

Since Collection is an interface, it supports polymorphism. Any class implementing Collection can be referenced by Collection. This allows flexibility in changing the specific collection type as needed.

Can use this all ways:

ArrayList<String> arr = new ArrayList<>();  
Collection<String> arr2 = new ArrayList<>();  
List<String> arr3 = new ArrayList<>();

## **2. List Interface and its Implementations**

The List interface in Java extends the Collection interface and represents an ordered collection of elements, known as a sequence. Lists allow duplicate elements and provide precise control over the position where elements are inserted or accessed.

Key Characteristics of List

* **Order**: Elements are ordered based on their insertion order, which is preserved.
* **Indexing**: Allows positional access using indexes, starting from 0.
* **Duplicates**: Lists can contain duplicate elements.

**Different List Implementations**

| **Feature** | **ArrayList** | **LinkedList** | **Vector** | **Stack** |
| --- | --- | --- | --- | --- |
| **Underlying DS** | Dynamic Array | Doubly Linked List | Dynamic Array | Dynamic Array |
| **Thread Safety** | No | No | Yes | Yes |
| **Access Speed** | Fast (index-based) | Slow | Fast | Fast |
| **Insert/Delete** | Slow (index-based) | Fast | Slow | Fast |
| **Use Case** | Random access | Frequent inserts | Thread-safe list | LIFO operations |

|  |  |
| --- | --- |
| ArrayList | 1. Dynamic array, random access, non-synchronized 2. Example of insertion and deletion 3. Key Use Cases: Ideal for applications where frequent access to elements is required, with relatively fewer insertions and deletions. |
| Vector | 1. Synchronized version of ArrayList, legacy class 2. Are thread-safe (should only be use in multithreading environment 3. Key Use Cases: Suitable for multithreaded applications where a thread-safe dynamic array is required. |
| LinkedList | 1. Doubly linked list, best for frequent insertions and deletions. 2. Key Use Cases: Ideal for applications where frequent insertions and deletions occur at both ends or in the middle of the list. |
| Stack | 1. Supports last-in-first-out access to elements. 2. Key Use Cases: Useful for LIFO operations like evaluating expressions, tracking call stacks, etc.. |

Key Methods

| **Method Signature** | **Description** |
| --- | --- |
| void add(int index, E element) | Inserts an element at the specified position. |
| E get(int index) | Returns the element at the specified position. |
| E set(int index, E element) | Replaces the element at the specified position with a new element. |
| E remove(int index) | Removes the element at the specified position. |
| int indexOf(Object o) | Returns the index of the first occurrence of the specified element. |
| int lastIndexOf(Object o) | Returns the index of the last occurrence of the specified element. |
| List<E> subList(int fromIndex, int toIndex) | Returns a view of a portion of the list. |

### # ArrayList

import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.Collection;  
import java.util.List;  
  
public class LearnArray {  
 public static void main(String args[]){  
 //Implement arraylist with data  
 Collection<String> array = new ArrayList<>(Arrays.*asList*("Apple","Ball","Cat"));  
  
 //ArrayList without data  
 ArrayList<String> arr = new ArrayList<>();  
 arr.add("Animal");  
 arr.add("Bat");  
 arr.add("Canon");  
  
 //Accessing an element  
 System.*out*.println(arr.get(1)); //Output: Bat  
  
 //get the size  
 System.*out*.println(array.size()); //Output:3  
  
 //Modify an element (Can't do arr[1] = "Banana")  
 arr.set(1, "Banana");  
  
 //remove an element  
 arr.remove(2);  
  
 //Print arrayLists  
 System.*out*.println(array); //[Apple,Ball,Cat]  
 System.*out*.println(arr); //[Animal,Banana]  
  
 //get subList  
 List<Integer> arr2 = new ArrayList<>(Arrays.*asList*(1,2,3,4,5));  
 System.*out*.println(arr2.subList(1,3)); //[2,3]  
 }  
}

### # Stack

import java.util.Stack;  
  
public class LearnStack {  
 public static void main(String[] args) {  
 Stack<String> books = new Stack<>();  
  
 // Adding elements  
 books.push("Java Programming");  
 books.push("Data Structures");  
 books.push("Algorithms");  
  
 // Accessing the top element  
 System.*out*.println("Top book: " + books.peek());  
  
 // Removing an element  
 books.pop();  
  
 System.*out*.println("Stack after pop: " + books);  
 }  
}

## **3. Set Interface and its Implementations**

The Set interface in Java extends the Collection interface and represents an unordered collection of unique elements. Unlike lists, sets do not allow duplicate elements, making them ideal for storing distinct values.

Key Characteristics of Set

* **No Duplicates:** Sets do not allow duplicate elements.
* **No Indexing:** Elements are not stored in a particular order, so they do not support accessing elements by index (except for LinkedHashSet which maintains insertion order).
* **Efficient Search**: Sets are optimized for quick searches, as they use hashing or ordering mechanisms.

Comparison of different implementations of Set

| **Feature** | **HashSet** | **LinkedHashSet** | **TreeSet** |
| --- | --- | --- | --- |
| **Underlying DS** | Hash Table | Hash Table + Linked List | Red-Black Tree |
| **Order** | No | Insertion Order | Sorted Order |
| **Null Values** | Allows one null | Allows one null | Does not allow null |
| **Performance** | O(1) for add/remove/contains | O(1) with ordered iteration | O(log n) for operations |
| **Use Case** | Unique elements with no specific order | Ordered unique elements | Sorted unique elements |

|  |  |
| --- | --- |
| HashSet | 1. Hash table, no duplicates, allows null 2. Key Use Cases: Suitable for storing unique items where order does not matter, such as storing unique IDs or tags. |
| LinkedHashSet | 1. Maintains insertion order 2. Key Use Cases: Useful when the uniqueness of elements is needed, but order is also significant, such as when tracking items recently accessed. |
| TreeSet | 1. Sorted order, no duplicates 2. Key Use Cases: Suitable for storing sorted data and when operations like range queries, min/max retrieval are needed. |

Different methods of set

| **Method Signature** | **Description** |
| --- | --- |
| boolean add(E e) | Adds an element to the set, returning false if it’s already present. |
| boolean remove(Object o) | Removes the specified element if present. |
| boolean contains(Object o) | Checks if the set contains the specified element. |
| int size() | Returns the number of elements in the set. |
| void clear() | Removes all elements from the set. |
| boolean isEmpty() | Checks if the set is empty. |
| boolean addAll(Collection<? extends E> c) | Adds all elements from another collection, ignoring duplicates. |

### # HashSet

import java.util.\*;  
  
public class LearnSet {  
 public static void main(String args[]){  
 Set<String> set = new HashSet<>();  
 set.add("Ajax");  
 set.add("Byte");  
 set.add("Code");  
  
 //Check if element exists  
 System.*out*.println(set.contains("Byte")); //true  
  
 //Add multiple elements  
 set.addAll(Arrays.*asList*("Debug","Eclipse"));  
  
 System.*out*.println(set); //[Eclipse, Byte, Code, Debug, Ajax]  
 }  
}

### # Union of Sets

import java.util.\*;  
  
public class SetUnion {  
  
 public Set<Integer> getUnion(Set<Integer> set1, Set<Integer> set2) {  
 Set<Integer> unionSet = new HashSet<>(set1);  
 unionSet.addAll(set2);  
 return unionSet;  
 }  
  
 public static void main(String args[]){  
 HashSet<Integer> set1 = new HashSet<>(Arrays.*asList*(1,2,4));  
  
 Set<Integer> set2 = new HashSet<>();  
 Integer[] arr = {5,6,3};  
 set2.addAll(Arrays.*asList*(arr));  
  
 SetUnion su = new SetUnion();  
 Set<Integer> set3 = su.getUnion(set1,set2);  
 System.*out*.println(set3); //[1,2,3,4,5,6]  
  
 }  
}

## **4. Queue Interface and its Implementations**

The Queue interface in Java represents a collection designed to hold elements in a sequence, with processing based on the First-In-First-Out (FIFO) principle. It is useful for scenarios where elements are processed in the order they arrive, such as task scheduling.

**Characteristics of Queue**

* **FIFO Order:** The first element added is the first one removed.
* **Addition and Removal:** New elements are added at the end, and elements are removed from the front.
* **Specialized Methods:** Unlike other collections, Queue includes specific methods for handling elements based on their order.

**Queue:-**

|  |  |
| --- | --- |
| PriorityQueue | 1. Special type of queue that orders elements based on priority rather than insertion order. 2. Natural Ordering: Elements are ordered according to their natural order (e.g., numbers from smallest to largest). 3. Custom Comparator: Custom ordering can be specified for more complex priority logic. 4. No Nulls: PriorityQueue does not allow null elements. |

Key methods of queue:

| **Method Signature** | **Description** |
| --- | --- |
| boolean add(E e) | Adds an element to the queue, throwing an exception if full. |
| boolean offer(E e) | Adds an element to the queue, returning false if it fails. |
| E remove() | Removes and returns the head element, throwing an exception if empty. |
| E poll() | Removes and returns the head element, or null if the queue is empty. |
| E element() | Retrieves, but does not remove, the head element. Throws an exception if empty. |
| E peek() | Retrieves, but does not remove, the head element, or returns null if empty. |

The primary implementations of Queue in Java are:

* **LinkedList** - Doubly linked list-based implementation of Queue.
* **PriorityQueue** - Special type of queue that orders elements based on priority rather than insertion order.

### # LinkedList

LinkedList can function as both a List and a Queue. When used as a Queue, it follows the FIFO ordering.

import java.util.LinkedList;  
import java.util.Queue;  
  
public class LinkedListQueueExample {  
 public static void main(String[] args) {  
 Queue<String> queue = new LinkedList<>();  
  
 // Adding elements to the queue  
 queue.add("Alice");  
 queue.add("Bob");  
 queue.add("Charlie");  
 queue.offer("Danny");  
  
 // Displaying the head element  
 System.*out*.println("Head of the queue: " + queue.peek());  
  
 // Removing and displaying each element  
 while (!queue.isEmpty()) {  
 System.*out*.println("Removing: " + queue.poll());  
 }  
 }  
}

### # PriorityQueue

import java.util.PriorityQueue;  
import java.util.Queue;  
  
public class PriorityQueueExample {  
 public static void main(String[] args) {  
 Queue<Integer> priorityQueue = new PriorityQueue<>();  
  
 // Adding elements to the priority queue  
 priorityQueue.add(10);  
 priorityQueue.add(5);  
 priorityQueue.add(15);  
 priorityQueue.add(1);  
  
 // Removing and displaying elements in priority order  
 while (!priorityQueue.isEmpty()) {  
 System.*out*.println("Removing: " + priorityQueue.poll());   
 }  
 //Output: 1 ,5 ,10 ,15  
 }  
}

## **5. Deque Interface and its Implementations**

The Deque (Double Ended Queue) interface extends Queue and allows elements to be added or removed from both ends. It supports both FIFO and LIFO (Last-In-First-Out) operations.

Characteristics of Deque

* **Double-Ended:** Elements can be added/removed from both front and back.
* **Flexible:** Supports both FIFO (as a queue) and LIFO (as a stack).
* **Null Values**: Does not permit null elements to prevent ambiguity in retrieval.

| **Method Signature** | **Description** |
| --- | --- |
| void addFirst(E e) | Adds an element at the front of the deque. |
| void addLast(E e) | Adds an element at the end of the deque. |
| E removeFirst() | Removes and returns the front element of the deque. |
| E removeLast() | Removes and returns the last element of the deque. |
| E getFirst() | Retrieves, but does not remove, the front element. |
| E getLast() | Retrieves, but does not remove, the last element. |
| E pollFirst() | Retrieves and removes the front element, or null if empty. |
| E pollLast() | Retrieves and removes the last element, or null if empty. |

### # ArrayDeque

ArrayDeque is a resizable array-based implementation of Deque. It is more efficient than LinkedList for Deque operations and is the preferred implementation for stacks and queues.

import java.util.ArrayDeque;  
import java.util.Deque;  
  
public class ArrayDequeStackExample {  
 public static void main(String[] args) {  
 Deque<String> stack = new ArrayDeque<>();  
  
 // Using ArrayDeque as a stack (LIFO)  
 stack.push("Java");  
 stack.push("Python");  
 stack.push("C++");  
  
 while (!stack.isEmpty()) {  
 System.*out*.println("Popped: " + stack.pop());  
 }  
 }  
}

## **6. Map Interface and its Implementations**

The Map interface represents a collection of key-value pairs. Each key is unique, and each key maps to exactly one value. Unlike Set or List, Map does not extend the Collection interface, as its structure and behavior differ significantly.

Key Characteristics of Map

* **Key-Value Pairs:** Map stores data as pairs of keys and values.
* **Unique Keys:** Keys must be unique, but values can be duplicated.
* **Efficient Lookup:** Maps provide efficient methods for adding, removing, and retrieving values based on keys.

Different implementations of map

|  |  |
| --- | --- |
| HashMap | 1. Key-value pairs, hash table, allows one null key 2. Does not maintain order of elements 3. Key Use Cases: Efficient when order is not a priority, such as for caching or data lookup tables. |
| LinkedHashMap | 1. Maintains insertion order 2. Key Use Cases: Suitable for caching applications or maintaining an access order where insertion order is essential |
| TreeMap | 1. Sorted map based on natural ordering or comparator 2. Does not allow null keys but allows null values. 3. Provides fast retrieval operations due to sorted order. 4. Key Use Cases: Ideal for applications where sorted order of entries is essential, such as in database indexing. |
| HashTable | 1. Synchronized, no null keys/values, legacy class |

Different methods of map interface

| **Method Signature** | **Description** |
| --- | --- |
| V put(K key, V value) | Adds a key-value pair to the map. Returns the previous value for the key, if any. |
| V get(Object key) | Retrieves the value associated with the specified key. |
| V remove(Object key) | Removes the key-value pair for the specified key. |
| boolean containsKey(Object key) | Checks if the map contains a mapping for the specified key. |
| boolean containsValue(Object value) | Checks if the map contains one or more keys mapped to the specified value. |
| Set<K> keySet() | Returns a set view of the map's keys. |
| Collection<V> values() | Returns a collection view of the map's values. |
| Set<Map.Entry<K, V>> entrySet() | Returns a set view of the map's key-value mappings. |
| void clear() | Removes all mappings from the map. |
| int size() | Returns the number of key-value mappings in the map. |

| **Feature** | **HashMap** | **LinkedHashMap** | **TreeMap** |
| --- | --- | --- | --- |
| **Underlying DS** | Hash Table | Hash Table + Linked List | Red-Black Tree |
| **Order** | No order | Insertion order | Sorted by key |
| **Null Keys** | Allows one null key | Allows one null key | Does not allow null keys |
| **Performance** | O(1) for add/get | O(1) with predictable order | O(log n) for add/get |
| **Use Case** | Lookup tables, caching | Ordered caching, access order | Sorted data collections |

### # HashMap

package LearnCollection;  
  
import java.util.\*;  
public class LearnHashMap {  
 public static void main(String args[]){  
 Map<String,Integer> marks = new HashMap<>();  
 marks.put("Maths",96);  
 marks.put("Physics",92);  
 marks.put("Chemistry",99);  
  
 //retrieve the value(Marks in maths is: 96)  
 System.*out*.println("Marks in maths is: " + marks.get("Maths"));  
  
 //get all key values (Output: Subjects: [Maths, Chemistry, Physics])  
 System.*out*.println("Subjects: " + marks.keySet());  
  
 //DefaultValue (output: Marks in English: 0)  
 System.*out*.println("Marks in English: "+ marks.getOrDefault("Englist",0));  
  
 //Run loop on all values  
 Set<String> keys = marks.keySet();  
 for(String key : keys ){  
 System.*out*.println("Key " + key + " Value " + marks.get(key));  
 }  
 //Key Maths Value 96  
 //Key Chemistry Value 99  
 //Key Physics Value 92  
  
 //other way to loop  
 marks.forEach((k,v)->System.*out*.println("subject : " + k + " marks : " + v));  
 }  
}

## **7. Collection utility class**

The Collections class provides a wide variety of utility methods for manipulating collections, including:

* Sorting and searching
* Reversing and rotating elements
* Shuffling and making collections unmodifiable or thread-safe
* Finding frequency of elements and handling maximum/minimum values

The methods in the Collections class are static and can be used directly without creating an instance of the class.

Important methods

| **Method** | **Description** |
| --- | --- |
| sort(List<T> list) | Sorts the specified list in ascending order. |
| sort(List<T> list, Comparator<? super T> c) | Sorts the specified list according to the specified comparator. |
| reverse(List<?> list) | Reverses the order of elements in the specified list. |
| shuffle(List<?> list) | Randomly shuffles the elements in the specified list. |
| rotate(List<?> list, int distance) | Rotates elements in the list by the specified distance. |
| swap(List<?> list, int i, int j) | Swaps elements at the specified positions in the list. |
| frequency(Collection<?> c, Object o) | Returns the number of occurrences of the specified element in the collection. |
| max(Collection<? extends T> c) | Returns the maximum element in the specified collection. |
| min(Collection<? extends T> c) | Returns the minimum element in the specified collection. |
| binarySearch(List<? extends T> list, T key) | Searches the list for the specified element using binary search. |
| fill(List<? super T> list, T obj) | Replaces all elements of the specified list with the specified element. |
| copy(List<? super T> dest, List<? extends T> src) | Copies elements from the source list to the destination list. |
| unmodifiableList(List<? extends T> list) | Returns an unmodifiable view of the specified list. |
| synchronizedList(List<T> list) | Returns a thread-safe view of the specified list. |

import java.util.\*;  
  
public class LearnCollectionUtility {  
 public static void main(String[] args) {  
 List<Integer> arr = new ArrayList<>(Arrays.*asList*(22,4,16,34));  
  
 //Sort in ascending order  
 Collections.*sort*(arr);  
 System.*out*.println(arr); //[4, 16, 22, 34]  
  
 //Reverse the order  
 Collections.*reverse*(arr);  
 System.*out*.println(arr); //[34, 22, 16, 4]  
  
 List<Integer> arr2 = new ArrayList<>(Arrays.*asList*(19,21,9,15,12));  
  
 //Sort in descending order ( use of comparator)  
 Collections.*sort*(arr2,Comparator.*reverseOrder*());  
 System.*out*.println(arr2); //[21, 19, 15, 12, 9]  
  
 //Get the max value  
 Integer max = Collections.*max*(arr2);  
 System.*out*.println(max); //21  
  
 List<String> arr3 = new ArrayList<>(Arrays.*asList*("Apple","Banana","Apple","Orange"));  
  
 //Get the frequency of element  
 int count = Collections.*frequency*(arr3,"Apple");  
 System.*out*.println(count); //2  
 }  
}

## **8. Generics**

Generics allow creating 'type variables' which can be used to create classes, functions & type aliases that don't need to explicitly define the types that they use. Generics makes it easier to write reusable code.

Generics were introduced in Java 5 to provide type safety and reduce runtime errors by allowing classes, interfaces, and methods to operate on specified types.

Instead of working with raw Object types, generics enable a way to parameterize types.

* Type Safety: Compile-time checks ensure type correctness.
* Code Reusability: Classes and methods can work with any type.
* Avoids Type Casting: Reduces verbosity and runtime errors.

Here, lets say we want to create a LearnG class with Integer value, but then also want to have all same functionality with Strings and Double. Without generics we would end up creating multiple classes with almost same code, in that place we can use generics.

class LearnG<T>{  
 T value;  
  
 public void getType(){  
 System.*out*.println(value.getClass().getName());  
 }  
  
 //Can even return it from functions  
 public T getValue(){  
 //do some modification on T  
 return this.value;  
 }  
}  
public class GenericsExample {  
 public static void main(String[] args) {  
 LearnG<Integer> lgInt = new LearnG<Integer>();  
 lgInt.value = 230;  
 lgInt.getType(); //java.lang.Integer  
 System.*out*.println(lgInt.getValue()); //230  
  
 LearnG<String> lgString = new LearnG<>();  
 lgString.value = "Hello";  
 lgString.getType(); //java.lang.String  
 System.*out*.println(lgString.getValue()); //Hello  
 }  
}

**Note: Generics work with reference types only. Only use wrapper classes (e.g., Integer for int).**

Now we see that all collections also had same condition, because they also use generics. That’s why we are able to create ArrayList of String and Integer and even user defined classes and get all functionalities of ArrayList working for all data types.

Why not use Object class?

We know Object class is parent of all classes, so it also possible to get all functionalities of generics from Object class, but issue is that generics enforce type safety at compile time, removing the need for typecasting and reducing errors. While we will get ClassCastException in runtime with Object class.

Without Generics

import java.util.ArrayList;  
  
public class WithoutGenerics {  
 public static void main(String[] args) {  
 ArrayList list = new ArrayList(); // Raw type, no generics  
 list.add("Hello");  
 list.add(10); // Allowed but may cause issues later  
  
 for (Object obj : list) {  
 String str = (String) obj; // Risk of ClassCastException  
 System.*out*.println(str);  
 }  
 }  
 //At runtime got error: class java.lang.Integer cannot be cast to class java.lang.String  
}

With Generics

import java.util.ArrayList;  
  
public class WithGenerics {  
 public static void main(String[] args) {  
 ArrayList<String> list = new ArrayList<>(); // Type-safe  
 list.add("Hello");  
 // list.add(10); // Compile-time error  
  
 for (String str : list) {  
 System.*out*.println(str); // No casting required  
 }  
 }  
}

Generics with Multiple Type Parameters

Generics can support multiple type parameters, allowing more flexibility in creating reusable code.

**Note:** To use generics with function, you need to add type before return type of function.

class Pair<T, V> {  
 private T first;  
 private V second;  
  
 public Pair(T first, V second) {  
 this.first = first;  
 this.second = second;  
 }  
   
 //To use Type X which is not defined for class  
 //we need to add <X> before return type of function  
 public static <X> void getDataValue(X val) {  
 System.*out*.println("Value is: " + val);  
 }  
  
 @Override  
 public String toString() {  
 return "Pair{" + "first=" + first + ", second=" + second + '}';  
 }  
}  
  
public class MultipleTypeParameters {  
 public static void main(String[] args) {  
 Pair<String, Integer> pair = new Pair<>("Age", 25);  
 System.*out*.println(pair); //Pair{first=Age, second=25}  
  
 Pair<Double, Boolean> anotherPair = new Pair<>(99.9, true);  
 System.*out*.println(anotherPair); //Pair{first=99.9, second=true}  
  
 Pair.*getDataValue*(34); //Value is: 34  
 Pair.*getDataValue*("String"); //Value is: String  
 }  
}

Bounded Generics

Lets say we want type T to be only of Type Number, and as we know Integer, Long, Float, Double all extends number, all this wrapper classes can be used as T. In this case we can make T to only take only one of this classes. Can also extend Interface(for interface also here use extands0

class LearnG<T extends Number,V extends T>{  
 T value;  
 V value2;  
  
 public void getDoubledValue(){  
 System.*out*.println(value.intValue() + value2.intValue());  
 }  
  
}  
public class GenericsExample {  
 public static void main(String[] args) {  
 LearnG<Integer,Integer> lgInt = new LearnG<>();  
 lgInt.value = 230;  
 lgInt.value2 = 12;  
 lgInt.getDoubledValue(); //242  
  
 LearnG<Double,Double> lgDouble = new LearnG<>();  
 lgDouble.value = 30.2;  
 lgDouble.value2 = 41.4;  
 lgDouble.getDoubledValue(); //71  
 }  
}

### # WildCards

Wildcards (?) in generics represent an unknown type. They allow flexibility when working with parameterized types.

* **Generics (<T>)** → Used in class definitions and methods to enforce type safety.
* **Wildcards (?)** → Used in method parameters when the type is unknown or needs flexibility

import java.util.Arrays;  
import java.util.List;  
  
public class WildCardExample {  
  
 // Method that accepts any type of list  
 public static void printList(List<?> list) {  
 for (Object obj : list) {  
 System.*out*.println(obj);  
 }  
 }  
  
 public static void main(String[] args) {  
 List list = Arrays.*asList*("Apple", "Banana", "Cherry", 2, 323);  
 WildCardExample.*printList*(list);  
 //Output Apple , Banana , Cherry , 2 , 323  
  
 List<String> stringList = List.*of*("ABC", "PQR", "XYZ");  
 WildCardExample.*printList*(stringList);  
 //Output "ABC", "PQR" , "XYZ  
 }  
}

| **Scenario** | **Use Generics (<T>)** | **Use Wildcards (?)** |
| --- | --- | --- |
| You need a class that stores a specific type safely | ✅ Yes | ❌ No |
| You need a method that works with different types | ✅ Yes | ✅ Yes |
| You need to pass a collection with different types but don’t need to modify it | ❌ No | ✅ Yes |
| You want to return a value with a specific type | ✅ Yes | ❌ No |

If we replace ? with T, the method becomes tied to a single type for each invocation. The flexibility to handle different types of lists is lost because T assumes a consistent type within the method.

Also it is better to write methods with wildcard when there are not much updates on data

| **Feature** | **Wildcards (?)** | **Type Parameter (T)** |
| --- | --- | --- |
| **Definition** | Represents an unknown type. | Represents a specific type. |
| **Flexibility** | Used for methods that can operate on multiple types without restrictions. | Used for strongly typed, reusable methods or classes. |
| **Read/Write** | Can read but not write (except null). | Can read and write. |
| **Use Case** | When the method/class needs flexibility and doesn't modify elements. | When specific type operations are needed. |

//with wildcard  
public void printList(List<?> list) {  
 for (Object obj : list) {  
 System.*out*.println(obj);  
 }  
}

//with generics  
public <T> void printList(List<T> list) {  
 for (T element : list) {  
 System.*out*.println(element);  
 }  
}

Bounded wildcard

import java.util.List;  
  
public class BoundedWildcard{  
  
 // Method to process any list of numbers  
 public static void processNumbers(List<? extends Number> list) {  
 double sum = 0;  
 for (Number num : list) {  
 sum += num.doubleValue(); // Works for Integer, Double, etc.  
 }  
 System.*out*.println("Sum: " + sum);  
 }  
  
 public static void main(String[] args) {  
 List<Integer> intList = List.*of*(1, 2, 3, 4);  
 List<Double> doubleList = List.*of*(1.5, 2.5, 3.5);  
  
 *processNumbers*(intList); // Works for List<Integer> //sum: 10.0  
 *processNumbers*(doubleList); // Works for List<Double> //sum: 7.5  
 }  
}