**Collections Framework: Overview, Collection Interfaces, Collection Classes, Accessing a collection via Iterator, Working with Maps, Generics**

**Overview of Collections**

Collections are the one-stop solutions for all the data manipulation jobs such as storing data, searching, sorting, insertion, deletion, and updating of data. Java collection responds as a single object, and a Java Collection Framework provides various Interfaces and Classes.

A Java Collection is a predefined architecture capable of storing a group of elements and behaving like a single unit such as an [object](https://www.simplilearn.com/tutorials/java-tutorial/java-classes-and-objects) or a group.

**What is a framework in Java?**

A framework provides a ready-made structure of classes and interfaces for building software applications efficiently. It simplifies adding new features by offering reusable components that perform similar tasks, eliminating the need to create a framework from scratch for each new project. This approach enhances object-oriented design, making development quicker, more consistent, and reliable.

1. It provides readymade architecture.
2. It represents a set of classes and interfaces.
3. It is optional.

**What is Collection framework**

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It enhances code efficiency and readability by offering various data structures, including arrays, linked lists, trees, and hash tables, tailored to different programming needs. It has:

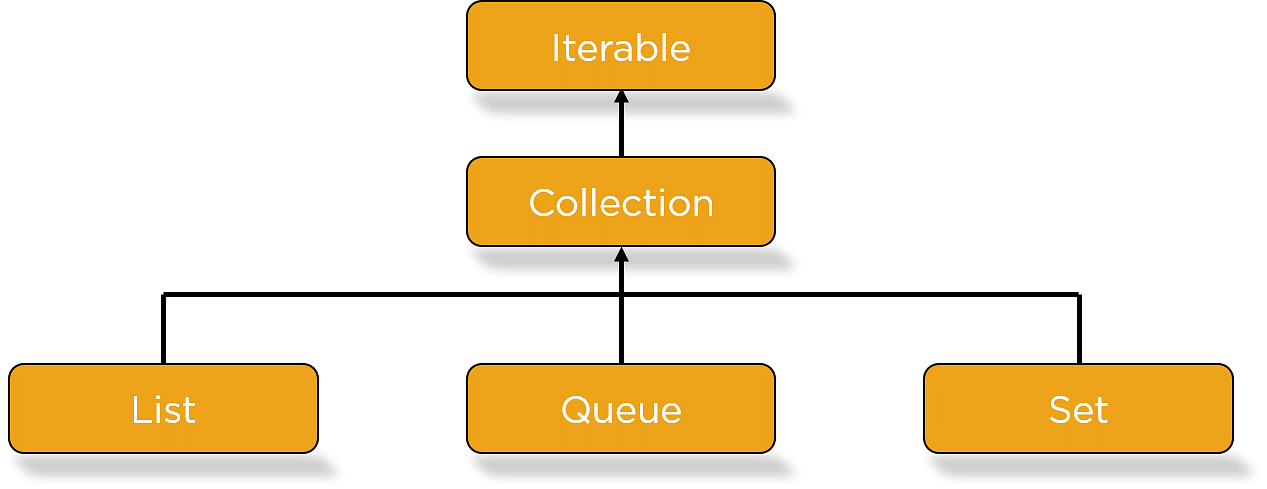
1. Interfaces and its implementations, i.e., classes
2. Algorithm

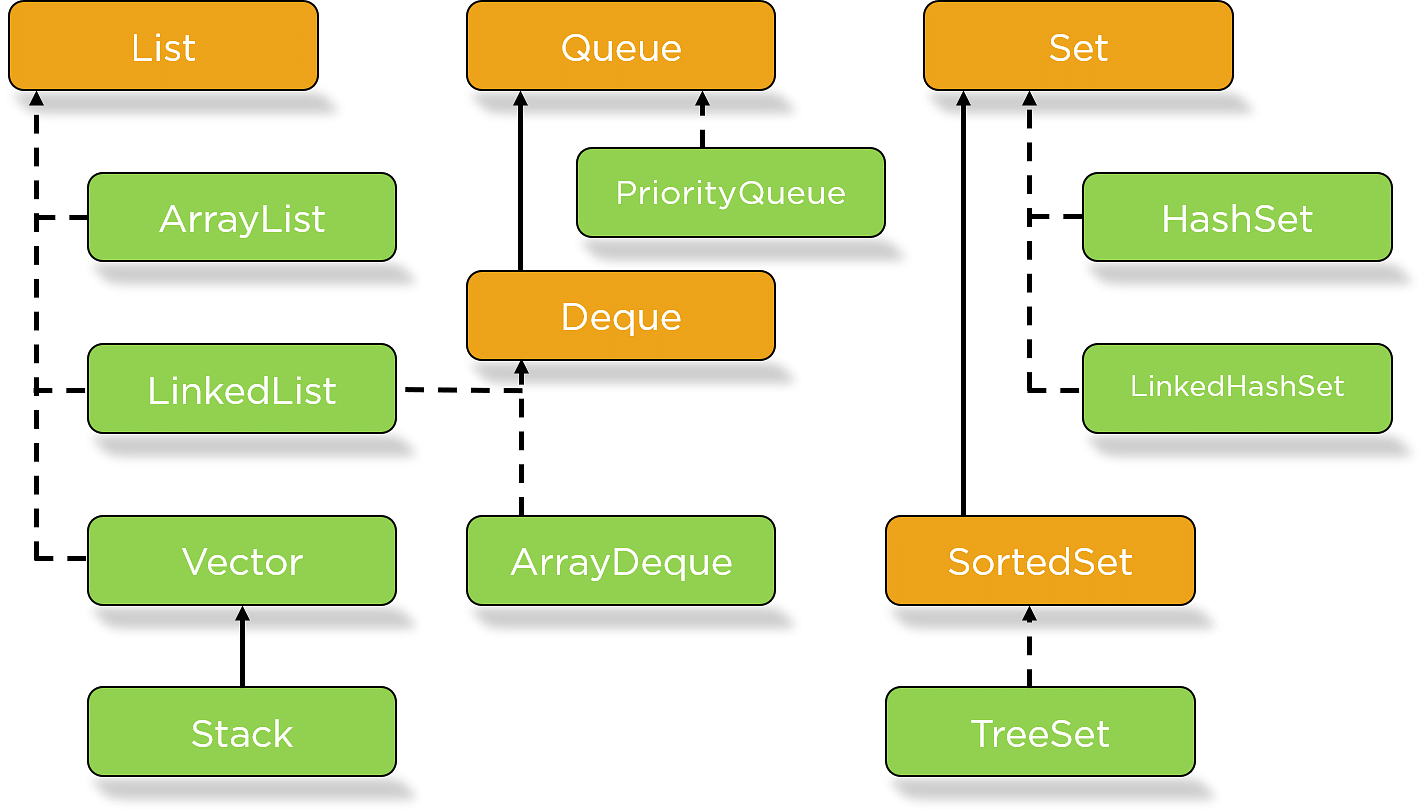
**Benefits of the Java Collections Framework**

The Java Collections Framework provides the following benefits:

1. **Reduces programming effort** by providing data structures and algorithms so you don't have to write them yourself.
2. **Increases performance** by providing high-performance implementations of data structures and algorithms. Because the various implementations of each interface are interchangeable, programs can be tuned by switching implementations.
3. **Provides interoperability** between unrelated APIs by establishing a common language to pass collections back and forth.
4. **Reduces the effort required to learn APIs** by requiring you to learn multiple ad hoc collection APIs.
5. **Reduces the effort required to design and implement APIs** by not requiring you to produce ad hoc collections APIs.
6. **Fosters software reuse** by providing a standard interface for collections and algorithms with which to manipulate them.

**Java Collection Framework Hierarchy**





## 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(): Returns an iterator over elements of type T.

**Collection Interface in Java**

1. The basic interface of the collections framework is the Collection interface which is the root interface of all collections in the API (Application Programming Interface).

It is placed at the top of the collection hierarchy in java. It provides the basic operations for adding and removing elements in the collection.

2. Collection interface extends the Iterable interface. The iterable interface has only one method called iterator(). The function of the iterator method is to return the iterator object. Using this iterator object, we can iterate over the elements of the collection.

3. [List](https://www.scientecheasy.com/2020/09/java-list-interface.html/), Queue, and [Set](https://www.scientecheasy.com/2020/09/java-set.html/) have three components which extends the Collection interface. A map is not inherited by Collection interface.

**List Interface**

1. This interface represents a collection of elements whose elements are arranged sequentially ordered.

2. List maintains an order of elements means the order is retained in which we add elements, and the same sequence we will get while retrieving elements.

3. We can insert elements into the list at any location. The list allows storing duplicate elements in Java.

4. [ArrayList](https://www.scientecheasy.com/2020/09/arraylist-in-java.html/), [vector](https://www.scientecheasy.com/2020/09/vector-in-java.html/), and [LinkedList](https://www.scientecheasy.com/2020/09/java-linkedlist.html/) are three concrete subclasses that implement the list interface.

**Set Interface**

1. This interface represents a collection of elements that contains unique elements. i.e, It is used to store the collection of unique elements.

2. Set interface does not maintain any order while storing elements and while retrieving, we may not get the same order as we put elements.  All the elements in a set can be in any order.

3. Set does not allow any duplicate elements.

4. HashSet, LinkedHashSet, TreeSet classes implements the set interface and sorted interface extends a set interface.

5. It can be iterated by using Iterator but cannot be iterated using ListIterator.

**SortedSet Interface**

1. This interface extends a set whose iterator transverse its elements according to their natural ordering.

2. TreeSet implements the sorted interface.

**Queue Interface**

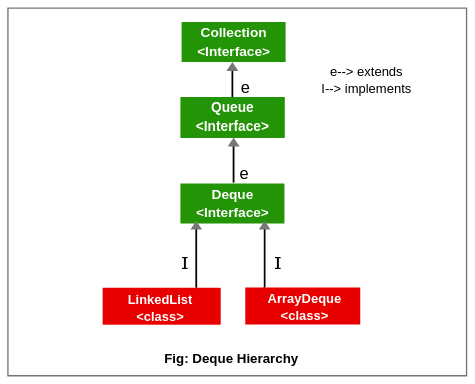
1. A queue is an ordered of the homogeneous group of elements in which new elements are added at one end(rear) and elements are removed from the other end(front). Just like a queue in a supermarket or any shop.

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

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

**Deque Interface**

1. A deque (double-ended queue) is a sub-interface of queue interface. It is usually pronounced “deck”.

2. This interface was added to the collection framework in Java SE 6. Deque interface extends the queue interface and uses its method to implement deque. The hierarchy of the deque interface is shown in the below figure.[](https://www.scientecheasy.com/2020/09/collection-hierarchy-in-java.html/)

4. 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.

5. LinkedList and ArrayDeque classes implement the Deque interface.

**Methods of Collection Interface in Java**

The Collection interface consists of a total of fifteen methods for manipulating elements in the collection. They are as follows:

1. **add():** This method is used to add or insert an element in the collection. The general syntax for add() method is as follow:

add(Object element) : boolean

If the element is added to the collection, it will return true otherwise false, if the element is already present and the collection doesn’t allow duplicates.

2. **addAll():** This method adds a collection of elements to the collection. It returns true if the elements are added otherwise returns false. The general syntax for this method is as follows:

addAll(Collection c) : boolean

3. **clear():** This method clears or removes all the elements from the collection. The general form of this method is as follows:

clear() : void

This method returns nothing.

4. **contains():** It checks that element is present or not in a collection. That is it is used to search an element. The general for contains() method is as follows:

contains(Object element) : boolean

This method returns true if the element is present otherwise false.

5. **containsAll():** This method checks that specified a collection of elements are present or not. It returns true if the calling collection contains all specified elements otherwise return false. The general syntax is as follows:

containsAll(Collection c) : boolean

6. **equals():** It checks for equality with another object. The general form is as follows:

equals(Object element) : boolean

7. **hashCode():** It returns the hash code number for the collection. Its return type is an integer. The general form for this method is:

hashCode() : int

8. **isEmpty():** It returns true if a collection is empty. That is, this method returns true if the collection contains no elements.

isEmpty() : boolean

9. **iterator():** It returns an iterator. The general form is given below:

iterator() : Iterator

10. **remove():** It removes a specified element from the collection. The general syntax is given below:

remove(Object element) : boolean

This method returns true if the element was removed. Otherwise, it returns false.

11. **removeAll():** The removeAll() method removes all elements from the collection. It returns true if all elements are removed otherwise returns false.

removeAll(Collection c) : boolean

12. **retainAll():** This method is used to remove all elements from the collection except the specified collection. It returns true if all the elements are removed otherwise returns false.

retainAll(Collection c) : boolean

13. **size():** The size() method returns the total number of elements in the collection. Its return type is an integer. The general syntax is given below:

size() : int

14. **toArray():** It returns the elements of a collection in the form of an array. The array elements are copies of the collection elements.

toArray() : Object[]

15. **Object[ ] toArray():** Returns an array that contains all the elements stored in the invoking collection. The array elements are the copies of the collection elements.

toArray(Object array[]) : Object[]

### **Collection classes**

### **LinkedList**

It is the most commonly used data structure that implements a doubly linked list to store the elements inside it. It can store duplicate elements. It implements the Dequeue interface extended by the Queue interface and the List interface. It is not synchronized. Now let’s see how to solve our problem discussed above (the FIFO concept) using LinkedList. The problem is to serve the customers in a manner they arrive i.e **first in first out**.

#### Example

**import** java.util.\*;

**public** **class** TestJavaCollection2{

**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());

}

}

}

### **ArrayList**

It simply implements the List interface. It maintains the insertion order and uses a dynamic array to store elements of different data types. Elements can be duplicated. It is also non-synchronized and can store null values. Now let’s see its different methods... These are useful when we do not know how many records or elements we need to insert. Let’s take an example of a library where we don’t know how many books we have to keep. So whenever we have a book, we need to insert it into ArrayList.

#### Example

**import** java.util.\*;

**class** TestJavaCollection1{

**public** **static** **void** main(String args[]){

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

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

list.add("Vijay");

list.add("Ravi");

list.add("Ajay");

//Traversing list through Iterator

Iterator itr=list.iterator();

**while**(itr.hasNext()){

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

}

}

}

### **HashSet**

It implements the Set interface and never contains duplicate values. It implements the hash table for storing the values. It also allows null values. It never maintains the insertion order but provides the constant time performance for add, remove, size, and contains methods. It is best for search operations and it is not synchronized.

#### Example

**import** java.util.\*;

**class** HashSetExample{

**public** **static** **void** main(String args[]){

//creating HashSet and adding elements to it

HashSet<Integer> hashSet=**new** HashSet();

hashSet.add(1);

hashSet.add(5);

hashSet.add(4);

hashSet.add(3);

hashSet.add(2);

//getting an iterator for the collection

Iterator<Integer> i=hashSet.iterator();

//iterating over the value

**while**(i.hasNext()) {

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

}

}

}

### Output

1 2 3 4 5

As you can see it does not maintains the insertion order.

### **ArrayDeque**

It implements the Deque interface so it allows operations from both ends. It does not allow null values. It is faster than Stack and LinkedList when implemented as Stack and LinkedList. ArrayDeque has no size restrictions as it grows and shrinks as per the requirements. It is unsynchronized, meaning it is not thread-safe. To keep it thread-safe we have to implement some external logic.

#### Example

**import** java.util.\*;

**public** **class** ArrayDequeExample {

**public** **static** **void** main(String[] args) {

//creating Deque and adding elements

Deque<String> deque = **new** ArrayDeque<String>();

//adding an element

deque.add("One");

//adding an element at the start

deque.addFirst("Two");

//adding an element at the end

deque.addLast("Three");

//traversing elements of the collection

**for** (String str : deque) {

System.out.println(str);

}

}

}

#### Output

Two One Three

### **HashMap**

It is the implementation of the Map interface backed by the hash table. It stores the key-value pairs. It does not allow null values. It is not synchronized. It never guarantees the insertion order. It provides constant time performance for methods like get, and put. Its performance depends on two factors — **initial capacity** and **load factor**. Capacity is the number of buckets in the hash table so the initial capacity is the number of buckets allocated at the time of creation. Load factor is the measure of how much a hash table can be populated before its capacity is increased. The rehash method is used to increase the capacity and it mainly doubles the number of buckets.

#### Example

**import** java.util.\*;

**public** **class** HashMapExample{

**public** **static** **void** main(String args[]){

//creating a HashMap

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

//putting elements into the map

map.put(1,"England");

map.put(2,"USA");

map.put(3,"China");

//get element at index 2

System.out.println("Value at index 2 is: "+map.get(2));

System.out.println("iterating map");

//iterating the map

**for**(Map.Entry m : map.entrySet()){

System.out.println(m.getKey()+" "+m.getValue());

}

}

}

# **Accessing a Java Collection using Iterators**

To access elements of a collection, either we can use index if collection is list based or we need to traverse the element. There are three possible ways to traverse through the elements of any collection.

1. Using Iterator interface
2. Using ListIterator interface
3. Using for-each loop

### **Accessing elements using Iterator**

Iterator is an interface that is used to iterate the collection elements. It is part of java collection framework. It provides some methods that are used to check and access elements of a collection.

Iterator Interface is used to traverse a list in forward direction, enabling you to remove or modify the elements of the collection. Each collection classes provide iterator() method to return an iterator.

### **Iterator Interface Methods**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean hasNext() | Returns **true** if there are more elements in the collection. Otherwise, returns false. |
| E next() | Returns the **next element present** in the collection. Throws NoSuchElementException if there is not a next element. |
| void remove() | Removes the current element. Throws IllegalStateException if an attempt is made to call remove() method that is not preceded by a call to next() method. |

### **Iterator Example**

In this example, we are using iterator() method of collection interface that returns an instance of Iterator interface. After that we are using hasNext() method that returns true of collection contains an elements and within the loop, obtain each element by calling next() method.

import java.util.\*;

class Demo

{

public static void main(String[] args)

{

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

ar.add("ab");

ar.add("bc");

ar.add("cd");

ar.add("de");

Iterator it = ar.iterator(); //Declaring Iterator

while(it.hasNext())

{

System.out.print(it.next()+" ");

}

}

}

### **Accessing elements using ListIterator**

ListIterator Interface is used to traverse a list in both **forward** and **backward** direction. It is available to only those collections that implements the **List** Interface.

### **Methods of ListIterator:**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(E obj) | Inserts obj into the list in front of the element that will be returned by the next call to next() method. |
| boolean hasNext() | Returns true if there is a next element. Otherwise, returns false. |
| boolean hasPrevious() | Returns true if there is a previous element. Otherwise, returns false. |
| E next() | Returns the next element. A NoSuchElementException is thrown if there is not a next element. |
| int nextIndex() | Returns the index of the next element. If there is not a next element, returns the size of the list. |
| E previous() | Returns the previous element. A NoSuchElementException is thrown if there is not a previous element. |
| int previousIndex() | Returns the index of the previous element. If there is not a previous element, returns -1. |
| void remove() | Removes the current element from the list. An IllegalStateException is thrown if remove() method is called before next() or previous() method is invoked. |
| void set(E obj) | Assigns obj to the current element. This is the element last returned by a call to either next() or previous() method. |

### **ListIterator Example**

Lets create an example to traverse the elements of ArrayList. ListIterator works only with list collection.

import java.util.\*;

class Demo

{

public static void main(String[] args)

{

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

ar.add("ab");

ar.add("bc");

ar.add("cd");

ar.add("de");

ListIterator litr = ar.listIterator();

while(litr.hasNext()) //In forward direction

{

System.out.print(litr.next()+" ");

}

while(litr.hasPrevious()) //In backward direction

{

System.out.print(litr.previous()+" ");

}

}

}

ab bc cd de de cd bc ab

### **for-each loop**

for-each version of for loop can also be used for traversing the elements of a collection. But this can only be used if we don't want to modify the contents of a collection and we don't want any **reverse** access. for-each loop can cycle through any collection of object that implements Iterable interface.

### **Exmaple:**

import java.util.\*;

class Demo

{

public static void main(String[] args)

{

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

ls.add("a");

ls.add("b");

ls.add("c");

ls.add("d");

for(String str : ls)

{

System.out.print(str+" ");

}

}

}

a b c d

### **Traversing using for loop**

we can use for loop to traverse the collection elements but only index-based collection can be accessed. For example, list is index-based collection that allows to access its elements using the index value.

import java.util.\*;

class Demo

{

public static void main(String[] args)

{

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

ls.add("a");

ls.add("b");

ls.add("c");

ls.add("d");

for(int i = 0; i<ls.size(); i++)

{

System.out.print(ls.get(i)+" ");

}

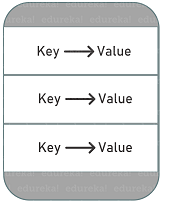
}

}

a b c d

## ****Java Map Interface****

**A Map in Java is an**[***object***](https://www.edureka.co/blog/java-object/)**that maps keys to values and is designed for the faster lookups. Data is stored in key-value pairs and every key is unique.  Each key maps to a value hence the name map.  These key-value pairs are called map entries.**



In the [JDK](https://www.edureka.co/blog/what-is-java/#ComponentsinJava), [java.util.Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html) is an [interface](https://www.edureka.co/blog/java-collections/#interface) that includes method signatures for insertion, removal, and retrieval of elements based on a key. With such methods, it’s a perfect tool to use for key-value association mapping such as dictionaries.

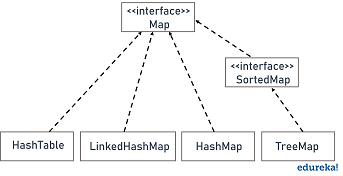
## ****Characteristics of Map Interface****

* The Map interface is not a true subtype of Collection interface, therefore, its characteristics and behaviors are different from the rest of the collection types.
* It provides three collection views – set of keys, set of key-value mappings and collection of values.
* A Map cannot contain duplicate keys and each key can map to at most one value. Some implementations allow null key and null value (HashMap and LinkedHashMap) but some does not (TreeMap).
* The Map interface doesn’t guarantee the order of mappings, however, it depends on the implementation. For instance, HashMap doesn’t guarantee the order of mappings but TreeMap does.
* AbstractMap class provides a skeletal implementation of the Java Map interface and most of the Map concrete [classes](https://www.edureka.co/blog/java-objects-and-classes/) extend AbstractMap class and implement required methods.

Now that you have an idea of what Map interface in[Java](https://www.edureka.co/blog/what-is-java/) is, let’s go ahead and check out the hierarchy of Java Map.

## ****Java Map Hierarchy****

There are two interfaces that implement the Map in java: Map and SortedMap. And popular implementation classes of Map in Java are HashMap, TreeMap, and LinkedHashMap. The hierarchy of Java Map is given below:



**Methods in Java Map Interface**

|  |  |
| --- | --- |
| **Methods** | **Description** |
| public put(Object key, Object value) | This method inserts an entry in the map |
| public void putAll(Map map) | This method inserts the specified map in this map |
| public Object remove(Object key) | It is used to delete an entry for the specified key |
| public Set keySet() | It returns the Set view containing all the keys |
| public Set entrySet() | It returns the Set view containing all the keys and values |
| void clear() | It is used to reset the map |
| public void putIfAbsent(K key, V value) | It inserts the specified value with the specified key in the map only if it is not already specified |
| public Object get(Object key) | It returns the value for the specified key |
| public boolean containsKey(Object key) | It is used to search the specified key from this map |

### **HashMap Class**

The most common class that implements the Java Map interface is the HashMap. It is a hash table based implementation of the Map interface. It implements all of the Map operations and allows null values and one null key. Also, this class does not maintain any order among its elements. Here’s an example program demonstrating the HashMap class.

**import** java.util.\*;

**public** **class** HashMapExample1

{

**public** **static** **void** main(String args[])

{

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

map.put(1,"Mango");  //Put elements in Map

map.put(2,"Apple");

map.put(3,"Banana");

map.put(4,"Grapes");

System.out.println("Iterating Hashmap...");

**for**(Map.Entry m : map.entrySet())

{

System.out.println(m.getKey()+" "+m.getValue());

}

}

}

### **TreeMap Class**

This implementation uses the Red-Black tree as the underlying [data structure](https://www.edureka.co/blog/data-structures-algorithms-in-java/). A TreeMap is sorted according to the natural ordering of its keys, or by a Comparator provided at creation time. This implementation doesn’t allow nulls but maintains order on its elements. Here’s an example program demonstrating the TreeMap class.

**import** java.util.\*;

**public** **class** TreeMap2

{

**public** **static** **void** main(String args[])

{

TreeMap<Integer,String> map=**new** TreeMap<Integer,String>();

map.put(100,"Amit");

map.put(102,"Ravi");

map.put(101,"Vijay");

map.put(103,"Rahul");

System.out.println("Before invoking remove() method");

**for**(Map.Entry m:map.entrySet())

{

System.out.println(m.getKey()+" "+m.getValue());

}

map.remove(102);

System.out.println("After invoking remove() method");

**for**(Map.Entry m:map.entrySet())

{

System.out.println(m.getKey()+" "+m.getValue());

}

}

}

### **LinkedHashMap Class**

As the name indicates this implementation of Java Map interface uses a hash table and a linked list as the underlying data structures. Thus the order of a LinkedHashMap is predictable, with insertion-order as the default order. Also, allows nulls like in HashMap. Here’s an example program demonstrating the TreeMap class

**import** java.util.\*;

**public** **class** LinkedHashMap3

{

**public** **static** **void** main(String args[])

{

Map<Integer,String> map=**new** LinkedHashMap<Integer,String>();

map.put(101,"Amit");

map.put(102,"Vijay");

map.put(103,"Rahul");

System.out.println("Before invoking remove() method: "+map);

map.remove(102);

System.out.println("After invoking remove() method: "+map);

}

}

**Java Generics**

Java Generics allows us to create a single class, interface, and method that can be used with different types of data (objects).

This helps us to reuse our code.

**Note**: **Generics** does not work with primitive types (int, float, char, etc).

## Java Generics Class

We can create a class that can be used with any type of data. Such a class is known as Generics Class.

Here's is how we can create a generics class in Java:

### Example: Create a Generics Class

class GenericExample

{

public static void main(String[] args)

{

// initialize generic class with Integer data

GenericsClass<Integer> intObj = new GenericsClass<>(5);

System.out.println("Generic Class returns: " + intObj.getData());\

// initialize generic class with String data

GenericsClass<String> stringObj = new GenericsClass<>("Java Programming");

System.out.println("Generic Class returns: " + stringObj.getData());

}

}

// create a generics class

class GenericsClass<T>

{

// variable of T type

private T data;

public GenericsClass(T data)

{

this.data = data;

}

// method that return T type variable

public T getData()

{

return this.data;

}

}

**Output**

Generic Class returns: 5

Generic Class returns: Java Programming

In the above example, we have created a generic class named GenericsClass. This class can be used to work with any type of data.

class GenericsClass<T> {...}

Here, T used inside the angle bracket <> indicates the **type parameter**. Inside the Main class, we have created two objects of GenericsClass

* intObj - Here, the type parameter T is replaced by Integer. Now, the GenericsClass works with integer data.
* stringObj - Here, the type parameter T is replaced by String. Now, the GenericsClass works with string data.

## Java Generics Method

Similar to the generics class, we can also create a method that can be used with any type of data. Such a class is known as Generics Method.

Here's is how we can create a generics method in Java:

### Example: Create a Generics Method

class Main {

public static void main(String[] args) {

// initialize the class with Integer data

DemoClass demo = new DemoClass();

// generics method working with String

demo.<String>genericsMethod("Java Programming");

// generics method working with integer

demo.<Integer>genericsMethod(25);

}

}

class DemoClass {

// creae a generics method

public <T> void genericsMethod(T data) {

System.out.println("Generics Method:");

System.out.println("Data Passed: " + data);

}

}

**Output**

Generics Method:

Data Passed: Java Programming

Generics Method:

Data Passed: 25

In the above example, we have created a generic method named genericsMethod.

public <T> void genericMethod(T data) {...}

Here, the type parameter <T> is inserted after the modifier public and before the return type void.

We can call the generics method by placing the actual type <String> and <Integer> inside the bracket before the method name.

## Bounded Types

In general, the **type parameter** can accept any data types (except primitive types).

However, if we want to use generics for some specific types (such as accept data of number types) only, then we can use bounded types.

In the case of bound types, we use the extends keyword. For example,

<T extends A>

This means T can only accept data that are subtypes of A.

### Example: Bounded Types

class GenericsClass <T extends Number> {

public void display() {

System.out.println("This is a bounded type generics class.");

}

}

class Main {

public static void main(String[] args) {

// create an object of GenericsClass

GenericsClass<String> obj = new GenericsClass<>();

}

}

In the above example, we have created a class named GenericsClass. Notice the expression, notice the expression

<T extends Number>

Here, GenericsClass is created with bounded type. This means GenericsClass can only work with data types that are children of Number (Integer, Double, and so on).

However, we have created an object of the generics class with String. In this case, we will get the following error.

GenericsClass<String> obj = new GenericsClass<>();

^

reason: inference variable T has incompatible bounds

equality constraints: String

lower bounds: Number

where T is a type-variable:

T extends Number declared in class GenericsClass

## Advantages of Java Generics

### 1. Code Reusability

With the help of generics in Java, we can write code that will work with different types of data. For example,

public <T> void genericsMethod(T data) {...}

Here, we have created a generics method. This same method can be used to perform operations on integer data, string data, and so on.

### 2. Compile-time Type Checking

The **type parameter** of generics provides information about the type of data used in the generics code. For example,

// using Generics

GenericsClass<Integer> list = new GenericsClass<>();

Here, we know that GenericsClass is working with Integer data only.

Now, if we try to pass data other than Integer to this class, the program will generate an error at compile time.

### 3. Used with Collections

The collections framework uses the concept of generics in Java. For example,

// creating a string type ArrayList

ArrayList<String> list1 = new ArrayList<>();

// creating a integer type ArrayList

ArrayList<Integer> list2 = new ArrayList<>();

In the above example, we have used the same [ArrayList class](https://www.programiz.com/java-programming/arraylist) to work with different types of data.

Similar to ArrayList, other collections (LinkedList, Queue, Maps, and so on) are also generic in Java.

**Difference between Comparable and Comparator**

Comparable and Comparator both are interfaces and can be used to sort collection elements.

However, there are many differences between Comparable and Comparator interfaces that are given below.

|  |  |
| --- | --- |
| **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. |