**What is a Collection in Java?**

Consider the example of a piggy bank. We all had it during our childhood where we used to store our coins. This piggy bank is called as Collection and the coins are nothing but objects. Technically, a collection is an object or a container that stores a group of other objects.

A Collection in Java is an object which represents a group of objects, known as its elements. It is a single unit. They are used to standardize the way in which objects are handled in the class.

Now that we know what Collection in Java is, **let us try to understand the real-life use cases**:

* Linked list emulates your browsing history, train coaches which are connected to each other etc.
* Stacks are like a stack of plates or trays in which the top-most one gets picked first.
* Queue is same as the real-life queues, the one who enters the **queue first**, it leaves it first too.

**What is Java Collections Framework?**

The Collections Framework is defined as a unified architecture for representing and manipulating collections. In Java, the Collections Framework is a hierarchy of **interfaces** and **classes** that provides an easy management of a group of objects.  
The java.util package contains the powerful tool of Collections Framework. It was defined in JDK 1.2 version which is one of the most used frameworks till date.

It provides a **ready-made architecture** for interfaces and classes and used for storing and manipulating a group of objects. All collections framework contain interfaces, classes and algorithms.

Now you might be wondering what is its need. Collections in Java were not a part of the original Java release. But prior to the release, **Vectors, Stacks and Arrays** were there. They had one major disadvantage, and that was lesser similarities. They didn't have a common interface and interconnection with each other. In this case, it becomes tedious for the user to remember all the **functions and syntax**. Plus, the conventional ways like arrays and stacks weren't providing the desired performance and flexibility.

This is why the Collections Framework and collection in Java was introduced in order to overcome the **drawbacks mentioned above**.

**Differences between the Collection and Collections Framework in Java**

* The following are the differences between Collection in Java and Collections Framework

| **Collection in Java** | **Collections Framework** |
| --- | --- |
| Collection in Java is a class. | Collections Framework is a framework. |
| It is a single unit which contains and **manipulates** a group of objects. | They are used to **manipulate collections**. |

**Advantages of the Collections Framework**

As mentioned above, the Collections Framework in Java carries a lot of advantages.

* Provides **high-performance and high efficiency**. This is due to the fact that various implementations of each interface are interchangeable, so programs can be written by **switching implementations**.
* Some methods of each interface of the Collections Framework have a **uniform implementation**. The Collections API has basic interfaces like Set, Map and List, so the classes that implement them have a few common set of methods.
* **Reduces the programming effort** by providing useful data structures and algorithms. We don't have to write our own data structure as it has already been provided to us.
* **Facilitates code reusability**. Collections Framework provides common classes and interfaces that can be used with different types of collections.

**Hierarchy of Collections Framework**

The Collections Framework in Java follows a certain hierarchy. All the classes and interfaces in the Collections Framework come under the java.util package.

But before getting acquainted with the hierarchy of the Collections Framework in Java, Let us understand these terms.

* Class: A class is a collection of similar types of objects. It is an **implementation of the collection interface**.
* Interface: An interface is the abstract data type. It is at the topmost position in the framework hierarchy. In Layman's terms, it is a group of related method with empty bodies. Abstraction is a process of hiding the implementation details from the user by only providing the functionality. Using interfaces, **we can achieve (complete) abstraction**.
* extends: It is a keyword which is used to develop inheritance between two classes and two interfaces.Inheritance in Java refers to the concept that one can create new classes that are built upon existing classes.
* implements: implements is also a keyword used to develop inheritance between class and interface. A class implements an interface.

The above figure illustrates the hierarchy of the Collections Framework. It consists of four core interfaces such as **Collection, List, Set, Queue and various classes** which get implemented through them.

**Collection Interface**

The Collection Interface is the **root** or the foundation on which the **Collections Framework** is built. It is a general interface that has the declaration:

interface Collection<E>

Here, E is the type of object that the collection will hold.

It provides the basic operations like adding, removing, clearing the elements in a collection, checking whether the collection is empty etc.

List, Queue and Set are the components that extend the Collection Interface.

**Methods of the Collection Interface**

The Collection Interface has the following methods. The methods declared in an interface are by default abstract **(only method signature, no body of the method)**.

| **Method** | **Description** |
| --- | --- |
| boolean add(E obj) | It is used to add an object obj to the collection. Returns true if obj was added, else returns false if the element obj was already a member and the collection does not allow duplicates. |
| boolean addAll(Collection<? extends E> c) | Adds all the elements of c to the collection.Returns true if the object was added, else returns false. |
| void clear() | Removes all the elements from the collection. |
| int size() | Returns the number of elements present in the collection. |
| Iterator iterator() | Returns an iterator for the collection. It is an object that can be used to loop through collections. |
| boolean contains(Object obj) | Checks if the object is present in the collection and returns true if found. Else, it returns false. |
| boolean containsAll(Collection<?> c) | Returns true if the collection contains all the elements of c, else returns false. |
| int hashCode() | It returns the hash code for the collection i.e., returns an integer or a 4 byte value which is generated by the hashing algorithm. |
| boolean equals(Object obj) | It returns true if the collection and obj are equal, else returns false. |
| boolean isEmpty() | Returns true if the collection is empty, else returns false. |
| boolean remove(Object obj) | Removes one instance of obj from the collection. It returns true if the element was removed, else returns false. |
| boolean removeAll(Collection<?> c) | Removes all elements of c from the collection. Returns true if the elements were removed, else returns false. |
| boolean retainAll(Collection<?> c) | It retains only those elements which are in c and removes the other elements from the collection. It returns true if the elements were removed, else returns false. |
| default Spliterator spliterator() | Returns a spliterator to the collection. A spliterator can be used to iterate over a collection and split it into \*\*multiple sets. |

**Iterator Interface**

Iterator is an object that can be **used to loop through collections**.  
As the name suggests, it is used to iterate over the elements. It is used to modify and iterate over the elements in a collection.

**There are 3 methods in the Iterator interface:**

* public Object next()- It returns the next element in the collection. It throws the exception of NoSuchElementException if there is no next element.
* public void remove()- It removes the current element from the collection.
* public boolean hasNext()- It returns true if there are more elements in the collection. Else, returns false.

**Iterable Interface**

The Iterable Interface allows the collection to be iterated over. The root interface for the entire collection framework is termed as the Iterable Interface. The collection interface extends the iterable interface, hence the sub-classes of the collection interface also implement the iterable interface, i.e, it **automatically becomes a part of the iterable interface**.

**Syntax**:

Iterator <E> iterator()

Returns an iterator of type E for the collection. It can be used to **iterate over** the elements of the collection.

**List Interface**

The list interface extends the collection interface. A list is used to store ordered collection of data and it may contain duplicates. **Ordered collection means** the order in which the elements are being inserted and they contain a specific value. The elements present, can be accessed or inserted by their position in the list using zero-based indexing. The list interface is implemented by LinkedList, ArrayList, Vectors and Stack classes. They are an **important part of collections in Java**.

There are three classes implemented by the list interface and they are given below.

**Syntax**

List<Data-Type>linkedlist= new LinkedList<Data-Type>();

List<Data-Type>arraylist= new ArrayList<Data-Type>();

List<Data-Type>vector= new Vector<Data-Type>();

**Let us learn about each class implemented by the List Interface in depth**.

**1. ArrayList**

* So, you **must've figured it out** from the name itself, that ArrayList is similar to Arrays. They are also called as dynamic arrays. That means, it does not have a fixed size. Its size can be increased or decreased if elements are added or removed.
* It implements the **List Interface**.
* It is similar to Vectors in **C++**.
* Since the ArrayList cannot be used for primitive data types like **int,char etc**. , we need to use a wrapper class.

In order to initialize an ArrayList, there are 3 ways.

**1. Using the add Keyword**

ArrayList<Data-Type> str = new ArrayList<Data-Type>();

str.add(<content>);

str.add(<content>);

str.add(<content>);

**Example**

*//initialising ArrayList using add keyword*

import java.util.\*;

public class ScalerTopics {

public static void main(String args[])

{

*// create an ArrayList of String type*

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

*// Initialize an ArrayList with add()*

str.add("Scaler");

str.add("Topics");

str.add("Rocks");

*// to print the ArrayList*

System.out.println("ArrayList is" + str);

}

}

**Output**

ArrayList is [Scaler, Topics, Rocks]

**2. Using asList()** AsList() method in Java is used to return a fixed-size list backed by the given array.

ArrayList<Data-Type> obj =

new ArrayList<Data-Type(Arrays.asList(Obj A, Obj B, Obj C, ....));

**Example**

*// initialise ArrayList using asList method*

import java.util.\*;

public class ScalerTopics {

public static void main(String args[])

{

*// create an ArrayList of String type*

*// and Initialize an ArrayList with asList()*

ArrayList<String> scaler = new ArrayList<String>(

Arrays.asList("I",

"love",

"Java"));

*// to print the ArrayList*

System.out.println("ArrayList is " + scaler);

}

}

**Output**

ArrayList is [I, Love, Java]

**3. Using list.of() Method** It is used to return immutable lists containing the specified elements.

List<Data-Type> obj = new ArrayList<>(

List.of(Obj 1, Obj 1, Obj 1, ....));

**Example**

*// initialise ArrayList using List.of() method*

import java.util.\*;

public class ScalerTopics {

public static void main(String args[])

{

*// create an ArrayList of String type*

*// and Initialize an ArrayList with List.of()*

List<String> str = new ArrayList<>(

List.of("Coding",

"is",

"Fun"));

*// to print the ArrayList*

System.out.println("ArrayList is " + str);

}

}

**Output**

ArrayList is [Coding, is, Fun]

Now that we are acquainted to the methods to **initialise an ArrayList**, let us look at an example.

import java.util.\*;

public class ScalerTopics{

public static void main(String args[]){

*//creating an ArrayList*

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

*//displaying the initial size*

System.out.println("Size at the beginning "+str.size());

*//add elements*

str.add("Hello");

str.add("Hi");

str.add("Namaste");

str.add("Bonjour");

*//displaying the ArrayList*

System.out.println(str);

*//displaying the size*

System.out.println("Size after addition "+str.size());

*//remove element at index 0*

str.remove(0);

*//display the new ArrayList*

System.out.println(str);

*//display the new size*

System.out.println("Size after removal "+str.size());

}

}

**Output**

Size at the beginning 0

[Hello, Hi, Namaste, Bonjour]

Size after addition 4

[Hi, Namaste, Bonjour]

Size after removal 3

**Explanation**: We saw the ArrayList methods like add(), remove() and size() which perform manipulations on ArrayLists.

**2. LinkedList**

* The LinkedList class extends the AbstractSequentialList and it also extends the **List, Deque and Queue interface**.
* By this, we get a linked-list data structure.
* Linked List is a linear data structure where the elements are called **as nodes**.
* Here, each node has two fields- data and next. Data stores the actual piece of information and **next points to the next node**. 'Next' field is actually the address of the next node.
* Elements are not **stored in a contiguous memory**, so direct access to that element is not possible.
* **LinkedList uses Doubly Linked List** to store its elements while ArrayList internally uses a dynamic array to store its elements. **LinkedList is faster** in manipulation of data as it is node based which makes it unique.
* LinkedList is **non-synchronised** means multiple threads at a time can access the code. This means if one thread is working on LinkedList, other threads can also get a hold of it. Multiple operations on LinkedList can be performed at a time. **For example**, if addition is being performed by one thread, other operation can be **performed by some other thread too**.

**How to create a Linked List?**

There are two ways in which we can create a linked list using constructors.

**1. Creating an empty Linked List**

LinkedList list=new LinkedList();

**2. Creating a Linked List from Collection**

It will create a Linked List with all the elements of Collection C. We will use the LinkedList class and new keyword to create a constructor which contains the **elements of collection**.

LinkedList list=new LinkedList(C);

Let us take a look at an example of Linked List.

import java.util.\*;

public class ScalerTopics{

public static void main(String args[]){

*//creating a LinkedList*

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

*//displaying the initial size*

System.out.println("Size at the beginning "+list.size());

*//add elements*

list.add("Java");

list.add("C++");

list.add("JavaScript");

list.addFirst("C#");

list.addLast("Kotlin");

list.add(2,"Python");

*//displaying the LinkedList*

System.out.println("Original Linked List " + list);

*//displaying the size*

System.out.println("Size after addition "+list.size());

*//remove element at index 5*

list.remove(5);

list.remove("C#");

*//display the new LinkedList*

System.out.println("New Linked List "+ list);

*//display the new size*

System.out.println("Size after removal "+list.size());

}

}

**Output**

Size at the beginning 0

Original Linked List [C#, Java, Python, C++, JavaScript, Kotlin]

Size after addition 6

New Linked List [Java, Python, C++, JavaScript]

Size after removal 4

**Explanation**: In the above code we used LinkedList methods like add(), remove() and size() which perform manipulations on LinkedLists. We are adding, removing elements and then printing their size. **The methods are similar to that of ArrayList**.

**3. Vector**

* Like **ArrayList, Vectors** in Java are used for dynamic arrays.
* It extends the AbstractList and implements the List interface.
* Vector is synchronised. Synchronised means only one thread at a time can access the code. This means if one thread is working on Vector, no other thread can get a hold of it. Only one operation on vector can be performed at a time. **For example**, if addition is being performed by one thread, other operation cannot be performed until the first one is over.
* Before the introduction of the Collection Framework, Vectors were categorised as Legacy Classes**(Classes which were a part of the earlier release of Java but now they are re-constructed)**.

**Methods to create the constructor of Vectors**

**1. Using the default method**

In this method, the size is not specified so the default size of Vectors is 10.

Vector<Data-type> v = new Vector<Data-Type>();

**2. By specifying the desired size**

In this method, we specify the size.

Vector<Data-type> v = new Vector<Data-Type>(int size);

**3. Using size and increment attributes**

This method is used to create a vector whose initial capacity is declared by *size* and the increment is declared by *incr*. Here, **the increment is the value** that specifies the number of elements to allocate each time that vector gets **resized upward**.

Vector<Data-type> v = new Vector<Data-Type>(int size,int incr);

**4. Creating a Vector from Collection**

It is used to **create a vector** which contains all the elements of a collection.

Vector<Data-type> v = new Vector<Data-Type>(Collection C);

**Example**

import java.util.\*;

public class ScalerTopics{

public static void main(String args[]){

*//creating a Vector*

Vector<Integer> v= new Vector<Integer>();

*//displaying the size*

System.out.println("Size at the beginning "+v.size());

*//add elements*

v.add(19);

v.add(88);

v.add(1);

v.add(39);

*//displaying the Vector*

System.out.println(v);

*//displaying the size*

System.out.println("Size after addition "+v.size());

*//remove element at index 3*

v.remove(3);

*//display the new Vector*

System.out.println(v);

*//display the new size*

System.out.println("Size after removal "+v.size());

}

}

**Output**

Size at the beginning 0

[19, 88, 1, 39]

Size after addition 4

[19, 88, 1]

Size after removal 3

**Explanation**: In the above code we used Vector methods like add(), remove() and size() which perform manipulations on Vectors. We are adding, removing elements and then printing their size. The methods are similar to that of **LinkedList and ArrayList**.

**4. Stack**

* Stack class extends the Vector class and it is its subclass.
* It works on the principle of Last-In, First-Out.
* In order to put an object on the top of the stack, we call the push() method.
* To remove and return the top element in the stack, we call pop() method.
* There are other methods like **peek(), search() and empty()** which are used to perform operations on the stack.

**Methods to create the constructor of Stack**

**1. Creating a default stack**

This creates an empty stack.

Stack<Data-Type> stack = new Stack<Data-Type>();

**Example**

import java.util.\*;

public class ScalerTopics{

public static void main(String args[]){

*//creating a Stack*

Stack<Integer> s= new Stack<Integer>();

*//displaying the initial size*

System.out.println("Size at the beginning "+s.size());

*//push elements*

s.push(99);

s.push(28);

s.push(17);

s.push(74);

s.push(1);

*//displaying the Stack*

System.out.println("New Stack" + s);

*//displaying the size*

System.out.println("Size after addition "+s.size());

*//pop the element and display it*

System.out.println("Popped element " + s.pop());

*//display the new Stack*

System.out.println("New Stack after popping"+ s);

*//display the new size*

System.out.println("Size after removal "+s.size());

*//peek method to find the top-most element and display it*

System.out.println("Top-most element " + s.peek());

*//the size remains the same as peek does not remove the element*

System.out.println("Size after Peek "+s.size());

}

}

**Output**

Size at the beginning 0

New Stack[99, 28, 17, 74, 1]

Size after addition 5

Popped element 1

New Stack after popping[99, 28, 17, 74]

Size after removal 4

Top-most element 74

Size after Peek 4

**Explanation**: In the above program saw methods like pop(), push() and peek(). We are adding elements using pop() function, removing the top using the push() function and getting the **top-most** element by the peek() function.

One thing to note is that Stack is thread-safe. It might be overhead in an environment where the thread-safety **concept is not needed**. So, ArrayDeque is preferred.

**Queue Interface**

* The **Queue Interface extends** the Collection interface.
* It uses the principle of *First-In, First-Out* (FIFO).
* A Queue is an ordered list where there is a need to maintain the **order of the elements**.
* It has classes like PriorityQueue and ArrayDeque.
* The most famous implementation is that of PriorityQueue.

Let us comprehend the concept of PriorityQueue.

**PriorityQueue**

* The **PriorityQueue class extends AbstractQueue** and implements the Queue Interface.
* As the name suggests, they follow the principle of priority of the elements.
* We know that we follow First-In,First-Out for queues, but at times, the elements needs to be processed in terms of their priority. This is where the PriorityQueue comes into play.
* It does not allow null values to be stored inside it.
* The add() method is used to add an element while the poll() method is used to remove the top-most element. While, peek() is used to display the top-most element.

**Methods to initialise a PriorityQueue using constructors**

**1. Creating an empty PriorityQueue**

PriorityQueue<Data-Type> pq = new PriorityQueue<Data-Type>();

**2. Creating a PriorityQueue with the specified size**

PriorityQueue<Data-Type> pq = new PriorityQueue<Data-Type>(int size);

**3. Creating a PriorityQueue from Collection**

It will create a PriorityQueue with all the elements of Collection C.

PriorityQueue<Data-Type> pq = new PriorityQueue<Data-Type>(Collection<Data-Type> C);

**Example**

import java.util.\*;

public class ScalerTopics{

public static void main(String args[])

{

*// Creating a priority queue*

PriorityQueue<Integer> pq = new PriorityQueue<Integer>();

*//displaying the initial size*

System.out.println("Size at the beginning "+pq.size());

*// Adding elements using add()*

pq.add(99);

pq.add(18);

pq.add(27);

pq.add(34);

*//displaying the PriorityQueue*

System.out.println("New PriorityQueue" + pq);

*//displaying the size*

System.out.println("Size after addition "+pq.size());

*// Printing the top element of the PriorityQueue*

System.out.println("Top-most element " +pq.peek());

*// Printing the top element and removing it*

System.out.println("Removing " +pq.poll());

*//displaying the PriorityQueue*

System.out.println("New PriorityQueue after removal" + pq);

*//display the new size*

System.out.println("Size after removal "+pq.size());

}

}

**Output**

Size at the beginning 0

New PriorityQueue[18, 34, 27, 99]

Size after addition 4

Top-most element 18

Removing 18

New PriorityQueue after removal[27, 34, 99]

Size after removal 3

**Explanation**: Here we added elements in the PriorityQueue by add() method, removed the **top-most** element by poll() method and got the **top-most** element by peek() method.

**Set Interface**

* The Set interface defines an **unordered collection**.
* It extends the Collection Interface.
* We cannot store **duplicate values** in this.
* The Set Interface is implemented by popular classes like **HashedSet, LinkedHashSet and TreeSet**.

**Method to instantiate the Set Interface**

Set<Data-Type> hs = new HashSet<Data-Type>();

Set<Data-Type> lhs = new LinkedHashSet<Data-Type>();

Set<Data-Type> ts = new TreeSet<Data-Type>();

**HashSet**

* The HashSet class implements the Set Interface.
* It uses a hash table for storage which uses a mechanism called Hashing.
* In hashing, the informational content of a key determines a unique value, called its hash code.
* The hash code is then used as an index, at which the data associated with the key is stored.
* When we insert elements into the HashSet, it is not guaranteed that it gets stored in the same order.
* We can store Null values in this.
* HashSet is **non-synchronized** means multiple threads at a time can access the code. This means if one thread is working on HashSet, other threads can also get a hold of it. Multiple operations on HashSet can be performed at a time.

**For example,** if addition is being performed by one thread, other operation can be performed by some **other thread too**.

**Methods to create the constructors of HashSet**

**1. Creating an empty HashSet** It is used to create an empty HashSet object in which the default initial capacity is 16.

HashSet<Data-type> hs = new HashSet<Data-type>();

**2. Creating a HashSet with a specified size** It is used to create a HashSet with the given size.

HashSet<Data-type> hs = new HashSet<Data-type>(int size);

**3. Creating a HashSet with a specified size and fill ratio** It is used to create a HashSet with a given size and fill ratio.

HashSet<Data-type> hs = new HashSet<Data-type>(int size,float fillRatio);

**4. Creating a HashSet from Collection** It is used to create a HashSet which contains all the elements from the collection.

HashSet<Data-type> hs = new HashSet<Data-type>(Collection C);

**Example**

import java.util.\*;

public class ScalerTopics{

public static void main(String args[]){

*//creating a HashSet*

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

*//displaying the initial size*

System.out.println("Size at the beginning "+str.size());

*//add elements*

str.add("Hello");

str.add("Hi");

str.add("Namaste");

str.add("Bonjour");

*//displaying the HashSet*

System.out.println(str);

*//displaying the size*

System.out.println("Size after addition "+str.size());

*//remove element using value*

str.remove("Bonjour");

*//display the new HashSet*

System.out.println(str);

*//display the new size*

System.out.println("Size after removal "+str.size());

}

}

**Output**

Size at the beginning 0

[Hi, Hello, Namaste, Bonjour]

Size after addition 4

[Hi, Hello, Namaste]

Size after removal 3

Explanation- The above code is an example of HashSet where we have added items using add() function, removed them using remove() function and fetched the size using size() function.

**LinkedHashSet**

* The LinkedHashSet class extends the HashSet class.
* It maintains a linked list of entries in the set and hence maintains the order in which they were inserted.
* LinkedHashSet is non-synchronized means multiple threads at a time can access the code. This means if one thread is working on **LinkedHashSet**, other threads can also get a hold of it. Multiple operations on **LinkedHashSet** can be performed at a time. For example, if addition is being performed by one thread, other operation can be **performed by some other thread** too.

**The figure below shows the illustration of a LinkedHashSet**

**Methods to create the constructors of HashSet**

**1. Creating an empty LinkedHashSet** It is used to create an empty LinkedHashSet object.

LinkedHashSet<Data-type> lhs = new LinkedHashSet<Data-type>();

**2. Creating a LinkedHashSet with a specified size** It is used to create a LinkedHashSet with the given size.

LinkedHashSet<Data-type> lhs = new LinkedHashSet<Data-type>(int size);

**3. Creating a LinkedHashSet with a specified size and fill ratio** It is used to create a LinkedHashSet with a given size and fill ratio. Fill ratio determines how full the hash set can be before it is resized.

LinkedHashSet<Data-type> lhs = new LinkedHashSet<Data-type>(int size,float fillRatio);

**4. Creating a LinkedHashSet from Collection** It is used to create a LinkedHashSet which contains all the elements from the collection.

LinkedHashSet<Data-type> lhs = new LinkedHashSet<Data-type>(Collection C);

**Example**

import java.util.\*;

public class ScalerTopics{

public static void main(String args[]){

*//creating a HashSet*

LinkedHashSet<String> str= new LinkedHashSet<String>();

*//displaying the initial size*

System.out.println("Size at the beginning "+str.size());

*//add elements*

str.add("Hello");

str.add("Hi");

str.add("Namaste");

str.add("Bonjour");

*//displaying the LinkedHashSet*

System.out.println(str);

*//displaying the size*

System.out.println("Size after addition "+str.size());

*//remove element using value*

str.remove("Bonjour");

*//display the new LinkedHashSet*

System.out.println(str);

*//display the new size*

System.out.println("Size after removal "+str.size());

}

}

**Output**

Size at the beginning 0

[Hello, Hi, Namaste, Bonjour]

Size after addition 4

[Hello, Hi, Namaste]

Size after removal 3

Explanation- In the above code, we created a LinkedHashSet and added elements to it using the add() method, remove the elements by remove() method, and get its size by size() method.

**SortedSet Interface**

* The SortedSet Interface extends the Set Interface.
* It is similar to the Set Interface but plays a **vital role to arrange** data in ascending or sorted manner.
* The TreeSet class implements this interface.

**The figure below illustrates the SortedSet Interface**.

**Method to instantiate the SortedSet Interface**

SortedSet<Data-Type> ts = new TreeSet<Data-type>();

**TreeSet**

* The TreeSet class **implements the Set** Interface.
* It uses a tree to store the elements.
* TreeSet contains unique elements.
* The access and retrieval time is very fast.
* TreeSet is **non-synchronized** means multiple threads at a time can access the code. This means if one thread is working on TreeSet, other threads can also get a hold of it. Multiple operations on **TreeSet** can be performed at a time. For example, if addition is being performed by **one thread**, other operations can be performed by some other thread too.

**Example**

import java.util.\*;

class ScalerTopics {

public static void main(String[] args)

{

*// Creating a TreeSet*

Set<String> ts = new TreeSet<>();

*//displaying the initial size*

System.out.println("Size at the beginning "+ts.size());

*// Elements are added using add() method*

ts.add("India");

ts.add("USA");

ts.add("Britain");

*//displaying the TreeSet*

System.out.println(ts);

*//displaying the size*

System.out.println("Size after addition "+ts.size());

*// Duplicates will not get inserted into the TreeSet*

ts.add("Britain");

*// Elements get stored in Ascending order*

System.out.println(ts);

}

}

**Output**

Size at the beginning 0

[Britain, India, USA]

Size after addition 3

[Britain, India, USA]

Explanation- The above program shows how we can create a TreeSet and add elements to it using the add() method and get its size by get() method.

**Map Interface**

* A map is an object that stores key and value pairs.
* It contains unique keys as the same key cannot have multiple mappings.
* Although a part of the Collections Framework, maps are not themselves collections because they do not **implement the Collection Interface**.

**Method to instantiate the Map Interface**

Map<Data-Type> hm = new HashMap<> ();

**HashMap**

* The HashMap class extends **AbstractMap and implements the Map interface**.
* It uses a hash table for storing key-value pairs.
* If we want to access a value in a\*\* hash map\*\*, we must know its key.

**Example**

import java.util.\*;

public class ScalerTopics {

public static void main(String args[])

{

*// Creating a HashMap*

HashMap<Integer, Double> hm = new HashMap<Integer, Double>();

*//adding key value pairs using put()*

hm.put(1, 1.9);

hm.put(2, 2.8);

hm.put(3, 3.7);

*// Finding the value for a key using get()*

System.out.println("The Value for 1 is " + hm.get(1));

}

}

**Output**

The Value for 1 is 1.9

**Explanation**: In the above program, we performed manipulations on HashMap like inserting key-value pairs by put() function and getting the value of a key by using get() method.

**When to use which Collection Framework?**

Now that you are well versed with the **Collections in Java and Collection Framework**, you might be facing the dilemma of when to use which collection in Java. Don't worry, we will clear this confusion and get you set.

**Conclusion**

* A Collection in Java refers to a single unit that contains a group of objects.
* A separate Collection Framework was needed to have a standard implementation for objects.
* The List interface is implemented by **LinkedList, ArrayList, Vectors,** and **Stack classes**.
* The Queue interface is implemented by **PriorityQueue**,
* The Set Interface is implemented by **HashedSet, LinkedHashSet,** and **TreeSet**.
* The Map interface is implemented by **HashMap**.
* ArrayList is good for searching a particular element using **array index**. ArrayLists are dynamic which is advantageous over arrays.
* Linked list is good for insertion and deletion operations.
* Set is used to store **unique elements**.
* Map is used when to store key-value pairs, Hash map is used in Java to build **caching mechanism**