CO3:

Collection Framework- Collection Basics: Iterator Interface, Methods of Collection Interface- List: Array List, Linked List, Vector; Set: Hash Set, Tree Set; Map: Hash Map, Tree Map, Hashtable. Collection Framework Advance Concepts: Comparable Interface, Comparator Interface, Collections class, Differentiate Comparable and Comparator

What are **Java collections?** Java collections refer to a collection of individual objects that are represented as a single unit. You can perform all operations such as searching, sorting, insertion, manipulation, deletion, etc., on Java collections just like you do it on data.

**What is a Java Collection Framework?**

A collections framework is a unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details. A Java collection framework includes the following:

* Interfaces
* Classes
* Algorithm

**Interfaces**: Interface in Java refers to the abstract data types. They allow Java collections to be manipulated independently from the details of their representation. Also, they form a hierarchy in object-oriented programming languages.

**Classes:** Classes in Java are the implementation of the collection interface. It basically refers to the data structures that are used again and again.

**Algorithm:** Algorithm refers to the methods which are used to perform operations such as searching and sorting, on objects that implement collection interfaces. Algorithms are polymorphic in nature as the same method can be used to take many forms or you can say perform different implementations of the Java collection interface.

So why do you think we need Java collections?

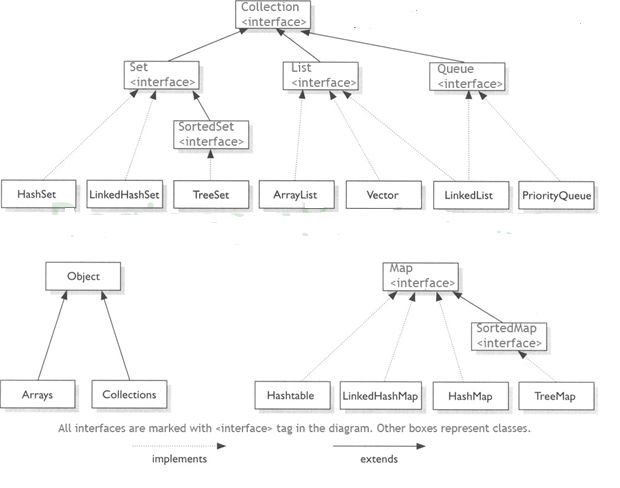
The Java collection framework provides the developers to access pre-packaged data structures as well as algorithms to manipulate data.

The primary advantages of a collection framework are that it:

* **Reduces programming effort** by providing data structures and algorithms so you don't have to write them yourself.
* **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.
* **Provides interoperability between unrelated APIs** by establishing a common language to pass collections back and forth.
* **Reduces the effort required to learn APIs** by requiring you to learn multiple ad hoc collection APIs.
* **Reduces the effort required to design and implement APIs** by not requiring you to produce ad hoc collections APIs.
* **Fosters software reuse** by providing a standard interface for collections and algorithms with which to manipulate them.

The collections framework consists of:

* **Collection interfaces**. Represent different types of collections, such as sets, lists, and maps. These interfaces form the basis of the framework.
* **General-purpose implementations**. Primary implementations of the collection interfaces.
* **Legacy implementations**. The collection classes from earlier releases, Vector and Hashtable, were retrofitted to implement the collection interfaces.
* **Special-purpose implementations**. Implementations designed for use in special situations. These implementations display nonstandard performance characteristics, usage restrictions, or behavior.
* **Concurrent implementations**. Implementations designed for highly concurrent use.
* **Wrapper implementations**. Add functionality, such as synchronization, to other implementations.
* **Convenience implementations**. High-performance "mini-implementations" of the collection interfaces.
* **Abstract implementations**. Partial implementations of the collection interfaces to facilitate custom implementations.
* **Algorithms**. Static methods that perform useful functions on collections, such as sorting a list.
* **Infrastructure**. Interfaces that provide essential support for the collection interfaces.
* **Array Utilities**. Utility functions for arrays of primitive types and reference objects. Not, strictly speaking, a part of the collections framework, this feature was added to the Java platform at the same time as the collections framework and relies on some of the same infrastructure.
* **Collections Framework hierarchy**



**Iterator interface:**

In Java, an **Iterator** is one of the Java cursors. **Java Iterator** is an interface that is practiced in order to iterate over a collection of Java object components entirety one by one. It is free to use in the Java programming language since the Java 1.2 Collection framework. It belongs to java.util package.

Though Java Iterator was introduced in Java 1.2, however, it is still not the oldest tool available to traverse through the elements of the Collection object. The oldest Iterator in the Java programming language is the Enumerator predated Iterator. Java Iterator interface succeeds the enumerator iterator that was practiced in the beginning to traverse over some accessible collections like the ArrayLists.

The Java Iterator is also known as the **universal cursor** of Java as it is appropriate for all the classes of the Collection framework. The Java Iterator also helps in the operations like READ and REMOVE. When we compare the Java Iterator interface with the enumeration iterator interface, we can say that the names of the methods available in Java Iterator are more precise and straightforward to use.

#### **METHODS OF ITERATOR INTERFACE**

There are only three methods in the Iterator interface. They are:

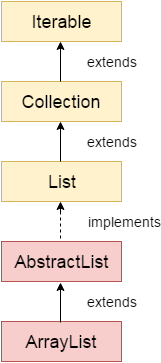
|  |  |  |
| --- | --- | --- |
| No. | Method | Description |
| 1 | public boolean hasNext() | It returns true if the iterator has more elements otherwise it returns false. |
| 2 | public Object next() | It returns the element and moves the cursor pointer to the next element. |
| 3 | public void remove() | It removes the last elements returned by the iterator. It is less used. |

## Methods of the Collection Interface

This interface contains various methods which can be directly used by all the collections which implement this interface. They are:

|  |  |
| --- | --- |
| **Method** | **Description** |
| [**add(Object)**](https://www.geeksforgeeks.org/collection-add-method-in-java-with-examples/) | This method is used to add an object to the collection. |
| [**addAll(Collection c)**](https://www.geeksforgeeks.org/collections-addall-method-in-java-with-examples/) | This method adds all the elements in the given collection to this collection. |
| [**clear()**](https://www.geeksforgeeks.org/collection-clear-method-in-java-with-examples/) | This method removes all of the elements from this collection. |
| [**contains(Object o)**](https://www.geeksforgeeks.org/collection-contains-method-in-java-with-examples/) | This method returns true if the collection contains the specified element. |
| **containsAll(Collection c)** | This method returns true if the collection contains all of the elements in the given collection. |
| **equals(Object o)** | This method compares the specified object with this collection for equality. |
| **hashCode()** | This method is used to return the hash code value for this collection. |
| [**isEmpty()**](https://www.geeksforgeeks.org/collection-isempty-method-in-java-with-examples/) | This method returns true if this collection contains no elements. |
| **iterator()** | This method returns an iterator over the elements in this collection. |
| [**max()**](https://www.geeksforgeeks.org/collections-max-method-in-java-with-examples/) | This method is used to return the maximum value present in the collection. |
| **parallelStream()** | This method returns a parallel Stream with this collection as its source. |
| **remove(Object o)** | This method is used to remove the given object from the collection. If there are duplicate values, then this method removes the first occurrence of the object. |
| **removeAll(Collection c)** | This method is used to remove all the objects mentioned in the given collection from the collection. |
| **removeIf(Predicate filter)** | This method is used to remove all the elements of this collection that satisfy the given [predicate](https://www.geeksforgeeks.org/mathematic-logic-predicates-quantifiers/). |
| **retainAll(Collection c)** | This method is used to retain only the elements in this collection that are contained in the specified collection. |
| **size()** | This method is used to return the number of elements in the collection. |
| **spliterator()** | This method is used to create a [Spliterator](https://www.geeksforgeeks.org/java-program-to-convert-iterator-to-spliterator/) over the elements in this collection. |
| **stream()** | This method is used to return a sequential Stream with this collection as its source. |
| **toArray()** | This method is used to return an array containing all of the elements in this collection. |

# **Array List**



ArrayList class uses a dynamic array for storing the elements. It is like an array, but there is no size limit. We can add or remove elements anytime. So, it is much more flexible than the traditional array. It is found in the java.util package. It is like the Vector in C++. In Java, we use the ArrayList class to implement the functionality of resizable-arrays. It implements the List interface of the collections framework.

The ArrayList in Java can have the duplicate elements also. It implements the List interface so we can use all the methods of the List interface here. The ArrayList maintains the insertion order internally.

**Java ArrayList Vs Array**

In Java, we need to declare the size of an array before we can use it. Once the size of an array is declared, it's hard to change it.

To handle this issue, we can use the ArrayList class. It allows us to create resizable arrays.

Unlike arrays, arraylists can automatically adjust their capacity when we add or remove elements from them. Hence, arraylists are also known as dynamic arrays.

It inherits the AbstractList class and implements List interface.

The important points about the Java ArrayList class are:

ArrayList class can contain duplicate elements.

ArrayList class maintains insertion order.

ArrayList class is non synchronized.

ArrayList allows random access because the array works on an index basis.

**Creating an ArrayList**

Before using ArrayList, we need to import the java.util.ArrayList package first. Here is how we can create arraylists in Java:

ArrayList<Type> arrayList= new ArrayList<>();

Here, Type indicates the type of an arraylist. For example,

// create Integer type arraylist

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

// create String type arraylist

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

We can not create an array list of the primitive types, such as int, float, char, etc. It is required to use the required wrapper class in such cases. For example:

ArrayList<int> al = ArrayList<int>(); // it will not work

Correct Syntax:: ArrayList<Integer> al = new ArrayList<Integer>();

Java ArrayList gets initialized by the size. The size is dynamic in the array list, which varies according to the elements getting added or removed from the list.

**Java ArrayList Example**

import java.util.\*;

public class ArrayListExample1{

public static void main(String args[]){

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

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

list.add("Apple");

list.add("Banana");

list.add("Grapes");

//Printing the arraylist object

System.out.println(list);

}

}

**Iterating ArrayList using Iterator**

import java.util.\*;

public class ArrayListExample2{

public static void main(String args[]){

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

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

list.add("Apple");

list.add("Banana");

list.add("Grapes");

//Traversing list through Iterator

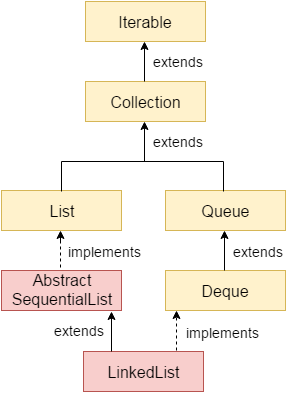
Iterator itr=list.iterator();//getting the Iterator

while(itr.hasNext()){//check if iterator has the elements

System.out.println(itr.next());//printing the element and move to next

} } }

# Java LinkedList class



Java LinkedList class uses a doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

The important points about Java LinkedList are:

* Java LinkedList class can contain duplicate elements.
* Java LinkedList class maintains insertion order.
* Java LinkedList class is non synchronized.
* In Java LinkedList class, manipulation is fast because no shifting needs to occur.
* Java LinkedList class can be used as a list, stack or queue.

### Hierarchy of LinkedList class

As shown in the above diagram, Java LinkedList class extends AbstractSequentialList class and implements List and Deque interfaces.

### Doubly Linked List

In the case of a doubly linked list, we can add or remove elements from both sides.



### LinkedList class declaration

Let's see the declaration for java.util.LinkedList class.

1. public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>, Deque<E>, Cloneable, Serializable

Java LinkedList Example

import java.util.\*;

public class LinkedList1{

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());

}

}

}

**Vector:**

Vectors are similar to arrays, where the elements of the vector object can be accessed via an index into the vector. Vector implements a dynamic array. Also, the vector is not limited to a specific size, it can shrink or grow automatically whenever required. It is similar to ArrayList, but with two differences :

* Vector is synchronized.
* Vector contains many legacy methods that are not part of the collections framework.

**Syntax**:

* Vector object = new Vector (size, increment);

// Java program to demonstrate the working of Vector

import java.io.\*;

import java.util.\*;

class GFG {

// Main Method

public static void main(String[] args)

{

// Declaring the Vector

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

// Appending new elements at

// the end of the list

for (int i = 1; i <= 5; i++)

v.add(i);

// Printing elements

System.out.println(v);

// Remove element at index 3

v.remove(3);

// Displaying the Vector

// after deletion

System.out.println(v);

// Printing elements one by one

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

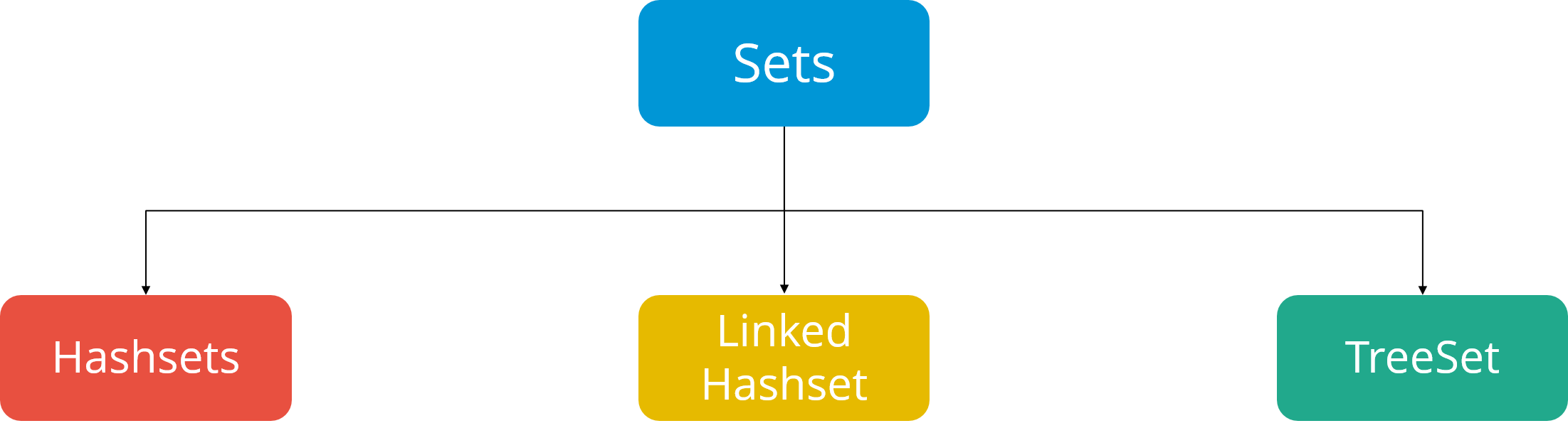
System.out.print(v.get(i) + " ");

}

}

Java Collections: Sets

A Set refers to a collection that cannot contain duplicate elements. It is mainly used to model the mathematical set abstraction. Set has its implementation in various classes such as HashSet, TreeSetand LinkedHashSet.



**HashSet**: Java HashSet class creates a collection that use a hash table for storage. Hashset only contain unique elements and it inherits the AbstractSet class and implements Set interface. Also, it uses a mechanism hashing to store the elements.

HashSet class declaration

Let's see the declaration for java.util.HashSet class.

public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, Serializable

The important points about Java HashSet class are:

* HashSet stores the elements by using a mechanism called **hashing.**
* HashSet contains unique elements only.
* HashSet allows null value.
* HashSet class is non synchronized.
* HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
* HashSet is the best approach for search operations.
* The initial default capacity of HashSet is 16, and the load factor is 0.75.

**Example:**

import java.util.\*;

class HashsetExample{

public static void main(String args[]){

HashSet&amp;amp;lt;String&amp;amp;gt; al=new HashSet(); // creating hashSet

al.add("Rachit"); // adding elements

al.add("Amit");

al.add("jack");

Iterator&amp;amp;lt;String&amp;amp;gt; itr=al.iterator();

while(itr.hasNext()){

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

}

}

}

TreeSet : TreeSet class implements the Set interface that uses a tree for storage. The objects of this class are stored in the ascending order. Also, it inherits AbstractSet class and implements NavigableSet interface. It contains only unique elements like HashSet. In TreeSet class, access and retrieval time are faster.

import java.util.\*;

class TreeSetExample{

public static void main(String args[]){

TreeSet&amp;amp;lt;String&amp;amp;gt; al=new TreeSet&amp;amp;lt;String&amp;amp;gt;();  // creating treeSet

al.add("John");                            // adding elements

al.add("Sam");

al.add("Rick");

Iterator&amp;amp;lt;String&amp;amp;gt; itr=al.iterator();

while(itr.hasNext()){

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

}

}

}

Now you must be wondering what is the difference between all these sets?

HashSet stores elements in random order whereas TreeHashSet stores according to natural ordering.

**Java TreeSet class**

Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order.

The important points about Java TreeSet class are:

Java TreeSet class contains unique elements only like HashSet.

Java TreeSet class access and retrieval times are quiet fast.

Java TreeSet class doesn't allow null element.

Java TreeSet class is non synchronized.

Java TreeSet class maintains ascending order.

### TreeSet class declaration

Let's see the declaration for java.util.TreeSet class.

public class TreeSet<E> extends AbstractSet<E> implements NavigableSet<E>, Cloneable, Serializable

Java TreeSet Examples

Java TreeSet Example 1:

Let's see a simple example of Java TreeSet.

import java.util.\*;

class TreeSet1{

public static void main(String args[]){

//Creating and adding elements

TreeSet<String> al=new TreeSet<String>();

al.add("Ravi");

al.add("Vijay");

al.add("Ravi");

al.add("Ajay");

//Traversing elements

Iterator<String> itr=al.iterator();

while(itr.hasNext()){

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

}

}

}

**Map Interface**

The Map interface is inherited from Java Collection Interface. The Map cannot store duplicate elements. A Map stores data using the key-value pair format. The manipulation operations are taken care of through accessing the key-value pairs.

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys.

A Map is useful if you have to search, update or delete elements on the basis of a key. This map interface is implemented by various classes like HashMap, TreeMap, etc. Since all the subclasses implement the map, we can instantiate a map object with any of these classes. A Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.

For example,

Map<T> hm = new HashMap<> ();

Map<T> tm = new TreeMap<> ();

Where T is the type of the object.

A Map can't be traversed, so you need to convert it into Set using keySet() or entrySet() method.

Example:

//Map Interface

package Simplilearn;

import java.util.\*;

import java.util.Map.Entry;

public class MapInterface {

public static void main(String args[]) {

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

map.put(1, "Cricket");

map.put(2, "Hockey");

map.put(3, "Archery");

for (Iterator<Entry<Integer, String>> iterator = map.entrySet().iterator(); iterator.hasNext();) {

Entry<Integer, String> m = iterator.next();

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

}

}

}

**HashMap:**

HashMap provides the basic implementation of the Map interface of Java. It stores the data in (Key, Value) pairs. To access a value in a HashMap, we must know its key. HashMap uses a technique called Hashing. Hashing is a technique of converting a large String to a small String that represents the same String so that the indexing and search operations are faster. HashSet also uses HashMap internally.

If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the java.util package.

HashMap in Java is like the legacy Hashtable class, but it is not synchronized. It allows us to store the null elements as well, but there should be only one null key. Since Java 5, it is denoted as HashMap<K,V>, where K stands for key and V for value. It inherits the AbstractMap class and implements the Map interface.

* Java HashMap contains values based on the key.
* Java HashMap contains only unique keys.
* Java HashMap may have one null key and multiple null values.
* Java HashMap is non synchronized.
* Java HashMap maintains no order.
* The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

HashMap class declaration

Let's see the declaration for java.util.HashMap class.

public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Cloneable, Serializable

HashMap class Parameters

Let's see the Parameters for java.util.HashMap class.

K: It is the type of keys maintained by this map.

V: It is the type of mapped values.

// Java program to demonstrate the

// working of a HashMap

import java.util.\*;

public class HashMapDemo {

// Main Method

public static void main(String args[])

{

// Creating HashMap and

// adding elements

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

hm.put(1, "Java");

hm.put(2, "is a");

hm.put(3, "Programming Language");

// Finding the value for a key

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

// Traversing through the HashMap

for (Map.Entry<Integer, String> e : hm.entrySet())

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

}

}

# Java TreeMap class

Java TreeMap class is a red-black tree based implementation. It provides an efficient means of storing key-value pairs in sorted order.

The important points about Java TreeMap class are:

* Java TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
* Java TreeMap contains only unique elements.
* Java TreeMap cannot have a null key but can have multiple null values.
* Java TreeMap is non synchronized.
* Java TreeMap maintains ascending order.

### TreeMap class declaration

Let's see the declaration for java.util.TreeMap class.

1. public class TreeMap<K,V> extends AbstractMap<K,V> implements NavigableMap<K,V>, Cloneable, Serializable

### TreeMap class Parameters

Let's see the Parameters for java.util.TreeMap class.

* **K**: It is the type of keys maintained by this map.
* **V**: It is the type of mapped values.

Java TreeMap Example

import java.util.\*;

class TreeMap1{

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");

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

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

}

}

}

**Hashtable in Java**

The Hashtable class implements a hash table, which maps keys to values. Any non-null object can be used as a key or as a value. To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method.

**Features of Hashtable**

It is similar to HashMap, but is synchronized.

Hashtable stores key/value pair in hash table.

In Hashtable we specify an object that is used as a key, and the value we want to associate to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.

HashMap doesn’t provide any Enumeration, while Hashtable provides not fail-fast Enumeration.

**Declaration:**

public class Hashtable<K,V> extends Dictionary<K,V> implements Map<K,V>, Cloneable, Serializable

Type Parameters:

K – the type of keys maintained by this map

V – the type of mapped values

Java Hashtable Example

import java.util.\*;

class Hashtable1{

public static void main(String args[]){

Hashtable<Integer,String> hm=new Hashtable<Integer,String>();

hm.put(100,"Amit");

hm.put(102,"Ravi");

hm.put(101,"Vijay");

hm.put(103,"Rahul");

for(Map.Entry m:hm.entrySet()){

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

}

}

}

# Difference between HashMap and Hashtable

HashMap and Hashtable both are used to store data in key and value form. Both are using hashing technique to store unique keys.

But there are many differences between HashMap and Hashtable classes that are given below.

|  |  |
| --- | --- |
| **HashMap** | **Hashtable** |
| 1) HashMap is **non synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code. | Hashtable is **synchronized**. It is thread-safe and can be shared with many threads. |
| 2) HashMap **allows one null key and multiple null values**. | Hashtable **doesn't allow any null key or value**. |
| 3) HashMap is a **new class introduced in JDK 1.2**. | Hashtable is a **legacy class**. |
| 4) HashMap is **fast**. | Hashtable is **slow**. |
| 5) We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap); | Hashtable is internally synchronized and can't be unsynchronized. |
| 6) HashMap is **traversed by Iterator**. | Hashtable is **traversed by Enumerator and Iterator**. |
| 7) Iterator in HashMap is **fail-fast**. | Enumerator in Hashtable is **not fail-fast**. |
| 8) HashMap inherits **AbstractMap** class. | Hashtable inherits **Dictionary** class. |

// Java program to demonstrate

// adding elements to Hashtable

import java.io.\*;

import java.util.\*;

class AddElementsToHashtable {

public static void main(String args[])

{

// No need to mention the

// Generic type twice

Hashtable<Integer, String> ht1 = new Hashtable<>();

// Initialization of a Hashtable

// using Generics

Hashtable<Integer, String> ht2

= new Hashtable<Integer, String>();

// Inserting the Elements

// using put() method

ht1.put(1, "one");

ht1.put(2, "two");

ht1.put(3, "three");

ht2.put(4, "four");

ht2.put(5, "five");

ht2.put(6, "six");

// Print mappings to the console

System.out.println("Mappings of ht1 : " + ht1);

System.out.println("Mappings of ht2 : " + ht2);

}

}

**Java Comparable interface**

Java Comparable interface is used to order the objects of the user-defined class. This interface is found in java.lang package and contains only one method named compareTo(Object). It provides a single sorting sequence only, i.e., you can sort the elements on the basis of single data member only. For example, it may be rollno, name, age or anything else.

public int compareTo(Object obj): It is used to compare the current object with the specified object. It returns

* positive integer, if the current object is greater than the specified object.
* negative integer, if the current object is less than the specified object.
* zero, if the current object is equal to the specified object.

We can sort the elements of:

* String objects
* Wrapper class objects
* User-defined class objects

**Collections class**

Collections class provides static methods for sorting the elements of collections. If collection elements are of Set or Map, we can use TreeSet or TreeMap. However, we cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements.

**Method of Collections class for sorting List elements**

public void sort(List list): It is used to sort the elements of List. List elements must be of the Comparable type.

**Java Comparable Example**

Let's see the example of the Comparable interface that sorts the list elements on the basis of age.

**File: Student.java**

class Student implements Comparable<Student>{

int rollno;

String name;

int age;

Student(int rollno,String name,int age){

this.rollno=rollno;

this.name=name;

this.age=age;

}

public int compareTo(Student st){

if(age==st.age)

return 0;

else if(age>st.age)

return 1;

else

return -1;

}

}

**File: TestSort1.java**

import java.util.\*;

public class TestSort1{

public static void main(String args[]){

ArrayList<Student> al=new ArrayList<Student>();

al.add(new Student(101,"Vijay",23));

al.add(new Student(106,"Ajay",27));

al.add(new Student(105,"Jai",21));

Collections.sort(al);

for(Student st:al){

System.out.println(st.rollno+" "+st.name+" "+st.age);

}

}

}

# **Java Comparator interface**

**Java Comparator interface** is used to order the objects of a user-defined class.

This interface is found in java.util package and contains 2 methods compare(Object obj1,Object obj2) and equals(Object element).

It provides multiple sorting sequences, i.e., you can sort the elements on the basis of any data member, for example, rollno, name, age or anything else.

**Collections class**

Collections class provides static methods for sorting the elements of a collection. If collection elements are of Set or Map, we can use TreeSet or TreeMap. However, we cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements also.

**Method of Collections class for sorting List elements**

public void sort(List list, Comparator c): is used to sort the elements of List by the given Comparator.

Java Comparator Example (Non-generic Old Style)

Let's see the example of sorting the elements of List on the basis of age and name. In this example, we have created 4 java classes:

Student.java

AgeComparator.java

NameComparator.java

Simple.java

Student.java

This class contains three fields rollno, name and age and a parameterized constructor.

class Student{

int rollno;

String name;

int age;

Student(int rollno,String name,int age){

this.rollno=rollno;

this.name=name;

this.age=age;

}

}

AgeComparator.java

This class defines comparison logic based on the age. If the age of the first object is greater than the second, we are returning a positive value. It can be anyone such as 1, 2, 10. If the age of the first object is less than the second object, we are returning a negative value, it can be any negative value, and if the age of both objects is equal, we are returning 0.

import java.util.\*;

class AgeComparator implements Comparator{

public int compare(Object o1,Object o2){

Student s1=(Student)o1;

Student s2=(Student)o2;

if(s1.age==s2.age)

return 0;

else if(s1.age>s2.age)

return 1;

else

return -1;

}

}

NameComparator.java

This class provides comparison logic based on the name. In such case, we are using the compareTo() method of String class, which internally provides the comparison logic.

import java.util.\*;

class NameComparator implements Comparator{

public int compare(Object o1,Object o2){

Student s1=(Student)o1;

Student s2=(Student)o2;

return s1.name.compareTo(s2.name);

}

}

Simple.java

In this class, we are printing the values of the object by sorting on the basis of name and age.

import java.util.\*;

import java.io.\*;

class Simple{

public static void main(String args[]){

ArrayList al=new ArrayList();

al.add(new Student(101,"Vijay",23));

al.add(new Student(106,"Ajay",27));

al.add(new Student(105,"Jai",21));

System.out.println("Sorting by Name");

Collections.sort(al,new NameComparator());

Iterator itr=al.iterator();

while(itr.hasNext()){

Student st=(Student)itr.next();

System.out.println(st.rollno+" "+st.name+" "+st.age);

}

System.out.println("Sorting by age");

Collections.sort(al,new AgeComparator());

Iterator itr2=al.iterator();

while(itr2.hasNext()){

Student st=(Student)itr2.next();

System.out.println(st.rollno+" "+st.name+" "+st.age);

}

}

}

**Java Collections class**

Java collection class is used exclusively with static methods that operate on or return collections. It inherits Object class.

The important points about Java Collections class are:

Java Collection class supports the polymorphic algorithms that operate on collections.

Java Collection class throws a NullPointerException if the collections or class objects provided to them are null.

Collections class declaration

Let's see the declaration for java.util.Collections class.

public class Collections extends Object

**Java Collections Example**

import java.util.\*;

public class CollectionsExample {

public static void main(String a[]){

List<String> list = new ArrayList<String>();

list.add("C");

list.add("Core Java");

list.add("Advance Java");

System.out.println("Initial collection value:"+list);

Collections.addAll(list, "Servlet","JSP");

System.out.println("After adding elements collection value:"+list);

String[] strArr = {"C#", ".Net"};

Collections.addAll(list, strArr);

System.out.println("After adding array collection value:"+list);

}

}

# **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. |