**File Handling:**

1.create folder

2.Delete folder

3.Create new file

4.delete file

5.write data into file

6.read data from file

**Collection:**

**What is collection in java:**

* Collection represents a single unit of objects. (Group of objects).
* collection in java is a framework that provides an architecture to store and manipulate the group of objects.
* All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion, etc.…can be achieved by Java Collections.
* Collection framework is a readymade utility which contains classes & interfaces.
* All the classes & interfaces related to collection framework are present in the ‘java.util’ package.
* In collection framework we can interact only with non-primitive datatypes

We have 2 major interfaces in collection framework

**1.Collection**

**2.Map:** Map interface is not child interface of Collection and hence we can't apply Collection interface methods here.

There are 4 types of data structures:

**1.List:** It is index based. It can take duplicate values

**2. Queue:** It implements FIFO, Queue allows duplicate values

**3.Set:** It is non-index based. It is a group of unique values. It does not allow duplicate values

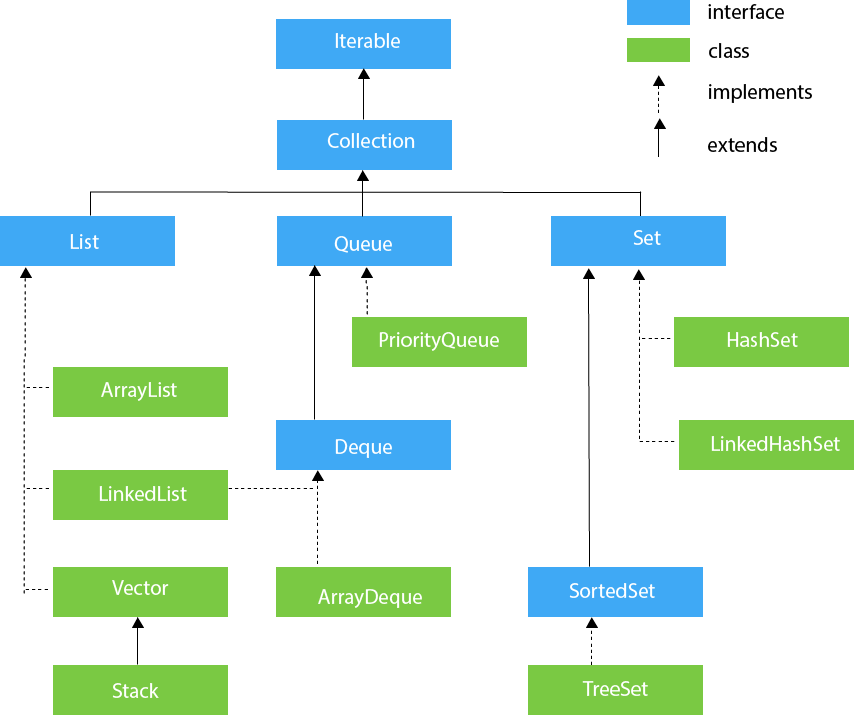
**4.Map:** It stores values with respect to a key. It allows duplicate values

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. Map contains only unique keys.

Map is useful if you have to search, update or delete elements on the basis of key.

**Hierarchy of Collection Framework**

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.



**The hierarchy of Java Map Interface**

Java Map Hierarchy

**Iterator and ListIterator interface:**

**Iterator Interface:**

It is an ‘interface’ present in ‘java.lang’ package.

It is used to retrieve the objects from collection (Set, List & Queue) in forward direction only.

It has 3-methods.

|  |
| --- |
| interface Iterator  {  boolean hasNext( );  Object next( );  void remove( );  } |

**boolean hasNext()** 🡪 It returns true if the iterator has more elements otherwise it returns false.

**Object next()** 🡪 It returns the element and moves the cursor pointer to the next element.

**void remove ()** 🡪 It removes the last elements returned by the iterator.

We get the ‘Iterator’ type object by calling ‘iterator ()’ method.

|  |
| --- |
| Iterator it=al.iterator();  while(it.hasNext())  {  Object obj = it.next();  System.out.println(obj);  } |

**ListIterator Interface:**

It is an ‘interface’ present in ‘java.lang’ package.

It is used to retrieve the objects from ‘List’ in both forward & reverse direction.

It has 5-methods.

|  |
| --- |
| interface ListIterator  {  boolean hasNext( );  Object next( );  void remove( );  boolean hasPrevious( );  Object previous( );  } |

We get ‘ListIterator’ type object by calling ‘listIterator()’ method.

|  |
| --- |
| ListIterator lit=al.listIterator();  //Output in forward direction  while(lit.hasNext())  {  Object obj = lit.next();  System.out.println(obj);  }  //Output in reverse direction  while(lit.hasPrevious())  {  Object obj=lit.previous();  System.out.println(obj);  } |

**Difference between Iterator & ListIterator:**

|  |  |
| --- | --- |
| **Iterator** | **ListIterator** |
| It is unidirectional.(Only Forward).  It is used to retrieve the objects from collection.(List, Set & Queue).  We get iterator type object by ‘iterator ()’ method.  It has methods i.e. hasNext (), next () & remove (). | It is bidirectional. (Forward & Reverse).  It is used to retrieve the objects from List.  We get list iterator object by ‘listIterator()’ method.  It has methods i.e. hasNext (), next (), remove (), hasPrevious () & previous (). |

### **Some Important Methods of Collection interface:**

1. **boolean add(Object r):**

This method adds the given object into last index & returns true if object gets added else it returns false.

1. **void add(int Index, Object r)**

This method adds the given object into given index. This method throws ‘IndexOutOfBounds’ exception if we pass a non-existing index.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 1.5 | 2.2 | 3.5 | 4.5 | 5.5 |   **al** 🡪   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 0 | 1 | **2** | 3 | 4 | 5 | | 1.5 | 2.2 | **6.5** | 3.5 | 4.5 | 5.5 |   **al.add(2, 6.5);** 🡪 |

1. **object set(int Index, Object r):**

This method will set a new value of object into given index by removing the existing object from the given index.

This method returns the object which is removed.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | 4 | | 1.5 | 2.2 | 3.5 | 4.5 | 5.5 |   al 🡪   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0 | 1 | **2** | 3 | 4 | | 1.5 | 2.2 | **6.5** | 4.5 | 5.5 |   **al.set(2, 6.5);** 🡪 |

1. **object remove(int Index):**

This method will remove the object from given index & it will return the removed object.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 0 |  | 1 | 2 | 3 | 4 | | 1.5 |  | 2.2 | 3.5 | 4.5 | 5.5 |   al 🡪   |  |  |  |  | | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | | 1.5 | 2.2 | 4.5 | 5.5 |   **al.remove(2);** 🡪 |

1. **boolean remove(Object r):**

This method removes the give object from the collection & returns true if the object gets removed else returns false.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 0 |  | 1 | 2 | 3 | 4 | | 1.5 |  | 2.2 | 3.5 | 4.5 | 5.5 |   al 🡪   |  |  |  |  | | --- | --- | --- | --- | | 0 | 1 | 2 | 3 | | 1.5 | 2.2 | 4.5 | 5.5 |   **al.remove(3.5);** 🡪 |

1. **boolean contains(Object r):**

This method returns true if the given object exists in the collection else it returns false.

1. **void addAll(collection r):**

This method will add the object of one collection into another.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  |  | 0 | 1 | 2 | 3 | 4 |  |  |  |  |  | | al 🡪 | | 1.5 | 2.2 | **6.5** | 3.5 | 4.5 |  |  |  |  |  | |  |  |  |  |  |  |  |  |  |  |  |  | |  |  | 0 | 1 | 2 | 3 | 4 |  |  |  |  |  | | bl 🡪 | | 50.5 | 60.5 | 70.5 | 80.5 | 90.5 |  |  |  |  |  | |  |  |  |  |  |  |  |  |  |  |  |  | |  |  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | **al.addAll(bl);** 🡪 | | 1.5 | 2.2 | **6.5** | 3.5 | 4.5 | 50.5 | 60.5 | 70.5 | 80.5 | 90.5 | |

1. **boolean containAll(Collection r):**

This method checks the all the objects of one collection present in another collection. It returns true of they are present else it returns false.

1. **boolean removeAll(Collection r):**

This method removes all the objects of one collection from another collection.

1. **boolean isEmpty():**

This method returns true if the collection is empty else it returns false.

1. **void clear():**

This method clears the entire collection by removing all the objects.

* Use “**Collections.sort (collection\_Name)**;” to sort in ascending order.

Ex: - Collections.sort (al); //Here “Collections” is a class

* Use “**Collections.sort (collection\_name, Collections.reverseOrder ( ))**” to sort in descending order.

Ex: - Collections.sort (al, Collections.reverseOrder());

**Difference between List &Set:**

|  |  |
| --- | --- |
| List | Set |
| 1. List gives ordered output. 2. It is index based. 3. It accepts duplicates. 4. It receives one or more null values. 5. It supports for, for-each, Iterator & ListIterator. | 1. Set gives un-ordered output. 2. It is non-index based. 3. No duplicates allowed. 4. It receives one or no null values. 5. It supports for-each & Iterators only. |

List

1. ArrayList
2. LinkedList
3. Vector

Set:

1. HashSet
2. LinkedHashSet
3. TreeSet

ArrayList:

* Java ArrayList class uses a dynamic array for storing the elements
* Java ArrayList class maintains insertion order
* It is non-legacy class. (Early versions of java did not include Collections framework. Instead it defined several classes and one interface to store objects. When collection came these classes reengineered to support the Collection interfaces. These old classes are known are legacy classes. Vector is one of them.)
* It is fast in retrieving objects as objects are sitting next to each other.
* Addition & deletion of objects is slow in ArrayList as numbers of swaps are more.
* ArrayList grows by half of its initial size.
* It does not have any additional methods.
* ArrayList class methods are ‘non-Synchronized’.

**Java Non-generic Vs Generic Collection:**

Java collection framework was non-generic before JDK 1.5. Since 1.5, it is generic.

Java new generic collection allows you to have only one type of object in collection. Now it is type safe so typecasting is not required at run time.

Below is the old non-generic example of creating java collection.

1. ArrayList al=new ArrayList();//creating old non-generic arraylist

Below is the new generic example of creating java collection.

1. ArrayList<String> al=new ArrayList<String>();//creating new generic arraylist

In generic collection, we specify the type in angular braces. Now ArrayList is forced to have only specified type of objects in it. If you try to add another type of object, it gives compile time error.

**LinkedList:**

* Java LinkedList class uses doubly linked list to store the elements
* Java ArrayList class maintains insertion order
* It is non-legacy class.
* It is slow in retrieving objects as objects are spreader out.
* Addition & deletion of object is fast in LinkedList as numbers of swaps are less.
* LinkedList grows by one of its initial size.
* It has some additional methods.
* LinkedList class methods are not ‘Synchronized’.

**Vector:**

* It is **legacy** class.
* Vector grows by **double of its initial size**.
* Vector class methods are ‘**Synchronized**’.

**Difference between ArrayList & LinkedList:**

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1. ArrayList is fast in retrieving objects because objects are sitting next to each other. 2. Addition & deletion of objects is slow in ArrayList because number of swaps are more. 3. ArrayList consumes **more** memory because it grows by half of its initial size. 4. It does not have any additional methods. | 1. LinkedList is slow in retrieving objects because objects are spreader out. 2. Addition & deletion of objects is fast in LinkedList because number of swaps are less. 3. It consumes **less** memory compared to ArrayList because it grows by one. 4. It has few additional methods like addFirst(), addLast(), getFirst(), getLast(), removeFirst(), removeLast()…etc. |

**Difference between ArrayList & Vector:**

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| 1. It is non-legacy class. 2. It consumes **less** memory compared to Vector because it grows by half of its initial size. 3. ArrayList class methods are non-synchronized. | 1. It is legacy class. 2. It consumes **more** memory because it grows by double of its initial size. 3. Vector class methods are synchronized. |

**Set:** It is non-index based.It is a group of unique values. It does not allow duplicate values

Since **Set interface** doesn't provide a **get()** method to retrieve elements, the only way to take out elements from a Set is to iterate over it by using **Iterator**, or loop over Set **using advanced for loop.** Hence it is non-index based.

**HashSet:**

* It gives **un-ordered** output.
* It receives any type of objects.
* HashSet class methods are ‘**Synchronized**’.
* It can receive only **one** **null** value.

**LinkedHashSet:**

* It gives **ordered** output.
* It receives any type of objects.
* LinkedHashSet class methods are **not ‘Synchronized’**.
* It can receive only **one** **null** value.

**TreeSet:**

* It gives **sorted** output.
* It can receive only one **unique** object.(same type of objects)
* TreeSet class methods are **not ‘Synchronized’**.
* It **cannot** have **null** value.

**Difference between HashSet & LinkedHashSet**:

|  |  |
| --- | --- |
| **HashSet** | **LinkedHashSet** |
| 1. It gives **un-ordered** output. 2. HashSet class methods are **synchronized**. | 1. It gives **ordered** output. 2. LinkedHashSet class methods are **not-synchronized.** |

Difference between TreeSet & LinkedHashSet:

|  |  |
| --- | --- |
| **TreeSet** | **LinkedHashSet** |
| 1. It gives **sorted** output. 2. It can receive only **unique objects**. 3. It **cannot** have **null** value. | 1. It gives **ordered** output. 2. It can receive **any** type of **objects**. 3. It can **have** only one **null** value. |

**Map:**

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. **Map** contains only **unique keys**.

Map is useful if you have to search, update or delete elements on the basis of key.

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

**HashMap:** HashMap is the implementation of Map but it doesn't maintain any order.

**LinkedHasMap:** LinkedHashMap is the implementation of Map, it inherits HashMap class. It maintains insertion order.

**TreeMap:** TreeMap is the implementation of Map and SortedMap, it maintains ascending order.

**Useful methods and description:**

**Object put(Object key, Object value):** It is used to insert an entry in this map.

**void putAll(Map map):** It is used to insert the specified map in this map.

**Object remove(Object key):** It is used to delete an entry for the specified key.

**Object get(Object key):** It is used to return the value for the specified key.

**boolean containsKey(Object key):** It is used to search the specified key from this map.

**Set keySet():** It is used to return the Set view containing all the keys.

**Set entrySet():** It is used to return the Set view containing all the keys and values.

**Map.Entry Interface:**

Entry is the sub interface of Map. So, we will be accessed it by Map.Entry name. It provides methods to get key and value.

**Methods of Map.Entry interface:**

Object getKey(): It is used to obtain key.

Object getValue(): It is used to obtain value.

HashMap :

* A HashMap contains values based on the key.
* It contains only unique elements.
* It may have one null key and multiple null values.
* It **maintains no order.**

LinkedHashMap:

* A LinkedHashMap contains values based on the key.
* It contains only unique elements.
* It may have one null key and multiple null values.
* It is same as HashMap instead **maintains insertion order**.

TreeMap:

* A TreeMap contains values based on the key.
* It contains only unique elements.
* It cannot have null key but can have multiple null values.
* It is same as HashMap instead maintains ascending order.

**Collections class:**

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

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

**Java Hashtable class:**

* Java Hashtable class implements a hashtable, which maps keys to values. It inherits Dictionary class and implements the Map interface.
* Hashtable contains values based on the key
* It contains only unique elements
* It may not have any null key or value

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 some differences.

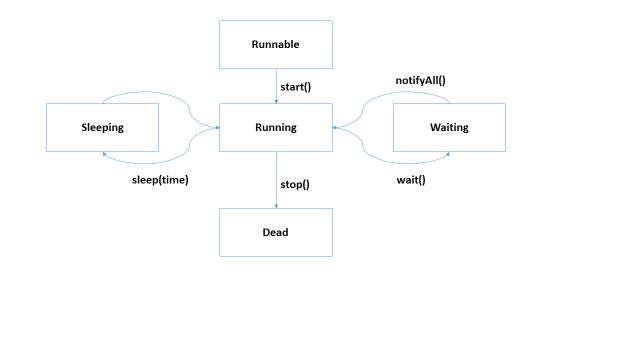
**Difference b/w HashMap and HashTable:**

|  |  |
| --- | --- |
| **HashMap** | **HashTable** |
| HashMap allows one null key and multiple null values. | Hashtable doesn't allow any null key or value. |
| It is **fast**. | It is slow |
| HashMap is a **new class introduced in JDK 1.2** | Hashtable is a **legacy class**. |
| HashMap is **non synchronized** | Hashtable is **synchronized** |
| HashMap is **traversed by Iterator**. | Hashtable is **traversed by Enumerator and Iterator**. |
| HashMap inherits **AbstractMap** class. | Hashtable inherits **Dictionary** class. |

**Thread:**

* Thread is a light weight container which divides the CPU execution time into multiple units.
* A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.Threads are independent.

**Thread Life Cycle**



A thread can be created in two ways.

1. By extending ‘Thread’ class.
2. By implementing ‘Runnable’ interface.

**🡪** Whenever we create a thread, we need to override the ‘**run ()**’ method.

**sleep():**

It is a public static method present inside the ‘Thread’ class. This method makes the thread to sleep for given interval of time.

* This method throws a checked exception i.e. “**InterruptedException**”.

**wait():**

This is a ‘Object’ class method which makes the thread to enter the waiting state. (Waiting pool)

* Threads in the waiting pool will get back to the running state once they are notify.

**notify():**

It is a ‘Object’ class method which notify one thread at a time which is present in the waiting pool.

**notifyAll():**

It will notify all the threads which are present in the waiting pool.