Unit - 4

Multithreading

Thread: A thread is a single sequential (separate) flow of control within program. Sometimes, it is called an execution context or light weight process.

Multithreading: Multithreading is a conceptual programming concept where a program (process) is divided into two or more subprograms (process), which can be implemented at the same time in parallel. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

Multitasking: Executing several tasks simultaneously is called multi-tasking.

There are 2 types of multi-tasking

1. Process-based multitasking
2. Thread-based multi-tasking

1. Process-based multi-tasking

Executing various jobs together where each job is a separate independent operation is called process-based multi-tasking.

2. Thread-based multi-tasking

Executing several tasks simultaneously where each task is a separate independent part of the same program is called Thread-based multitasking and each independent part is called Thread. It is best suitable for the programmatic level. The main goal of multi-tasking is to make or do a better performance of the system by reducing response time

Process based multitasking	Thread based multitasking
Process is a program under execution	Thread is one part of a program
Two or more programs run concurrently	Two or more parts of a single program run concurrently
Heavyweight process.	Lightweight process.
Programs requires separate address spaces	Same address space is shared by threads.

Interprocess communication is expensive and limited	Interthread communication is inexpensive.
Context switching from one process to another is also costly	Context switching from one thread to the next is lower in cost.
May create more idle time	Reduces the idle time
Ex: Running a Java compiler and downloading a file from a web site at the same time	Ex: We can format a text using a Text editor and printing the data at the same time.

Life Cycle of Thread

A thread can be in any of the five following states

1.Newborn State:

When a thread object is created a new thread is born and said to be in Newborn state.

2.Runnable State:

If a thread is in this state it means that the thread is ready for execution and waiting for the availability of the processor. If all threads in queue are of same priority then they are given time slots for execution in round robin fashion

3.Running State:

It means that the processor has given its time to the thread for execution. A thread keeps running until the following conditions occurs

a) Thread give up its control on its own and it can happen in the following situations

   i. A thread gets suspended using suspend() method which can only be revived with resume() method

   ii. A thread is made to sleep for a specified period of time using sleep(time) method, where time in milliseconds

   iii. A thread is made to wait for some event to occur using wait () method. In this case a thread can be scheduled to run again using notify () method.

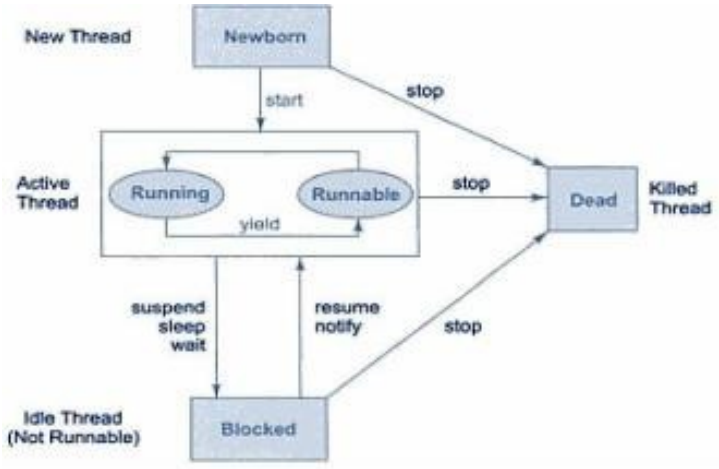
b) A thread is pre-empted by a higher priority thread

4. Blocked State:

If a thread is prevented from entering into runnable state and subsequently running state, then a thread is said to be in Blocked state.

5. Dead State:

A runnable thread enters the Dead or terminated state when it completes its task or otherwise



The Main Thread:

When we run any java program, the program begins to execute its code starting from the main method. Therefore, the JVM creates a thread to start executing the code present in main method. This thread is called as main thread. Although the main thread is automatically created, you can control it by obtaining a reference to it by calling currentThread() method

Two important things to know about main thread are,

- It is the thread from which other threads will be produced.
- main thread must be always the last thread to finish execution

```
class MainThread
{
 public static void main(String[] args)
 {
  Thread t1=Thread.currentThread();
  t.setName("MainThread");
  System.out.println("Name of thread is "+t1);
 }
}
```

Output: Name of thread is Thread[MainThread,5,main]

Creation Of Thread

Thread Can Be Implemented In Two Ways

1) Implementing Runnable Interface

2) Extending Thread Class

1.Create Thread by Implementing Runnable

The easiest way to create a thread is to create a class that implements the Runnable interface. To implement Runnable, a class need only implement a single method called run()

Example:

```
public class ThreadSample implements Runnable
  {
        public void run()  {
          try
        {
        for (int i = 5; i > 0; i--)
        {
        System.out.println("Child Thread" + i);
        Thread.sleep(1000);
        } }
        catch (InterruptedException e)
        {
        System.out.println("Child interrupted");
        }
          System.out.println("Exiting Child Thread");
        } }
        public class MainThread
        {
        public static void main(String[] arg)
        {
        ThreadSample d = new ThreadSample();
        Thread s = new Thread(d);
        s.start();
 try
{
for (int i = 5; i > 0; i--)
{
System.out.println("Main Thread" + i);
```

```
Thread.sleep(5000);
}   }
catch (InterruptedException e)
{
System.out.println("Main interrupted");
}
System.out.println("Exiting Main Thread");
}}
```

2. **Extending Thread Class**

The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class. The extending class must override the run( ) method, which is the entry point for the new thread. It must also call start( ) to begin execution of the new thread.

Example:

```
public class ThreadSample extends Thread
{
public void run()
{
try
{
for (int i = 5; i > 0; i--)
{
System.out.println("Child Thread" + i);
Thread.sleep(1000);
}   }
  catch (InterruptedException e)
{
System.out.println("Child interrupted");
}
System.out.println("Exiting Child Thread");
} }
public class MainThread
{
public static void main(String[] arg)
{
ThreadSample d = new ThreadSample();
d.start();
try
{
for (int i = 5; i > 0; i--)
{
System.out.println("Main Thread" + i);
Thread.sleep(5000);
}   }
catch (InterruptedException e)
```

```
{
System.out.println("Main interrupted");
}
System.out.println("Exiting Main Thread");   } }
```

Thread priority:

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

Three constants defined in Thread class:

public static int MIN_PRIORITY public static int NORM_PRIORITY public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY).

The value of MIN_PRIORITY is 1 and

the value of MAX_PRIORITY is 10.

Example :

```
public class MyThread1 extends Thread
{
MyThread1(String s)
{
  super(s);
start();
}
public void run()
{
for(int i=0;i<5;i++)
{
Thread cur=Thread.currentThread();
cur.setPriority(Thread.MAX_PRIORITY);
int p=cur.getPriority();
System.out.println("Thread Name"+Thread.currentThread().getName());
System.out.println("Thread Priority"+cur);
}  } }
class MyThread2 extends Thread
{
MyThread2(String s)
{
super(s);
start();
}
public void run()
{
for(int i=0;i<5;i++)
{
```

```
Thread cur=Thread.currentThread();
cur.setPriority(Thread.MIN_PRIORITY);
System.out.println(cur.getPriority());
int p=cur.getPriority();
System.out.println("Thread Name"+Thread.currentThread().getName());
System.out.println("Thread Priority"+cur);
} } }
public class ThreadPriority
{
public static void main(String[] args)
{
MyThread1 m1=new MyThread1("MyThread1");
MyThread2 m2=new MyThread2("MyThread2");
} }
```

Synchronizing Threads

- Synchronization in java is the capability to control the access of multiple threads to any shared resource.
- Java Synchronization is better option where we want to allow only one thread to access the shared resource

General Syntax :

```
synchronized(object)
{
//statement to be synchronized
}
```

Why use Synchronization

The synchronization is mainly used to

- To prevent thread interference.
- To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

- Process Synchronization
- Thread Synchronization

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
   - Synchronized method.
   - Synchronized block.
   - static synchronization.
2. Cooperation (Inter-thread communication in java)

Synchronized method

- If you declare any method as synchronized, it is known as synchronized method.
- Synchronized method is used to lock an object for any shared resource.
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Example of synchronized method

```
package Thread;
public class SynThread
{
public static void main(String args[])
{
share s = new share();
MyThread m1 = new MyThread(s, "Thread1");
MyThread m2 = new MyThread(s, "Thread2");
MyThread m3 = new MyThread(s, "Thread3");
} }
class MyThread extends Thread
{
share s;
MyThread(share s, String str)
{
super(str);
this.s = s;
start();
}
public void run()
{
s.doword(Thread.currentThread().getName());
} }
class share
{
public synchronized void doword(String str)
{
for (int i = 0; i < 5; i++)
{
System.out.println("Started:" + str);
try
{
Thread.sleep(1000);
}
catch (Exception e)
{
} } } }
```

Synchronized block.

- Synchronized block can be used to perform synchronization on any specific resource of the method.
- Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.
- If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Example of synchronized block

```
class Table
{
void printTable(int n)
{
synchronized(this)
{
//synchronized block
for(int i=1;i<=5;i++)
{
System.out.println(n*i);
try{
Thread.sleep(400);
}
catch(Exception e)
{
System.out.println(e);
} }  } }
//end of the method
}
class MyThread1 extends Thread
{
Table t;
MyThread1(Table t)
{
this.t=t;
}
public void run()
{
t.printTable(5);
} }
class MyThread2 extends Thread
{
```

```
Table t;
MyThread2(Table t)
{
this.t=t;
}
public void run()
{
t.printTable(100);
} }
public class TestSynchronizedBlock1
{
public static void main(String args[])
{
Table obj = new Table(); //only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
} }
```

Static synchronization
If you make any static method as synchronized, the lock will be on the class not on object.

Example of static synchronization
In this example we are applying synchronized keyword on the static method to perform static synchronization.

```
class Table
{
synchronized static void printTable(int n)
{
for(int i=1;i<=10;i++)
{
    System.out.println(n*i);
Try
{
Thread.sleep(400);
}
catch(Exception e)
{
}}}}

class MyThread1 extends Thread
{
public void run()
{
```

```
Table.printTable(1);
} }
class MyThread2 extends Thread
{
public void run()
{
Table.printTable(10);
} }
class MyThread3 extends Thread
{
public void run()
{
Table.printTable(100);
} }

class MyThread4 extends Thread
{
public void run()
{
Table.printTable(1000);
}}
public class TestSynchronization4
{
public static void main(String t[])
{
MyThread1 t1=new MyThread1();
MyThread2 t2=new MyThread2();
MyThread3 t3=new MyThread3();
MyThread4 t4=new MyThread4();
t1.start();   t2.start();   t3.start();   t4.start();   }}
```

Inter-thread communication

Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.

Inter-thread communication is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of Object class:

- wait()
- notify()
- notifyAll()

wait() - tells calling thread to give up monitor and go to sleep until some other thread enters the same monitor and call notify.

notify() - wakes up a thread that called wait() on same object.

notifyAll() - wakes up all the thread that called wait() on same object.

Wait ( )	Sleep( )
wait() method releases the lock	sleep() method doesn't release the lock.
is the method of Object class	is the method of Thread class
is the non-static method	is the static method
should be notified by notify() or notifyAll() methods	after the specified amount of time, sleep is completed.

Example of inter thread communication in java

```
class Customer{
int amount=10000;
synchronized void withdraw(int amount)
{
System.out.println("going to withdraw...");
if(this.amount<amount){
System.out.println("Less balance; waiting for deposit..."); try{wait();}catch(Exception e)
{
} }
this.amount-=amount;
System.out.println("withdraw completed...");
}
synchronized void deposit(int amount)
{
System.out.println("going to deposit...");
this.amount+=amount;
System.out.println("deposit completed... ");
notify();
} }
class Test
{
  public static void main(String args[])
{
final Customer c=new Customer();
```

```
new Thread()
{
public void run()
{
c.withdraw(15000);
} }
start();
new Thread()
{
public void run()
{
c.deposit(10000);
}   }
.start();
}}
```

What is collection Framework
Collections in Java

The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

What is Collection in Java

A Collection represents a single unit of objects, i.e., a group.

What is a framework in Java

- It provides readymade architecture.
- It represents a set of classes and interfaces.
- It is optional.

What is Collection framework

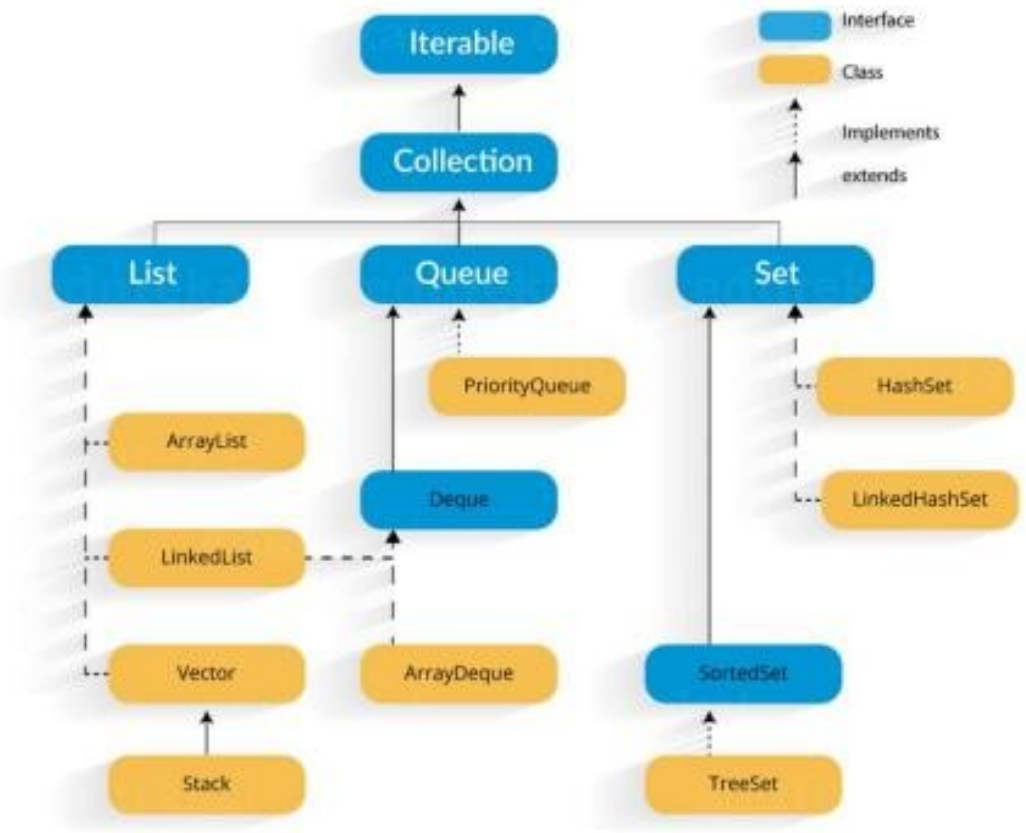
The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

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

2.Algorithm

Hierarchy of collection framework.

Hierarchy of Collection Framework Let us see the hierarchy of Collection framework. The java.util package contains all the classes and interfaces for the Collection framework.



Methods of Collection interface

There are many methods declared in the Collection interface. They are as follows:

No.	Method	Description
1	public boolean add(E e)	It is used to insert an element in this collection.
2	public boolean addAll(Collection<? extends E> c)	It is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	It is used to delete an element from the collection.
4	public boolean removeAll(Collection<?> c)	It is used to delete all the elements of the specified collection from the invoking collection.
5	default boolean removeIf(Predicate<? super E> filter)	It is used to delete all the elements of the collection that satisfy the specified predicate.
6	public boolean retainAll(Collection<?> c)	It is used to delete all the elements of invoking collection except the specified collection.
7	public int size()	It returns the total number of elements in the collection.
8	public void clear()	It removes the total number of elements from the collection.
9	public boolean contains(Object element)	It is used to search an element.
10	public boolean containsAll(Collection<?> c)	It is used to search the specified collection in the collection.
11	public Iterator iterator()	It returns an iterator.
12	public Object[] toArray()	It converts collection into array.
13	public <T> T[] toArray(T[] a)	It converts collection into array. Here, the runtime type of the returned array is that of the specified array.
14	public boolean isEmpty()	It checks if collection is empty.

15	default Stream<E> parallelStream()	It returns a possibly parallel Stream with the collection as its source.
16	default Stream<E> stream()	It returns a sequential Stream with the collection as its source.
17	default Spliterator<E> spliterator()	It generates a Spliterator over the specified elements in the collection.
18	public boolean equals(Object element)	It matches two collections.
19	public int hashCode()	It returns the hash code number of the collection.

Iterator Interface: Iterator is an interface that contains methods to retrieve the elements one by one from a collection object. It retrieves elements only in forward direction. It has 3 methods:

Method	Description
boolean hasNext()	This method returns true if the iterator has more elements.
element next()	This method returns the next element in the iterator.
void remove()	This method removes the last element from the collection returned by the iterator.

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<T> iterator()

It returns the iterator over the elements of type T.

Collection Interface

The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

Some of the methods of Collection interface are Boolean add (Object obj), Boolean addAll (Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

List <data-type> list1= new ArrayList();
List <data-type> list2 = new LinkedList();
List <data-type> list3 = new Vector();
List <data-type> list4 = new Stack();

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

ArrayList

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

```
import java.util.*;
class TestJavaCollection1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();//Creating arraylist
list.add("Babu");//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());
} } }
```

Output:

```
Babu
Vijay
Ravi
Ajay
```

LinkedList

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection2{
public static void main(String args[]){
LinkedList<String> al=new LinkedList<String>();
al.add("Babu");
al.add("Vijay");
al.add("Ravi");
al.add("Ajay");
Iterator<String> itr=al.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
} } }
```

Output:

```
Babu
Vijay
Ravi
Ajay
```

Vector

Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection3{
public static void main(String args[]){
Vector<String> v=new Vector<String>();
v.add("Ayush");
v.add("Amit");
v.add("Ashish");
v.add("Garima");
Iterator<String> itr=v.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
} } }
```

Output:

```
Ayush
Amit
Ashish
Garima
```

Stack

The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection4{
public static void main(String args[]){
Stack<String> stack = new Stack<String>();
stack.push("Ayush");
stack.push("Garvit");
stack.push("Amit");
stack.push("Ashish");
```

```
stack.push("Garima");
stack.pop();
Iterator<String> itr=stack.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
} } }
```

Output:

```
Ayush
Garvit
Amit
Ashish
```

Queue Interface

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated as:

```
Queue<String> q1 = new PriorityQueue();
Queue<String> q2 = new ArrayDeque();
```

There are various classes that implement the Queue interface, some of them are given below.

PriorityQueue

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection5{
public static void main(String args[]){
PriorityQueue<String> queue=new PriorityQueue<String>();
```

```
queue.add("Amit Sharma");
queue.add("Vijay Raj");
queue.add("JaiShankar");
queue.add("Raj");
System.out.println("head:"+queue.element());
System.out.println("head:"+queue.peek());
System.out.println("iterating the queue elements:");
Iterator itr=queue.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
queue.remove();
queue.poll();
System.out.println("after removing two elements:");
Iterator<String> itr2=queue.iterator();
while(itr2.hasNext()){
System.out.println(itr2.next());
} } }
```

Output:

```
head:Amit Sharma
head:Amit Sharma
iterating the queue elements:
Amit Sharma
Raj
JaiShankar
Vijay Raj
after removing two elements:
Raj
Vijay Raj
```

Deque Interface

Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

Deque can be instantiated as:

Deque d = new ArrayDeque();

ArrayDeque

ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.

ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection6{
public static void main(String[] args) {
//Creating Deque and adding elements
Deque<String> deque = new ArrayDeque<String>();
deque.add("Gautam");
deque.add("Karan");
deque.add("Ajay");
//Traversing elements
for (String str : deque) {
System.out.println(str);
} } }
```

Output:

```
Gautam
Karan
Ajay
```

Set Interface

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

Set<data-type> s1 = new HashSet<data-type>();
Set<data-type> s2 = new LinkedHashSet<data-type>();
Set<data-type> s3 = new TreeSet<data-type>();

HashSet

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection7{
public static void main(String args[]){
//Creating HashSet and adding elements
HashSet<String> set=new HashSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//Traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
} } }
```

Output:

```
Vijay
Ravi
Ajay
```

LinkedHashSet

LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection8{
public static void main(String args[]){
LinkedHashSet<String> set=new LinkedHashSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
} } }
```

Output:

```
Ravi
Vijay
Ajay
```

SortedSet Interface

SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order. The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

The SortedSet can be instantiated as:

```
SortedSet<data-type> set = new TreeSet();
```

TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

Consider the following example:

```
import java.util.*;
```

```
public class TestJavaCollection9{
public static void main(String args[]){
//Creating and adding elements
TreeSet<String> set=new TreeSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
} }  }
```

Output:

```
Ajay
Ravi
Vijay
```

ArrayList Class: An ArrayList is like an array, which can grow in memory dynamically.

ArrayList is not synchronized. This means that when more than one thread acts simultaneously on the ArrayList object, the results may be incorrect in some cases.

ArrayList class can be written as:

class ArrayList <E>

We can create an object to ArrayList as:

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

ArrayList Class Methods:

Method	Description
boolean add (element obj)	This method appends the specified element to the end of the ArrayList. If the element is added successfully then the method returns true.
void add(int position, element obj)	This method inserts the specified element at the specified positioin in the ArrayList.
element remove(int position)	This method removes the element at the specified position in the ArrayList and returns it.
boolean remove (Object obj)	This method removes the first occurrence of the specified element obj from the ArrayList, if it is present.
void clear ()	This method removes all the elements from the ArrayList.
element set(int position, element obj)	This method replaces an element at the specified position in the ArrayList with the specified element obj.
boolean contains (Object obj)	This method returns true if the ArrayList contains the specified element obj.
element get (int position)	This method returns the element available at the specified position in the ArrayList.
int size ()	Returns number of elements in the ArrayList.
int indexOf (Object obj)	This method returns the index of the first occurrence of the specified element in the list, or -1 if the list does not contain the element.
int lastIndexOf (Object obj)	This method returns the index of the last occurrence of the specified element in the list, or -1 if the list does not contain the element.
Object[] toArray ()	This method converts the ArrayLlist into an array of Object class type. All the elements of the ArrayList will be stored into the array in this sequence.

LinkedList Class: A linked list contains a group of elements in the form of nodes. Each node will have three fields- the data field contatins data and the link fields contain references to previous and next nodes.A linked list is written in the form of:

class LinkedList<E>

we can create an empty linked list for storing String type elements (objects) as:

LinkedList <String> ll = new LinkedList<String> ();
LinkedList Class methods:

Method	Description
boolean add (element obj)	This method adds an element to the linked list. It returns true if the element is added successfully.
void add(int position, element obj)	This method inserts an element obj into the linked list at a specified position.
void addFirst(element obj)	This method adds the element obj at the first position of the linked list.
void addLast(element obj)	This method adds the element obj at the last position of the linked list.
element removeFirst ()	This method removes the first element from the linked list and returns it.
element removeLast ()	This method removes the last element from the linked list and returns it.
element remove (int position)	This method removes an element at the specified position in the linked list.
void clear ()	This method removes all the elements from the linked list.
element get (int position)	This method returns the element at the specified position in the linked list.
element getFirst ()	This method returns the first element from the list.
element getLast ()	This method returns the last element from the list.
element set(int position, element obj)	This method replaces the element at the specified position in the list with the specified element obj.
int size ()	Returns number of elements in the linked list.
int indexOf (Object obj)	This method returns the index of the first occurrence of the specified element in the list, or -1 if the list does not contain the element.
int lastIndexOf (Object obj)	This method returns the index of the last occurrence of the specified element in the list, or -1 if the list does not contain the element.
Object[] toArray()	This method converts the linked list into an array of Object class type. All the elements of the linked list will be stored into the array in the same sequence.

HashSet Class

HashSet Class: HashSet represents a set of elements (objects). It does **not guarantee the order** of elements. Also it **does not allow the duplicate** elements to be stored.

- We can write the HashSet class as : class HashSet <T>
- We can create the object as : HashSet <String> hs = new HashSet<String> ();

The following constructors are available in HashSet:

- **HashSet();**
- **HashSet (int capacity);** Here capacity represents how many elements can be stored into the HashSet initially. This capacity may increase automatically when more number of elements is being stored.

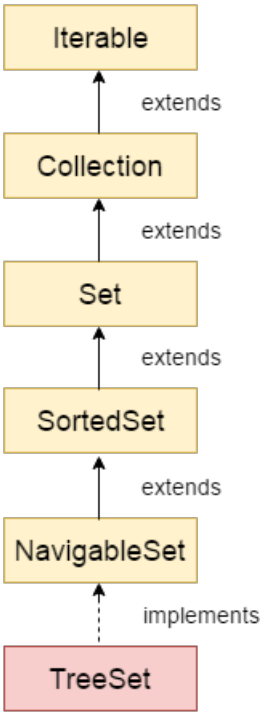
Constructors of Java HashSet class

SN	Constructor	Description
1)	HashSet()	It is used to construct a default HashSet.
2)	HashSet(int capacity)	It is used to initialize the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet.
3)	HashSet(int capacity, float loadFactor)	It is used to initialize the capacity of the hash set to the given integer value capacity and the specified load factor.
4)	HashSet(Collection<? extends E> c)	It is used to initialize the hash set by using the elements of the collection c.

HashSet Class Methods:

Method	Description
boolean add(obj)	This method adds an element obj to the HashSet. It returns true if the element is added to the HashSet, else it returns false. If the same element is already available in the HashSet, then the present element is not added.
boolean remove(obj)	This method removes the element obj from the HashSet, if it is present. It returns true if the element is removed successfully otherwise false.
void clear()	This removes all the elements from the HashSet
boolean contains(obj)	This returns true if the HashSet contains the specified element obj.
boolean isEmpty()	This returns true if the HashSet contains no elements.
int size()	This returns the number of elements present in the HashSet.

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 the 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.
- Java TreeSet class contains unique elements only like HashSet.
- Java TreeSet class access and retrieval times are quite fast.
- Java TreeSet class doesn't allow null elements.
- Java TreeSet class is non-synchronized.
- Java TreeSet class maintains ascending order.

- The TreeSet can only allow those generic types that are comparable. For example The Comparable interface is being implemented by the StringBuffer class.

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

Constructors of Java TreeSet Class

Constructor	Description
TreeSet()	It is used to construct an empty tree set that will be sorted in ascending order according to the natural order of the tree set.
TreeSet(Collection<? extends E> c)	It is used to build a new tree set that contains the elements of the collection c.
TreeSet(Comparator<? super E> comparator)	It is used to construct an empty tree set that will be sorted according to given comparator.
TreeSet(SortedSet<E> s)	It is used to build a TreeSet that contains the elements of the given SortedSet.

PriorityQueue Class

PriorityQueue is also class that is defined in the collection framework that gives us a way for processing the objects on the basis of priority. It is already described that the insertion and deletion of objects follows FIFO pattern in the Java queue. However, sometimes the elements of the queue are needed to be processed according to the priority, that's where a PriorityQueue comes into action.

Java Hashtable class

Java Hashtable class implements a hashtable, which maps keys to values. It inherits Dictionary class and implements the Map interface.

Points to remember

- A Hashtable is an array of a list. Each list is known as a bucket. The position of the bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.
- Java Hashtable class contains unique elements.
- Java Hashtable class doesn't allow null key or value.
- Java Hashtable class is synchronized.
- The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.

Hashtable class declaration

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

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

Hashtable class Parameters

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

- **K**: It is the type of keys maintained by this map.
- **V**: It is the type of mapped values.
- Constructors of Java Hashtable class

Constructor	Description
Hashtable()	It creates an empty hashtable having the initial default capacity and load factor.
Hashtable(int capacity)	It accepts an integer parameter and creates a hash table that contains a specified initial capacity.
Hashtable(int capacity, float loadFactor)	It is used to create a hash table having the specified initial capacity and loadFactor.
Hashtable(Map<? extends K,? extends V> t)	It creates a new hash table with the same mappings as the given Map.