UNIT - IV

MULTI THREADING, COLLECTIONS

Multi threading: java.lang.Thread, main thread, creation of new threads, thread priority, multithreading - using isalive() and join(), Synchronization, Suspending and resuming threads, Communication between threads.

Collections: Overview of Collection Framework - ArrayList, Vector, TreeSet, HashMap, Hashable, Iterator, Comparator

Multitasking:

Multitasking is a process of executing multiple tasks simultaneously.We use multitasking to utilize the CPU. There are two types of multitasking:

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

1. Process-Based Multitasking

Executing several tasks simultaneously, where each task is a separate independent process, is called **process-based multitasking**.

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

Example:
While typing a Java program in the editor, we can listen to MP3 audio songs. At the same time, we can download a file from the Internet. All these tasks are executing simultaneously and are independent of each other.

2) Thread-Based Multitasking (Multithreading)

Multithreading in java is a process of executing multiple threads simultaneously. Executing several tasks simultaneously, where each task is a separate independent part of the same program, such type of multitasking is called **Thread Based Multitasking** and each independent part is called a **thread**.

- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

Note: Java provides inbuilt support for multithreading by providing a rich library (**Thread, Runnable, ThreadGroup, ThreadLocal**, etc.).

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context switching between the threads takes less time than process.

Applications of Multithreading

Java Multithreading is mostly used in games, animation, etc.

1) It doesn't block the user because threads are independent and you can perform multiple operations at the same time.

2) You can perform many operations together, so it saves time.

3)Threads are independent, so it doesn't affect other threads if an exception occurs in a single thread

Thread:

Definition: A thread is a lightweight subprocess and an independent path of execution within a program.

Thread Class Declaration

public class Thread implements Runnable

Life Cycle of a Thread

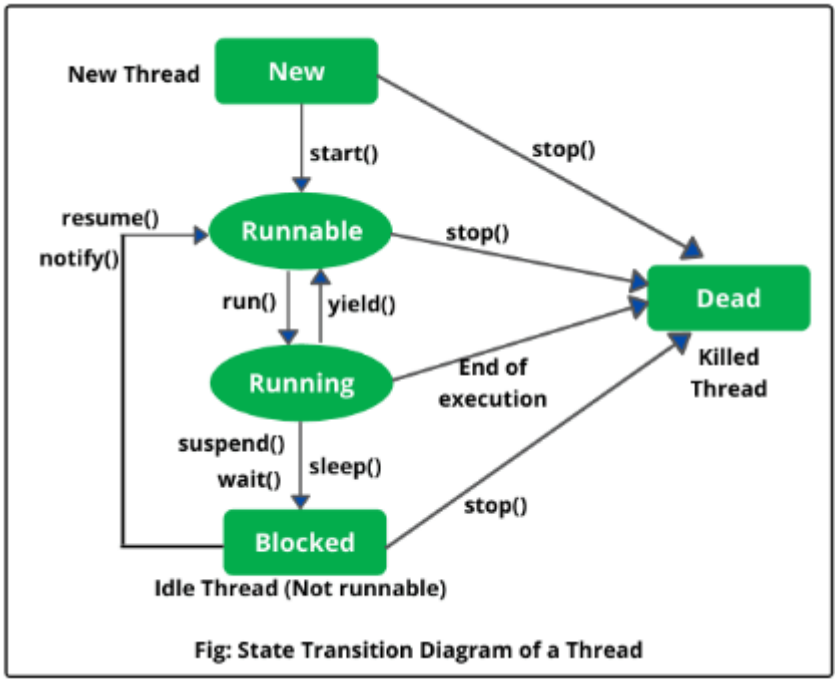


Fig: State Transition Diagram of a Thread

New State: A new thread begins its life cycle in the new state. It remains in this state

until the program starts the thread. It is also referred to as a **born thread**.

Runnable state: After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

Waiting state: Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

Blocked state: A runnable thread can enter the blocked state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

Terminated (Dead) state: A runnable thread enters the terminated state when it completeits task or otherwise terminates.

Thread class:

- Thread class is available in the java.lang package.
- Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface

Commonly used Constructors of Thread class:

1. Thread()
2. Thread(String name)
3. Thread(Runnable r)
4. Thread(Runnable r, String name)

- Every thread executes its run() method.
- start() creates a new thread; run() contains the task.

Commonly Used Methods of Thread Class

Method	Description
public void start()	Starts a new thread and internally calls run().
`public void` run()	Contains the code executed by the thread
public void sleep(long ms)	Pauses the current thread for the specified time
public void join()	Waits for a thread to complete(die).
public void yield()	Gives a chance to other threads of the same priority to execute
public void suspend()	is used to suspend the thread(depricated)
public void resume()	is used to resume the suspended thread(depricated)
public StringgetName()	Returns the thread name
`public void` setName(String name)	Changes the thread name
Public void currentThread()	Returns the currently executing thread.

Public booleanisAlive()	Checks whether the thread is still running
public int getPriority()	returns the priority of the thread
public int setPriority(int priority)	returns the name of the thread

Job of a Thread

- Defining a Thread.
- Starting a Thread.

Ex:

```
class Thread{
    public void run() {
      // not important
    }
    public void start(){
      // Registers the thread with the Thread Scheduler
      // Invokes the run() method
    }
  }
```

Main Thread and Creation of New Threads in Java

When a Java program starts, the JVM automatically creates a thread called the **Main Thread**.

- It executes the main() method.
- It is the parent of all user-created threads.
- Every Java application has one main thread.

```
public class Test {
   public static void main(String[] args) {
System.out.println("Main Thread");
   }
}
```

Creationof new threads:

Java supports multithreading through the Thread class and the Runnable interface. We can define a thread in the following two ways

1. By extending **Thread** class.
2. By implementing **Runnable** interface.

In Java multithreading, **java.lang.Thread** is the predefined class used to create and manage threads. It belongs to the **java.lang** package and provides methods for thread creation, execution, and control.

Creating a Thread by Extending Thread Class

```
class MyThread extends Thread {
   public void run() {
for(int i = 1; i<= 5; i++) {
System.out.println("Child Thread");
      }
   }
}
class MyThreadDemo {
   public static void main(String[] args) {
MyThread t = new MyThread(); // Thread creation
t.start();          // Starts new thread
for(int i = 1; i<= 5; i++) {
System.out.println("Main Thread");
      }
   }
}
```

Output:

```
C:\nrcm>javac MyThreadDemo.java

C:\nrcm>java MyThreadDemo
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
Child Thread
Child Thread
Child Thread
Child Thread
Child Thread
```

When you callt.start(), the JVM performs two important tasks:

1. Registers the thread with the Thread Scheduler.
2. Invokes the run() method automatically.

If you callt.run()directly, no new thread is created, it behaves like a normal method call.

2.By Implementing Runnable Interface

```
class MyRunnable implements Runnable {
   public void run() {
System.out.println("Child Thread");
   }
}
class MyThreadDemo1 {
   public static void main (String [] args) {
MyRunnable r = new MyRunnable();
      Thread t = new Thread(r);
t.start();
System.out.println("Main Thread");
   }
}
```

Output:

```
C:\nrcm>javac MyThreadDemo1.java

C:\nrcm>java MyThreadDemo1
Main Thread
Child Thread
```

```
MyThread t = new MyThread();  // Thread object created
t.start();                    // New thread starts
```

When start() is called:

1. A new thread is registered with the Thread Scheduler.
2. The thread moves to the Runnable state.
3. JVM invokes the run() method.
4. The new thread executes concurrently with the main thread.

Main Thread vs Child Thread

Main Thread	**Child Thread**
Created automatically by JVM	Created by programmer
Executes main() method	Executes run() method
Starts first	Starts after start() is called
Parent thread	Child thread

Main Thread: The thread created by JVM to execute the main() method.

Creating a New Thread:

1. Define a thread (extend Thread or implement Runnable).
2. Create a thread object.
3. Call start() to begin execution.

Job of start() method:

- Register the thread with the Thread Scheduler.
- Invoke the run() method automatically.

Thread Scheduler in Java:

- Thread Scheduler in Java Thread scheduler in java is the part of the JVM that decides which thread should run.
- There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.
- Only one thread at a time can run in a single process

THREAD PRIORITY

Thread priority is a scheduling hint that tells the operating system how important a thread is relative to other threads in the same process (and sometimes system-wide). Higher-priority threads generally get CPU time before lower-priority threads.

Every thread in java has some priority. The valid Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority (known as preemptive scheduling) i.e uses these priorities while allocating CPU. The thread having high priority will get chance first for execution. If two threads having same priority, it is not guaranteed which gets executed first because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

The default value of NORM_PRIORITY is 5, the value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

The Thread class defines two methods to get and set the priority of a thread

1. **public final int getPriority():** returns the priority of the thread.
2. **public final int setPriority(int priority):** changes the priority of the thread

The default priority of main() thread is 5. Whatever parent thread has the same will be inherited to the child.

```
// Thread Priority demonstration
class MyThread extends Thread {
public void run() {
System.out.println("Child Thread1 running....");
  }
}
class MyRunnable implements Runnable{
public void run(){
System.out.println("Child Thread2 running....");
}
}
class ThreadPriority {
public static void main(String args[]){
System.out.println(Thread.currentThread().getPriority());
Thread.currentThread().setPriority(10);
System.out.println(Thread.currentThread().getPriority());
MyThread t1=new MyThread();
t1.start();
System.out.println(t1.getPriority());
MyRunnable r1=new MyRunnable();
Thread t2=new Thread(r1);
t2.start();
System.out.println(t2.getPriority());
t2.setPriority(2);
System.out.println(t2.getPriority());
}
}
```

Output:

```
C:\nrcm>javac ThreadPriority.java

C:\nrcm>java ThreadPriority
5
10
10
Child Thread1 running....
10
2
Child Thread2 running....
```

Note:

- Thread priority is **not a guarantee**. The OS scheduler ultimately decides when threads run.

- Excessive use of high priorities can cause **starvation**, where lower-priority threads rarely get CPU time.

Multithreading using isAlive() and join():

IsAlive() and join() are methods of the Thread class that help coordinate multiple threads.

1. isAlive()

The isAlive() method checks whether a thread is still running.

Syntax: threadName.isAlive();

Returns:

- true – if the thread has been started and has not yet finished.
- false – if the thread has not started or has already completed.

Program :

```
class MyThread extends Thread {
   public void run() {
for(int i = 1; i<= 3; i++) {
System.out.println("Child Thread: " + i);
       }
   }
}
public class IsAliveDemo {
   public static void main(String[] args) {
MyThread t = new MyThread();
System.out.println("Before start: " + t.isAlive());
t.start();
System.out.println("After start: " + t.isAlive());
   }
}
```

Output:

```
C:\nrcm>javac IsAliveDemo.java

C:\nrcm>java IsAliveDemo
Before start: false
After start: true
Child Thread: 1
Child Thread: 2
Child Thread: 3
```

2. join()

The join() method makes one thread wait until another thread finishes its execution.

Syntax:

```
threadName.join();
```

Note:

- Ensures that a thread completes before the next statement executes.
- Useful when one task depends on the completion of another.

Example:

```
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i<= 5; i++) {
System.out.println("Child Thread: " + i);
        }
    }
}
public class JoinDemo {
    public static void main(String[] args) throws InterruptedException {
MyThread t = new MyThread();
t.start();
t.join();  // Main thread waits for child thread
System.out.println("Main thread resumes after child thread finishes.");
        for (int i = 1; i<= 5; i++) {
System.out.println("main Thread: " + i);
        }
    }
}
```

Output:

```
C:\nrcm>javac JoinDemo.java

C:\nrcm>java JoinDemo
Child Thread: 1
Child Thread: 2
Child Thread: 3
Child Thread: 4
Child Thread: 5
Main thread resumes after child thread finishes.
main Thread: 1
main Thread: 2
main Thread: 3
main Thread: 4
main Thread: 5
```

- Main thread starts → Creates t → Calls t.start()
- Two threads are now running concurrently (Main and Child).
- Main thread hits t.join(): It immediately goes into a waiting state. It freezes right there.
- Child thread (MyThread) runs to completion, printing its 1 to 5 loop.
- Child thread dies: The lock is released, and the Main thread wakes up.
- Main thread resumes, printing its confirmation message and its own 1 to 5 loop.

Difference between isAlive() and join()

isAlive()	join()
Checks whether a thread is currently running.	Waits until a thread completes execution.
Returns a boolean (true or false).	Returns no value (void).
Does not block the calling thread.	Blocks the calling thread until the target thread finishes.
Used to monitor thread status.	Used to synchronize thread execution.

Synchronization

Synchronization is a modifier applicable only to **methods** and **blocks**, but **not** to variables and classes.

If a method (or) block is declared as **synchronized**, then at any given time **only one thread** is allowed to execute that method (or) block on the given object.

Advantages of Synchronization

- Prevents **data inconsistency** problems.
- Ensures that shared resources are accessed by only one thread at a time.
- Avoids **race conditions** in multithreaded programs.

Disadvantages of Synchronization

- Increases the **waiting time** of threads.
- Reduces the overall **performance** of the system because only one thread can access the synchronized resource at a time.
- Excessive use of synchronization can decrease application efficiency.

Synchronizing Threads

- Synchronization in java is the capability to control the access of multiple threads to a shared resource.
- Synchronization is better option where we want to allow only one thread to access the shared resource.
- Synchronized is the **modifier,** applicable only for methods & blocks but not for variables and classes.

The **advantages** of synchronization are, used to
1. To prevent thread interference.
2. To prevent data consistency problem

.
There are two types of thread synchronization

1. Mutual Exclusion
2. Inter thread communication

1. Process Synchronization (Mutual Exclusion)

This ensures that **only one thread can access a shared resource at a time**.

It is achieved using:

- synchronized method
- synchronized block

```
class Counter {
   private int count = 0;
   synchronized void increment() {
  count++;
  }
}
```

2. Thread Synchronization (Inter-thread Communication)

This allows **threads to communicate and coordinate with each other** instead of just blocking one another.

It is achieved using:

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

Example:

```
synchronized (obj) {
        obj.wait();      // Thread waits
        obj.notify();    // Wakes one waiting thread
        obj.notifyAll(); // Wakes all waiting threads
        }
```

Type	Purpose	Methods Used
Process Synchronization (Mutual Exclusion)	Prevents multiple threads from accessing shared data simultaneously	synchronized, Lock
Thread Synchronization (Inter-thread Communication)	Enables threads to coordinate and communicate	wait(), notify(), notifyAll()

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.

```
class Table{
 synchronized void printTable(int n) { //synchronized method
for(int i=1;i<=5;i++){
System.out.println(n*i);
try{
Thread.sleep(400);
     }
catch(Exception e){
System.out.println(e);
     }
   }
 }
}
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 MethodSynchronizationDemo{
 public static void main(String args[]){
   Table ob1= new Table(); //only one object
   MyThread1 t1=new MyThread1(ob1);
   MyThread2 t2=new MyThread2(ob1);
   t1.start();
   t2.start();
  }
}
```

Output:

```
C:\nrcm>javac SynchronizationDemo.java

C:\nrcm>java SynchronizationDemo
5
10
15
20
25
100
200
300
400
500
```

(ii) 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.

```
// Synchronized Block
class Table{
  void printTable(int n){ // method
   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 block
  } //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 BlockSynchronizedDemo {
public static void main(String args[]) {
Table ob1= new Table();//only one object
MyThread1 t1=new MyThread1(ob1);
 MyThread2 t2=new MyThread2(ob1);
t1.start();
t2.start();
```

```
}
}
```

Output:

```
C:\nrcm>javac BlockSynchronizedDemo.java

C:\nrcm>java BlockSynchronizedDemo
5
10
15
20
25
100
200
300
400
500
```

2. Inter-thread communication in java (Cooperation)

Definition: Communication between threads is a mechanism by which one thread communicates with another thread to coordinate execution. In Java, this is achieved using the methods **wait()**, **notify()**, and **notifyAll()** of the Object class.

Thread Communication Needed

- To avoid unnecessary CPU usage caused by continuous polling.
- To synchronize the execution of multiple threads.
- To implement the Producer–Consumer problem efficiently.

Methods Used

1. wait()

- Causes the current thread to release the lock and enter the waiting state.
- The thread remains waiting until another thread calls notify() or notifyAll().

Syntax:object.wait();

2. notify()

- Wakes up one waiting thread.
- The awakened thread can continue only after acquiring the object's lock.

Syntax:object.notify();

3. notifyAll()

- Wakes up all waiting threads on the object.
- Only one thread acquires the lock at a time.

Syntax:object.notifyAll();

Notes:

- wait(), notify(), and notifyAll() belong to the **Object** class.
- They must be called inside a **synchronized** block or synchronized method.
- Calling them outside synchronization throws an **IllegalMonitorStateException**.

Example

```
class Shared {
   synchronized void display() {
      try {
System.out.println("Thread is waiting...");
wait();
System.out.println("Thread resumed.");
      } catch (InterruptedException e) {
e.printStackTrace();
      }
   }
   synchronized void resumeThread() {
System.out.println("Notifying thread...");
notify();
   }
}
```

Advantages

- Efficient synchronization between threads.
- Prevents busy waiting.
- Reduces CPU utilization.
- Helps solve producer–consumer and other synchronization problems.

Disadvantages

- Programming becomes more complex.
- Incorrect use may lead to deadlock or missed notifications.

```
class Queue {
   int data;
booleanvalueSet = false;
   synchronized void put(int value) {
      try {
         while (valueSet)
wait();

         data = value;
System.out.println("Produced: " + data);

valueSet = true;
notify();
      }
catch (InterruptedException e) {
System.out.println(e);
      }
   }

   synchronized void get() {
      try {
         while (!valueSet)
wait();
System.out.println("Consumed: " + data);
valueSet = false;
notify();
      }
catch (InterruptedException e) {
System.out.println(e);
}
}
}

class Producer extends Thread {
   Queue q;
Producer(Queue q) {
this.q = q;
   }
   public void run() {
      int i = 1;
      while (true) {
q.put(i++);
         try {
Thread.sleep(1000);
         } catch (InterruptedException e) {
         }
      }
   }
}
```

```
class Consumer extends Thread {
   Queue q;
Consumer(Queue q) {
this.q = q;
   }

   public void run() {
      while (true) {
q.get();
         try {
Thread.sleep(1000);
         } catch (InterruptedException e) {
         }
      }
   }
}

public class ProducerConsumerDemo {
   public static void main(String args[]) {
      Queue q = new Queue();

      Producer p = new Producer(q);
      Consumer c = new Consumer(q);

p.start();
c.start();
   }
}
```

Output:

```
C:\nrcm>javac ProducerConsumerDemo.java

C:\nrcm>java ProducerConsumerDemo
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
Produced: 4
Consumed: 4
Produced: 5
Consumed: 5
Produced: 6
Consumed: 6
Produced: 7
Consumed: 7
Produced: 8
Consumed: 8
Produced: 9
Consumed: 9
```

Collection Framework

Overview of Collection Framework:

An Array is an indexed collection of fixed number of homogeneous data elements.

Main limitations of object type Arrays:

1. Fixed in size, i.e., once an array is created there is no chance of increasing or decreasing its size based on requirements.
2. Arrays can hold only homogeneous data elements.

Example:

```
Student[] s = new Student[100];
s[0] = new Student(); ✓
s[1] = new Customer(); ✗
```

Compile-time error: Incompatible type found Customer, required Student.

3. There is no underlying data structure for arrays; hence there is no ready-made method support. For every requirement we have to write code manually.

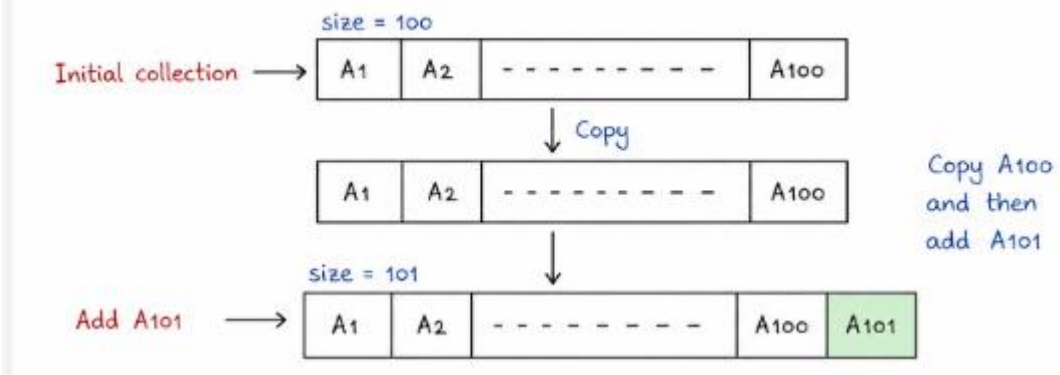
To overcome the above problems, some people introduced the **Collections** concept.

1. **Collections are growable in nature**, i.e., based on our requirement we can increase the size or we can decrease the size.
2. We can use collections to hold both **homogeneous** and **heterogeneous** objects.
3. For every collection class, an underlying data structure is available. Hence, for every requirement, ready-made methods are available. Being a programmer, we have to know how to use those methods, and we are not responsible for implementing those methods.

Limitation of Collections

The main limitation of collections is:

1. **It will affect the performance of the system.** Hence, if we know the size in advance, it is highly recommended to use **arrays** but not **collections**.



Collection

We can use a collection to represent a group of individual objects as a single entity.

Collection Framework

It contains several classes and interfaces which can be used to represent a group of objects as a single entity.

Java	C++
Collection	Container
Collection Framework	STL (Standard Template Library)

Key Interfaces of Collection Framework

1. Collection Interface

- If we want to represent a group of individual objects as a single entity, then we should go for a **Collection**.
- In general, the **Collection Interface** is treated as the **root interface** of the entire Collection Framework.
- This interface defines the most common methods which can be applicable for any collection object.
- There is **no concrete class** that implements the **Collection Interface** directly.

Example:

ArrayList indirectly provides the functionality defined by the Collection Interface.

Collection vs Collections

Collection

Collection is an interface which can be used to represent a group of objects as a single entity.

Collections

Collections is a utility class present in the java.util package to define several utility methods for Collection objects.

Ex: Collection vs Collections

Arraylist l

↓

class(util package)

↓

collections.sort(l);

Ex: Collection vs Collections

Arraylist l

class(util package)

collections.sort(l);

2. List Interface

(i) Child Interface of Collection

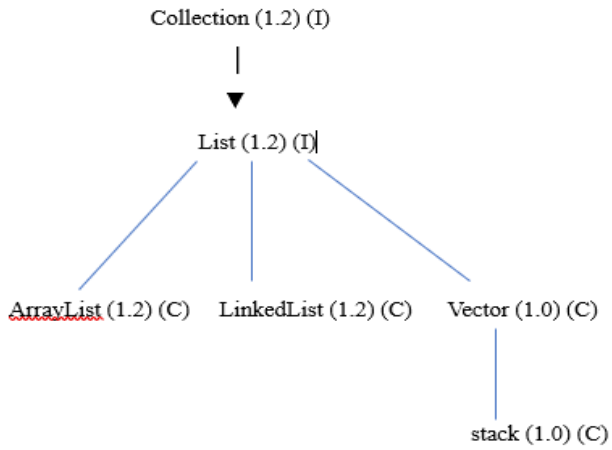
List is the child interface of Collection.

(ii) When to Use List

If we want to represent a group of individual objects where

- Insertion order is preserved.
- Duplicate objects are allowed. Then we should go for the List Interface.

Hierarchy of List Interface



Legend:

- **I** = Interface
- **C** = Class

Note:

- In **Java 1.2**, the existing **Vector** and **Stack** classes were re-engineered (modified) to fit into the new Collection Framework.

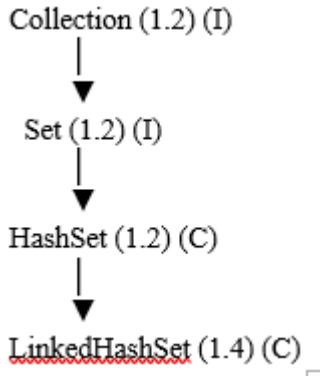
3. Set Interface

(i) It is the Child Interface of Collection If we want to represent a group of individual objects where

- Insertion order is **not preserved**
- Duplicate objects are **not allowed**

then we should go for the **Set Interface**.

Hierarchy of Set Interface

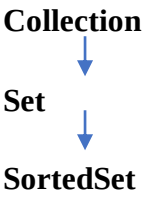


Legend:

- **I** = Interface
- **C** = Class

4. SortedSet Interface

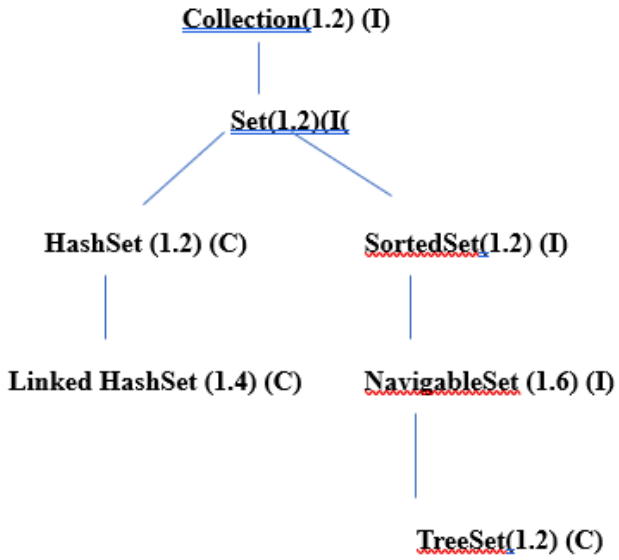
1. It is the Child Interface of Set
2. If we want to represent a group of individual objects whereDuplicate objects are not allowed., but all objects are arranged according to some sorting order. then we should go for SortedSet.

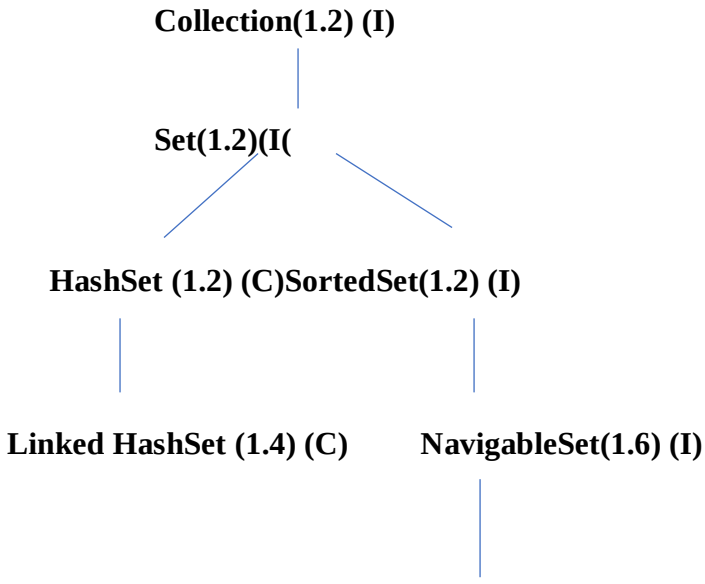


5. NavigableSet Interface

(i).It is the child Interface of SortedSet

(ii). This interface provides support for navigation operations on a sorted set.



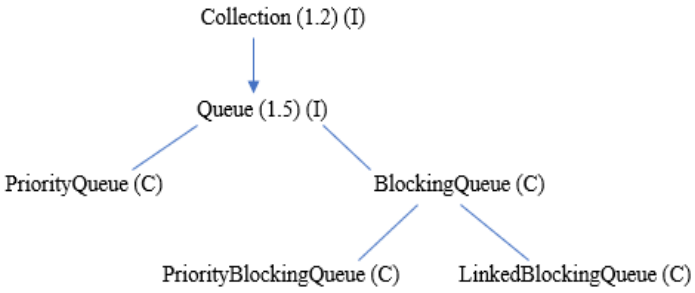


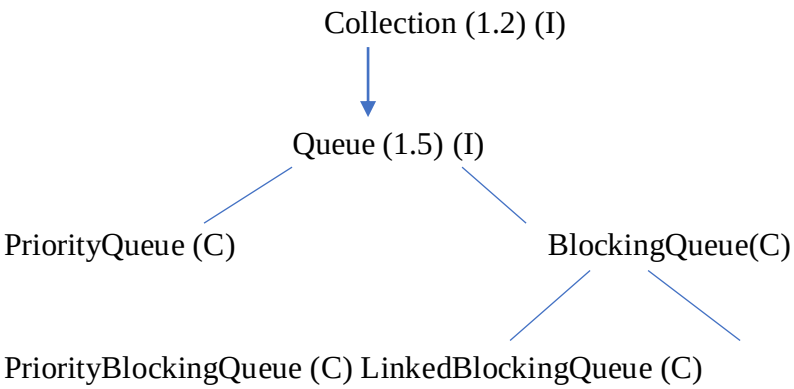
TreeSet(1.2) (C)

6. Queue Interface

(i) It is the child interface of Collection.

(ii) If we want to represent a group of individual objects prior to processing, then we should go for Queue.





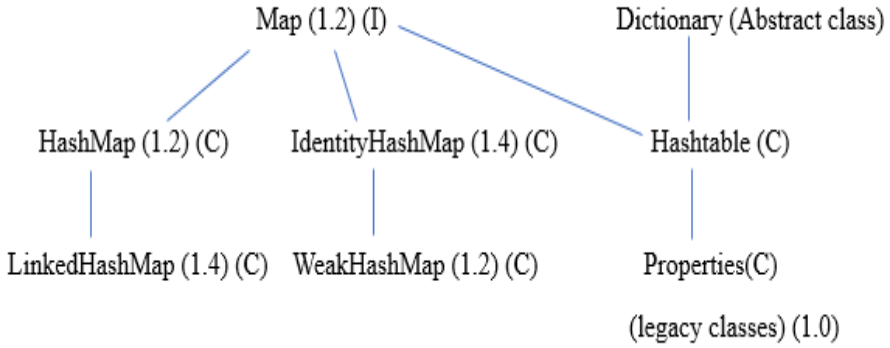
Note: All the above interfaces (**Collection, List, Set, SortedSet, NavigableSet, Queue**) are meant for representing a group of individual objects.

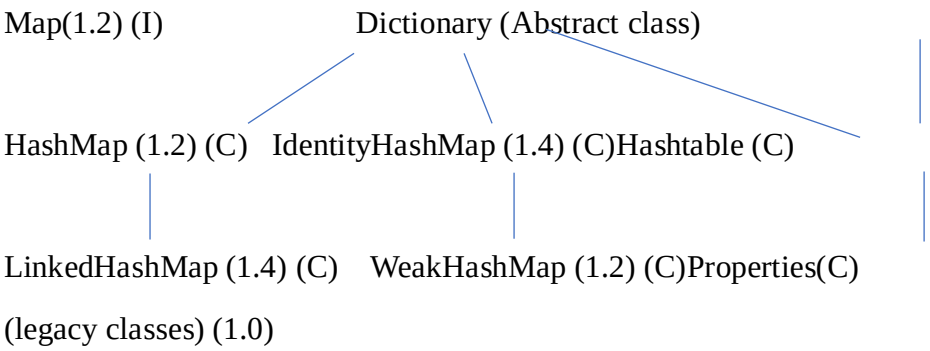
If we want to represent a group of objects as **key–value pairs**, we can't use the above interfaces. Compulsorily, we should go for the **Map** interface.

7) Map Interface

(i) It is **not** a child interface of Collection. If we want to represent a group of objects as **key–value pairs**, then we should go for **Map**.

(ii) Duplicate **keys** are not allowed, but **values** can be duplicated.





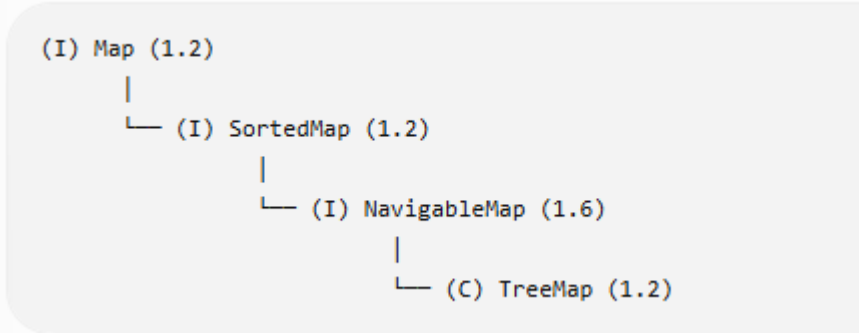
8) SortedMap Interface

(i) It is the child interface of Map.

(ii) If we want to represent a group of objects as key–value pairs according to some sorting order of keys, then we should go for the SortedMap interface.

9) NavigableMap Interface

It is the child interface of SortedMap and defines several methods for navigation.



Cursors

1. Enumeration
2. Iterator
3. ListIterator

Interfaces

1. Comparable
2. Comparator

Utility Classes

1. Collections
2. Arrays

Legacy Interfaces

1.Enumeration

Legacy Abstract Classes

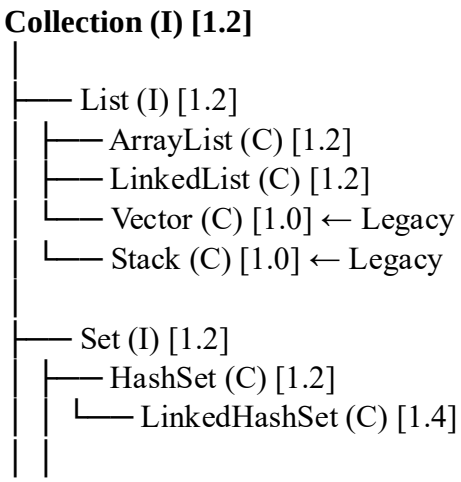
1. Dictionary

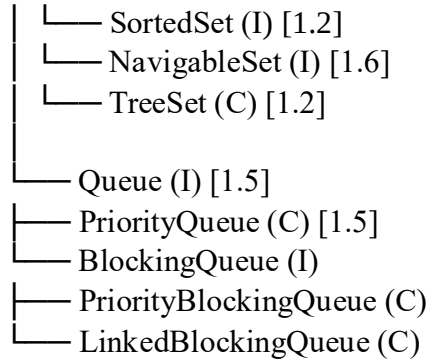
Legacy Concrete Classes

1. Vector
2. Stack
3. Hashtable
4. Properties

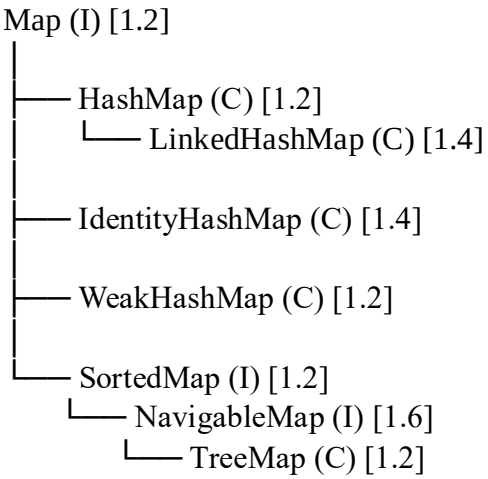
The above '6' legacy characters are 1.0 version.

Java Collections Framework Hierarchy

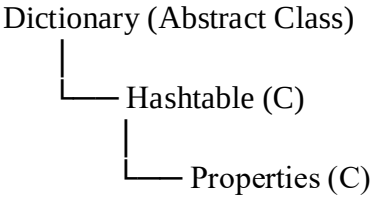




Map Hierarchy



Legacy Hierarchy (Java 1.0)



Legacy Components (Java 1.0)

1. Enumeration (Interface)
2. Dictionary (Abstract Class)
3. Vector (Class)
4. Stack (Class)
5. Hashtable (Class)
6. Properties (Class)

1) Collection Interface

- It is the **base interface** for all collection classes.
- If we want to represent a **group of individual objects as a single entity**, we should go for **Collection**.
- It defines the most common methods that can be applied to any collection object.

Important Methods of Collection Interface

1. **booleanadd(Object o)**
   - Adds the specified object to the collection.
2. **booleanaddAll(Collection c)**
   - Adds all elements of the specified collection to the current collection.
3. **booleanremove(Object o)**
   - Removes the specified object from the collection.
4. **booleanremoveAll(Collection c)**
   - Removes all elements from the current collection that are also present in the specified collection.
5. **void clear()**
   - Removes all elements from the collection.
6. **booleanretainAll(Collection c)**
   - Removes all elements from the current collection **except** those present in the specified collection c.
   - Retains only the common elements.
7. **booleanisEmpty()**
   - Returns true if the collection contains no elements.
   - Otherwise returns false.
8. **int size()**
   - Returns the number of elements present in the collection.
9. **booleancontains(Object o)**
   - Returns true if the specified object is present in the collection.

- Otherwise returns false.

10. booleancontainsAll(Collection c)

- Returns true if all elements of collection c are present in the current collection.
- Otherwise returns false.

11. Object[] toArray()

- Converts the collection into an array and returns it.

12. Iterator iterator()

- Returns an Iterator object for traversing the collection elements one by one.

2) List Interface

→ It is the child interface of Collection.

→ If we want to represent a group of individual objects where insertion order is preserved and duplicates are allowed then we should go for List interface.

l→ [a][b][c][d][a][d]
0 1 2 3 4 5

```
ArrayList l = new ArrayList();
        l.add("a");
        l.add("b");
        l.add("c");
        l.add("d");
        l.add("a");
        l.add("d");
System.out.println(l);
```

Output: [a, b, c, d, a, d]

- Insertion order will be preserved by means of index.
- We can differentiate duplicate elements by using index only.
- Hence, in a List, the index plays a very important role.

List Interface defines the following more specific methods

1. booleanadd(int index, Object o)
2. booleanaddAll(int index, Collection c)
3. Object remove(int index)
4. Object get(int index)
5. Object set(int index, Object new)

It replaces the object present at specified index withprovided new object and

old object will be return.

6. int indexOf(Object o)
7. int lastIndexOf(Object o)
8. ListIteratorlistIterator();

1) ArrayList:

- The underlying data structure is resizable Array (or) growable Array.
- Insertion order is preserved
- duplicate objects are allowed
- Heterogeneous objects are allowed.
- Null insertion is possible.

Constructors:

1. ArrayList l = new ArrayList();

Creates an empty ArrayList object with default initial capacity **10**.Once the ArrayList reaches its maximum capacity, a new ArrayList object will be created
with:

$$\text{new capacity} = (\text{current capacity} \times \frac{3}{2}) + 1$$

Ex:

$$10 \times \frac{3}{2} + 1 = 16$$

2.ArrayList l = new ArrayList(int initialCapacity);

Creates an empty ArrayList object with the specified initial capacity.

3. ArrayList l = new ArrayList(Collection c);

Creates an equivalent ArrayList object for the given collection object, i.e., this constructor is meant for **inter-conversion between collection objects**.

```
import java.util.*;

class ArrayListDemo{
    public static void main(String[] args) {
ArrayList a = new ArrayList();
        //ArrayList<Object> a = new ArrayList<>();
        a.add("A");
        a.add(10);
```

```
        a.add("A");
        a.add(null);
        System.out.println(a); // [A, 10, A, null]
        a.remove(2);
        System.out.println(a); // [A, 10, null]
        a.add(2, "M");
        a.add("N");
        System.out.println(a); // [A, 10, M, null, N]
         }
}
```

Output:

```
C:\nrcm>javac ArrayListDemo.java

C:\nrcm>java ArrayListDemo
[A, 10, A, null]
[A, 10, null]
[A, 10, M, null, N]
```

Usually, we can use collections to hold objects and transporting objects across the network. To provide support for this requirement, every collection class implements **Serializable** and **Cloneable** interfaces.

- **ArrayList** and **Vector** classes implement **RandomAccess** interface, so that any random element we can access with the same speed and same response time.
- **RandomAccess** interface doesn't contain any methods; it is a **marker interface**.

Ex: ArrayList l1 = new ArrayList();

LinkedList l2 = new LinkedList ();
System.out.println(l1 instanceof Serializable); // true
System.out.println(l1 instanceof Cloneable); // true
System.out.println(l1 instanceofRandomAccess); // true
System.out.println(l2 instanceofRandomAccess); // false

Difference between ArrayList& Vector

ArrayList	Vector
1. No method is synchronized.	**1.** All methods are synchronized.

2. Multiple threads can access simultaneously and hence it is **not thread-safe**.	**2.** Multiple threads can't access simultaneously and hence Vector is **thread-safe**.
3. Threads are not required to wait to access ArrayList; hence performance is high.	**3.** Threads are required to wait to access Vector; hence waiting time of threads increases and affects performance.
4. Introduced in **JDK 1.2** and it is a part of the **Collections Framework**.	**4.** Introduced in **JDK 1.0** and it is a **legacy class**.

Vector:

- The underlying data structure is a **resizable array (or growable array)**.
- Insertion order is preserved.
- Duplicate objects are allowed.
- Heterogeneous objects are allowed.
- Implements **Serializable**, **Cloneable**, and **RandomAccess** interfaces.
- Best choice if our frequent operation is **retrieval**.
- Worst choice if our frequent operation is **insertion** (or **deletion**) in the middle.
- All methods are synchronized; hence a **Vector** object is **thread-safe**.

Vector Class Constructors:

1) Vector v = new Vector();

Creates an empty Vector object with default initial capacity **10**.

If it reaches maximum capacity, a new Vector object will be created with:

$$\text{new capacity} = 2 \times \text{current capacity}$$

2) Vector v = new Vector(int initialCapacity);

Creates an empty Vector object with the specified initial capacity.

3) Vector v = new Vector(int initialCapacity, int incrementalCapacity);

Creates an empty Vector object with the specified initial capacity and incremental capacity.

4) Vector v = new Vector(Collection c);

Creates a Vector object containing all elements of the specified collection. This constructor is used for **inter-conversion between collection objects**.

Methods of Vector Class:

For Adding Objects

1. add(Object o); → Collection
2. add(int index, Object o); → List
3. addElement(Object o); → Vector

For Removing Elements

4. remove(Object o); → Collection

5. removeElement(Object o); → Vector
6. remove(int index); → List
7. removeElementAt(int index); → Vector
8. clear(); → Collection
9. removeAllElements(); → Vector

For Accessing Objects

10. Object get(int index); → **List**
11. Object elementAt(int index); → **Vector**
12. Object firstElement(); → **Vector**
13. Object lastElement(); → **Vector**

Other Methods

14. int size();
15. int capacity();
16. Enumeration elements(); → **Vector**

```
import java.util.*;
class VectorDemo{
  public static void main (String [] args) {
        Vector v = new Vector();
          //Vector<Object> v = new Vector<>();
        System.out.println(v.capacity());  // 10
         for (int i = 1; i<= 10; i++){
        v.addElement(i);
         }
        System.out.println(v.capacity());   // 10
        v.addElement("A");
        System.out.println(v.capacity());   // 20 = 10 × 2
        System.out.println(v);              // [1,2,3,4,5,6,7,8,9,10,A]
           }
        }
```

Output:

```
C:\nrcm>javac VectorDemo.java

C:\nrcm>java VectorDemo
10
10
20
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, A]
```

TreeSet:

- The underlying data structure is a balanced tree (Red-Black Tree).
- Duplicate objects are not allowed.
- Insertion order is not preserved.
- But all objects will be inserted according to some sorting order.
- Heterogeneous objects are not allowed, otherwise we will get ClassCastException.
- Null insertion is possible (only once)

TreeSet Constructors:

1. TreeSet t = new TreeSet();

Creates an empty TreeSet object where all the elements will be inserted according to **natural sorting order**.

2. TreeSet t = new TreeSet(Comparator c);

Creates an empty TreeSet object where all the elements will be inserted according to a **customized sorting order** specified by the Comparator object.

3. TreeSet t = new TreeSet(Collection c);

Creates a TreeSet containing the elements of the specified collection, sorted according to the natural ordering of its elements.

4. TreeSet t = new TreeSet(SortedSet s);

Creates a TreeSet containing the same elements and using the same ordering as the specified sorted set.

null acceptance:

- For the empty TreeSet as the first element null insertion is possible. But after inserting that null if we are trying to insert any other, we will get NullPointerException.
- For the non-empty TreeSet if we are trying to insert null, then we will get NullPointerException.

Program:

```
import java.util.*;
class TreeSetDemo {
  public static void main(String[] args) {
        TreeSet t = new TreeSet();
              //TreeSet< Object> t = new TreeSet<>();
        t.add("A");       // t.add(new String("A")); written adjacent
        t.add("a");
        t.add("B");
        t.add("Z");
        t.add("L");
          // t.add(new Integer(10)); ->ClassCastException
          // t.add(null); ->NullPointerException
        System.out.println(t);  // -> [A, B, L, Z, a]
```

```
        }
}
```

Output :

```
C:\nrcm>javac TreeSetDemo.java

C:\nrcm>java TreeSetDemo
[A, B, L, Z, a]
```

- If we are depending on default natural sorting order, compulsory objects should be homogeneous and comparable. Otherwise we will get Runtime Exception saying ClassCastException.
- An object is said to be comparable iff [if and only if] the corresponding class implements Comparable interface.
- All wrapper classes and String class implements Comparable interface. But StringBuffer doesn't implement Comparable.
- All wrapper classes & string class implements comparable interface except these, no predefined class implements Comparable interface.
- Any user defined class must provide interface implementation for compareTo(Obj).

Comparable Interface:

- Comparable interface present in java.lang package.
- This interface contains only one method compareTo() method.

public int compareTo(Object obj);
obj1.compareTo(obj2)

return '-ve' number iff [if and only if] obj1 has to come before obj2.
return '+ve' number iff obj1 has to come after obj2.
return '0' iff obj1 & obj2 both are equal.

S.o.p("A". compareTo("Z")); **'-ve' number**
S.o.p("Z". compareTo("A")); '**+ve' number**
S.o.p("A". compareTo("A")); **'0'**
S.o.p("A". compareTo(new Integer(10))); **R.E: ClassCastException**
S.o.p("A". compareTo(null)); **R.E: NullPointerException**

Whenever we are depending on Natural Sorting order compulsory the objects should be comparable because JVM internally calls compareTo() method to arrange objects in sorting order.

Code:

```
interface Comparable{
  public int compareTo(Object o);
}
```

```
class String implements Comparable {
   public int compareTo(Object obj1) {
     // downcasting is required to corresponding class
      if (obj1 < obj2){ // Note: circled '10' near obj2
return -ve;
}
else if (obj1 > obj2){
 return +ve;
}
else {
return 0; // Note near brackets: "In String this logic won't write"
       }
}
}
```

Rules:

- Two Objects must be homogeneous (similar type).
- The object'scorresponding classes must implement Comparable interface**.**

```
import java.util.*;
class Treesetdemo1{
public static void main(String[] args){
TreeSet t=new TreeSet();
t.add(10);
t.add(20);
t.add(5);
t.add(1);
t.add(3);
//t.add(new Integer(10));    //ClassCastException
//t.add(null);               //NullPointerException
System.out.println(t);
}
}
```

Output :

```
C:\nrcm>javac Treesetdemo1.java

C:\nrcm>java Treesetdemo1
[1, 3, 5, 10, 20]
```

Ex: TreeSet t = new TreeSet();

t.add(20); // 20 becomes the root node.

t.add(0);//0.compareTo(20) **'-ve' number**

//While adding the 2nd element to treeset it will check two conditions above.

Twoconditions are satisfied.

//Current structure order: (0, 20)

t.add(15);

//15.compareTo(0) **'+ve' number**

//15.compareTo(20) '-ve' number

t.add(5);

/* 5.compareTo(0) '+ve' number

5.compareTo(15) '-ve' number*/

t.add(10);

/* 10.compareTo(0) '+ve' number

10.compareTo(5) '+ve' number

10.compareTo(15) '-ve' number */

t.add(5);

/* 5.compareTo(0);

5.compareTo(5); returns 0 (Duplicate detected, will not be added)*/

System.out.println(t); // -> [0, 5, 10, 15, 20]

Sometimes our requirement may notdefault natural sorting order. To define our own customized sorting we should go for "Comparator Interface."

- Comparable is meant for default natural sorting order.
- Comparator is meant for customized sorting order.

Comparator interface:

- This interface present in util package.
- This interface defines the following two methods.

1. public int compare(Object obj1, Object obj2)

- Returns '-ve' iff [if and only if] obj1 has to come before obj2.
- '+ve' iff obj1 has to come after obj2.
- '0' iff obj1 and obj2 are equal.

2. public booleanequals(Object obj);

- Whenever we are implementing Comparator interface compulsory, we should provide implementation for compare () method. equals () method implementation is optional. Because it is already available to our class from object class through inheritance.

```
/*Java program using Treeset for customised sorted order
in this program we must use comparator interface to get descending order
in this program we must use constructor2 to create TreeSet object */
import java.util.*;
class Mycomparator implements Comparator<Integer>{
 public int compare(Integer i1, Integer i2){
     if(i1 < i2)
     return +100;
      else if(i1 > i2)
         return -100;
       else
         return 0;
        }
}
class CustomisedTreeSet{
public static void main(String[] args){
     TreeSet<Integer> t=new TreeSet<>(new Mycomparator());
     t.add(20);
     t.add(11);
     t.add(5);
     t.add(1);
     t.add(3);
     t.add(7);
     System.out.println(t);
}
}
```

Output:

```
C:\nrcm>javac CustomisedTreeSet.java

C:\nrcm>java CustomisedTreeSet
[20, 11, 7, 5, 3, 1]
```

Write a program to insert strings into the TreeSet where the sorting order is

6reverse of alphabetical order

```
/*Java program using Treeset for customised sorted order
in this program we must use comparator interface to get descending order
in this program we must use constructor2 to create TreeSet object */

import java.util.*;
```

```
 // Using <String> generic type to eliminate raw type warnings
class Mycomparator implements Comparator<String> {

   public int compare(String s1, String s2) {
      // Direct comparison without any manual type casting
      int result = s1.compareTo(s2);

      if (result < 0) {
         return +100;
      } else if (result > 0) {
         return -100;
      } else {
         return 0;
      }
   }
}

class StringCustomisedTreeSet {
   public static void main(String[] args) {
      // Declaring TreeSet with generic type <String>
TreeSet<String> t = new TreeSet<>(new Mycomparator());

t.add("ANURADHA");
t.add("ANU");
t.add("ANITHA");
t.add("ANIL");
t.add("SREEDEVI");
t.add("SAILAJA");

System.out.println(t);
   }
}
```

Output:

```
C:\nrcm>javac StringCustomisedTreeSet.java

C:\nrcm>java StringCustomisedTreeSet
[SREEDEVI, SAILAJA, ANURADHA, ANU, ANITHA, ANIL]
```

Note:

- For the predefined comparable classes default natural sorting order is already available. If we are not satisfied with that, then we should go for Comparator to define our own sorting. Ex: String.
- For the predefined non-comparable classes default natural sorting order is not already available, we can define our own sorting by Comparator. Ex: StringBuffer.

- For our own classes by implementing Comparable we can define natural sorting order. The person who is using our class, if is not satisfied with that default natural sorting order, He can define his own customized sorting order by using comparator.

"Comparator" PredefinedComparable classes (String)	"Comparator" Predefined non-comparable classes (StringBuffer)	"comparable", "Comparator" Ourown classes ("Employee")

Comparison between Comparable & Comparator:

Feature	Comparable	Comparator
Purpose	Used to define the **natural sorting order** of a class.	Used to define a **customized sorting order**.
Package	Present in java.lang	Present in java.util
Methods	Contains only **one** method: • compareTo(Object o)	Contains **two** core methods: • compare(Object o1, Object o2) • equals(Object obj)
Implementation	**Implemented by:** String and all Wrapper classes (Integer, Double, etc.).	**No typical predefined classes implement it** (except specific text utilities like Collator and RuleBasedCollator).

Map interface:

- If we want to represent a group of objects as key value pairs then we should go for Map interface.
- Both key & value are objects.
- Duplicate keys are not allowed.
- But values can be duplicated.
- Each key value pair is considered as one entry object.

Ex: Rollno Name

Key	Value
101	Durga
102	Pavan
103	Siva
104	Ravi

Map Hierarchy

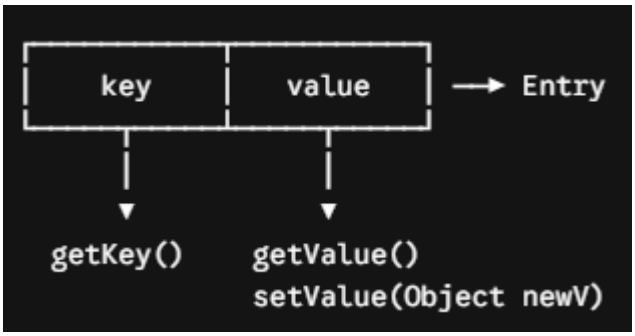
Map interface defines the following methods:

1. **Object put(Object key, Object value);**
    - To add an entry to the Map.
    - If the specified key is already available then old value will be replaced with new value and that old value will be returned.
2. **void putAll(Map m)**
3. **Object get(Object key)**
    - returns the value associated with the specified key. If the key is not available then it returns null.
4. **Object remove(Object key)**
    - It removes the entry associated with specified key and the corresponding value will be return.
5. **booleancontainsKey(Object key);**
6. **booleancontainsValue(Object value);**
7. **booleanisEmpty();**
8. **int size();**
    - returns how many entries are there.
9. **void clear();**
10. **Set keySet();**
11. **Collection values();**
12. **Set entrySet();**

Entry Interface:

- Each key-value pair is called Entry.
- Hence a Map object is considered as a collection of Entries.
- Without existing Map object there is no chance of existing Entry object. Hence interface Entry is defined inside Map as Inner interface.

```
interface Map{
   interface Entry {
      Object getKey();
      Object getValue();
      Object setValue(Object newV);
 }
}
```



HashMap:

- The underlying data structure is Hashtable.
- Insertion order is not preserved and it is based on hash code of the keys.
- Duplicate keys are not allowed.
- Values can be duplicated.
- Heterogeneous objects are allowed for both keys & values.
- Null keysis allowed (only once).
- But we can use null for value any no. of times.

Constructors:

1. **HashMap h = new HashMap();**
   creates an empty HashMap object with default initial capacity 16 and default fill ratio 0.75
2. **HashMap h = new HashMap(int initialCapacity);**
3. **HashMap h = new HashMap(int initialCapacity, float fillRatio);**
   Note under fillRatio: "0 to 1"
4. **HashMap h = new HashMap(Map m);**

We can get synchronized version of HashMap. By using synchronizedMap() method of Collections class.

Program on HashMap

```
import java.util.*;

class HashMapDemo {
   public static void main(String[] args){
      HashMap m = new HashMap();
     // HashMap<String, Integer> m = new HashMap<>();
m.put("chiranjeevi", 700);
```

```
m.put("balaiah", 8000);
m.put("venkatesh", 2000);
m.put("nagarjuna", 500);
System.out.println("Initial Map: " +m);
System.out.println(m.put("chiranjeevi", 1000));
        Set s = m.keySet();
        // Set<String> s = m.keySet();
System.out.println("Keys: " + s);
          Collection c = m.values();
         // Collection<Integer> c = m.values();
System.out.println(c);
        Set s1 = m.entrySet();
        Iterator itr = s1.iterator();
        // Set<Map.Entry<String, Integer>> s1 = m.entrySet();
        // Iterator<Map.Entry<String, Integer>>itr = s1.iterator();

        while (itr.hasNext()){
Map.Entry<String, Integer> m1 = itr.next(); // No manual casting needed!
         // Map.Entry m1 = (Map.Entry) itr.next();
System.out.println(m1.getKey() + "..." + m1.getValue());
        }
itr = s1.iterator();
        while (itr.hasNext()){
// Map.Entry m1 = (Map.Entry) itr.next();
Map.Entry<String, Integer> m1 = itr.next();
System.out.println(m1.getKey() + "..." + m1.getValue());
          if (m1.getKey().equals("nagarjuna")){
              m1.setValue(10000);
           }
        }
System.out.println(m);
   }
}
```

Output:

```
C:\nrcm>javac HashMapDemo.java

C:\nrcm>java HashMapDemo
Initial Map: {balaiah=8000, chiranjeevi=700, venkatesh=2000, nagarjuna=500}
700
Keys: [balaiah, chiranjeevi, venkatesh, nagarjuna]
[8000, 1000, 2000, 500]
balaiah...8000
chiranjeevi...1000
venkatesh...2000
nagarjuna...500
balaiah...8000
chiranjeevi...1000
venkatesh...2000
nagarjuna...500
{balaiah=8000, chiranjeevi=1000, venkatesh=2000, nagarjuna=10000}
```

Difference between HashMap &Hashtable:

HashMap	Hashtable
1. No method is synchronized. 2. It is not thread safe 3. Performance is high. 4. Null is allowed for both key & value 5. Introduced in 1.2 version & non-legacy.	1. Every method is synchronized. 2. It is thread safe. 3. Performance is low. 4. null is not allowed for both key & values. Otherwise NullPointerException. 5. Introduced in 1.0 version & legacy.

Hashtable:

- The underlying data structure is Hashtable.
- Insertion order is not preserved and it is based on **hashcode of the keys**.
- Duplicate keys are not allowed, but values can be duplicated.
- Heterogeneous objects are allowed for both key and values.
- Null is not allowed for both key & value. Otherwise, NullPointerException.
- Every method is synchronized and hence Hashtable object is thread safe.
- Best suitable if our frequent operation is search operation.

Constructors:

1. **Hashtable h = new Hashtable();**
    - creates an empty Hashtable object with default initial capacity '11' and default fill ratio 0.75.
2. **Hashtable h = new Hashtable(int initialCapacity);**
3. **Hashtable h = new Hashtable(int initialCapacity, float fillRatio);**

4. **Hashtable h = new Hashtable(Map m);**

Program on HashTable

```
import java.util.*;
class Temp {
   int i;
Temp(int i) {
this.i = i;
   }
   public int hashCode(){
      return i;
   }
   public String toString() {
      return i + "";
   }
}
class HashtableDemo{
   public static void main(String[] args){
Hashtable h = new Hashtable();
//Hashtable<Temp, String> h = new Hashtable<>();
h.put(new Temp(5), "A");
h.put(new Temp(2), "B");
h.put(new Temp(6), "C");
h.put(new Temp(15), "D");
h.put(new Temp(23), "E");
h.put(new Temp(16), "F");
      // h.put("durga", null); // NullPointerException
System.out.println(h);
   }
}
```

Output:

```
C:\nrcm>javac HashtableDemo.java

C:\nrcm>java HashtableDemo
{6=C, 16=F, 5=A, 15=D, 2=B, 23=E}
```

Cursors:

3 Cursors of Java: (for navigation & get elements only) If u want to get objects one by onewe required cursor.

There are three types of cursors available in Java.

1. Enumeration

2. Iterator
3. ListIterator

1.**Enumeration**

(It is used for Vector & Stack only)

- It is the legacy interface introduced in 1.0 version.
- Enumeration interface contains the following two methods:
    1. public booleanhasMoreElements();
    2. public Object nextElement();

```
import java.util.Vector;
import java.util.Enumeration; // Corrected interface name
class EnumerationDemo{
   public static void main(String[] args) {
      Vector v = new Vector();
      //Vector<Integer> v = new Vector<>();
      for (int i = 0; i<= 10; i++){
v.addElement(i);
      }
      Enumeration<Integer> e = v.elements();
      while (e.hasMoreElements()){
         Integer I = (Integer)e.nextElement();
         if (I % 2 == 0){
System.out.println(I);
         }
      }
System.out.println(v);
   }
}
```

Output:

```
C:\nrcm>javac EnumerationDemo.java

C:\nrcm>java EnumerationDemo
0
2
4
6
8
10
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Limitations of Enumeration:

- **Enumeration is not a universal cursor**, it is applicable only for legacy classes.
- While iterating objects **we will have only read permissions** but **we can't remove** useless objects.

2.Iterator interface: (Universal cursor)

- It is introduced in 1.2 version.
- This interface defines the following three methods:
    1. public booleanhasNext();
    2. public Object next();
    3. public void remove();
- Iterator is the **universal cursor** and we can apply for any collection object.

```
import java.util.*;
class IteratorDemo{
   public static void main(String[] args){
      //  ArrayList l = new ArrayList();
ArrayList<Integer> l = new ArrayList<>();
      for (int i = 0; i<= 10; i++){
l.add(i);
      }
System.out.println(l);
      Iterator itr = l.iterator(); // Creation of Iterator
      //Iterator<Integer>itr = l.iterator();
      while (itr.hasNext()){
        Integer I = (Integer) itr.next();
        if (I % 2 == 0){
System.out.print(I+" ");
        }
        else {
itr.remove();
        }
      }
System.out.println("Modified List: " + l);
       }
}
```

Output:

```
C:\nrcm>javac IteratorDemo.java

C:\nrcm>java IteratorDemo
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
0 2 4 6 8 10
```

Limitations of Iterator:

1. While iterating the elements we can perform **only read and remove operations** and we **can't perform adding new objects and replacing existing object**.
2. **Iterator** and **Enumeration** will always move towards **forward direction**. These are **not bidirectional cursors**.

ListIterator:

- It is the child interface of Iterator
- while iterating the objects we can perform read, remove, replace and addition of new objects also.
- ListIterator is a Bi-directional cursor we can move either to the forward direction (or) to the backward direction.

ListIterator interface defines the following '9' methods:

1. public booleanhasNext();
2. public booleanhasPrevious();
3. public Object next();
4. public Object previous();
5. public int nextIndex();
6. public int previousIndex();
7. public void remove();
8. public void add(Object o);
9. public void set(Object new);

Program:

```
import java.util.*;

class ListIteratorExample1 {
   public static void main(String[] args) {

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

l.add("balakrishna");
l.add("venki");
l.add("chiru");
l.add("nag");
System.out.println("Original List: " + l);
ListIterator<String>ltr = l.listIterator();
```

```
        while (ltr.hasNext()) {
            String s =(String)ltr.next();
            if (s.equals("venki")) {
ltr.remove();
            }
            else if (s.equals("nag")) {
ltr.add("chaitu");
            }
            else if (s.equals("chiru")) {
ltr.set("charan");
            }
        }
System.out.println("Modified List: " + l);
    }
}
```

Output:

```
C:\nrcm>javac ListIteratorExample1.java

C:\nrcm>java ListIteratorExample1
Original List: [balakrishna, venki, chiru, nag]
Modified List: [balakrishna, charan, nag, chaitu]
```

Note: ListIterator is the most powerful cursor but its limitation is it is applicable only for list implemented class objects.

Comparison Table of Three Cursors:

S.No	Property	Enumeration	Iterator	ListIterator
1	**Is it legacy?**	Yes	No	No
2	**Applicable**	Only for legacy classes	For any collection object	Only for List objects
3	**Movement**	Single direction (Forward)	Single direction (Forward)	Bi-directional (Forward & Backward)
4	**How to get**	By using elements()	By using iterator()	By using listIterator()
5	**Accessibility**	Only read	Read and remove	Read, remove, replace, add
6	**Methods**	hasMoreElements() nextElement()	hasNext() next()	hasNext() hasPrevious()

			remove()	next() previous() nextIndex() previousIndex() remove() add(Object o) set(Object new)