

Synchronization in Java

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.

Why use Synchronization

The synchronization is mainly used to

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

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

Thread Synchronization

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

1. Mutual Exclusive
 1. Synchronized method.
 2. Synchronized block.
 3. static synchronization.

2. Cooperation (Inter-thread communication in java)
-

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

1. by synchronized method
 2. by synchronized block
 3. by static synchronization
-

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has an lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package `java.util.concurrent.locks` contains several lock implementations.

Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

1. `class Table{`
2. `void printTable(int n){//method not synchronized`
3. `for(int i=1;i<=5;i++){`
4. `System.out.println(n*i);`
5. `try{`
6. `Thread.sleep(400);`
7. `}catch(Exception e){System.out.println(e);}`

```
8.     }
9.
10.  }
11.  }
12.
13.  class MyThread1 extends Thread{
14.      Table t;
15.      MyThread1(Table t){
16.          this.t=t;
17.      }
18.      public void run(){
19.          t.printTable(5);
20.      }
21.
22.  }
23.  class MyThread2 extends Thread{
24.      Table t;
25.      MyThread2(Table t){
26.          this.t=t;
27.      }
28.      public void run(){
29.          t.printTable(100);
30.      }
31.  }
32.
33.  class TestSynchronization1{
34.      public static void main(String args[]){
35.          Table obj = new Table();//only one object
```

```
36. MyThread1 t1=new MyThread1(obj);
37. MyThread2 t2=new MyThread2(obj);
38. t1.start();
39. t2.start();
40. }
41. }
```

Output: 5

```
100
10
200
15
300
20
400
25
500
```

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.

1. `//example of java synchronized method`
2. `class Table{`
3. `synchronized void printTable(int n){//synchronized method`
4. `for(int i=1;i<=5;i++){`
5. `System.out.println(n*i);`
6. `try{`
7. `Thread.sleep(400);`

```
8.     }catch(Exception e){System.out.println(e);}
9.     }
10.
11. }
12. }
13.
14. class MyThread1 extends Thread{
15.     Table t;
16.     MyThread1(Table t){
17.         this.t=t;
18.     }
19.     public void run(){
20.         t.printTable(5);
21.     }
22.
23. }
24. class MyThread2 extends Thread{
25.     Table t;
26.     MyThread2(Table t){
27.         this.t=t;
28.     }
29.     public void run(){
30.         t.printTable(100);
31.     }
32. }
33.
34. public class TestSynchronization2{
35.     public static void main(String args[]){
```

```
36. Table obj = new Table();//only one object
37. MyThread1 t1=new MyThread1(obj);
38. MyThread2 t2=new MyThread2(obj);
39. t1.start();
40. t2.start();
41. }
42. }
```

Output: 5

```
10
15
20
25
100
200
300
400
500
```

Example of synchronized method by using anonymous class

In this program, we have created the two threads by anonymous class, so less coding is required.

```
1. //Program of synchronized method by using anonymous class
2. class Table{
3.     synchronized void printTable(int n){//synchronized method
4.         for(int i=1;i<=5;i++){
5.             System.out.println(n*i);
6.             try{
7.                 Thread.sleep(400);
8.             }catch(Exception e){System.out.println(e);}
```

```
9.     }  
10.  
11. }  
12. }  
13.  
14. public class TestSynchronization3{  
15. public static void main(String args[]){  
16. final Table obj = new Table();//only one object  
17.  
18. Thread t1=new Thread(){  
19. public void run(){  
20. obj.printTable(5);  
21. }  
22. };  
23. Thread t2=new Thread(){  
24. public void run(){  
25. obj.printTable(100);  
26. }  
27. };  
28.  
29. t1.start();  
30. t2.start();  
31. }  
32. }
```

Output: 5

```
10  
15  
20  
25  
100
```

200
300
400
500

Synchronized Block in Java

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.

Points to remember for Synchronized block

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.

Syntax to use synchronized block

1. **synchronized** (object reference expression) {
2. //code block
3. }

Example of synchronized block

Let's see the simple example of synchronized block.

Program of synchronized block

1. **class** Table{
- 2.
3. **void** printTable(**int** n){
4. **synchronized(this){**//synchronized block


```
5.     for(int i=1;i<=5;i++){
6.         System.out.println(n*i);
7.         try{
8.             Thread.sleep(400);
9.         }catch(Exception e){System.out.println(e);}
10.    }
11. }
12. }//end of the method
13. }
14.
15. class MyThread1 extends Thread{
16.     Table t;
17.     MyThread1(Table t){
18.         this.t=t;
19.     }
20.     public void run(){
21.         t.printTable(5);
22.     }
23.
24. }
25. class MyThread2 extends Thread{
26.     Table t;
27.     MyThread2(Table t){
28.         this.t=t;
29.     }
30.     public void run(){
31.         t.printTable(100);
32.     }
```

```
33. }  
34.  
35. public class TestSynchronizedBlock1{  
36. public static void main(String args[]){  
37. Table obj = new Table();//only one object  
38. MyThread1 t1=new MyThread1(obj);  
39. MyThread2 t2=new MyThread2(obj);  
40. t1.start();  
41. t2.start();  
42. }  
43. }
```

Test it Now

Output:5

```
10  
15  
20  
25  
100  
200  
300  
400  
500
```

Same Example of synchronized block by using anonymous class:

//Program of synchronized block by using anonymous class

```
1. class Table{  
2.  
3. void printTable(int n){  
4. synchronized(this){//synchronized block
```

```
5.     for(int i=1;i<=5;i++){
6.         System.out.println(n*i);
7.         try{
8.             Thread.sleep(400);
9.         }catch(Exception e){System.out.println(e);}
10.    }
11. }
12. }//end of the method
13. }
14.
15. public class TestSynchronizedBlock2{
16.     public static void main(String args[]){
17.         final Table obj = new Table();//only one object
18.
19.         Thread t1=new Thread(){
20.             public void run(){
21.                 obj.printTable(5);
22.             }
23.         };
24.         Thread t2=new Thread(){
25.             public void run(){
26.                 obj.printTable(100);
27.             }
28.         };
29.
30.         t1.start();
31.         t2.start();
32.     }
```

33. }

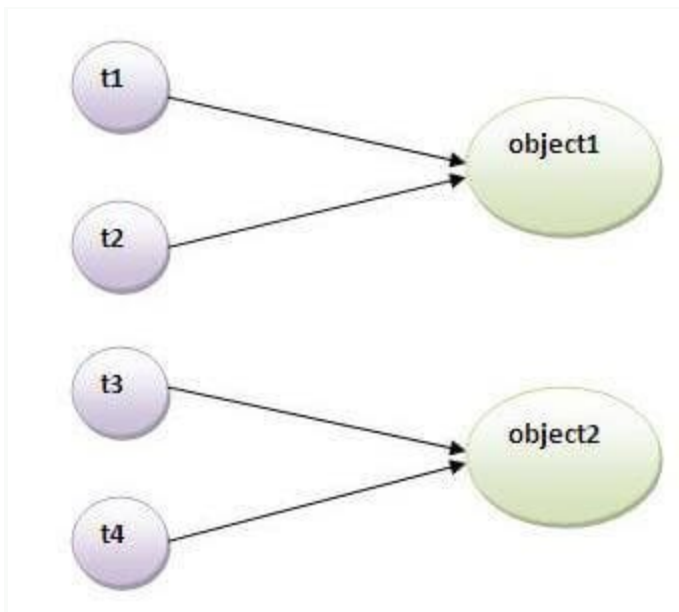
Test it Now

Output:5

10
15
20
25
100
200
300
400
500

Static Synchronization

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



Problem without static synchronization

Suppose there are two objects of a shared class(e.g. Table) named object1 and object2. In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refer to a common object that has a single lock. But there can be interference between t1 and t3 or t2

and t4 because t1 acquires another lock and t3 acquires another lock. I want no interference between t1 and t3 or t2 and t4. Static synchronization solves this problem.

Example of static synchronization

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

```
1.  class Table{
2.
3.  synchronized static void printTable(int n){
4.      for(int i=1;i<=10;i++){
5.          System.out.println(n*i);
6.          try{
7.              Thread.sleep(400);
8.          }catch(Exception e){}
9.      }
10. }
11. }
12.
13. class MyThread1 extends Thread{
14. public void run(){
15.     Table.printTable(1);
16. }
17. }
18.
19. class MyThread2 extends Thread{
20. public void run(){
21.     Table.printTable(10);
22. }
23. }
24.
```

```
25. class MyThread3 extends Thread{
26. public void run(){
27. Table.printTable(100);
28. }
29. }
30.
31.
32.
33.
34. class MyThread4 extends Thread{
35. public void run(){
36. Table.printTable(1000);
37. }
38. }
39.
40. public class TestSynchronization4{
41. public static void main(String t[]){
42. MyThread1 t1=new MyThread1();
43. MyThread2 t2=new MyThread2();
44. MyThread3 t3=new MyThread3();
45. MyThread4 t4=new MyThread4();
46. t1.start();
47. t2.start();
48. t3.start();
49. t4.start();
50. }
51. }
```

Test it Now

Output: 1

2

3

4

5

6

7

8

9

10

10

20

30

40

50

60

70

80

90

100

100

200

300

400

500

600

700

800

900

1000

1000

2000

3000

4000
5000
6000
7000
8000
9000
10000

Same example of static synchronization by anonymous class

In this example, we are using anonymous class to create the threads.

```
1.  class Table{
2.
3.  synchronized static void printTable(int n){
4.      for(int i=1;i<=10;i++){
5.          System.out.println(n*i);
6.          try{
7.              Thread.sleep(400);
8.          }catch(Exception e){}
9.      }
10. }
11. }
12.
13. public class TestSynchronization5 {
14.     public static void main(String[] args) {
15.
16.         Thread t1=new Thread(){
```



```
17.     public void run(){
18.         Table.printTable(1);
19.     }
20. };
21.
22. Thread t2=new Thread(){
23.     public void run(){
24.         Table.printTable(10);
25.     }
26. };
27.
28. Thread t3=new Thread(){
29.     public void run(){
30.         Table.printTable(100);
31.     }
32. };
33.
34. Thread t4=new Thread(){
35.     public void run(){
36.         Table.printTable(1000);
37.     }
38. };
39. t1.start();
40. t2.start();
41. t3.start();
42. t4.start();
43.
44. }
```

45. }

Test it Now

Output: 1

2

3

4

5

6

7

8

9

10

10

20

30

40

50

60

70

80

90

100

100

200

300

400

500

600

700

800

900

1000
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000

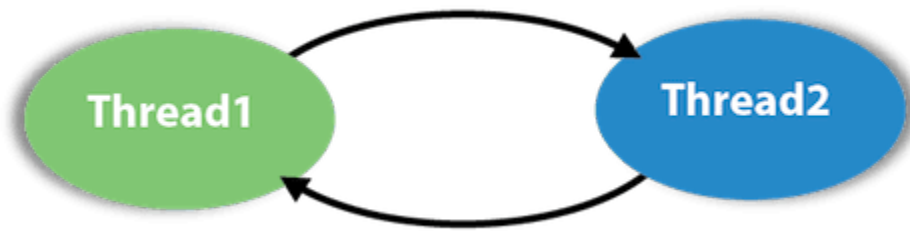
Synchronized block on a class lock:

The block synchronizes on the lock of the object denoted by the reference .class name .class. A static synchronized method printTable(int n) in class Table is equivalent to the following declaration:

```
1. static void printTable(int n) {  
2.     synchronized (Table.class) {    // Synchronized block on class A  
3.         // ...  
4.     }  
5. }
```

Deadlock in java

Deadlock in java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



Example of Deadlock in java

```
1. public class TestDeadlockExample1 {
2.     public static void main(String[] args) {
3.         final String resource1 = "ratan jaiswal";
4.         final String resource2 = "vimal jaiswal";
5.         // t1 tries to lock resource1 then resource2
6.         Thread t1 = new Thread() {
7.             public void run() {
8.                 synchronized (resource1) {
9.                     System.out.println("Thread 1: locked resource 1");
10.
11.                     try { Thread.sleep(100);} catch (Exception e) {}
12.
13.                     synchronized (resource2) {
14.                         System.out.println("Thread 1: locked resource 2");
15.                     }
16.                 }
17.             }
18.         };
19.
20.         // t2 tries to lock resource2 then resource1
21.         Thread t2 = new Thread() {
```

```

22.     public void run() {
23.         synchronized (resource2) {
24.             System.out.println("Thread 2: locked resource 2");
25.
26.             try { Thread.sleep(100);} catch (Exception e) {}
27.
28.         synchronized (resource1) {
29.             System.out.println("Thread 2: locked resource 1");
30.         }
31.     }
32. }
33. };
34.
35.
36. t1.start();
37. t2.start();
38. }
39. }
40.

```

Output: Thread 1: locked resource 1

Thread 2: locked resource 2

Inter-thread communication in Java

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


Cooperation (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()`

1) `wait()` method

Causes current thread to release the lock and wait until either another thread invokes the `notify()` method or the `notifyAll()` method for this object, or a specified amount of time has elapsed.



volume is gedempt

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
<code>public final void wait()throws InterruptedException</code>	waits until object is notified.
<code>public final void wait(long timeout)throws InterruptedException</code>	waits for the specified amount of time.

2) `notify()` method

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

Syntax:

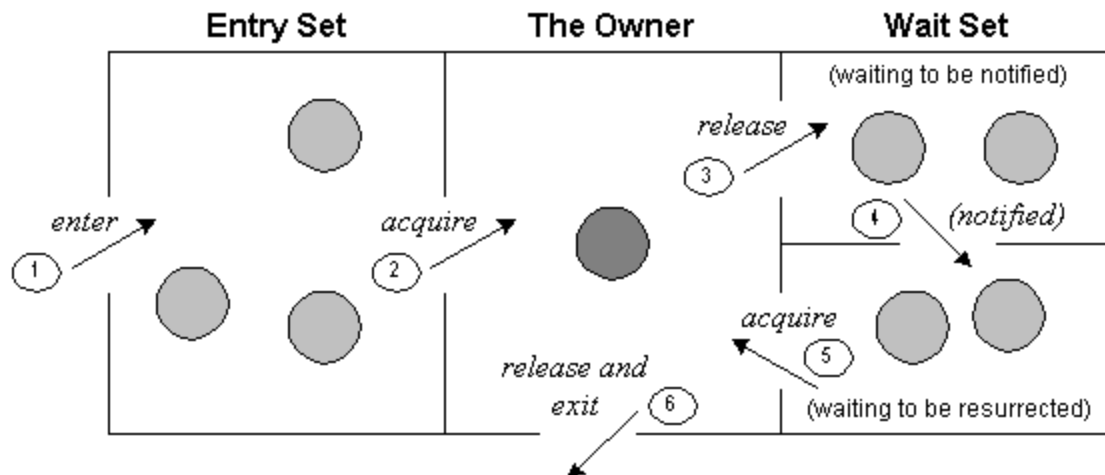
```
public final void notify()
```

3) `notifyAll()` method

Wakes up all threads that are waiting on this object's monitor. Syntax:

```
public final void notifyAll()
```

Understanding the process of inter-thread communication



The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
 2. Lock is acquired by one thread.
 3. Now thread goes to waiting state if you call `wait()` method on the object. Otherwise it releases the lock and exits.
 4. If you call `notify()` or `notifyAll()` method, thread moves to the notified state (runnable state).
 5. Now thread is available to acquire lock.
 6. After completion of the task, thread releases the lock and exits the monitor state of the object.
-

Why `wait()`, `notify()` and `notifyAll()` methods are defined in Object class not Thread class?

It is because they are related to lock and object has a lock.

Difference between wait and sleep?

Let's see the important differences between wait and sleep methods.

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
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

Let's see the simple example of inter thread communication.

```
1. class Customer{
2.     int amount=10000;
3.
4.     synchronized void withdraw(int amount){
5.         System.out.println("going to withdraw...");
6.
7.         if(this.amount<amount){
8.             System.out.println("Less balance; waiting for deposit...");
9.             try{wait();}catch(Exception e){}
10.        }
11.        this.amount-=amount;
12.        System.out.println("withdraw completed...");
13.    }
```



```
14.  
15. synchronized void deposit(int amount){  
16. System.out.println("going to deposit...");  
17. this.amount+=amount;  
18. System.out.println("deposit completed... ");  
19. notify();  
20. }  
21. }  
22.  
23. class Test{  
24. public static void main(String args[]){  
25. final Customer c=new Customer();  
26. new Thread(){  
27. public void run(){c.withdraw(15000);}  
28. }.start();  
29. new Thread(){  
30. public void run(){c.deposit(10000);}  
31. }.start();  
32.  
33. }}
```

Output: going to withdraw...

Less balance; waiting for deposit...

going to deposit...

deposit completed...

withdraw completed

Interrupting a Thread:

If any thread is in sleeping or waiting state (i.e. `sleep()` or `wait()` is invoked), calling the `interrupt()` method on the thread, breaks out the sleeping or waiting state throwing `InterruptedException`. If the thread is not in the sleeping or waiting state, calling the `interrupt()` method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true. Let's first see the methods provided by the `Thread` class for thread interruption.

The 3 methods provided by the `Thread` class for interrupting a thread

- `public void interrupt()`
 - `public static boolean interrupted()`
 - `public boolean isInterrupted()`
-

Example of interrupting a thread that stops working

In this example, after interrupting the thread, we are propagating it, so it will stop working. If we don't want to stop the thread, we can handle it where `sleep()` or `wait()` method is invoked. Let's first see the example where we are propagating the exception.

1. `class TestInterruptingThread1 extends Thread{`
2. `public void run(){`
3. `try{`
4. `Thread.sleep(1000);`
5. `System.out.println("task");`
6. `}catch(InterruptedException e){`

```
7.  throw new RuntimeException("Thread interrupted..." + e);
8.  }
9.
10. }
11.
12. public static void main(String args[]){
13.     TestInterruptingThread1 t1=new TestInterruptingThread1();
14.     t1.start();
15.     try{
16.         t1.interrupt();
17.     }catch(Exception e){System.out.println("Exception handled " + e);}
18.
19. }
20. }
```

Test it Now

[download this example](#)

Output:Exception in thread-0

```
java.lang.RuntimeException: Thread interrupted...
java.lang.InterruptedException: sleep interrupted
at A.run(A.java:7)
```

Example of interrupting a thread that doesn't stop working

In this example, after interrupting the thread, we handle the exception, so it will break out the sleeping but will not stop working.

```
1. class TestInterruptingThread2 extends Thread{
2.     public void run(){
3.         try{
4.             Thread.sleep(1000);
5.             System.out.println("task");
6.         }catch(InterruptedException e){
7.             System.out.println("Exception handled "+e);
8.         }
9.         System.out.println("thread is running...");
10.    }
11.
12.    public static void main(String args[]){
13.        TestInterruptingThread2 t1=new TestInterruptingThread2();
14.        t1.start();
15.
16.        t1.interrupt();
17.
18.    }
19. }
```

Test it Now

[download this example](#)

Output:Exception handled

java.lang.InterruptedException: sleep interrupted

thread is running...

Example of interrupting thread that behaves normally

If thread is not in sleeping or waiting state, calling the interrupt() method sets the interrupted flag to true that can be used to stop the thread by the java programmer later.

```
1. class TestInterruptingThread3 extends Thread{
2.
3. public void run(){
4. for(int i=1;i<=5;i++)
5. System.out.println(i);
6. }
7.
8. public static void main(String args[]){
9. TestInterruptingThread3 t1=new TestInterruptingThread3();
10. t1.start();
11.
12. t1.interrupt();
13.
14. }
15. }
```

Test it Now

Output:1

2
3
4
5

What about isInterrupted and interrupted method?

The `isInterrupted()` method returns the interrupted flag either true or false. The static `interrupted()` method returns the interrupted flag after that it sets the flag to false if it is true.

```
1. public class TestInterruptingThread4 extends Thread{
2.
3.     public void run(){
4.         for(int i=1;i<=2;i++){
5.             if(Thread.interrupted()){
6.                 System.out.println("code for interrupted thread");
7.             }
8.             else{
9.                 System.out.println("code for normal thread");
10.            }
11.
12.        } //end of for loop
13.    }
14.
15.    public static void main(String args[]){
16.
17.        TestInterruptingThread4 t1=new TestInterruptingThread4();
18.        TestInterruptingThread4 t2=new TestInterruptingThread4();
19.
20.        t1.start();
21.        t1.interrupt();
22.
23.        t2.start();
24.
25.    }
```

26. }

Test it Now

Output:Code for interrupted thread

code for normal thread

code for normal thread

code for normal thread

Reentrant Monitor in Java

According to Sun Microsystems, **Java monitors are reentrant** means java thread can reuse the same monitor for different synchronized methods if method is called from the method.

Advantage of Reentrant Monitor

It eliminates the possibility of single thread deadlocking

Let's understand the java reentrant monitor by the example given below:

```
1. class Reentrant {
2.     public synchronized void m() {
3.         n();
4.         System.out.println("this is m() method");
5.     }
6.     public synchronized void n() {
7.         System.out.println("this is n() method");
8.     }
9. }
```

In this class, m and n are the synchronized methods. The m() method internally calls the n() method.

Now let's call the m() method on a thread. In the class given below, we are creating thread using anonymous class.

```
1. public class ReentrantExample{
2.     public static void main(String args[]){
3.         final ReentrantExample re=new ReentrantExample();
4.
5.         Thread t1=new Thread(){
6.             public void run(){
7.                 re.m();//calling method of Reentrant class
8.             }
9.         };
10.        t1.start();
11.    }}
```

Test it Now

Output: this is n() method

this is m() method

Serialization and Deserialization in Java

1. [Serialization](#)
2. [Serializable Interface](#)
3. [Example of Serialization](#)
4. [Example of Deserialization](#)
5. [Serialization with Inheritance](#)
6. [Externalizable interface](#)
7. [Serialization and static data member](#)

Serialization in Java is a mechanism of *writing the state of an object into a byte-stream*. It is mainly used in Hibernate, RMI, JPA, EJB and JMS technologies.

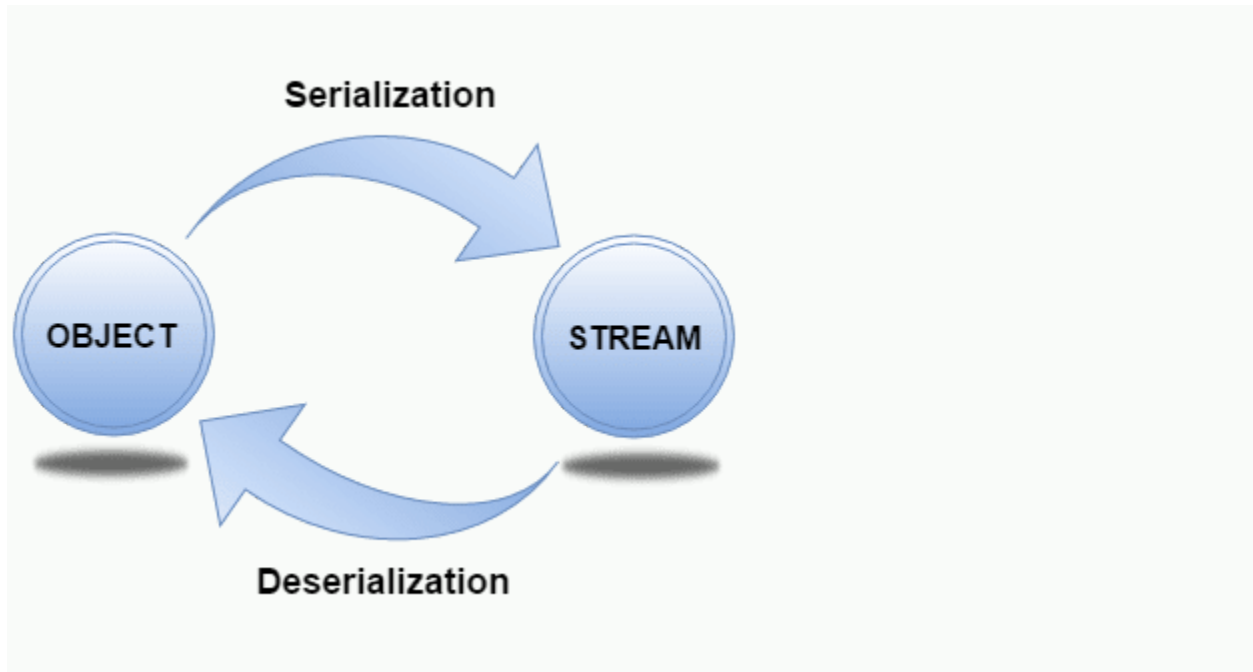
The reverse operation of serialization is called *deserialization* where byte-stream is converted into an object. The serialization and deserialization process is platform-independent, it means you can serialize an object in a platform and deserialize in different platform.

For serializing the object, we call the **writeObject()** method of *ObjectOutputStream*, and for deserialization we call the **readObject()** method of *ObjectInputStream* class.

We must have to implement the *Serializable* interface for serializing the object.

Advantages of Java Serialization

It is mainly used to travel object's state on the network (which is known as marshaling).



java.io.Serializable interface

Serializable is a marker interface (has no data member and method). It is used to "mark" Java classes so that the objects of these classes may get a certain capability. The Cloneable and Remote are also marker interfaces.

It must be implemented by the class whose object you want to persist.

The String class and all the wrapper classes implement the *java.io.Serializable* interface by default.

Let's see the example given below:

```
1. import java.io.Serializable;
2. public class Student implements Serializable{
3.     int id;
4.     String name;
5.     public Student(int id, String name) {
6.         this.id = id;
7.         this.name = name;
```

```
8.  }  
9.  }
```

In the above example, Student class implements Serializable interface. Now its objects can be converted into stream.

ObjectOutputStream class

The ObjectOutputStream class is used to write primitive data types, and Java objects to an OutputStream. Only objects that support the java.io.Serializable interface can be written to streams.

Constructor

1) public ObjectOutputStream(OutputStream out) throws IOException {}	creates an ObjectOutputStream that writes to the specified OutputStream.
---	--

Important Methods

Method	Description
1) public final void writeObject(Object obj) throws IOException {}	writes the specified object to the ObjectOutputStream.
2) public void flush() throws IOException {}	flushes the current output stream.
3) public void close() throws IOException {}	closes the current output stream.

ObjectInputStream class

An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

Constructor

1) public ObjectInputStream(InputStream in) throws IOException {}	creates an ObjectInputStream that reads from the specified InputStream.
--	---

Important Methods

Method	Description
1) public final Object readObject() throws IOException, ClassNotFoundException {}	reads an object from the input stream.
2) public void close() throws IOException {}	closes ObjectInputStream.

Example of Java Serialization

In this example, we are going to serialize the object of Student class. The writeObject() method of ObjectOutputStream class provides the functionality to serialize the object. We are saving the state of the object in the file named f.txt.

```

1. import java.io.*;
2. class Persist{
3.     public static void main(String args[]){
4.         try{
5.             //Creating the object
6.             Student s1 =new Student(211,"ravi");
7.             //Creating stream and writing the object
8.             FileOutputStream fout=new FileOutputStream("f.txt");
9.             ObjectOutputStream out=new ObjectOutputStream(fout);
10.            out.writeObject(s1);
11.            out.flush();
12.            //closing the stream

```

```
13. out.close();
14. System.out.println("success");
15. }catch(Exception e){System.out.println(e);}
16. }
17. }
```

success

[download this example of serialization](#)

Example of Java Deserialization

Deserialization is the process of reconstructing the object from the serialized state. It is the reverse operation of serialization. Let's see an example where we are reading the data from a deserialized object.

```
1. import java.io.*;
2. class Depersist{
3.     public static void main(String args[]){
4.         try{
5.             //Creating stream to read the object
6.             ObjectInputStream in=new ObjectInputStream(new FileInputStream("f.txt"));
7.             Student s=(Student)in.readObject();
8.             //printing the data of the serialized object
9.             System.out.println(s.id+" "+s.name);
10.            //closing the stream
11.            in.close();
12.        }catch(Exception e){System.out.println(e);}
13.    }
14. }
```

211 ravi

[download this example of deserialization](#)

Java Serialization with Inheritance (IS-A Relationship)

If a class implements serializable then all its sub classes will also be serializable. Let's see the example given below:

```
1. import java.io.Serializable;
2. class Person implements Serializable{
3.     int id;
4.     String name;
5.     Person(int id, String name) {
6.         this.id = id;
7.         this.name = name;
8.     }
9. }

1. class Student extends Person{
2.     String course;
3.     int fee;
4.     public Student(int id, String name, String course, int fee) {
5.         super(id,name);
6.         this.course=course;
7.         this.fee=fee;
8.     }
9. }
```

Now you can serialize the Student class object that extends the Person class which is Serializable. Parent class properties are inherited to subclasses so if parent class is Serializable, subclass would also be.

Java Serialization with Aggregation (HAS-A Relationship)

If a class has a reference to another class, all the references must be Serializable otherwise serialization process will not be performed. In such case, *NotSerializableException* is thrown at runtime.

```
1. class Address{
2.     String addressLine,city,state;
3.     public Address(String addressLine, String city, String state) {
4.         this.addressLine=addressLine;
5.         this.city=city;
6.         this.state=state;
7.     }
8. }

1. import java.io.Serializable;
2. public class Student implements Serializable{
3.     int id;
4.     String name;
5.     Address address;//HAS-A
6.     public Student(int id, String name) {
7.         this.id = id;
8.         this.name = name;
9.     }
10. }
```

Since Address is not Serializable, you can not serialize the instance of Student class.

Note: All the objects within an object must be Serializable.

Java Serialization with the static data member

If there is any static data member in a class, it will not be serialized because static is the part of class not object.

```
1. class Employee implements Serializable{
2.     int id;
3.     String name;
4.     static String company="SSS IT Pvt Ltd";//it won't be serialized
5.     public Student(int id, String name) {
6.         this.id = id;
7.         this.name = name;
8.     }
9. }
```

Java Serialization with array or collection

Rule: In case of array or collection, all the objects of array or collection must be serializable. If any object is not serializable, serialization will be failed.

Externalizable in java

The Externalizable interface provides the facility of writing the state of an object into a byte stream in compress format. It is not a marker interface.

The Externalizable interface provides two methods:

- **public void writeExternal(ObjectOutput out) throws IOException**
- **public void readExternal(ObjectInput in) throws IOException**

Java Transient Keyword

If you don't want to serialize any data member of a class, you can mark it as transient.

```
1. class Employee implements Serializable{
2.     transient int id;
3.     String name;
4.     public Student(int id, String name) {
5.         this.id = id;
6.         this.name = name;
7.     }
8. }
```

Now, id will not be serialized, so when you deserialize the object after serialization, you will not get the value of id. It will return default value always. In such case, it will return 0 because the data type of id is an integer.

Visit next page for more details.

SerialVersionUID

The serialization process at runtime associates an id with each Serializable class which is known as serialVersionUID. It is used to verify the sender and receiver of the serialized object. The sender and receiver must be the same. To verify it, serialVersionUID is used. The sender and receiver must have the same serialVersionUID, otherwise, **InvalidClassException** will be thrown when you deserialize the object. We can also declare our own serialVersionUID in the Serializable class. To do so, you need to create a field serialVersionUID and assign a value to it. It must be of the long type with static and final. It is suggested to explicitly declare the serialVersionUID field in the class and have it private also. For example:

```
1. private static final long serialVersionUID=1L;
```

Now, the Serializable class will look like this:

```
1. import java.io.Serializable;
2. class Employee implements Serializable{
3.     private static final long serialVersionUID=1L;
4.     int id;
5.     String name;
6.     public Student(int id, String name) {
```



```
7.  this.id = id;
8.  this.name = name;
9.  }
10. }
```

Java Transient Keyword

Java transient keyword is used in serialization. If you define any data member as transient, it will not be serialized.

Let's take an example, I have declared a class as Student, it has three data members id, name and age. If you serialize the object, all the values will be serialized but I don't want to serialize one value, e.g. age then we can declare the age data member as transient.

Example of Java Transient Keyword

In this example, we have created the two classes Student and PersistExample. The age data member of the Student class is declared as transient, its value will not be serialized.

If you deserialize the object, you will get the default value for transient variable.

Let's create a class with transient variable.

```
1.  import java.io.Serializable;
2.  public class Student implements Serializable{
3.      int id;
4.      String name;
5.      transient int age;//Now it will not be serialized
6.      public Student(int id, String name,int age) {
7.          this.id = id;
8.          this.name = name;
9.          this.age=age;
10.     }
11. }
```

Now write the code to serialize the object.

```
1. import java.io.*;
2. class PersistExample{
3.     public static void main(String args[])throws Exception{
4.         Student s1 =new Student(211,"ravi",22);//creating object
5.         //writing object into file
6.         FileOutputStream f=new FileOutputStream("f.txt");
7.         ObjectOutputStream out=new ObjectOutputStream(f);
8.         out.writeObject(s1);
9.         out.flush();
10.
11.        out.close();
12.        f.close();
13.        System.out.println("success");
14.    }
15. }
```

Output:

success

Now write the code for deserialization.

```
1. import java.io.*;
2. class DePersist{
3.     public static void main(String args[])throws Exception{
4.         ObjectInputStream in=new ObjectInputStream(new FileInputStream("f.txt"));
5.         Student s=(Student)in.readObject();
6.         System.out.println(s.id+" "+s.name+" "+s.age);
7.         in.close();
8.     }
```

8. }

9. }

211 ravi 0

Java Networking

Java Networking is a concept of connecting two or more computing devices together so that we can share resources.

Java socket programming provides facility to share data between different computing devices.

Advantage of Java Networking

1. sharing resources
2. centralize software management

Do You Know ?

- How to perform connection-oriented Socket Programming in networking ?
- How to display the data of any online web page ?
- How to get the IP address of any host name e.g. www.google.com ?
- How to perform connection-less socket programming in networking ?

Java Networking Terminology

The widely used java networking terminologies are given below:

1. IP Address
2. Protocol
3. Port Number
4. MAC Address
5. Connection-oriented and connection-less protocol
6. Socket

1) IP Address

IP address is a unique number assigned to a node of a network e.g. 192.168.0.1 . It is composed of octets that range from 0 to 255.

It is a logical address that can be changed.

2) Protocol

A protocol is a set of rules basically that is followed for communication. For example:

- TCP
- FTP
- Telnet
- SMTP
- POP etc.

3) Port Number

The port number is used to uniquely identify different applications. It acts as a communication endpoint between applications.

The port number is associated with the IP address for communication between two applications.

4) MAC Address

MAC (Media Access Control) Address is a unique identifier of NIC (Network Interface Controller). A network node can have multiple NIC but each with unique MAC.

5) Connection-oriented and connection-less protocol

In connection-oriented protocol, acknowledgement is sent by the receiver. So it is reliable but slow. The example of connection-oriented protocol is TCP.

But, in connection-less protocol, acknowledgement is not sent by the receiver. So it is not reliable but fast.
The example of connection-less protocol is UDP.

6) Socket

A socket is an endpoint between two way communication.

Visit next page for java socket programming.

java.net package

The java.net package provides many classes to deal with networking applications in Java. A list of these classes is given below:

- Authenticator
- CacheRequest
- CacheResponse
- ContentHandler
- CookieHandler
- CookieManager
- DatagramPacket
- DatagramSocket
- DatagramSocketImpl
- InterfaceAddress
- JarURLConnection
- MulticastSocket

- InetAddress
- InetSocketAddress
- Inet4Address
- Inet6Address
- IDN
- HttpURLConnection
- HttpCookie
- NetPermission
- NetworkInterface
- PasswordAuthentication
- Proxy
- ProxySelector
- ResponseCache
- SecureCacheResponse
- ServerSocket
- Socket
- SocketAddress
- SocketImpl
- SocketPermission
- StandardSocketOptions
- URI
- URL
- URLClassLoader

- URLConnection
- URLDecoder
- URLEncoder
- URLStreamHandler

Java Socket Programming

Java Socket programming is used for communication between the applications running on different JRE.

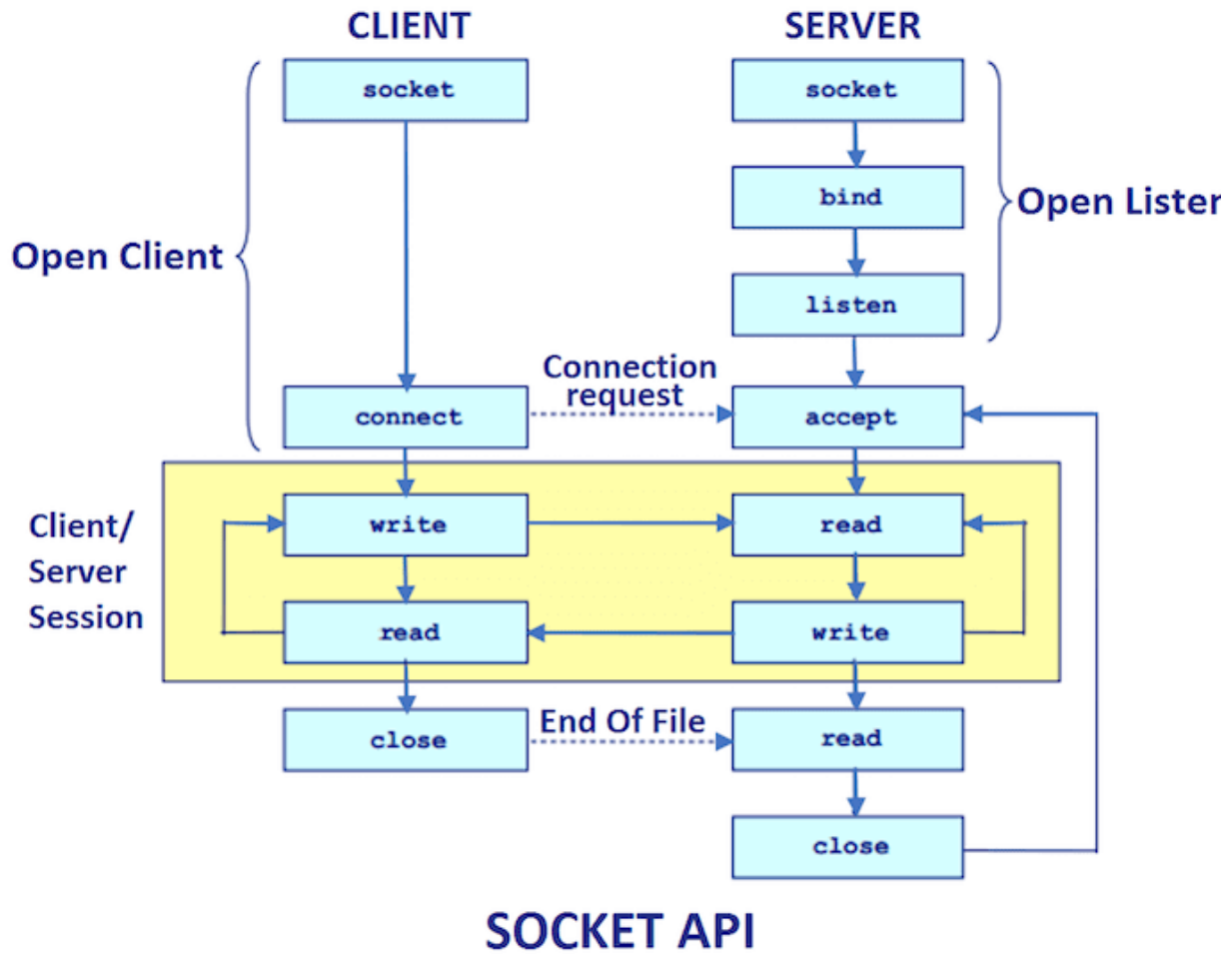
Java Socket programming can be connection-oriented or connection-less.

Socket and ServerSocket classes are used for connection-oriented socket programming and DatagramSocket and DatagramPacket classes are used for connection-less socket programming.

The client in socket programming must know two information:

1. IP Address of Server, and
2. Port number.

Here, we are going to make one-way client and server communication. In this application, client sends a message to the server, server reads the message and prints it. Here, two classes are being used: Socket and ServerSocket. The Socket class is used to communicate client and server. Through this class, we can read and write message. The ServerSocket class is used at server-side. The accept() method of ServerSocket class blocks the console until the client is connected. After the successful connection of client, it returns the instance of Socket at server-side.



Socket class

A socket is simply an endpoint for communications between the machines. The Socket class can be used to create a socket.

Important methods

Method	Description
1) public InputStream getInputStream()	returns the InputStream attached with this socket.
2) public OutputStream getOutputStream()	returns the OutputStream attached with this socket.
3) public void close()	closes this socket.

ServerSocket class

The ServerSocket class can be used to create a server socket. This object is used to establish communication with the clients.

Important methods

Method	Description
1) public Socket accept()	returns the socket and establish a connection between server and client.
2) public synchronized void close()	closes the server socket.

Example of Java Socket Programming

Creating Server:

To create the server application, we need to create the instance of ServerSocket class. Here, we are using 6666 port number for the communication between the client and server. You may also choose any other port number. The accept() method waits for the client. If clients connects with the given port number, it returns an instance of Socket.

1. `ServerSocket ss=new ServerSocket(6666);`
2. `Socket s=ss.accept();//establishes connection and waits for the client`

Creating Client:

To create the client application, we need to create the instance of Socket class. Here, we need to pass the IP address or hostname of the Server and a port number. Here, we are using "localhost" because our server is running on same system.

1. Socket s=**new** Socket("localhost",6666);

Let's see a simple of Java socket programming where client sends a text and server receives and prints it.

File: MyServer.java

1. **import** java.io.*;
2. **import** java.net.*;
3. **public class** MyServer {
4. **public static void** main(String[] args){
5. **try**{
6. ServerSocket ss=**new** ServerSocket(6666);
7. Socket s=ss.accept();//establishes connection
8. DataInputStream dis=**new** DataInputStream(s.getInputStream());
9. String str=(String)dis.readUTF();
10. System.out.println("message= "+str);
11. ss.close();
12. }**catch**(Exception e){System.out.println(e);}
13. }
14. }

File: MyClient.java

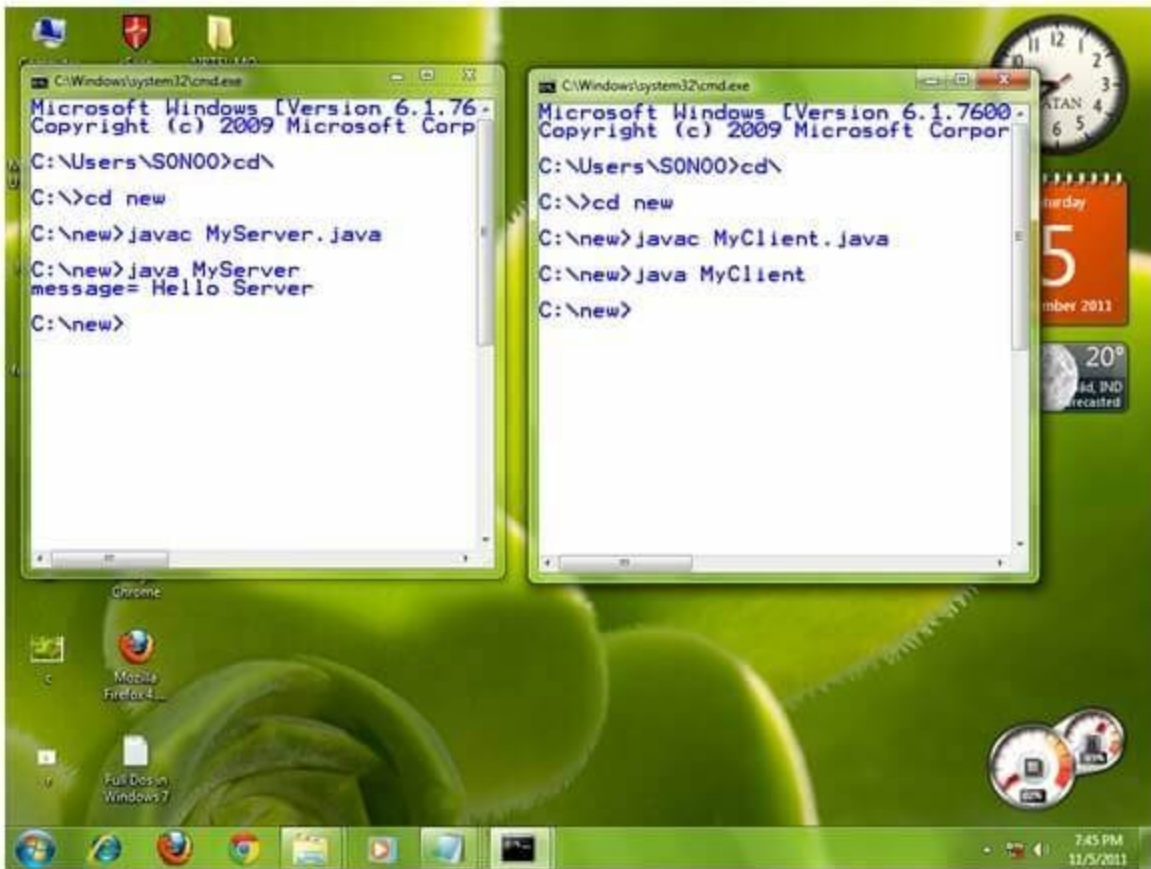
1. **import** java.io.*;
2. **import** java.net.*;
3. **public class** MyClient {
4. **public static void** main(String[] args) {

```
5. try{
6.  Socket s=new Socket("localhost",6666);
7.  DataOutputStream dout=new DataOutputStream(s.getOutputStream());
8.  dout.writeUTF("Hello Server");
9.  dout.flush();
10. dout.close();
11. s.close();
12. }catch(Exception e){System.out.println(e);}
13. }
14. }
```

[download this example](#)

To execute this program open two command prompts and execute each program at each command prompt as displayed in the below figure.

After running the client application, a message will be displayed on the server console.



Example of Java Socket Programming (Read-Write both side)

In this example, client will write first to the server then server will receive and print the text. Then server will write to the client and client will receive and print the text. The step goes on.

File: MyServer.java

1. **import** java.net.*;
2. **import** java.io.*;
3. **class** MyServer{

```

4.  public static void main(String args[])throws Exception{
5.  ServerSocket ss=new ServerSocket(3333);
6.  Socket s=ss.accept();
7.  DataInputStream din=new DataInputStream(s.getInputStream());
8.  DataOutputStream dout=new DataOutputStream(s.getOutputStream());
9.  BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
10.
11. String str="",str2="";
12. while(!str.equals("stop")){
13. str=din.readUTF();
14. System.out.println("client says: "+str);
15. str2=br.readLine();
16. dout.writeUTF(str2);
17. dout.flush();
18. }
19. din.close();
20. s.close();
21. ss.close();
22. }}

```

File: MyClient.java

```

1.  import java.net.*;
2.  import java.io.*;
3.  class MyClient{
4.  public static void main(String args[])throws Exception{
5.  Socket s=new Socket("localhost",3333);
6.  DataInputStream din=new DataInputStream(s.getInputStream());
7.  DataOutputStream dout=new DataOutputStream(s.getOutputStream());

```

```
8.  BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
9.
10. String str="",str2="";
11. while(!str.equals("stop")){
12. str=br.readLine();
13. dout.writeUTF(str);
14. dout.flush();
15. str2=din.readUTF();
16. System.out.println("Server says: "+str2);
17. }
18.
19. dout.close();
20. s.close();
21. }}
```

Java URL

The **Java URL** class represents an URL. URL is an acronym for Uniform Resource Locator. It points to a resource on the World Wide Web. For example:

1. <https://www.javatpoint.com/java-tutorial>



A URL contains many information:

1. **Protocol:** In this case, http is the protocol.
2. **Server name or IP Address:** In this case, www.javatpoint.com is the server name.

3. **Port Number:** It is an optional attribute. If we write `http://www.javatpoint.com:80/sonoojaiswal/` , 80 is the port number. If port number is not mentioned in the URL, it returns -1.
 4. **File Name or directory name:** In this case, `index.jsp` is the file name.
-

Constructors of Java URL class

URL(String spec)

Creates an instance of a URL from the String representation.

URL(String protocol, String host, int port, String file)

Creates an instance of a URL from the given protocol, host, port number, and file.

URL(String protocol, String host, int port, String file, URLStreamHandler handler)

Creates an instance of a URL from the given protocol, host, port number, file, and handler.

URL(String protocol, String host, String file)

Creates an instance of a URL from the given protocol name, host name, and file name.

URL(URL context, String spec)

Creates an instance of a URL by parsing the given spec within a specified context.

URL(URL context, String spec, URLStreamHandler handler)

Creates an instance of a URL by parsing the given spec with the specified handler within a given context.

Commonly used methods of Java URL class

The `java.net.URL` class provides many methods. The important methods of URL class are given below.

Method	Description
<code>public String getProtocol()</code>	it returns the protocol of the URL.
<code>public String getHost()</code>	it returns the host name of the URL.
<code>public String getPort()</code>	it returns the Port Number of the URL.
<code>public String getFile()</code>	it returns the file name of the URL.
<code>public String getAuthority()</code>	it returns the authority of the URL.
<code>public String toString()</code>	it returns the string representation of the URL.
<code>public String getQuery()</code>	it returns the query string of the URL.
<code>public String getDefaultPort()</code>	it returns the default port of the URL.
<code>public URLConnection openConnection()</code>	it returns the instance of URLConnection i.e. associated with this URL.
<code>public boolean equals(Object obj)</code>	it compares the URL with the given object.
<code>public Object getContent()</code>	it returns the content of the URL.

<code>public String getRef()</code>	it returns the anchor or reference of the URL.
<code>public URI toURI()</code>	it returns a URI of the URL.

Example of Java URL class

```
1. //URLDemo.java
2. import java.net.*;
3. public class URLDemo{
4.     public static void main(String[] args){
5.         try{
6.             URL url=new URL("http://www.javatpoint.com/java-tutorial");
7.
8.             System.out.println("Protocol: "+url.getProtocol());
9.             System.out.println("Host Name: "+url.getHost());
10.            System.out.println("Port Number: "+url.getPort());
11.            System.out.println("File Name: "+url.getFile());
12.
13.        }catch(Exception e){System.out.println(e);}
14.    }
15. }
```

Test it Now

Output:

Protocol: http

Host Name: www.javatpoint.com

Port Number: -1

File Name: /java-tutorial

Let us see another example URL class in Java.

```
1. //URLDemo.java
2. import java.net.*;
3. public class URLDemo{
4.     public static void main(String[] args){
5.         try{
6.             URL url=new
               URL("https://www.google.com/search?q=javatpoint&oq=javatpoint&sourceid=chrome&ie=UTF-8
               ");
7.
8.             System.out.println("Protocol: "+url.getProtocol());
9.             System.out.println("Host Name: "+url.getHost());
10.            System.out.println("Port Number: "+url.getPort());
11.            System.out.println("Default Port Number: "+url.getDefaultPort());
12.            System.out.println("Query String: "+url.getQuery());
13.            System.out.println("Path: "+url.getPath());
14.            System.out.println("File: "+url.getFile());
15.
16.        }catch(Exception e){System.out.println(e);}
17.    }
18. }
```

Output:

Protocol: https

Host Name: www.google.com

Port Number: -1

Default Port Number: 443

Query String: q=javatpoint&oq=javatpoint&sourceid=chrome&ie=UTF-8

Path: /search

File: /search?q=javatpoint&oq=javatpoint&sourceid=chrome&ie=UTF-8

Java URLConnection class

The **Java URLConnection** class represents a communication link between the URL and the application. This class can be used to read and write data to the specified resource referred by the URL.

How to get the object of URLConnection class

The `openConnection()` method of URL class returns the object of URLConnection class. Syntax:

1. **public** URLConnection `openConnection()`**throws** IOException{}
-

Displaying source code of a webpage by URLConnecton class

The URLConnection class provides many methods, we can display all the data of a webpage by using the `getInputStream()` method. The `getInputStream()` method returns all the data of the specified URL in the stream that can be read and displayed.

Example of Java URLConnection class

1. **import** java.io.*;

```
2. import java.net.*;
3. public class URLConnectionExample {
4.     public static void main(String[] args){
5.     try{
6.         URL url=new URL("http://www.javatpoint.com/java-tutorial");
7.         URLConnection urlcon=url.openConnection();
8.         InputStream stream=urlcon.getInputStream();
9.         int i;
10.        while((i=stream.read())!=-1){
11.            System.out.print((char)i);
12.        }
13.    }catch(Exception e){System.out.println(e);}
14.    }
15. }
```

Java HttpURLConnection class

The **Java HttpURLConnection** class is http specific URLConnection. It works for HTTP protocol only.

By the help of HttpURLConnection class, you can information of any HTTP URL such as header information, status code, response code etc.

The java.net.HttpURLConnection is subclass of URLConnection class.

How to get the object of HttpURLConnection class

The openConnection() method of URL class returns the object of URLConnection class. Syntax:

```
1. public URLConnection openConnection()throws IOException{}
```

You can typecast it to HttpURLConnection type as given below.

```
1. URL url=new URL("http://www.javatpoint.com/java-tutorial");
2. HttpURLConnection huc=(HttpURLConnection)url.openConnection();
```

Java HttpURLConnection Example

```
1. import java.io.*;
2. import java.net.*;
3. public class HttpURLConnectionDemo{
4.     public static void main(String[] args){
5.         try{
6.             URL url=new URL("http://www.javatpoint.com/java-tutorial");
7.             HttpURLConnection huc=(HttpURLConnection)url.openConnection();
8.             for(int i=1;i<=8;i++){
9.                 System.out.println(huc.getHeaderFieldKey(i)+" = "+huc.getHeaderField(i));
10.            }
11.            huc.disconnect();
12.        }catch(Exception e){System.out.println(e);}
13.    }
14. }
```

Test it Now

Output:

```
Date = Wed, 10 Dec 2014 19:31:14 GMT
Set-Cookie = JSESSIONID=D70B87DBB832820CACA5998C90939D48; Path=/
Content-Type = text/html
Cache-Control = max-age=2592000
Expires = Fri, 09 Jan 2015 19:31:14 GMT
Vary = Accept-Encoding,User-Agent
Connection = close
Transfer-Encoding = chunked
```

Java InetAddress class

Java InetAddress class represents an IP address. The `java.net.InetAddress` class provides methods to get the IP of any host name *for example* `www.javatpoint.com`, `www.google.com`, `www.facebook.com`, etc.

An IP address is represented by 32-bit or 128-bit unsigned number. An instance of `InetAddress` represents the IP address with its corresponding host name. There are two types of address types: Unicast and Multicast. The Unicast is an identifier for a single interface whereas Multicast is an identifier for a set of interfaces.

Moreover, `InetAddress` has a cache mechanism to store successful and unsuccessful host name resolutions.

Commonly used methods of InetAddress class

Method	Description
<code>public static InetAddress getByName(String host) throws UnknownHostException</code>	it returns the instance of <code>InetAddress</code> containing LocalHost IP and name.
<code>public static InetAddress getLocalHost() throws UnknownHostException</code>	it returns the instance of <code>InetAddress</code> containing local host name and address.
<code>public String getHostName()</code>	it returns the host name of the IP address.
<code>public String.getHostAddress()</code>	it returns the IP address in string format.

Example of Java InetAddress class

Let's see a simple example of InetAddress class to get ip address of www.javatpoint.com website.

```
1. import java.io.*;
2. import java.net.*;
3. public class InetDemo{
4.     public static void main(String[] args){
5.         try{
6.             InetAddress ip=InetAddress.getByName("www.javatpoint.com");
7.
8.             System.out.println("Host Name: "+ip.getHostName());
9.             System.out.println("IP Address: "+ip.getHostAddress());
10.        }catch(Exception e){System.out.println(e);}
11.    }
12. }
```

Test it Now

Output:

Host Name: www.javatpoint.com

IP Address: 206.51.231.148

Java DatagramSocket and DatagramPacket

Java DatagramSocket and DatagramPacket classes are used for connection-less socket programming.

Java DatagramSocket class

Java DatagramSocket class represents a connection-less socket for sending and receiving datagram packets.

A datagram is basically an information but there is no guarantee of its content, arrival or arrival time.

Commonly used Constructors of DatagramSocket class

- **DatagramSocket()** throws **SocketEeption**: it creates a datagram socket and binds it with the available Port Number on the localhost machine.
 - **DatagramSocket(int port)** throws **SocketEeption**: it creates a datagram socket and binds it with the given Port Number.
 - **DatagramSocket(int port, InetAddress address)** throws **SocketEeption**: it creates a datagram socket and binds it with the specified port number and host address.
-

Java DatagramPacket class

Java DatagramPacket is a message that can be sent or received. If you send multiple packet, it may arrive in any order. Additionally, packet delivery is not guaranteed.

Commonly used Constructors of DatagramPacket class

- **DatagramPacket(byte[] barr, int length)**: it creates a datagram packet. This constructor is used to receive the packets.
 - **DatagramPacket(byte[] barr, int length, InetAddress address, int port)**: it creates a datagram packet. This constructor is used to send the packets.
-

Example of Sending DatagramPacket by DatagramSocket

1. `//DSender.java`
2. `import java.net.*;`
3. `public class DSender{`
4. `public static void main(String[] args) throws Exception {`
5. `DatagramSocket ds = new DatagramSocket();`

```
6.   String str = "Welcome java";
7.   InetAddress ip = InetAddress.getByName("127.0.0.1");
8.
9.   DatagramPacket dp = new DatagramPacket(str.getBytes(), str.length(), ip, 3000);
10.  ds.send(dp);
11.  ds.close();
12.  }
13. }
```

Example of Receiving DatagramPacket by DatagramSocket

```
1.  //DReceiver.java
2.  import java.net.*;
3.  public class DReceiver{
4.      public static void main(String[] args) throws Exception {
5.          DatagramSocket ds = new DatagramSocket(3000);
6.          byte[] buf = new byte[1024];
7.          DatagramPacket dp = new DatagramPacket(buf, 1024);
8.          ds.receive(dp);
9.          String str = new String(dp.getData(), 0, dp.getLength());
10.         System.out.println(str);
11.         ds.close();
12.     }
13. }
```

Collections in Java

1. [Java Collection Framework](#)
2. [Hierarchy of Collection Framework](#)
3. [Collection interface](#)
4. [Iterator interface](#)

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:

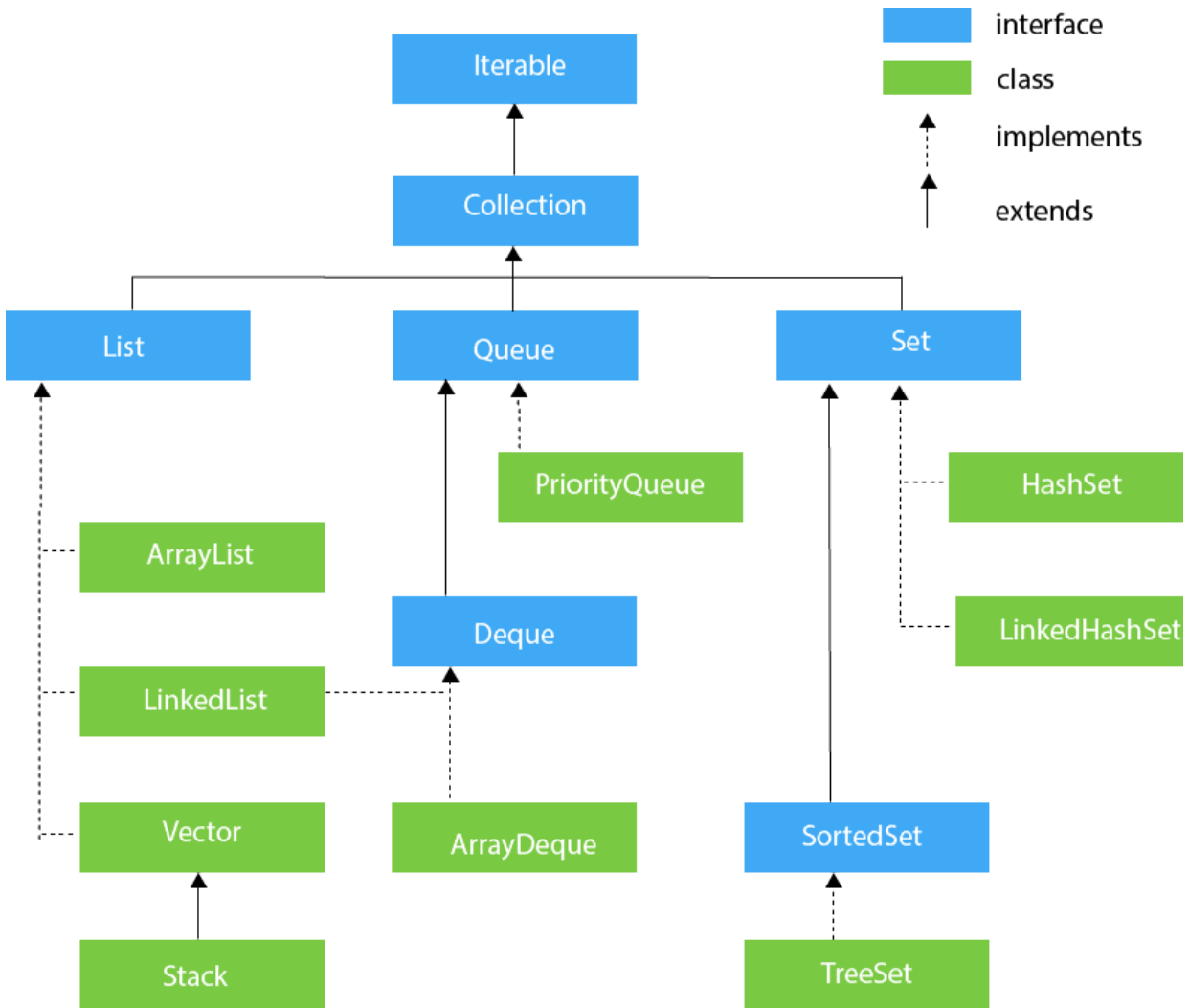
1. Interfaces and its implementations, i.e., classes
2. Algorithm

Do You Know?

- What are the two ways to iterate the elements of a collection?
- What is the difference between ArrayList and LinkedList classes in collection framework?
- What is the difference between ArrayList and Vector classes in collection framework?
- What is the difference between HashSet and HashMap classes in collection framework?
- What is the difference between HashMap and Hashtable class?
- What is the difference between Iterator and Enumeration interface in collection framework?
- How can we sort the elements of an object? What is the difference between Comparable and Comparator interfaces?
- What does the hashCode() method?
- What is the difference between Java collection and Java collections?

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:

N	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.

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.
1 0	public boolean containsAll(Collection<?> c)	It is used to search the specified collection in the collection.
1 1	public Iterator iterator()	It returns an iterator.
1 2	public Object[] toArray()	It converts collection into array.
1 3	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.
1 4	public boolean isEmpty()	It checks if collection is empty.
1 5	default Stream<E> parallelStream()	It returns a possibly parallel Stream with the collection as its source.
1 6	default Stream<E> stream()	It returns a sequential Stream with the collection as its source.

1 7	default Spliterator<E> spliterator()	It generates a Spliterator over the specified elements in the collection.
1 8	public boolean equals(Object element)	It matches two collections.
1 9	public int hashCode()	It returns the hash code number of the collection.

Iterator interface

Iterator interface provides the facility of iterating the elements in a forward direction only.

Methods of Iterator interface

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

No. of Method			Description
1	public boolean	hasNext()	It returns true if the iterator has more elements

	ean has Nex t()	otherwise it returns false.
2	publ ic Obj ect next () ()	It returns the element and moves the cursor pointer to the next element.
3	publ ic void rem ove()	It removes the last elements returned by the iterator. It is less used.

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.,

1. `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 :

1. List <data-type> list1= **new** ArrayList();
2. List <data-type> list2 = **new** LinkedList();
3. List <data-type> list3 = **new** Vector();
4. 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.

The classes that implement the List interface are given below.

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.

```
1. import java.util.*;
2. class TestJavaCollection1{
3.     public static void main(String args[]){
4.         ArrayList<String> list=new ArrayList<String>();//Creating arraylist
5.         list.add("Ravi");//Adding object in arraylist
6.         list.add("Vijay");
7.         list.add("Ravi");
8.         list.add("Ajay");
9.         //Traversing list through Iterator
10.        Iterator itr=list.iterator();
11.        while(itr.hasNext()){
12.            System.out.println(itr.next());
13.        }
14.    }
15. }
```

Output:

```
Ravi
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.

```
1. import java.util.*;
2. public class TestJavaCollection2{
3.     public static void main(String args[]){
4.         LinkedList<String> al=new LinkedList<String>();
5.         al.add("Ravi");
6.         al.add("Vijay");
7.         al.add("Ravi");
8.         al.add("Ajay");
9.         Iterator<String> itr=al.iterator();
10.        while(itr.hasNext()){
11.            System.out.println(itr.next());
12.        }
13.    }
14. }
```

Output:

```
Ravi
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.

```
1. import java.util.*;
2. public class TestJavaCollection3{
3.     public static void main(String args[]){
4.         Vector<String> v=new Vector<String>();
5.         v.add("Ayush");
6.         v.add("Amit");
7.         v.add("Ashish");
8.         v.add("Garima");
9.         Iterator<String> itr=v.iterator();
10.        while(itr.hasNext()){
11.            System.out.println(itr.next());
12.        }
13.    }
14. }
```

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.

```
1. import java.util.*;
2. public class TestJavaCollection4{
3.     public static void main(String args[]){
4.         Stack<String> stack = new Stack<String>();
5.         stack.push("Ayush");
6.         stack.push("Garvit");
7.         stack.push("Amit");
8.         stack.push("Ashish");
9.         stack.push("Garima");
10.        stack.pop();
11.        Iterator<String> itr=stack.iterator();
12.        while(itr.hasNext()){
13.            System.out.println(itr.next());
14.        }
15.    }
16. }
```

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:

1. `Queue<String> q1 = new PriorityQueue();`
2. `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.

1. `import java.util.*;`
2. `public class TestJavaCollection5{`
3. `public static void main(String args[]){`
4. `PriorityQueue<String> queue=new PriorityQueue<String>();`
5. `queue.add("Amit Sharma");`
6. `queue.add("Vijay Raj");`
7. `queue.add("JaiShankar");`
8. `queue.add("Raj");`
9. `System.out.println("head:"+queue.element());`

```
10. System.out.println("head:"+queue.peek());
11. System.out.println("iterating the queue elements:");
12. Iterator itr=queue.iterator();
13. while(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. queue.remove();
17. queue.poll();
18. System.out.println("after removing two elements:");
19. Iterator<String> itr2=queue.iterator();
20. while(itr2.hasNext()){
21. System.out.println(itr2.next());
22. }
23. }
24. }
```

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:

1. 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.

1. **import** java.util.*;
2. **public class** TestJavaCollection6{
3. **public static void** main(String[] args) {
4. *//Creating Deque and adding elements*
5. Deque<String> deque = **new** ArrayDeque<String>();
6. deque.add("Gautam");
7. deque.add("Karan");
8. deque.add("Ajay");
9. *//Traversing elements*
10. **for** (String str : deque) {
11. System.out.println(str);
12. }
13. }
14. }

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:

1. `Set<data-type> s1 = new HashSet<data-type>();`
2. `Set<data-type> s2 = new LinkedHashSet<data-type>();`
3. `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.

1. `import java.util.*;`
2. `public class TestJavaCollection7{`
3. `public static void main(String args[]){`
4. `//Creating HashSet and adding elements`
5. `HashSet<String> set=new HashSet<String>();`
6. `set.add("Ravi");`

```
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //Traversing elements
11. Iterator<String> itr=set.iterator();
12. while(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }
```

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.

```
1. import java.util.*;
2. public class TestJavaCollection8{
3. public static void main(String args[]){
4. LinkedHashSet<String> set=new LinkedHashSet<String>();
5. set.add("Ravi");
```

```
6. set.add("Vijay");
7. set.add("Ravi");
8. set.add("Ajay");
9. Iterator<String> itr=set.iterator();
10. while(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }
```

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:

```
1. 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:

```
1. import java.util.*;
2. public class TestJavaCollection9{
3.     public static void main(String args[]){
4.         //Creating and adding elements
5.         TreeSet<String> set=new TreeSet<String>();
6.         set.add("Ravi");
7.         set.add("Vijay");
8.         set.add("Ravi");
9.         set.add("Ajay");
10.        //traversing elements
11.        Iterator<String> itr=set.iterator();
12.        while(itr.hasNext()){
13.            System.out.println(itr.next());
14.        }
15.    }
16. }
```

Output:

Ajay

Ravi

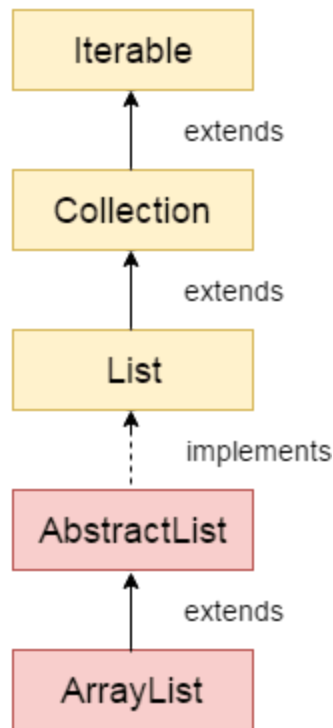
Vijay

What are we going to learn in Java Collections Framework

1. ArrayList class
2. LinkedList class
3. List interface

4. [HashSet class](#)
5. [LinkedHashSet class](#)
6. [TreeSet class](#)
7. [PriorityQueue class](#)
8. [Map interface](#)
9. [HashMap class](#)
10. [LinkedHashMap class](#)
11. [TreeMap class](#)
12. [Hashtable class](#)
13. [Sorting](#)
14. [Comparable interface](#)
15. [Comparator interface](#)
16. [Properties class in Java](#)

Java ArrayList



Java **ArrayList** class uses a *dynamic array* for storing the elements. It is like an array, but there is *no size limit*. We can add or remove elements anytime. So, it is much more flexible than the traditional array. It is found in the *java.util* package. It is like the Vector in C++.

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

It inherits the AbstractList class and implements *List interface*.

The important points about Java ArrayList class are:

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non *synchronized*.
- Java ArrayList allows random access because array works at the index basis.
- In ArrayList, manipulation is little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.

Hierarchy of ArrayList class

As shown in the above diagram, Java ArrayList class extends AbstractList class which implements List interface. The List interface extends the **Collection** and Iterable interfaces in hierarchical order.

ArrayList class declaration

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

1. **public class** ArrayList<E> **extends** AbstractList<E> **implements** List<E>, RandomAccess, Cloneable, Serializable

Constructors of ArrayList

Constructor	Description
ArrayList()	It is used to build an empty array list.
ArrayList(Collection<? extends E> c)	It is used to build an array list that is initialized with the elements of the collection c.
ArrayList(int capacity)	It is used to build an array list that has the specified initial capacity.

Methods of ArrayList

Method	Description
void add (int index, E element)	It is used to insert the specified element at the specified position in a list.
boolean add (E e)	It is used to append the specified element at the end of a list.
boolean addAll (Collection<? extends E> c)	It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.

boolean addAll (int index, Collection<? extends E> c)	It is used to append all the elements in the specified collection, starting at the specified position of the list.
void clear ()	It is used to remove all of the elements from this list.
void ensureCapacity(int requiredCapacity)	It is used to enhance the capacity of an ArrayList instance.
E get(int index)	It is used to fetch the element from the particular position of the list.
boolean isEmpty()	It returns true if the list is empty, otherwise false.
Iterator ()	
listIterator ()	
int lastIndexOf(Object o)	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
Object[] toArray()	It is used to return an array containing all of the elements in this list in the correct order.
<T> T[] toArray(T[] a)	It is used to return an array containing all of the elements in this list in the correct order.
Object clone()	It is used to return a shallow copy of an ArrayList.
boolean contains(Object o)	It returns true if the list contains the specified element
int indexOf(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.

E remove(int index)	It is used to remove the element present at the specified position in the list.
boolean remove(Object o)	It is used to remove the first occurrence of the specified element.
boolean removeAll(Collection<?> c)	It is used to remove all the elements from the list.
boolean removeIf(Predicate<? super E> filter)	It is used to remove all the elements from the list that satisfies the given predicate.
protected void removeRange(int fromIndex, int toIndex)	It is used to remove all the elements lies within the given range.
void replaceAll(UnaryOperator<E> operator)	It is used to replace all the elements from the list with the specified element.
void retainAll(Collection<?> c)	It is used to retain all the elements in the list that are present in the specified collection.
E set(int index, E element)	It is used to replace the specified element in the list, present at the specified position.
void sort(Comparator<? super E> c)	It is used to sort the elements of the list on the basis of specified comparator.
Splitter<E> splitter()	It is used to create splitter over the elements in a list.
List<E> subList(int fromIndex, int toIndex)	It is used to fetch all the elements lies within the given range.

int size()	It is used to return the number of elements present in the list.
void trimToSize()	It is used to trim the capacity of this ArrayList instance to be the list's current size.

Java Non-generic Vs. Generic Collection

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

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

Let's see the old non-generic example of creating java collection.

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

Let's see the new generic example of creating java collection.

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

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

For more information on Java generics, click here [Java Generics Tutorial](#).

Java ArrayList Example

1. `import java.util.*;`
2. `public class ArrayListExample1{`
3. `public static void main(String args[]){`
4. `ArrayList<String> list=new ArrayList<String>();//Creating arraylist`
5. `list.add("Mango");//Adding object in arraylist`
6. `list.add("Apple");`
7. `list.add("Banana");`
8. `list.add("Grapes");`
9. `//Printing the arraylist object`

```
10.    System.out.println(list);
11. }
12. }
```

Test it Now

Output:

[Mango, Apple, Banana, Grapes]

Iterating ArrayList using Iterator

Let's see an example to traverse ArrayList elements using the Iterator interface.

```
1.  import java.util.*;
2.  public class ArrayListExample2{
3.      public static void main(String args[]){
4.          ArrayList<String> list=new ArrayList<String>();//Creating arraylist
5.          list.add("Mango");//Adding object in arraylist
6.          list.add("Apple");
7.          list.add("Banana");
8.          list.add("Grapes");
9.          //Traversing list through Iterator
10.     Iterator itr=list.iterator();//getting the Iterator
11.     while(itr.hasNext()){//check if iterator has the elements
12.         System.out.println(itr.next());//printing the element and move to next
13.     }
14. }
15. }
```

Test it Now

Output:

Mango

Apple
Banana
Grapes

Iterating ArrayList using For-each loop

Let's see an example to traverse the ArrayList elements using the for-each loop

```
1. import java.util.*;
2. public class ArrayListExample3{
3.     public static void main(String args[]){
4.         ArrayList<String> list=new ArrayList<String>();//Creating arraylist
5.         list.add("Mango");//Adding object in arraylist
6.         list.add("Apple");
7.         list.add("Banana");
8.         list.add("Grapes");
9.         //Traversing list through for-each loop
10.        for(String fruit:list)
11.            System.out.println(fruit);
12.
13.    }
14. }
```

Output:

Test it Now

Mango
Apple
Banana
Grapes

Get and Set ArrayList

The *get()* method returns the element at the specified index, whereas the *set()* method changes the element.

```
1. import java.util.*;
2. public class ArrayListExample4{
3.     public static void main(String args[]){
4.         ArrayList<String> al=new ArrayList<String>();
5.         al.add("Mango");
6.         al.add("Apple");
7.         al.add("Banana");
8.         al.add("Grapes");
9.         //accessing the element
10.        System.out.println("Returning element: "+al.get(1));//it will return the 2nd element, because index
        starts from 0
11.        //changing the element
12.        al.set(1,"Dates");
13.        //Traversing list
14.        for(String fruit:al)
15.            System.out.println(fruit);
16.
17.    }
18. }
```

Test it Now

Output:

Returning element: Apple
Mango
Dates
Banana
Grapes

How to Sort ArrayList

The *java.util* package provides a utility class **Collections** which has the static method `sort()`. Using the **Collections.sort()** method, we can easily sort the `ArrayList`.

```
1. import java.util.*;
2. class SortArrayList{
3.     public static void main(String args[]){
4.         //Creating a list of fruits
5.         List<String> list1=new ArrayList<String>();
6.         list1.add("Mango");
7.         list1.add("Apple");
8.         list1.add("Banana");
9.         list1.add("Grapes");
10.        //Sorting the list
11.        Collections.sort(list1);
12.        //Traversing list through the for-each loop
13.        for(String fruit:list1)
14.            System.out.println(fruit);
15.
16.        System.out.println("Sorting numbers...");
17.        //Creating a list of numbers
18.        List<Integer> list2=new ArrayList<Integer>();
19.        list2.add(21);
20.        list2.add(11);
21.        list2.add(51);
22.        list2.add(1);
23.        //Sorting the list
24.        Collections.sort(list2);
25.        //Traversing list through the for-each loop
26.        for(Integer number:list2)
27.            System.out.println(number);
```

28. }

29.

30. }

Output:

Apple

Banana

Grapes

Mango

Sorting numbers...

1

11

21

51

Ways to iterate the elements of the collection in Java

There are various ways to traverse the collection elements:

1. By Iterator interface.
2. By for-each loop.
3. By ListIterator interface.
4. By for loop.
5. By forEach() method.
6. By forEachRemaining() method.

Iterating Collection through remaining ways

Let's see an example to traverse the ArrayList elements through other ways

1. **import** java.util.*;
2. **class** ArrayList4{
3. **public static void** main(String args[]){


```
4.    ArrayList<String> list=new ArrayList<String>();//Creating arraylist
5.        list.add("Ravi");//Adding object in arraylist
6.        list.add("Vijay");
7.        list.add("Ravi");
8.        list.add("Ajay");
9.
10.       System.out.println("Traversing list through List Iterator:");
11.       //Here, element iterates in reverse order
12.       ListIterator<String> list1=list.listIterator(list.size());
13.       while(list1.hasPrevious())
14.       {
15.           String str=list1.previous();
16.           System.out.println(str);
17.       }
18.       System.out.println("Traversing list through for loop:");
19.       for(int i=0;i<list.size();i++)
20.       {
21.           System.out.println(list.get(i));
22.       }
23.
24.       System.out.println("Traversing list through forEach() method:");
25.       //The forEach() method is a new feature, introduced in Java 8.
26.       list.forEach(a->{ //Here, we are using lambda expression
27.           System.out.println(a);
28.       });
29.
30.       System.out.println("Traversing list through forEachRemaining() method:");
31.       Iterator<String> itr=list.iterator();
```

```
32.         itr.forEachRemaining(a-> //Here, we are using lambda expression
33.         {
34.             System.out.println(a);
35.         });
36.     }
37. }
```

Output:

Traversing list through List Iterator:

Ajay

Ravi

Vijay

Ravi

Traversing list through for loop:

Ravi

Vijay

Ravi

Ajay

Traversing list through forEach() method:

Ravi

Vijay

Ravi

Ajay

Traversing list through forEachRemaining() method:

Ravi

Vijay

Ravi

Ajay

User-defined class objects in Java ArrayList

Let's see an example where we are storing Student class object in an array list.

1. **class** Student{
2. **int** rollno;

```
3. String name;
4. int age;
5. Student(int rollno,String name,int age){
6.     this.rollno=rollno;
7.     this.name=name;
8.     this.age=age;
9. }
10. }
```

```
1. import java.util.*;
2. class ArrayList5{
3.     public static void main(String args[]){
4.         //Creating user-defined class objects
5.         Student s1=new Student(101,"Sonoo",23);
6.         Student s2=new Student(102,"Ravi",21);
7.         Student s3=new Student(103,"Hanumat",25);
8.         //creating arraylist
9.         ArrayList<Student> al=new ArrayList<Student>();
10.        al.add(s1);//adding Student class object
11.        al.add(s2);
12.        al.add(s3);
13.        //Getting Iterator
14.        Iterator itr=al.iterator();
15.        //traversing elements of ArrayList object
16.        while(itr.hasNext()){
17.            Student st=(Student)itr.next();
18.            System.out.println(st.rollno+" "+st.name+" "+st.age);
19.        }
```

20. }

21. }

Output:

```
101 Sonoo 23
102 Ravi 21
103 Hanumat 25
```

Java ArrayList Serialization and Deserialization Example

Let's see an example to serialize an ArrayList object and then deserialize it.

```
1.  import java.io.*;
2.  import java.util.*;
3.  class ArrayList6 {
4.
5.      public static void main(String [] args)
6.      {
7.          ArrayList<String> al=new ArrayList<String>();
8.          al.add("Ravi");
9.          al.add("Vijay");
10.         al.add("Ajay");
11.
12.         try
13.         {
14.             //Serialization
15.             FileOutputStream fos=new FileOutputStream("file");
16.             ObjectOutputStream oos=new ObjectOutputStream(fos);
17.             oos.writeObject(al);
18.             fos.close();
```

```
19.         oos.close();
20.         //Deserialization
21.         FileInputStream fis=new FileInputStream("file");
22.         ObjectInputStream ois=new ObjectInputStream(fis);
23.         ArrayList list=(ArrayList)ois.readObject();
24.         System.out.println(list);
25.     }catch(Exception e)
26.     {
27.         System.out.println(e);
28.     }
29. }
30. }
```

Output:

```
[Ravi, Vijay, Ajay]
```

Java ArrayList example to add elements

Here, we see different ways to add an element.

```
1. import java.util.*;
2. class ArrayList7{
3.     public static void main(String args[]){
4.         ArrayList<String> al=new ArrayList<String>();
5.         System.out.println("Initial list of elements: "+al);
6.         //Adding elements to the end of the list
7.         al.add("Ravi");
8.         al.add("Vijay");
9.         al.add("Ajay");
```

```

10.    System.out.println("After invoking add(E e) method: "+al);
11.    //Adding an element at the specific position
12.    al.add(1, "Gaurav");
13.    System.out.println("After invoking add(int index, E element) method: "+al);
14.    ArrayList<String> al2=new ArrayList<String>();
15.    al2.add("Sonoo");
16.    al2.add("Hanumat");
17.    //Adding second list elements to the first list
18.    al.addAll(al2);
19.    System.out.println("After invoking addAll(Collection<? extends E> c) method: "+al);
20.    ArrayList<String> al3=new ArrayList<String>();
21.    al3.add("John");
22.    al3.add("Rahul");
23.    //Adding second list elements to the first list at specific position
24.    al.addAll(1, al3);
25.    System.out.println("After invoking addAll(int index, Collection<? extends E> c) method:
    "+al);
26.
27. }
28. }

```

Output:

Initial list of elements: []

After invoking add(E e) method: [Ravi, Vijay, Ajay]

After invoking add(int index, E element) method: [Ravi, Gaurav, Vijay, Ajay]

After invoking addAll(Collection<? extends E> c) method:

[Ravi, Gaurav, Vijay, Ajay, Sonoo, Hanumat]

After invoking addAll(int index, Collection<? extends E> c) method:

[Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat]

Java ArrayList example to remove elements

Here, we see different ways to remove an element.

```
1.  import java.util.*;
2.  class ArrayList8 {
3.
4.      public static void main(String [] args)
5.      {
6.          ArrayList<String> al=new ArrayList<String>();
7.          al.add("Ravi");
8.          al.add("Vijay");
9.          al.add("Ajay");
10.         al.add("Anuj");
11.         al.add("Gaurav");
12.         System.out.println("An initial list of elements: "+al);
13.         //Removing specific element from arraylist
14.         al.remove("Vijay");
15.         System.out.println("After invoking remove(object) method: "+al);
16.         //Removing element on the basis of specific position
17.         al.remove(0);
18.         System.out.println("After invoking remove(index) method: "+al);
19.
20.         //Creating another arraylist
21.         ArrayList<String> al2=new ArrayList<String>();
22.         al2.add("Ravi");
23.         al2.add("Hanumat");
24.         //Adding new elements to arraylist
25.         al.addAll(al2);
26.         System.out.println("Updated list : "+al);
```

```

27.      //Removing all the new elements from arraylist
28.      al.removeAll(al2);
29.      System.out.println("After invoking removeAll() method: "+al);
30.      //Removing elements on the basis of specified condition
31.      al.removeIf(str -> str.contains("Ajay")); //Here, we are using Lambda expression
32.      System.out.println("After invoking removeIf() method: "+al);
33.      //Removing all the elements available in the list
34.      al.clear();
35.      System.out.println("After invoking clear() method: "+al);
36.  }
37.  }

```

Output:

An initial list of elements: [Ravi, Vijay, Ajay, Anuj, Gaurav]
 After invoking remove(object) method: [Ravi, Ajay, Anuj, Gaurav]
 After invoking remove(index) method: [Ajay, Anuj, Gaurav]
 Updated list : [Ajay, Anuj, Gaurav, Ravi, Hanumat]
 After invoking removeAll() method: [Ajay, Anuj, Gaurav]
 After invoking removeIf() method: [Anuj, Gaurav]
 After invoking clear() method: []

Java ArrayList example of retainAll() method

```

1.  import java.util.*;
2.  class ArrayList9{
3.      public static void main(String args[]){
4.          ArrayList<String> al=new ArrayList<String>();
5.          al.add("Ravi");
6.          al.add("Vijay");
7.          al.add("Ajay");
8.          ArrayList<String> al2=new ArrayList<String>();

```



```
9.  al2.add("Ravi");
10. al2.add("Hanumat");
11. al.retainAll(al2);
12. System.out.println("iterating the elements after retaining the elements of al2");
13. Iterator itr=al.iterator();
14. while(itr.hasNext()){
15.     System.out.println(itr.next());
16. }
17. }
18. }
```

Output:

```
iterating the elements after retaining the elements of al2
Ravi
```

Java ArrayList example of isEmpty() method

```
1. import java.util.*;
2. class ArrayList10{
3.
4.     public static void main(String [] args)
5.     {
6.         ArrayList<String> al=new ArrayList<String>();
7.         System.out.println("Is ArrayList Empty: "+al.isEmpty());
8.         al.add("Ravi");
9.         al.add("Vijay");
10.        al.add("Ajay");
11.        System.out.println("After Insertion");
12.        System.out.println("Is ArrayList Empty: "+al.isEmpty());
13.    }
```

```
14. }
```

Output:

Is ArrayList Empty: true

After Insertion

Is ArrayList Empty: false

Java ArrayList Example: Book

Let's see an ArrayList example where we are adding books to list and printing all the books.

```
1. import java.util.*;
2. class Book {
3.     int id;
4.     String name,author,publisher;
5.     int quantity;
6.     public Book(int id, String name, String author, String publisher, int quantity) {
7.         this.id = id;
8.         this.name = name;
9.         this.author = author;
10.        this.publisher = publisher;
11.        this.quantity = quantity;
12.    }
13. }
14. public class ArrayListExample20 {
15.     public static void main(String[] args) {
16.         //Creating list of Books
17.         List<Book> list=new ArrayList<Book>();
18.         //Creating Books
19.         Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
```

```

20. Book b2=new Book(102,"Data Communications and Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to list
23. list.add(b1);
24. list.add(b2);
25. list.add(b3);
26. //Traversing list
27. for(Book b:list){
28.     System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
29. }
30. }
31. }

```

Test it Now

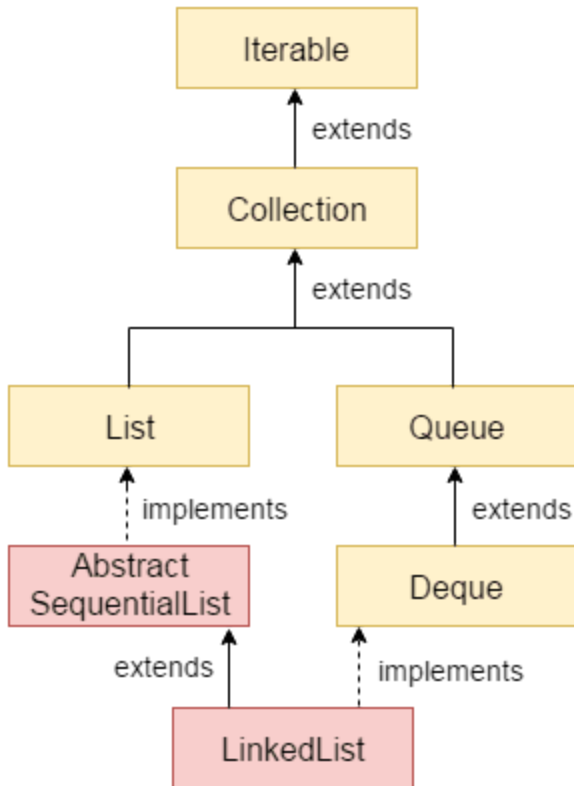
Output:

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications and Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

Java LinkedList class



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

The important points about Java **LinkedList** are:

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

Hierarchy of LinkedList class

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

Doubly Linked List

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



fig- doubly linked list

LinkedList class declaration

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

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

Constructors of Java LinkedList

Construct or	Description
LinkedList()	It is used to construct an empty list.
LinkedList(Collection<? extends E> c)	It is used to construct a list containing the elements of the specified collection, in the order, they are returned by the collection's iterator.

Methods of Java LinkedList

Method	Description
boolean add(E e)	It is used to append the specified element to the end of a list.
void add(int index, E element)	It is used to insert the specified element at the specified position index in a list.
boolean addAll(Collection collection ? extends E> c)	It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
boolean addAll(Collection collection ? extends E> c)	It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
boolean addAll(int index, Collection <? extends E> c)	It is used to append all the elements in the specified collection, starting at the specified position of the list.

void addFirst(E e)	It is used to insert the given element at the beginning of a list.
void addLast(E e)	It is used to append the given element to the end of a list.
void clear()	It is used to remove all the elements from a list.
Object clone()	It is used to return a shallow copy of an ArrayList.
boolean contains(Object o)	It is used to return true if a list contains a specified element.
Iterator<E > descendingIterator()	It is used to return an iterator over the elements in a deque in reverse sequential order.
E element()	It is used to retrieve the first element of a list.
E get(int index)	It is used to return the element at the specified position in a list.
E getFirst()	It is used to return the first element in a list.

E getLast()	It is used to return the last element in a list.
int indexOf(Object o)	It is used to return the index in a list of the first occurrence of the specified element, or -1 if the list does not contain any element.
int lastIndex Of(Object o)	It is used to return the index in a list of the last occurrence of the specified element, or -1 if the list does not contain any element.
ListIterato r<E> listIterator (int index)	It is used to return a list-iterator of the elements in proper sequence, starting at the specified position in the list.
boolean offer(E e)	It adds the specified element as the last element of a list.
boolean offerFirst(E e)	It inserts the specified element at the front of a list.
boolean offerLast(E e)	It inserts the specified element at the end of a list.
E peek()	It retrieves the first element of a list

E peekFirst())	It retrieves the first element of a list or returns null if a list is empty.
E peekLast())	It retrieves the last element of a list or returns null if a list is empty.
E poll()	It retrieves and removes the first element of a list.
E pollFirst()	It retrieves and removes the first element of a list, or returns null if a list is empty.
E pollLast()	It retrieves and removes the last element of a list, or returns null if a list is empty.
E pop()	It pops an element from the stack represented by a list.
void push(E e)	It pushes an element onto the stack represented by a list.
E remove()	It is used to retrieve and removes the first element of a list.
E remove(int index)	It is used to remove the element at the specified position in a list.

boolean remove(Object o)	It is used to remove the first occurrence of the specified element in a list.
E removeFirst()	It removes and returns the first element from a list.
boolean removeFirstOccurrence(Object o)	It is used to remove the first occurrence of the specified element in a list (when traversing the list from head to tail).
E removeLast()	It removes and returns the last element from a list.
boolean removeLastOccurrence(Object o)	It removes the last occurrence of the specified element in a list (when traversing the list from head to tail).
E set(int index, E element)	It replaces the element at the specified position in a list with the specified element.
Object[] toArray()	It is used to return an array containing all the elements in a list in

	proper sequence (from first to the last element).
<code><T> T[] toArray(T[] a)</code>	It returns an array containing all the elements in the proper sequence (from first to the last element); the runtime type of the returned array is that of the specified array.
<code>int size()</code>	It is used to return the number of elements in a list.

Java LinkedList Example

```

1. import java.util.*;
2. public class LinkedList1{
3.   public static void main(String args[]){
4.
5.     LinkedList<String> al=new LinkedList<String>();
6.     al.add("Ravi");
7.     al.add("Vijay");
8.     al.add("Ravi");
9.     al.add("Ajay");
10.
11.    Iterator<String> itr=al.iterator();
12.    while(itr.hasNext()){
13.      System.out.println(itr.next());
14.    }

```

15. }

16. }

Output: Ravi

Vijay

Ravi

Ajay

Java LinkedList example to add elements

Here, we see different ways to add elements.

```
1. import java.util.*;
2. public class LinkedList2{
3.     public static void main(String args[]){
4.         LinkedList<String> ll=new LinkedList<String>();
5.         System.out.println("Initial list of elements: "+ll);
6.         ll.add("Ravi");
7.         ll.add("Vijay");
8.         ll.add("Ajay");
9.         System.out.println("After invoking add(E e) method: "+ll);
10.        //Adding an element at the specific position
11.        ll.add(1, "Gaurav");
12.        System.out.println("After invoking add(int index, E element) method: "+ll);
13.        LinkedList<String> ll2=new LinkedList<String>();
14.        ll2.add("Sonoo");
15.        ll2.add("Hanumat");
16.        //Adding second list elements to the first list
17.        ll.addAll(ll2);
```

```

18.      System.out.println("After invoking addAll(Collection<? extends E> c) method: "+ll);
19.      LinkedList<String> ll3=new LinkedList<String>();
20.      ll3.add("John");
21.      ll3.add("Rahul");
22.      //Adding second list elements to the first list at specific position
23.      ll.addAll(1, ll3);
24.      System.out.println("After invoking addAll(int index, Collection<? extends E> c) method:
      "+ll);
25.      //Adding an element at the first position
26.      ll.addFirst("Lokesh");
27.      System.out.println("After invoking addFirst(E e) method: "+ll);
28.      //Adding an element at the last position
29.      ll.addLast("Harsh");
30.      System.out.println("After invoking addLast(E e) method: "+ll);
31.
32.  }
33.  }

```

Initial list of elements: []

After invoking add(E e) method: [Ravi, Vijay, Ajay]

After invoking add(int index, E element) method: [Ravi, Gaurav, Vijay, Ajay]

After invoking addAll(Collection<? extends E> c) method:

[Ravi, Gaurav, Vijay, Ajay, Sonoo, Hanumat]

After invoking addAll(int index, Collection<? extends E> c) method:

[Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat]

After invoking addFirst(E e) method:

[Lokesh, Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat]

After invoking addLast(E e) method:

[Lokesh, Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat, Harsh]

Java LinkedList example to remove elements

Here, we see different ways to remove an element.

```
1. import java.util.*;
2. public class LinkedList3 {
3.
4.     public static void main(String [] args)
5.     {
6.         LinkedList<String> ll=new LinkedList<String>();
7.         ll.add("Ravi");
8.         ll.add("Vijay");
9.         ll.add("Ajay");
10.        ll.add("Anuj");
11.        ll.add("Gaurav");
12.        ll.add("Harsh");
13.        ll.add("Virat");
14.        ll.add("Gaurav");
15.        ll.add("Harsh");
16.        ll.add("Amit");
17.        System.out.println("Initial list of elements: "+ll);
18.        //Removing specific element from arraylist
19.        ll.remove("Vijay");
20.        System.out.println("After invoking remove(object) method: "+ll);
21.        //Removing element on the basis of specific position
22.        ll.remove(0);
23.        System.out.println("After invoking remove(index) method: "+ll);
24.        LinkedList<String> ll2=new LinkedList<String>();
25.        ll2.add("Ravi");
26.        ll2.add("Hanumat");
```

```

27.    // Adding new elements to arraylist
28.        ll.addAll(l12);
29.        System.out.println("Updated list : "+l1);
30.    //Removing all the new elements from arraylist
31.        ll.removeAll(l12);
32.        System.out.println("After invoking removeAll() method: "+l1);
33.    //Removing first element from the list
34.        ll.removeFirst();
35.        System.out.println("After invoking removeFirst() method: "+l1);
36.    //Removing first element from the list
37.        ll.removeLast();
38.        System.out.println("After invoking removeLast() method: "+l1);
39.    //Removing first occurrence of element from the list
40.        ll.removeFirstOccurrence("Gaurav");
41.        System.out.println("After invoking removeFirstOccurrence() method: "+l1);
42.    //Removing last occurrence of element from the list
43.        ll.removeLastOccurrence("Harsh");
44.        System.out.println("After invoking removeLastOccurrence() method: "+l1);
45.
46.    //Removing all the elements available in the list
47.        ll.clear();
48.        System.out.println("After invoking clear() method: "+l1);
49.    }
50. }

```

Initial list of elements: [Ravi, Vijay, Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]

After invoking remove(object) method: [Ravi, Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]

After invoking remove(index) method: [Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]

Updated list : [Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit, Ravi, Hanumat]

After invoking removeAll() method: [Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]

After invoking removeFirst() method: [Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]

After invoking removeLast() method: [Gaurav, Harsh, Virat, Gaurav, Harsh]

After invoking removeFirstOccurrence() method: [Harsh, Virat, Gaurav, Harsh]

After invoking removeLastOccurrence() method: [Harsh, Virat, Gaurav]

After invoking clear() method: []

Java LinkedList Example to reverse a list of elements

```
1. import java.util.*;
2. public class LinkedList4{
3.     public static void main(String args[]){
4.
5.         LinkedList<String> ll=new LinkedList<String>();
6.         ll.add("Ravi");
7.         ll.add("Vijay");
8.         ll.add("Ajay");
9.         //Traversing the list of elements in reverse order
10.        Iterator i=ll.descendingIterator();
11.        while(i.hasNext())
12.        {
13.            System.out.println(i.next());
14.        }
15.
16.    }
17. }
```

Output: Ajay

Vijay

Ravi

Java LinkedList Example: Book

```
1. import java.util.*;
2. class Book {
3.     int id;
4.     String name,author,publisher;
5.     int quantity;
6.     public Book(int id, String name, String author, String publisher, int quantity) {
7.         this.id = id;
8.         this.name = name;
9.         this.author = author;
10.        this.publisher = publisher;
11.        this.quantity = quantity;
12.    }
13. }
14. public class LinkedListExample {
15.     public static void main(String[] args) {
16.         //Creating list of Books
17.         List<Book> list=new LinkedList<Book>();
18.         //Creating Books
19.         Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20.         Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21.         Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
22.         //Adding Books to list
23.         list.add(b1);
24.         list.add(b2);
25.         list.add(b3);
```

```

26. //Traversing list
27. for(Book b:list){
28.     System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
29. }
30. }
31. }

```

Output:

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications & Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

Difference between ArrayList and LinkedList

ArrayList and LinkedList both implements List interface and maintains insertion order. Both are non synchronized classes.

However, there are many differences between ArrayList and LinkedList classes that are given below.

ArrayList	LinkedList
1) ArrayList internally uses a dynamic array to store the elements.	LinkedList internally uses a doubly linked list to store the elements.
2) Manipulation with ArrayList is slow because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is faster than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
3) An ArrayList class can act as a list only because it implements List only.	LinkedList class can act as a list and queue both because it implements List and Deque interfaces.

4) ArrayList is **better for storing and accessing** data.

LinkedList is **better for manipulating** data.

Example of ArrayList and LinkedList in Java

Let's see a simple example where we are using ArrayList and LinkedList both.

```
1. import java.util.*;
2. class TestArrayLinked{
3.     public static void main(String args[]){
4.
5.         List<String> al=new ArrayList<String>();//creating arraylist
6.         al.add("Ravi");//adding object in arraylist
7.         al.add("Vijay");
8.         al.add("Ravi");
9.         al.add("Ajay");
10.
11.        List<String> al2=new LinkedList<String>();//creating linkedlist
12.        al2.add("James");//adding object in linkedlist
13.        al2.add("Serena");
14.        al2.add("Swati");
15.        al2.add("Junaid");
16.
17.        System.out.println("arraylist: "+al);
18.        System.out.println("linkedlist: "+al2);
19.    }
20. }
```

Test it Now

Output:

arraylist: [Ravi,Vijay,Ravi,Ajay]

linkedlist: [James,Serena,Swati,Junaid]

Java List

List in Java provides the facility to maintain the *ordered collection*. It contains the index-based methods to insert, update, delete and search the elements. It can have the duplicate elements also. We can also store the null elements in the list.

The List interface is found in the `java.util` package and inherits the Collection interface. It is a factory of ListIterator interface. Through the ListIterator, we can iterate the list in forward and backward directions. The implementation classes of List interface are `ArrayList`, `LinkedList`, `Stack` and `Vector`. The `ArrayList` and `LinkedList` are widely used in Java programming. The `Vector` class is deprecated since Java 5.

List Interface declaration

1. `public interface List<E> extends Collection<E>`

Java List Methods

Method	Description	
void add(int index, E element)	It is used to insert the specified element at the specified position in a list.	
boolean add(E e)	It is used to append the specified element at the end of a list.	
boolean addAll(Collection<? extends E> c)	It is used to append all of the elements in the specified collection to the end of a list.	
boolean addAll(int index, Collection<? extends E> c)	It is used to append all the elements in the specified collection, starting at the specified position of the list.	
void clear()	It is used to remove all of the elements from this list.	

<code>boolean equals(Object o)</code>	It is used to compare the specified object with the elements of a list.	
<code>int hashCode()</code>	It is used to return the hash code value for a list.	
<code>E get(int index)</code>	It is used to fetch the element from the particular position of the list.	
<code>boolean isEmpty()</code>	It returns true if the list is empty, otherwise false.	
<code>int lastIndexOf(Object o)</code>	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.	
<code>Object[] toArray()</code>	It is used to return an array containing all of the elements in this list in the correct order.	
<code><T> T[] toArray(T[] a)</code>	It is used to return an array containing all of the elements in this list in the correct order.	
<code>boolean contains(Object o)</code>	It returns true if the list contains the specified element	
<code>boolean containsAll(Collection<?> c)</code>	It returns true if the list contains all the specified element	
<code>int indexOf(Object o)</code>	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.	
<code>E remove(int index)</code>	It is used to remove the element present at the specified position in the list.	
<code>boolean remove(Object o)</code>	It is used to remove the first occurrence of the specified element.	
<code>boolean removeAll(Collection<?> c)</code>	It is used to remove all the elements from the list.	

void replaceAll(UnaryOperator<E> operator)	It is used to replace all the elements from the list with the specified element.	
void retainAll(Collection<?> c)	It is used to retain all the elements in the list that are present in the specified collection.	
E set(int index, E element)	It is used to replace the specified element in the list, present at the specified position.	
void sort(Comparator<? super E> c)	It is used to sort the elements of the list on the basis of specified comparator.	
Splitterator<E> spliterator()	It is used to create spliterator over the elements in a list.	
List<E> subList(int fromIndex, int toIndex)	It is used to fetch all the elements lies within the given range.	
int size()	It is used to return the number of elements present in the list.	

Java List vs ArrayList

List is an interface whereas ArrayList is the implementation class of List.

How to create List

The ArrayList and LinkedList classes provide the implementation of List interface. Let's see the examples to create the List:

1. `//Creating a List of type String using ArrayList`
2. `List<String> list=new ArrayList<String>();`
- 3.
4. `//Creating a List of type Integer using ArrayList`
5. `List<Integer> list=new ArrayList<Integer>();`

- 6.
7. *//Creating a List of type Book using ArrayList*
8. List<Book> list=**new** ArrayList<Book>();
- 9.
10. *//Creating a List of type String using LinkedList*
11. List<String> list=**new** LinkedList<String>();

In short, you can create the List of any type. The ArrayList<T> and LinkedList<T> classes are used to specify the type. Here, T denotes the type.

Java List Example

Let's see a simple example of List where we are using the ArrayList class as the implementation.

1. **import** java.util.*;
2. **public class** ListExample1{
3. **public static void** main(String args[]){
4. *//Creating a List*
5. List<String> list=**new** ArrayList<String>();
6. *//Adding elements in the List*
7. list.add("Mango");
8. list.add("Apple");
9. list.add("Banana");
10. list.add("Grapes");
11. *//Iterating the List element using for-each loop*
12. **for**(String fruit:list)
13. System.out.println(fruit);
- 14.
15. }
16. }

Test it Now

Output:

Mango
Apple
Banana
Grapes

How to convert Array to List

We can convert the Array to List by traversing the array and adding the element in list one by one using `list.add()` method. Let's see a simple example to convert array elements into List.

```
1. import java.util.*;
2. public class ArrayToListExample{
3.     public static void main(String args[]){
4.         //Creating Array
5.         String[] array={"Java","Python","PHP","C++"};
6.         System.out.println("Printing Array: "+Arrays.toString(array));
7.         //Converting Array to List
8.         List<String> list=new ArrayList<String>();
9.         for(String lang:array){
10.             list.add(lang);
11.         }
12.         System.out.println("Printing List: "+list);
13.
14.     }
15. }
```

Test it Now

Output:

Printing Array: [Java, Python, PHP, C++]
Printing List: [Java, Python, PHP, C++]

How to convert List to Array

We can convert the List to Array by calling the `list.toArray()` method. Let's see a simple example to convert list elements into array.

```
1. import java.util.*;
2. public class ListToArrayExample{
3.     public static void main(String args[]){
4.         List<String> fruitList = new ArrayList<>();
5.         fruitList.add("Mango");
6.         fruitList.add("Banana");
7.         fruitList.add("Apple");
8.         fruitList.add("Strawberry");
9.         //Converting ArrayList to Array
10.        String[] array = fruitList.toArray(new String[fruitList.size()]);
11.        System.out.println("Printing Array: "+Arrays.toString(array));
12.        System.out.println("Printing List: "+fruitList);
13.    }
14. }
```

Test it Now

Output:

Printing Array: [Mango, Banana, Apple, Strawberry]

Printing List: [Mango, Banana, Apple, Strawberry]

Get and Set Element in List

The `get()` method returns the element at the given index, whereas the `set()` method changes or replaces the element.

```
1. import java.util.*;
2. public class ListExample2{
```

```
3.  public static void main(String args[]){
4.      //Creating a List
5.      List<String> list=new ArrayList<String>();
6.      //Adding elements in the List
7.      list.add("Mango");
8.      list.add("Apple");
9.      list.add("Banana");
10.     list.add("Grapes");
11.     //accessing the element
12.     System.out.println("Returning element: "+list.get(1));//it will return the 2nd element, because index
        starts from 0
13.     //changing the element
14.     list.set(1,"Dates");
15.     //Iterating the List element using for-each loop
16.     for(String fruit:list)
17.         System.out.println(fruit);
18.
19. }
20. }
```

Test it Now

Output:

```
Returning element: Apple
Mango
Dates
Banana
Grapes
```

How to Sort List

There are various ways to sort the List, here we are going to use Collections.sort() method to sort the list element. The *java.util* package provides a utility class **Collections** which has the static method sort(). Using the **Collections.sort()** method, we can easily sort any List.

```
1. import java.util.*;
2. class SortArrayList{
3.     public static void main(String args[]){
4.         //Creating a list of fruits
5.         List<String> list1=new ArrayList<String>();
6.         list1.add("Mango");
7.         list1.add("Apple");
8.         list1.add("Banana");
9.         list1.add("Grapes");
10.        //Sorting the list
11.        Collections.sort(list1);
12.        //Traversing list through the for-each loop
13.        for(String fruit:list1)
14.            System.out.println(fruit);
15.
16.        System.out.println("Sorting numbers...");
17.        //Creating a list of numbers
18.        List<Integer> list2=new ArrayList<Integer>();
19.        list2.add(21);
20.        list2.add(11);
21.        list2.add(51);
22.        list2.add(1);
23.        //Sorting the list
24.        Collections.sort(list2);
25.        //Traversing list through the for-each loop
26.        for(Integer number:list2)
```

```
27.    System.out.println(number);  
28. }  
29.  
30. }
```

Output:

```
Apple  
Banana  
Grapes  
Mango  
Sorting numbers...  
1  
11  
21  
51
```

Java ListIterator Interface

ListIterator Interface is used to traverse the element in a backward and forward direction.

ListIterator Interface declaration

1. **public interface** ListIterator<E> **extends** Iterator<E>

Methods of Java ListIterator Interface:

Method	Description
void add(E e)	This method inserts the specified element into the list.
boolean hasNext()	This method returns true if the list iterator has more elements while traversing the list in the forward direction.
E next()	This method returns the next element in the list and advances the cursor position.

int nextIndex()	This method returns the index of the element that would be returned by a subsequent call to next()
boolean hasPrevious()	This method returns true if this list iterator has more elements while traversing the list in the reverse direction.
E previous()	This method returns the previous element in the list and moves the cursor position backward.
E previousIndex()	This method returns the index of the element that would be returned by a subsequent call to previous().
void remove()	This method removes the last element from the list that was returned by next() or previous() methods
void set(E e)	This method replaces the last element returned by next() or previous() methods with the specified element.

Example of ListIterator Interface

```

1. import java.util.*;
2. public class ListIteratorExample1{
3.     public static void main(String args[]){
4.         List<String> al=new ArrayList<String>();
5.         al.add("Amit");
6.         al.add("Vijay");
7.         al.add("Kumar");
8.         al.add(1,"Sachin");
9.         ListIterator<String> itr=al.listIterator();
10.        System.out.println("Traversing elements in forward direction");
11.        while(itr.hasNext()){

```

```
12.  
13.    System.out.println("index:"+itr.nextIndex()+" value:"+itr.next());  
14.    }  
15.    System.out.println("Traversing elements in backward direction");  
16.    while(itr.hasPrevious()){  
17.  
18.        System.out.println("index:"+itr.previousIndex()+" value:"+itr.previous());  
19.    }  
20. }  
21. }
```

Output:

```
Traversing elements in forward direction  
index:0 value:Amit  
index:1 value:Sachin  
index:2 value:Vijay  
index:3 value:Kumar  
Traversing elements in backward direction  
index:3 value:Kumar  
index:2 value:Vijay  
index:1 value:Sachin  
index:0 value:Amit
```

Example of List: Book

Let's see an example of List where we are adding the Books.

```
1.  import java.util.*;  
2.  class Book {  
3.      int id;  
4.      String name,author,publisher;  
5.      int quantity;  
6.      public Book(int id, String name, String author, String publisher, int quantity) {
```

```

7.  this.id = id;
8.  this.name = name;
9.  this.author = author;
10. this.publisher = publisher;
11. this.quantity = quantity;
12. }
13. }
14. public class ListExample5 {
15. public static void main(String[] args) {
16.     //Creating list of Books
17.     List<Book> list=new ArrayList<Book>();
18.     //Creating Books
19.     Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20.     Book b2=new Book(102,"Data Communications and Networking","Forouzan","Mc Graw Hill",4);
21.     Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
22.     //Adding Books to list
23.     list.add(b1);
24.     list.add(b2);
25.     list.add(b3);
26.     //Traversing list
27.     for(Book b:list){
28.         System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
29.     }
30. }
31. }

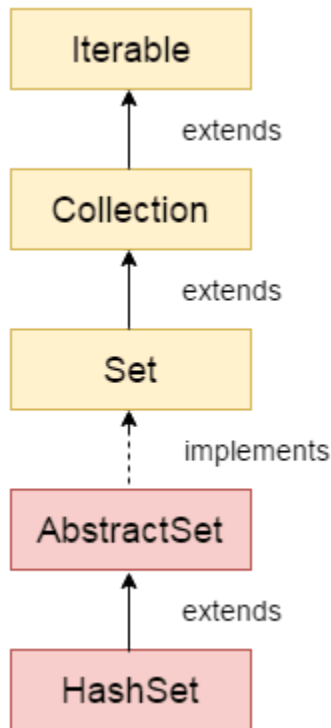
```

Test it Now

Output:

101 Let us C Yashwant Kanetkar BPB 8

Java HashSet



Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

The important points about Java HashSet class are:

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

Difference between List and Set

A list can contain duplicate elements whereas Set contains unique elements only.

Hierarchy of HashSet class

The HashSet class extends AbstractSet class which implements Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.

HashSet class declaration

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

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

Constructors of Java HashSet class

S N	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 n<? extends E> c)	It is used to initialize the hash set by using the elements of the collection c.

Methods of Java HashSet class

Various methods of Java HashSet class are as follows:

S N	Modifier & Type	Method	Description
1)	boolean	<code>add(E e)</code>	It is used to add the specified element to this set if it is not already present.
2)	void	<code>clear()</code>	It is used to remove all of the elements from the set.
3)	object	<code>clone()</code>	It is used to return a shallow copy of this HashSet instance: the elements themselves are not cloned.
4)	boolean	<code>contains(Object o)</code>	It is used to return true if this set contains the specified element.
5)	boolean	<code>isEmpty()</code>	It is used to return true if this set contains no elements.
6)	Iterator<E>	<code>iterator()</code>	It is used to return an iterator over the elements in this set.
7)	boolean	<code>remove(Object o)</code>	It is used to remove the specified element from this set if it is present.
8)	int	<code>size()</code>	It is used to return the number of elements in the set.
9)	Splitterator<E>	<code>spliterator()</code>	It is used to create a late-binding and fail-fast Spliterator over the elements in the set.

Java HashSet Example

Let's see a simple example of HashSet. Notice, the elements iterate in an unordered collection.

1. `import java.util.*;`
2. `class HashSet1{`

```

3.  public static void main(String args[]){
4.      //Creating HashSet and adding elements
5.      HashSet<String> set=new HashSet();
6.          set.add("One");
7.          set.add("Two");
8.          set.add("Three");
9.          set.add("Four");
10.         set.add("Five");
11.         Iterator<String> i=set.iterator();
12.         while(i.hasNext())
13.         {
14.             System.out.println(i.next());
15.         }
16.     }
17. }

```

Five
 One
 Four
 Two
 Three

Java HashSet example ignoring duplicate elements

In this example, we see that HashSet doesn't allow duplicate elements.

```

1.  import java.util.*;
2.  class HashSet2{
3.      public static void main(String args[]){
4.          //Creating HashSet and adding elements
5.          HashSet<String> set=new HashSet<String>();
6.          set.add("Ravi");

```

```

7.  set.add("Vijay");
8.  set.add("Ravi");
9.  set.add("Ajay");
10. //Traversing elements
11. Iterator<String> itr=set.iterator();
12. while(itr.hasNext()){
13.     System.out.println(itr.next());
14. }
15. }
16. }

```

Ajay
 Vijay
 Ravi

Java HashSet example to remove elements

Here, we see different ways to remove an element.

```

1.  import java.util.*;
2.  class HashSet3{
3.      public static void main(String args[]){
4.          HashSet<String> set=new HashSet<String>();
5.          set.add("Ravi");
6.          set.add("Vijay");
7.          set.add("Arun");
8.          set.add("Sumit");
9.          System.out.println("An initial list of elements: "+set);
10.         //Removing specific element from HashSet
11.         set.remove("Ravi");
12.         System.out.println("After invoking remove(object) method: "+set);

```

```

13.     HashSet<String> set1=new HashSet<String>();
14.     set1.add("Ajay");
15.     set1.add("Gaurav");
16.     set.addAll(set1);
17.     System.out.println("Updated List: "+set);
18.     //Removing all the new elements from HashSet
19.     set.removeAll(set1);
20.     System.out.println("After invoking removeAll() method: "+set);
21.     //Removing elements on the basis of specified condition
22.     set.removeIf(str->str.contains("Vijay"));
23.     System.out.println("After invoking removeIf() method: "+set);
24.     //Removing all the elements available in the set
25.     set.clear();
26.     System.out.println("After invoking clear() method: "+set);
27. }
28. }

```

An initial list of elements: [Vijay, Ravi, Arun, Sumit]

After invoking remove(object) method: [Vijay, Arun, Sumit]

Updated List: [Vijay, Arun, Gaurav, Sumit, Ajay]

After invoking removeAll() method: [Vijay, Arun, Sumit]

After invoking removeIf() method: [Arun, Sumit]

After invoking clear() method: []

Java HashSet from another Collection

```

1.  import java.util.*;
2.  class HashSet4{
3.      public static void main(String args[]){
4.          ArrayList<String> list=new ArrayList<String>();
5.          list.add("Ravi");

```

```
6.      list.add("Vijay");
7.      list.add("Ajay");
8.
9.      HashSet<String> set=new HashSet(list);
10.     set.add("Gaurav");
11.     Iterator<String> i=set.iterator();
12.     while(i.hasNext())
13.     {
14.         System.out.println(i.next());
15.     }
16. }
17. }
```

Vijay
Ravi
Gaurav
Ajay

Java HashSet Example: Book

Let's see a HashSet example where we are adding books to set and printing all the books.

```
1.  import java.util.*;
2.  class Book {
3.      int id;
4.      String name,author,publisher;
5.      int quantity;
6.      public Book(int id, String name, String author, String publisher, int quantity) {
7.          this.id = id;
8.          this.name = name;
9.          this.author = author;
```

```

10.  this.publisher = publisher;
11.  this.quantity = quantity;
12. }
13. }
14. public class HashSetExample {
15. public static void main(String[] args) {
16.     HashSet<Book> set=new HashSet<Book>();
17.     //Creating Books
18.     Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
19.     Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
20.     Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
21.     //Adding Books to HashSet
22.     set.add(b1);
23.     set.add(b2);
24.     set.add(b3);
25.     //Traversing HashSet
26.     for(Book b:set){
27.         System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
28.     }
29. }
30. }

```

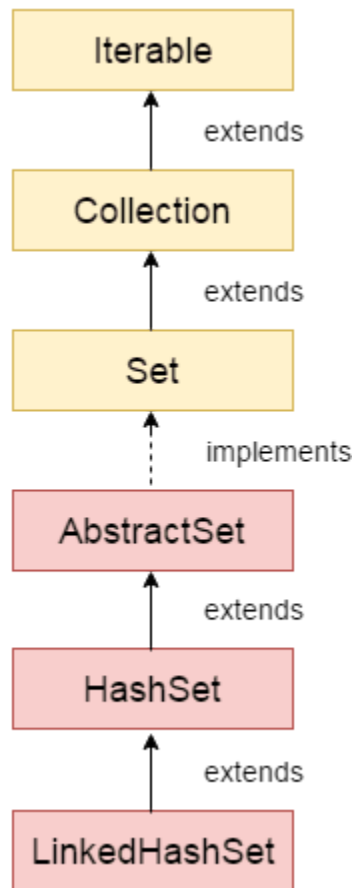
Output:

```

101 Let us C Yashwant Kanetkar BPB 8
102 Data Communications & Networking Forouzan Mc Graw Hill 4
103 Operating System Galvin Wiley 6

```

Java LinkedHashSet class



Java `LinkedHashSet` class is a `Hashtable` and `Linked list` implementation of the `set` interface. It inherits `HashSet` class and implements `Set` interface.

The important points about Java `LinkedHashSet` class are:

- Java `LinkedHashSet` class contains unique elements only like `HashSet`.
- Java `LinkedHashSet` class provides all optional set operation and permits null elements.
- Java `LinkedHashSet` class is non synchronized.
- Java `LinkedHashSet` class maintains insertion order.

Hierarchy of `LinkedHashSet` class

The `LinkedHashSet` class extends `HashSet` class which implements `Set` interface. The `Set` interface inherits `Collection` and `Iterable` interfaces in hierarchical order.

`LinkedHashSet` class declaration

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

1. **public class** LinkedHashSet<E> **extends** HashSet<E> **implements** Set<E>, Cloneable, Serializable

Constructors of Java LinkedHashSet class

Constructor	Description
HashSet()	It is used to construct a default HashSet.
HashSet(Collection c)	It is used to initialize the hash set by using the elements of the collection c.
LinkedHashSet(int capacity)	It is used initialize the capacity of the linked hash set to the given integer value capacity.
LinkedHashSet(int capacity, float fillRatio)	It is used to initialize both the capacity and the fill ratio (also called load capacity) of the hash set from its argument.

Java LinkedHashSet Example

Let's see a simple example of Java LinkedHashSet class. Here you can notice that the elements iterate in insertion order.

1. **import** java.util.*;
2. **class** LinkedHashSet1{
3. **public static void** main(String args[]){
4. *//Creating HashSet and adding elements*
5. LinkedHashSet<String> set=**new** LinkedHashSet();
6. set.add("One");
7. set.add("Two");
8. set.add("Three");
9. set.add("Four");

```
10.         set.add("Five");
11.         Iterator<String> i=set.iterator();
12.         while(i.hasNext())
13.         {
14.             System.out.println(i.next());
15.         }
16.     }
17. }
```

One

Two

Three

Four

Five

Java LinkedHashSet example ignoring duplicate Elements

```
1.  import java.util.*;
2.  class LinkedHashSet2{
3.      public static void main(String args[]){
4.          LinkedHashSet<String> al=new LinkedHashSet<String>();
5.          al.add("Ravi");
6.          al.add("Vijay");
7.          al.add("Ravi");
8.          al.add("Ajay");
9.          Iterator<String> itr=al.iterator();
10.         while(itr.hasNext()){
11.             System.out.println(itr.next());
12.         }
13.     }
```

14. }

Ravi

Vijay

Ajay

Java LinkedHashSet Example: Book

```
1. import java.util.*;
2. class Book {
3.     int id;
4.     String name,author,publisher;
5.     int quantity;
6.     public Book(int id, String name, String author, String publisher, int quantity) {
7.         this.id = id;
8.         this.name = name;
9.         this.author = author;
10.        this.publisher = publisher;
11.        this.quantity = quantity;
12.    }
13. }
14. public class LinkedHashSetExample {
15.     public static void main(String[] args) {
16.         LinkedHashSet<Book> hs=new LinkedHashSet<Book>();
17.         //Creating Books
18.         Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPP",8);
19.         Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
20.         Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
21.         //Adding Books to hash table
22.         hs.add(b1);
```

```
23.    hs.add(b2);
24.    hs.add(b3);
25.    //Traversing hash table
26.    for(Book b:hs){
27.        System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
28.    }
29. }
30. }
```

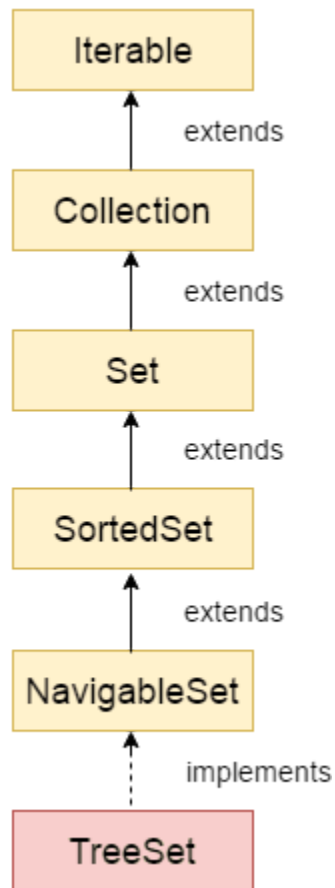
Output:

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications & Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

Java TreeSet class



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

The important points about Java TreeSet class are:

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

Hierarchy of TreeSet class

As shown in the above diagram, Java TreeSet class implements the NavigableSet interface. The NavigableSet interface extends SortedSet, Set, Collection and Iterable interfaces in hierarchical order.

TreeSet class declaration

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

1. **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.

Methods of Java TreeSet class

Method	Description
boolean add(E e)	It is used to add the specified element to this set if it is not already present.
boolean addAll(Collection<? extends E> c)	It is used to add all of the elements in the specified collection to this set.

E ceiling(E e)	It returns the equal or closest greatest element of the specified element from the set, or null there is no such element.
Comparator<? super E> comparator()	It returns comparator that arranged elements in order.
Iterator descendingIterator()	It is used iterate the elements in descending order.
NavigableSet descendingSet()	It returns the elements in reverse order.
E floor(E e)	It returns the equal or closest least element of the specified element from the set, or null there is no such element.
SortedSet headSet(E toElement)	It returns the group of elements that are less than the specified element.
NavigableSet headSet(E toElement, boolean inclusive)	It returns the group of elements that are less than or equal to(if, inclusive is true) the specified element.
E higher(E e)	It returns the closest greatest element of the specified element from the set, or null there is no such element.
Iterator iterator()	It is used to iterate the elements in ascending order.
E lower(E e)	It returns the closest least element of the specified element from the set, or null there is no such element.
E pollFirst()	It is used to retrieve and remove the lowest(first) element.

E pollLast()	It is used to retrieve and remove the highest(last) element.
Spliterator spliterator()	It is used to create a late-binding and fail-fast spliterator over the elements.
NavigableSet subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)	It returns a set of elements that lie between the given range.
SortedSet subSet(E fromElement, E toElement))	It returns a set of elements that lie between the given range which includes fromElement and excludes toElement.
SortedSet tailSet(E fromElement)	It returns a set of elements that are greater than or equal to the specified element.
NavigableSet tailSet(E fromElement, boolean inclusive)	It returns a set of elements that are greater than or equal to (if, inclusive is true) the specified element.
boolean contains(Object o)	It returns true if this set contains the specified element.
boolean isEmpty()	It returns true if this set contains no elements.
boolean remove(Object o)	It is used to remove the specified element from this set if it is present.
void clear()	It is used to remove all of the elements from this set.
Object clone()	It returns a shallow copy of this TreeSet instance.

E first()	It returns the first (lowest) element currently in this sorted set.
E last()	It returns the last (highest) element currently in this sorted set.
int size()	It returns the number of elements in this set.

Java TreeSet Examples

Java TreeSet Example 1:

Let's see a simple example of Java TreeSet.

```

1.  import java.util.*;
2.  class TreeSet1{
3.      public static void main(String args[]){
4.          //Creating and adding elements
5.          TreeSet<String> al=new TreeSet<String>();
6.          al.add("Ravi");
7.          al.add("Vijay");
8.          al.add("Ravi");
9.          al.add("Ajay");
10.         //Traversing elements
11.         Iterator<String> itr=al.iterator();
12.         while(itr.hasNext()){
13.             System.out.println(itr.next());
14.         }
15.     }
16. }
```

Test it Now

Output:

Ajay
Ravi
Vijay

Java TreeSet Example 2:

Let's see an example of traversing elements in descending order.

```
1. import java.util.*;
2. class TreeSet2{
3.     public static void main(String args[]){
4.         TreeSet<String> set=new TreeSet<String>();
5.         set.add("Ravi");
6.         set.add("Vijay");
7.         set.add("Ajay");
8.         System.out.println("Traversing element through Iterator in descending order");
9.         Iterator i=set.descendingIterator();
10.        while(i.hasNext())
11.        {
12.            System.out.println(i.next());
13.        }
14.
15.    }
16. }
```

Test it Now

Output:

Traversing element through Iterator in descending order

Vijay
Ravi
Ajay
Traversing element through NavigableSet in descending order
Vijay
Ravi
Ajay

Java TreeSet Example 3:

Let's see an example to retrieve and remove the highest and lowest Value.

```
1. import java.util.*;  
2. class TreeSet3{  
3.     public static void main(String args[]){  
4.         TreeSet<Integer> set=new TreeSet<Integer>();  
5.         set.add(24);  
6.         set.add(66);  
7.         set.add(12);  
8.         set.add(15);  
9.         System.out.println("Highest Value: "+set.pollFirst());  
10.        System.out.println("Lowest Value: "+set.pollLast());  
11.    }  
12. }
```

Output:

Highest Value: 12
Lowest Value: 66

Java TreeSet Example 4:

In this example, we perform various NavigableSet operations.

```
1. import java.util.*;
2. class TreeSet4{
3.     public static void main(String args[]){
4.         TreeSet<String> set=new TreeSet<String>();
5.         set.add("A");
6.         set.add("B");
7.         set.add("C");
8.         set.add("D");
9.         set.add("E");
10.        System.out.println("Initial Set: "+set);
11.
12.        System.out.println("Reverse Set: "+set.descendingSet());
13.
14.        System.out.println("Head Set: "+set.headSet("C", true));
15.
16.        System.out.println("SubSet: "+set.subSet("A", false, "E", true));
17.
18.        System.out.println("TailSet: "+set.tailSet("C", false));
19.    }
20. }
```

Output:

```
Initial Set: [A, B, C, D, E]
Reverse Set: [E, D, C, B, A]
Head Set: [A, B, C]
SubSet: [B, C, D, E]
TailSet: [D, E]
```

Java TreeSet Example 4:

In this example, we perform various SortedSet operations.

```
1.  import java.util.*;
2.  class TreeSet4{
3.      public static void main(String args[]){
4.          TreeSet<String> set=new TreeSet<String>();
5.          set.add("A");
6.          set.add("B");
7.          set.add("C");
8.          set.add("D");
9.          set.add("E");
10.
11.         System.out.println("Initial Set: "+set);
12.
13.         System.out.println("Head Set: "+set.headSet("C"));
14.
15.         System.out.println("SubSet: "+set.subSet("A", "E"));
16.
17.         System.out.println("TailSet: "+set.tailSet("C"));
18.     }
19. }
```

Output:

Initial Set: [A, B, C, D, E]

Head Set: [A, B]

SubSet: [A, B, C, D]

TailSet: [C, D, E]

Java TreeSet Example: Book

Let's see a TreeSet example where we are adding books to set and printing all the books. The elements in TreeSet must be of a Comparable type. String and Wrapper classes are Comparable by default. To add user-defined objects in TreeSet, you need to implement the Comparable interface.

```
1. import java.util.*;
2. class Book implements Comparable<Book>{
3.     int id;
4.     String name,author,publisher;
5.     int quantity;
6.     public Book(int id, String name, String author, String publisher, int quantity) {
7.         this.id = id;
8.         this.name = name;
9.         this.author = author;
10.        this.publisher = publisher;
11.        this.quantity = quantity;
12.    }
13.    public int compareTo(Book b) {
14.        if(id>b.id){
15.            return 1;
16.        }else if(id<b.id){
17.            return -1;
18.        }else{
19.            return 0;
20.        }
21.    }
22. }
23. public class TreeSetExample {
24.     public static void main(String[] args) {
```

```

25. Set<Book> set=new TreeSet<Book>();
26. //Creating Books
27. Book b1=new Book(121,"Let us C","Yashwant Kanetkar","BPB",8);
28. Book b2=new Book(233,"Operating System","Galvin","Wiley",6);
29. Book b3=new Book(101,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
30. //Adding Books to TreeSet
31. set.add(b1);
32. set.add(b2);
33. set.add(b3);
34. //Traversing TreeSet
35. for(Book b:set){
36.     System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
37. }
38. }
39. }

```

Output:

```

101 Data Communications & Networking Forouzan Mc Graw Hill 4
121 Let us C Yashwant Kanetkar BPB 8
233 Operating System Galvin Wiley 6

```

Java Queue Interface

Java Queue interface orders the element in FIFO(First In First Out) manner. In FIFO, first element is removed first and last element is removed at last.

Queue Interface declaration

1. **public interface** Queue<E> **extends** Collection<E>

Methods of Java Queue Interface

Method	Description
boolean add(object)	It is used to insert the specified element into this queue and return true upon success.
boolean offer(object)	It is used to insert the specified element into this queue.
Object remove()	It is used to retrieves and removes the head of this queue.
Object poll()	It is used to retrieves and removes the head of this queue, or returns null if this queue is empty.
Object element()	It is used to retrieves, but does not remove, the head of this queue.
Object peek()	It is used to retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

PriorityQueue class

The PriorityQueue class provides the facility of using queue. But it does not orders the elements in FIFO manner. It inherits AbstractQueue class.

PriorityQueue class declaration

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

1. **public class** PriorityQueue<E> **extends** AbstractQueue<E> **implements** Serializable

Java PriorityQueue Example

1. **import** java.util.*;


```
2. class TestCollection12{
3.     public static void main(String args[]){
4.         PriorityQueue<String> queue=new PriorityQueue<String>();
5.         queue.add("Amit");
6.         queue.add("Vijay");
7.         queue.add("Karan");
8.         queue.add("Jai");
9.         queue.add("Rahul");
10.        System.out.println("head:"+queue.element());
11.        System.out.println("head:"+queue.peek());
12.        System.out.println("iterating the queue elements:");
13.        Iterator itr=queue.iterator();
14.        while(itr.hasNext()){
15.            System.out.println(itr.next());
16.        }
17.        queue.remove();
18.        queue.poll();
19.        System.out.println("after removing two elements:");
20.        Iterator<String> itr2=queue.iterator();
21.        while(itr2.hasNext()){
22.            System.out.println(itr2.next());
23.        }
24.    }
25. }
```

Test it Now

Output:head:Amit

head:Amit

iterating the queue elements:

Amit

Jai

Karan
Vijay
Rahul
after removing two elements:
Karan
Rahul
Vijay

Java PriorityQueue Example: Book

Let's see a PriorityQueue example where we are adding books to queue and printing all the books. The elements in PriorityQueue must be of Comparable type. String and Wrapper classes are Comparable by default. To add user-defined objects in PriorityQueue, you need to implement Comparable interface.

```
1. import java.util.*;
2. class Book implements Comparable<Book>{
3.     int id;
4.     String name,author,publisher;
5.     int quantity;
6.     public Book(int id, String name, String author, String publisher, int quantity) {
7.         this.id = id;
8.         this.name = name;
9.         this.author = author;
10.        this.publisher = publisher;
11.        this.quantity = quantity;
12.    }
13.    public int compareTo(Book b) {
14.        if(id>b.id){
15.            return 1;
16.        }else if(id<b.id){
17.            return -1;
18.        }else{
```

```

19.     return 0;
20. }
21. }
22. }
23. public class LinkedListExample {
24.     public static void main(String[] args) {
25.         Queue<Book> queue=new PriorityQueue<Book>();
26.         //Creating Books
27.         Book b1=new Book(121,"Let us C","Yashwant Kanetkar","BPB",8);
28.         Book b2=new Book(233,"Operating System","Galvin","Wiley",6);
29.         Book b3=new Book(101,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
30.         //Adding Books to the queue
31.         queue.add(b1);
32.         queue.add(b2);
33.         queue.add(b3);
34.         System.out.println("Traversing the queue elements:");
35.         //Traversing queue elements
36.         for(Book b:queue){
37.             System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
38.         }
39.         queue.remove();
40.         System.out.println("After removing one book record:");
41.         for(Book b:queue){
42.             System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
43.         }
44.     }
45. }

```

Output:

Traversing the queue elements:

101 Data Communications & Networking Forouzan Mc Graw Hill 4

233 Operating System Galvin Wiley 6

121 Let us C Yashwant Kanetkar BPB 8

After removing one book record:

121 Let us C Yashwant Kanetkar BPB 8

233 Operating System Galvin Wiley 6

Java Deque Interface

Java Deque Interface is a linear collection that supports element insertion and removal at both ends. Deque is an acronym for "**double ended queue**".

Deque Interface declaration

1. **public interface** Deque<E> **extends** Queue<E>

Methods of Java Deque Interface

Method	Description
boolean add(object)	It is used to insert the specified element into this deque and return true upon success.
boolean offer(object)	It is used to insert the specified element into this deque.
Object remove()	It is used to retrieves and removes the head of this deque.
Object poll()	It is used to retrieves and removes the head of this deque, or returns null if this deque is empty.
Object element()	It is used to retrieves, but does not remove, the head of this deque.

Object peek()	It is used to retrieve, but does not remove, the head of this deque, or returns null if this deque is empty.
---------------	--

ArrayDeque class

The ArrayDeque class provides the facility of using deque and resizable-array. It inherits AbstractCollection class and implements the Deque interface.

The important points about ArrayDeque class are:

- Unlike Queue, we can add or remove elements from both sides.
- Null elements are not allowed in the ArrayDeque.
- ArrayDeque is not thread safe, in the absence of external synchronization.
- ArrayDeque has no capacity restrictions.
- ArrayDeque is faster than LinkedList and Stack.

ArrayDeque Hierarchy

The hierarchy of ArrayDeque class is given in the figure displayed at the right side of the page.

ArrayDeque class declaration

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

1. **public class** ArrayDeque<E> **extends** AbstractCollection<E> **implements** Deque<E>, Cloneable, Serializable

Java ArrayDeque Example

1. **import** java.util.*;
2. **public class** ArrayDequeExample {

```
3.  public static void main(String[] args) {
4.      //Creating Deque and adding elements
5.      Deque<String> deque = new ArrayDeque<String>();
6.      deque.add("Ravi");
7.      deque.add("Vijay");
8.      deque.add("Ajay");
9.      //Traversing elements
10.     for (String str : deque) {
11.         System.out.println(str);
12.     }
13. }
14. }
```

Output:

Ravi
Vijay
Ajay

Java ArrayDeque Example: offerFirst() and pollLast()

```
1.  import java.util.*;
2.  public class DequeExample {
3.      public static void main(String[] args) {
4.          Deque<String> deque=new ArrayDeque<String>();
5.          deque.offer("arvind");
6.          deque.offer("vimal");
7.          deque.add("mukul");
8.          deque.offerFirst("jai");
9.          System.out.println("After offerFirst Traversal...");
```

```
10.  for(String s:deque){
11.      System.out.println(s);
12.  }
13.  //deque.poll();
14.  //deque.pollFirst();//it is same as poll()
15.  deque.pollLast();
16.  System.out.println("After pollLast() Traversal...");
17.  for(String s:deque){
18.      System.out.println(s);
19.  }
20. }
21. }
```

Output:

After offerFirst Traversal...

jai
arvind
vimal
mukul

After pollLast() Traversal...

jai
arvind
vimal

Java ArrayDeque Example: Book

```
1.  import java.util.*;
2.  class Book {
3.      int id;
4.      String name,author,publisher;
5.      int quantity;
```

```

6.  public Book(int id, String name, String author, String publisher, int quantity) {
7.      this.id = id;
8.      this.name = name;
9.      this.author = author;
10.     this.publisher = publisher;
11.     this.quantity = quantity;
12. }
13. }
14. public class ArrayDequeExample {
15.     public static void main(String[] args) {
16.         Deque<Book> set=new ArrayDeque<Book>();
17.         //Creating Books
18.         Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
19.         Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
20.         Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
21.         //Adding Books to Deque
22.         set.add(b1);
23.         set.add(b2);
24.         set.add(b3);
25.         //Traversing ArrayDeque
26.         for(Book b:set){
27.             System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
28.         }
29.     }
30. }

```

Output:

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications & Networking Forouzan Mc Graw Hill 4

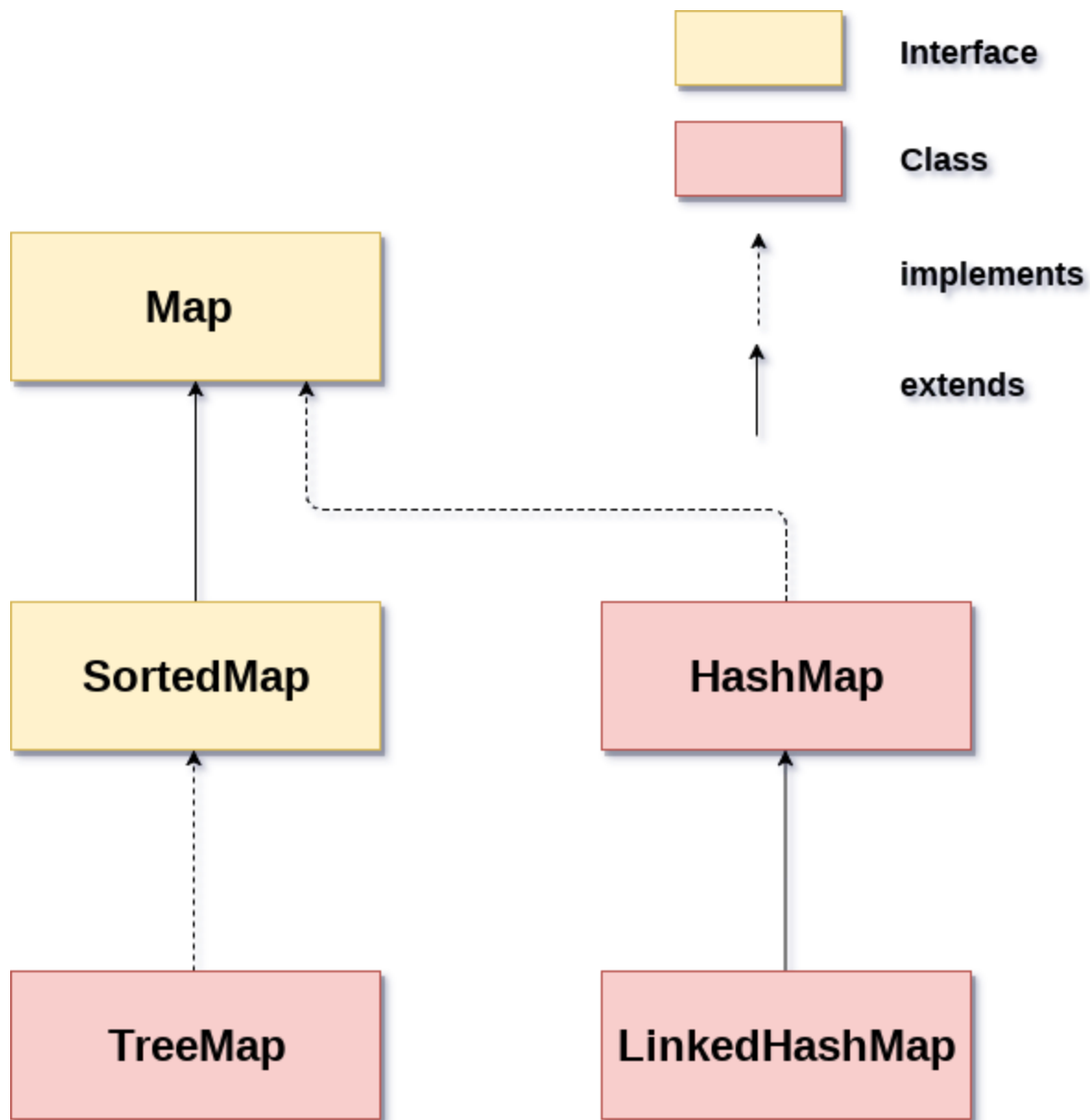
Java Map Interface

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

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

Java Map Hierarchy

There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap. The hierarchy of Java Map is given below:



A Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.

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

Class	Description
HashMap	HashMap is the implementation of Map, but it doesn't maintain any order.

LinkedHashMap	LinkedHashMap is the implementation of Map. It inherits HashMap class. It maintains insertion order.
TreeMap	TreeMap is the implementation of Map and SortedMap. It maintains ascending order.

Useful methods of Map interface

Method	Description
V put(Object key, Object value)	It is used to insert an entry in the map.
void putAll(Map map)	It is used to insert the specified map in the map.
V putIfAbsent(K key, V value)	It inserts the specified value with the specified key in the map only if it is not already specified.
V remove(Object key)	It is used to delete an entry for the specified key.
boolean remove(Object key, Object value)	It removes the specified values with the associated specified keys from the map.
Set keySet()	It returns the Set view containing all the keys.
Set<Map.Entry<K,V>> entrySet()	It returns the Set view containing all the keys and values.
void clear()	It is used to reset the map.
V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).

V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)	It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null.
V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null.
boolean containsValue(Object value)	This method returns true if some value equal to the value exists within the map, else return false.
boolean containsKey(Object key)	This method returns true if some key equal to the key exists within the map, else return false.
boolean equals(Object o)	It is used to compare the specified Object with the Map.
void forEach(BiConsumer<? super K,? super V> action)	It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.
V get(Object key)	This method returns the object that contains the value associated with the key.
V getOrDefault(Object key, V defaultValue)	It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key.
int hashCode()	It returns the hash code value for the Map
boolean isEmpty()	This method returns true if the map is empty; returns false if it contains at least one key.

V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)	If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
V replace(K key, V value)	It replaces the specified value for a specified key.
boolean replace(K key, V oldValue, V newValue)	It replaces the old value with the new value for a specified key.
void replaceAll(BiFunction<? super K,? super V,? extends V> function)	It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
Collection values()	It returns a collection view of the values contained in the map.
int size()	This method returns the number of entries in the map.

Map.Entry Interface

Entry is the subinterface of Map. So we will be accessed it by Map.Entry name. It returns a collection-view of the map, whose elements are of this class. It provides methods to get key and value.

Methods of Map.Entry interface

Method	Description
K getKey()	It is used to obtain a key.
V getValue()	It is used to obtain value.
int hashCode()	It is used to obtain hashCode.

V setValue(V value)	It is used to replace the value corresponding to this entry with the specified value.
boolean equals(Object o)	It is used to compare the specified object with the other existing objects.
static <K extends Comparable<? super K>,V> Comparator<Map.Entry<K,V>> comparingByKey()	It returns a comparator that compare the objects in natural order on key.
static <K,V> Comparator<Map.Entry<K,V>> comparingByKey(Comparator<? super K> cmp)	It returns a comparator that compare the objects by key using the given Comparator.
static <K,V extends Comparable<? super V>> Comparator<Map.Entry<K,V>> comparingByValue()	It returns a comparator that compare the objects in natural order on value.
static <K,V> Comparator<Map.Entry<K,V>> comparingByValue(Comparator<? super V> cmp)	It returns a comparator that compare the objects by value using the given Comparator.

Java Map Example: Non-Generic (Old Style)

1. `//Non-generic`
2. `import java.util.*;`
3. `public class MapExample1 {`
4. `public static void main(String[] args) {`
5. `Map map=new HashMap();`
6. `//Adding elements to map`
7. `map.put(1,"Amit");`
8. `map.put(5,"Rahul");`
9. `map.put(2,"Jai");`
10. `map.put(6,"Amit");`
11. `//Traversing Map`

```
12. Set set=map.entrySet();//Converting to Set so that we can traverse
13. Iterator itr=set.iterator();
14. while(itr.hasNext()){
15.     //Converting to Map.Entry so that we can get key and value separately
16.     Map.Entry entry=(Map.Entry)itr.next();
17.     System.out.println(entry.getKey()+" "+entry.getValue());
18. }
19. }
20. }
```

Output:

```
1 Amit
2 Jai
5 Rahul
6 Amit
```

Java Map Example: Generic (New Style)

```
1. import java.util.*;
2. class MapExample2{
3.     public static void main(String args[]){
4.         Map<Integer,String> map=new HashMap<Integer,String>();
5.         map.put(100,"Amit");
6.         map.put(101,"Vijay");
7.         map.put(102,"Rahul");
8.         //Elements can traverse in any order
9.         for(Map.Entry m:map.entrySet()){
10.            System.out.println(m.getKey()+" "+m.getValue());
11.        }
12.    }
```

```
13. }
```

Output:

102 Rahul

100 Amit

101 Vijay

Java Map Example: comparingByKey()

```
1. import java.util.*;
2. class MapExample3{
3.     public static void main(String args[]){
4.         Map<Integer,String> map=new HashMap<Integer,String>();
5.         map.put(100,"Amit");
6.         map.put(101,"Vijay");
7.         map.put(102,"Rahul");
8.         //Returns a Set view of the mappings contained in this map
9.         map.entrySet()
10.        //Returns a sequential Stream with this collection as its source
11.        .stream()
12.        //Sorted according to the provided Comparator
13.        .sorted(Map.Entry.comparingByKey())
14.        //Performs an action for each element of this stream
15.        .forEach(System.out::println);
16.     }
17. }
```

Output:

100=Amit

101=Vijay

102=Rahul

Java Map Example: comparingByKey() in Descending Order

```
1. import java.util.*;
2. class MapExample4{
3.     public static void main(String args[]){
4.         Map<Integer,String> map=new HashMap<Integer,String>();
5.         map.put(100,"Amit");
6.         map.put(101,"Vijay");
7.         map.put(102,"Rahul");
8.         //Returns a Set view of the mappings contained in this map
9.         map.entrySet()
10.        //Returns a sequential Stream with this collection as its source
11.        .stream()
12.        //Sorted according to the provided Comparator
13.        .sorted(Map.Entry.comparingByKey(Comparator.reverseOrder()))
14.        //Performs an action for each element of this stream
15.        .forEach(System.out::println);
16.     }
17. }
```

Output:

```
102=Rahul
101=Vijay
100=Amit
```

Java Map Example: comparingByValue()

```
1. import java.util.*;
2. class MapExample5{
```

```

3.  public static void main(String args[]){
4.  Map<Integer,String> map=new HashMap<Integer,String>();
5.      map.put(100,"Amit");
6.      map.put(101,"Vijay");
7.      map.put(102,"Rahul");
8.      //Returns a Set view of the mappings contained in this map
9.      map.entrySet()
10.     //Returns a sequential Stream with this collection as its source
11.     .stream()
12.     //Sorted according to the provided Comparator
13.     .sorted(Map.Entry.comparingByValue())
14.     //Performs an action for each element of this stream
15.     .forEach(System.out::println);
16. }
17. }

```

Output:

```

100=Amit
102=Rahul
101=Vijay

```

Java Map Example: comparingByValue() in Descending Order

```

1.  import java.util.*;
2.  class MapExample6{
3.      public static void main(String args[]){
4.      Map<Integer,String> map=new HashMap<Integer,String>();
5.          map.put(100,"Amit");
6.          map.put(101,"Vijay");
7.          map.put(102,"Rahul");

```

```

8.    //Returns a Set view of the mappings contained in this map
9.    map.entrySet()
10.   //Returns a sequential Stream with this collection as its source
11.   .stream()
12.   //Sorted according to the provided Comparator
13.   .sorted(Map.Entry.comparingByValue(Comparator.reverseOrder()))
14.   //Performs an action for each element of this stream
15.   .forEach(System.out::println);
16. }
17. }

```

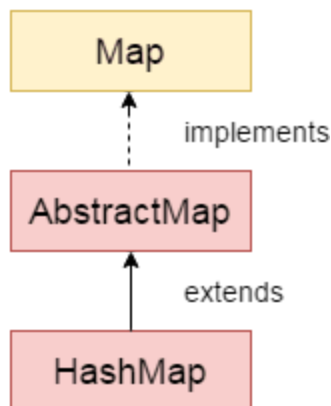
Output:

101=Vijay

102=Rahul

100=Amit

JJava HashMap



Java **HashMap** class implements the Map interface which allows us *to store key and value pair*, where keys should be unique. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the `java.util` package.

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

Points to remember

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

Hierarchy of HashMap class

As shown in the above figure, HashMap class extends AbstractMap class and implements Map interface.

HashMap class declaration

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

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

HashMap class Parameters

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

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

Constructors of Java HashMap class

Constructor	Description
HashMap()	It is used to construct a default HashMap.
HashMap(Map<? extends K,? extends V> m)	It is used to initialize the hash map by using the elements of the given Map object m.
HashMap(int capacity)	It is used to initializes the capacity of the hash map to the given integer value, capacity.
HashMap(int capacity, float loadFactor)	It is used to initialize both the capacity and load factor of the hash map by using its arguments.

Methods of Java HashMap class

Method	Description
void clear()	It is used to remove all of the mappings from this map.
boolean isEmpty()	It is used to return true if this map contains no key-value mappings.
Object clone()	It is used to return a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
Set entrySet()	It is used to return a collection view of the mappings contained in this map.

Set keySet()	It is used to return a set view of the keys contained in this map.
V put(Object key, Object value)	It is used to insert an entry in the map.
void putAll(Map map)	It is used to insert the specified map in the map.
V putIfAbsent(K key, V value)	It inserts the specified value with the specified key in the map only if it is not already specified.
V remove(Object key)	It is used to delete an entry for the specified key.
boolean remove(Object key, Object value)	It removes the specified values with the associated specified keys from the map.
V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)	It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null.
V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null.
boolean containsValue(Object value)	This method returns true if some value equal to the value exists within the map, else return false.

<code>boolean containsKey(Object key)</code>	This method returns true if some key equal to the key exists within the map, else return false.
<code>boolean equals(Object o)</code>	It is used to compare the specified Object with the Map.
<code>void forEach(BiConsumer<? super K,? super V> action)</code>	It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.
<code>V get(Object key)</code>	This method returns the object that contains the value associated with the key.
<code>V getOrDefault(Object key, V defaultValue)</code>	It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key.
<code>boolean isEmpty()</code>	This method returns true if the map is empty; returns false if it contains at least one key.
<code>V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)</code>	If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
<code>V replace(K key, V value)</code>	It replaces the specified value for a specified key.
<code>boolean replace(K key, V oldValue, V newValue)</code>	It replaces the old value with the new value for a specified key.

<code>void replaceAll(BiFunction<? super K,? super V,? extends V> function)</code>	It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
<code>Collection<V> values()</code>	It returns a collection view of the values contained in the map.
<code>int size()</code>	This method returns the number of entries in the map.

Java HashMap Example

Let's see a simple example of HashMap to store key and value pair.

```

1.  import java.util.*;
2.  public class HashMapExample1{
3.      public static void main(String args[]){
4.          HashMap<Integer,String> map=new HashMap<Integer,String>();//Creating HashMap
5.          map.put(1,"Mango"); //Put elements in Map
6.          map.put(2,"Apple");
7.          map.put(3,"Banana");
8.          map.put(4,"Grapes");
9.
10.         System.out.println("Iterating Hashmap...");
11.         for(Map.Entry m : map.entrySet()){
12.             System.out.println(m.getKey()+" "+m.getValue());
13.         }
14.     }
15. }
```

Test it Now

Iterating Hashmap...

1 Mango

2 Apple

3 Banana

4 Grapes

In this example, we are storing Integer as the key and String as the value, so we are using `HashMap<Integer,String>` as the type. The `put()` method inserts the elements in the map.

To get the key and value elements, we should call the `getKey()` and `getValue()` methods. The `Map.Entry` interface contains the `getKey()` and `getValue()` methods. But, we should call the `entrySet()` method of Map interface to get the instance of Map.Entry.

No Duplicate Key on HashMap

You cannot store duplicate keys in HashMap. However, if you try to store duplicate key with another value, it will replace the value.

```
1. import java.util.*;
2. public class HashMapExample2{
3.     public static void main(String args[]){
4.         HashMap<Integer,String> map=new HashMap<Integer,String>();//Creating HashMap
5.         map.put(1,"Mango"); //Put elements in Map
6.         map.put(2,"Apple");
7.         map.put(3,"Banana");
8.         map.put(1,"Grapes"); //trying duplicate key
9.
10.        System.out.println("Iterating Hashmap...");
11.        for(Map.Entry m : map.entrySet()){
12.            System.out.println(m.getKey()+" "+m.getValue());
13.        }
14.    }
15. }
```

Test it Now

Iterating Hashmap...

1 Grapes

2 Apple

3 Banana

Java HashMap example to add() elements

Here, we see different ways to insert elements.

```
1.  import java.util.*;
2.  class HashMap1{
3.      public static void main(String args[]){
4.          HashMap<Integer,String> hm=new HashMap<Integer,String>();
5.          System.out.println("Initial list of elements: "+hm);
6.          hm.put(100,"Amit");
7.          hm.put(101,"Vijay");
8.          hm.put(102,"Rahul");
9.
10.         System.out.println("After invoking put() method ");
11.         for(Map.Entry m:hm.entrySet()){
12.             System.out.println(m.getKey()+" "+m.getValue());
13.         }
14.
15.         hm.putIfAbsent(103, "Gaurav");
16.         System.out.println("After invoking putIfAbsent() method ");
17.         for(Map.Entry m:hm.entrySet()){
18.             System.out.println(m.getKey()+" "+m.getValue());
19.         }
```

```
20.    HashMap<Integer,String> map=new HashMap<Integer,String>();
21.    map.put(104,"Ravi");
22.    map.putAll(hm);
23.    System.out.println("After invoking putAll() method ");
24.    for(Map.Entry m:map.entrySet()){
25.        System.out.println(m.getKey()+" "+m.getValue());
26.    }
27. }
28. }
```

Initial list of elements: {}

After invoking put() method

100 Amit

101 Vijay

102 Rahul

After invoking putIfAbsent() method

100 Amit

101 Vijay

102 Rahul

103 Gaurav

After invoking putAll() method

100 Amit

101 Vijay

102 Rahul

103 Gaurav

104 Ravi

Java HashMap example to remove() elements

Here, we see different ways to remove elements.

```
1. import java.util.*;
2. public class HashMap2 {
3.     public static void main(String args[]) {
4.         HashMap<Integer,String> map=new HashMap<Integer,String>();
5.         map.put(100,"Amit");
6.         map.put(101,"Vijay");
7.         map.put(102,"Rahul");
8.         map.put(103, "Gaurav");
9.         System.out.println("Initial list of elements: "+map);
10.        //key-based removal
11.        map.remove(100);
12.        System.out.println("Updated list of elements: "+map);
13.        //value-based removal
14.        map.remove(101);
15.        System.out.println("Updated list of elements: "+map);
16.        //key-value pair based removal
17.        map.remove(102, "Rahul");
18.        System.out.println("Updated list of elements: "+map);
19.    }
20. }
```

Output:

Initial list of elements: {100=Amit, 101=Vijay, 102=Rahul, 103=Gaurav}

Updated list of elements: {101=Vijay, 102=Rahul, 103=Gaurav}

Updated list of elements: {102=Rahul, 103=Gaurav}

Updated list of elements: {103=Gaurav}

Java HashMap example to replace() elements

Here, we see different ways to replace elements.

```
1.  import java.util.*;
2.  class HashMap3{
3.      public static void main(String args[]){
4.          HashMap<Integer,String> hm=new HashMap<Integer,String>();
5.          hm.put(100,"Amit");
6.          hm.put(101,"Vijay");
7.          hm.put(102,"Rahul");
8.          System.out.println("Initial list of elements:");
9.          for(Map.Entry m:hm.entrySet())
10.         {
11.             System.out.println(m.getKey()+" "+m.getValue());
12.         }
13.          System.out.println("Updated list of elements:");
14.          hm.replace(102, "Gaurav");
15.          for(Map.Entry m:hm.entrySet())
16.         {
17.             System.out.println(m.getKey()+" "+m.getValue());
18.         }
19.          System.out.println("Updated list of elements:");
20.          hm.replace(101, "Vijay", "Ravi");
21.          for(Map.Entry m:hm.entrySet())
22.         {
23.             System.out.println(m.getKey()+" "+m.getValue());
24.         }
```

```
25.    System.out.println("Updated list of elements:");
26.    hm.replaceAll((k,v) -> "Ajay");
27.    for(Map.Entry m:hm.entrySet())
28.    {
29.        System.out.println(m.getKey()+" "+m.getValue());
30.    }
31. }
32. }
```

Initial list of elements:

100 Amit

101 Vijay

102 Rahul

Updated list of elements:

100 Amit

101 Vijay

102 Gaurav

Updated list of elements:

100 Amit

101 Ravi

102 Gaurav

Updated list of elements:

100 Ajay

101 Ajay

102 Ajay

Difference between HashSet and HashMap

HashSet contains only values whereas HashMap contains an entry(key and value).

Java HashMap Example: Book

```
1. import java.util.*;
2. class Book {
3.     int id;
4.     String name,author,publisher;
5.     int quantity;
6.     public Book(int id, String name, String author, String publisher, int quantity) {
7.         this.id = id;
8.         this.name = name;
9.         this.author = author;
10.        this.publisher = publisher;
11.        this.quantity = quantity;
12.    }
13. }
14. public class MapExample {
15.     public static void main(String[] args) {
16.         //Creating map of Books
17.         Map<Integer,Book> map=new HashMap<Integer,Book>();
18.         //Creating Books
19.         Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20.         Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21.         Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
22.         //Adding Books to map
23.         map.put(1,b1);
24.         map.put(2,b2);
25.         map.put(3,b3);
26.
27.         //Traversing map
```

```
28.  for(Map.Entry<Integer, Book> entry:map.entrySet()){
29.      int key=entry.getKey();
30.      Book b=entry.getValue();
31.      System.out.println(key+" Details:");
32.      System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
33.  }
34. }
35. }
```

Test it Now

Output:

1 Details:

101 Let us C Yashwant Kanetkar BPB 8

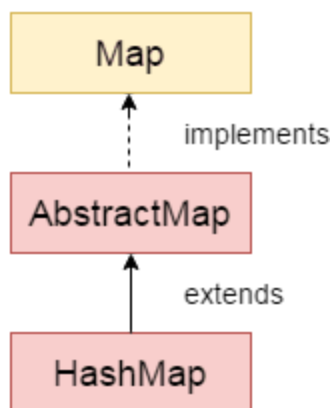
2 Details:

102 Data Communications and Networking Forouzan Mc Graw Hill 4

3 Details:

103 Operating System Galvin Wiley 6

Java HashMap



Java **HashMap** class implements the Map interface which allows us to store *key and value pair*, where keys should be unique. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the `java.util` package.

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

Points to remember

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

Hierarchy of HashMap class

As shown in the above figure, HashMap class extends AbstractMap class and implements Map interface.

HashMap class declaration

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

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

HashMap class Parameters

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

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

Constructors of Java HashMap class

Constructor	Description
HashMap()	It is used to construct a default HashMap.
HashMap(Map<? extends K,? extends V> m)	It is used to initialize the hash map by using the elements of the given Map object m.
HashMap(int capacity)	It is used to initializes the capacity of the hash map to the given integer value, capacity.
HashMap(int capacity, float loadFactor)	It is used to initialize both the capacity and load factor of the hash map by using its arguments.

Methods of Java HashMap class

Method	Description
void clear()	It is used to remove all of the mappings from this map.
boolean isEmpty()	It is used to return true if this map contains no key-value mappings.
Object clone()	It is used to return a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
Set entrySet()	It is used to return a collection view of the mappings contained in this map.
Set keySet()	It is used to return a set view of the keys contained in this map.
V put(Object key, Object value)	It is used to insert an entry in the map.
void putAll(Map map)	It is used to insert the specified map in the map.

V putIfAbsent(K key, V value)	It inserts the specified value with the specified key in the map only if it is not already specified.
V remove(Object key)	It is used to delete an entry for the specified key.
boolean remove(Object key, Object value)	It removes the specified values with the associated specified keys from the map.
V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)	It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null.
V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null.
boolean containsValue(Object value)	This method returns true if some value equal to the value exists within the map, else return false.
boolean containsKey(Object key)	This method returns true if some key equal to the key exists within the map, else return false.
boolean equals(Object o)	It is used to compare the specified Object with the Map.
void forEach(BiConsumer<? super K,? super V> action)	It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.

<code>V get(Object key)</code>	This method returns the object that contains the value associated with the key.
<code>V getOrDefault(Object key, V defaultValue)</code>	It returns the value to which the specified key is mapped, or <code>defaultValue</code> if the map contains no mapping for the key.
<code>boolean isEmpty()</code>	This method returns true if the map is empty; returns false if it contains at least one key.
<code>V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)</code>	If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
<code>V replace(K key, V value)</code>	It replaces the specified value for a specified key.
<code>boolean replace(K key, V oldValue, V newValue)</code>	It replaces the old value with the new value for a specified key.
<code>void replaceAll(BiFunction<? super K,? super V,? extends V> function)</code>	It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
<code>Collection<V> values()</code>	It returns a collection view of the values contained in the map.
<code>int size()</code>	This method returns the number of entries in the map.

Java HashMap Example

Let's see a simple example of HashMap to store key and value pair.

1. **import** java.util.*;
2. **public class** HashMapExample1{
3. **public static void** main(String args[]){

```

4.  HashMap<Integer,String> map=new HashMap<Integer,String>();//Creating HashMap
5.  map.put(1,"Mango"); //Put elements in Map
6.  map.put(2,"Apple");
7.  map.put(3,"Banana");
8.  map.put(4,"Grapes");
9.
10. System.out.println("Iterating Hashmap...");
11. for(Map.Entry m : map.entrySet()){
12.     System.out.println(m.getKey()+" "+m.getValue());
13. }
14. }
15. }

```

Test it Now

Iterating Hashmap...

```

1 Mango
2 Apple
3 Banana
4 Grapes

```

In this example, we are storing Integer as the key and String as the value, so we are using `HashMap<Integer,String>` as the type. The `put()` method inserts the elements in the map.

To get the key and value elements, we should call the `getKey()` and `getValue()` methods. The `Map.Entry` interface contains the `getKey()` and `getValue()` methods. But, we should call the `entrySet()` method of Map interface to get the instance of Map.Entry.

No Duplicate Key on HashMap

You cannot store duplicate keys in HashMap. However, if you try to store duplicate key with another value, it will replace the value.

```

1.  import java.util.*;
2.  public class HashMapExample2{
3.  public static void main(String args[]){

```

```

4.  HashMap<Integer,String> map=new HashMap<Integer,String>();//Creating HashMap
5.  map.put(1,"Mango"); //Put elements in Map
6.  map.put(2,"Apple");
7.  map.put(3,"Banana");
8.  map.put(1,"Grapes"); //trying duplicate key
9.
10. System.out.println("Iterating Hashmap...");
11. for(Map.Entry m : map.entrySet()){
12.     System.out.println(m.getKey()+" "+m.getValue());
13. }
14. }
15. }

```

Test it Now

Iterating Hashmap...

1 Grapes

2 Apple

3 Banana

Java HashMap example to add() elements

Here, we see different ways to insert elements.

```

1.  import java.util.*;
2.  class HashMap1{
3.      public static void main(String args[]){
4.          HashMap<Integer,String> hm=new HashMap<Integer,String>();
5.          System.out.println("Initial list of elements: "+hm);
6.          hm.put(100,"Amit");
7.          hm.put(101,"Vijay");
8.          hm.put(102,"Rahul");
9.

```

```

10.    System.out.println("After invoking put() method ");
11.    for(Map.Entry m:hm.entrySet()){
12.        System.out.println(m.getKey()+" "+m.getValue());
13.    }
14.
15.    hm.putIfAbsent(103, "Gaurav");
16.    System.out.println("After invoking putIfAbsent() method ");
17.    for(Map.Entry m:hm.entrySet()){
18.        System.out.println(m.getKey()+" "+m.getValue());
19.    }
20.    HashMap<Integer,String> map=new HashMap<Integer,String>();
21.    map.put(104,"Ravi");
22.    map.putAll(hm);
23.    System.out.println("After invoking putAll() method ");
24.    for(Map.Entry m:map.entrySet()){
25.        System.out.println(m.getKey()+" "+m.getValue());
26.    }
27. }
28. }

```

Initial list of elements: {}

After invoking put() method

100 Amit

101 Vijay

102 Rahul

After invoking putIfAbsent() method

100 Amit

101 Vijay

102 Rahul

103 Gaurav

After invoking putAll() method

100 Amit

101 Vijay

102 Rahul

103 Gaurav

104 Ravi

Java HashMap example to remove() elements

Here, we see different ways to remove elements.

```
1. import java.util.*;
2. public class HashMap2 {
3.     public static void main(String args[]) {
4.         HashMap<Integer,String> map=new HashMap<Integer,String>();
5.         map.put(100,"Amit");
6.         map.put(101,"Vijay");
7.         map.put(102,"Rahul");
8.         map.put(103, "Gaurav");
9.         System.out.println("Initial list of elements: "+map);
10.        //key-based removal
11.        map.remove(100);
12.        System.out.println("Updated list of elements: "+map);
13.        //value-based removal
14.        map.remove(101);
15.        System.out.println("Updated list of elements: "+map);
16.        //key-value pair based removal
17.        map.remove(102, "Rahul");
18.        System.out.println("Updated list of elements: "+map);
19.    }
20. }
```

Output:

Initial list of elements: {100=Amit, 101=Vijay, 102=Rahul, 103=Gaurav}

Updated list of elements: {101=Vijay, 102=Rahul, 103=Gaurav}

Updated list of elements: {102=Rahul, 103=Gaurav}

Updated list of elements: {103=Gaurav}

Java HashMap example to replace() elements

Here, we see different ways to replace elements.

```
1. import java.util.*;
2. class HashMap3{
3.     public static void main(String args[]){
4.         HashMap<Integer,String> hm=new HashMap<Integer,String>();
5.         hm.put(100,"Amit");
6.         hm.put(101,"Vijay");
7.         hm.put(102,"Rahul");
8.         System.out.println("Initial list of elements:");
9.         for(Map.Entry m:hm.entrySet())
10.        {
11.            System.out.println(m.getKey()+" "+m.getValue());
12.        }
13.         System.out.println("Updated list of elements:");
14.         hm.replace(102, "Gaurav");
15.         for(Map.Entry m:hm.entrySet())
16.        {
17.            System.out.println(m.getKey()+" "+m.getValue());
18.        }
19.         System.out.println("Updated list of elements:");
20.         hm.replace(101, "Vijay", "Ravi");
21.         for(Map.Entry m:hm.entrySet())
22.        {
```

```
23.     System.out.println(m.getKey()+" "+m.getValue());
24.     }
25.     System.out.println("Updated list of elements:");
26.     hm.replaceAll((k,v) -> "Ajay");
27.     for(Map.Entry m:hm.entrySet())
28.     {
29.         System.out.println(m.getKey()+" "+m.getValue());
30.     }
31. }
32. }
```

Initial list of elements:

100 Amit

101 Vijay

102 Rahul

Updated list of elements:

100 Amit

101 Vijay

102 Gaurav

Updated list of elements:

100 Amit

101 Ravi

102 Gaurav

Updated list of elements:

100 Ajay

101 Ajay

102 Ajay

Difference between HashSet and HashMap

HashSet contains only values whereas HashMap contains an entry(key and value).

Java HashMap Example: Book

```
1. import java.util.*;
2. class Book {
```

```
3.  int id;
4.  String name,author,publisher;
5.  int quantity;
6.  public Book(int id, String name, String author, String publisher, int quantity) {
7.      this.id = id;
8.      this.name = name;
9.      this.author = author;
10.     this.publisher = publisher;
11.     this.quantity = quantity;
12. }
13. }
14. public class MapExample {
15.     public static void main(String[] args) {
16.         //Creating map of Books
17.         Map<Integer,Book> map=new HashMap<Integer,Book>();
18.         //Creating Books
19.         Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20.         Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21.         Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
22.         //Adding Books to map
23.         map.put(1,b1);
24.         map.put(2,b2);
25.         map.put(3,b3);
26.
27.         //Traversing map
28.         for(Map.Entry<Integer, Book> entry:map.entrySet()){
29.             int key=entry.getKey();
30.             Book b=entry.getValue();
```

```
31.    System.out.println(key+" Details:");
32.    System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
33. }
34. }
35. }
```

Test it Now

Output:

1 Details:

101 Let us C Yashwant Kanetkar BPB 8

2 Details:

102 Data Communications and Networking Forouzan Mc Graw Hill 4

3 Details:

103 Operating System Galvin Wiley 6

Working of HashMap in Java

What is Hashing

It is the process of converting an object into an integer value. The integer value helps in indexing and faster searches.

What is HashMap

HashMap is a part of the Java collection framework. It uses a technique called Hashing. It implements the map interface. It stores the data in the pair of Key and Value. HashMap contains an array of the nodes, and the node is represented as a class. It uses an array and LinkedList data structure internally for storing Key and Value. There are four fields in HashMap.

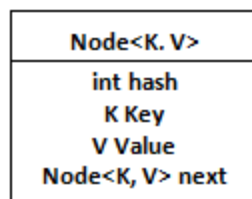


Figure: Representation of a Node

Before understanding the internal working of HashMap, you must be aware of `hashCode()` and `equals()` method.

- **equals():** It checks the equality of two objects. It compares the Key, whether they are equal or not. It is a method of the Object class. It can be overridden. If you override the `equals()` method, then it is mandatory to override the `hashCode()` method.

- **hashCode():** This is the method of the object class. It returns the memory reference of the object in integer form. The value received from the method is used as the bucket number. The bucket number is the address of the element inside the map. Hash code of null Key is 0.
- **Buckets:** Array of the node is called buckets. Each node has a data structure like a LinkedList. More than one node can share the same bucket. It may be different in capacity.

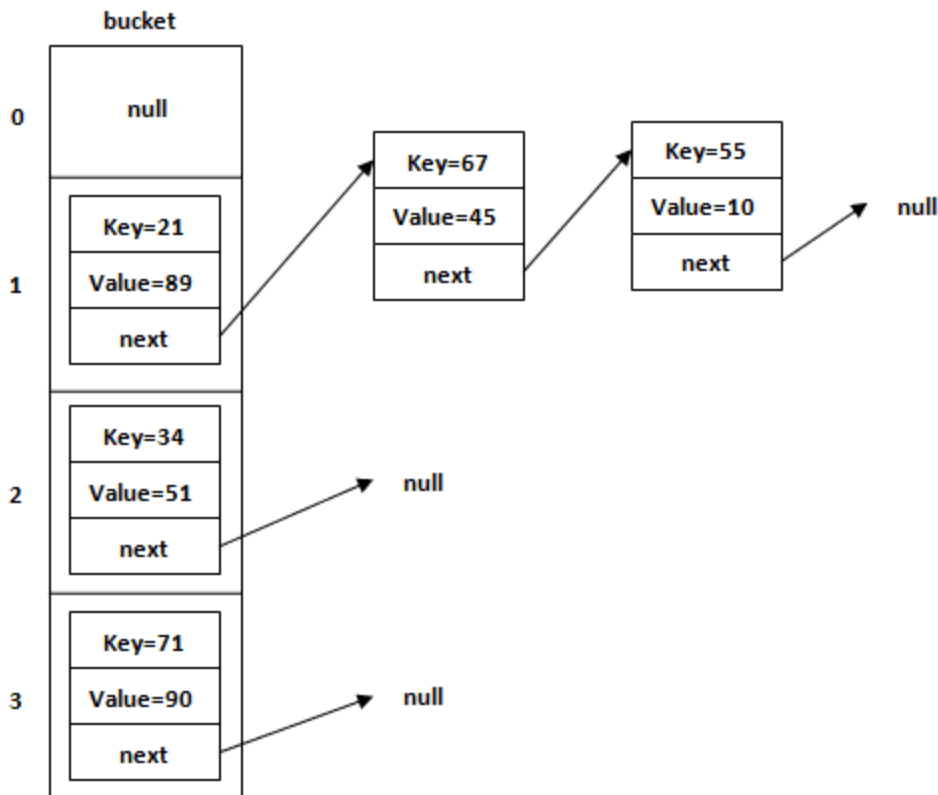


Figure: Allocation of nodes in Bucket

Insert Key, Value pair in HashMap

We use put() method to insert the Key and Value pair in the HashMap. The default size of HashMap is 16 (0 to 15).

Example

In the following example, we want to insert three (Key, Value) pair in the HashMap.

1. `HashMap<String, Integer> map = new HashMap<>();`
2. `map.put("Aman", 19);`
3. `map.put("Sunny", 29);`
4. `map.put("Ritesh", 39);`

Let's see at which index the Key, value pair will be saved into HashMap. When we call the put() method, then it calculates the hash code of the Key "Aman." Suppose the hash code of "Aman" is 2657860. To store the Key in memory, we have to calculate the index.

Calculating Index

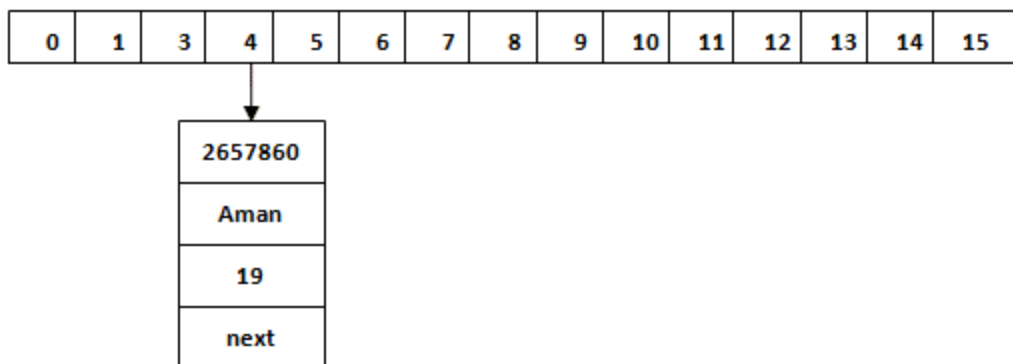
Index minimizes the size of the array. The Formula for calculating the index is:

1. $\text{Index} = \text{hashCode}(\text{Key}) \& (n-1)$

Where n is the size of the array. Hence the index value for "Aman" is:

1. $\text{Index} = 2657860 \& (16-1) = 4$

The value 4 is the computed index value where the Key and value will store in HashMap.

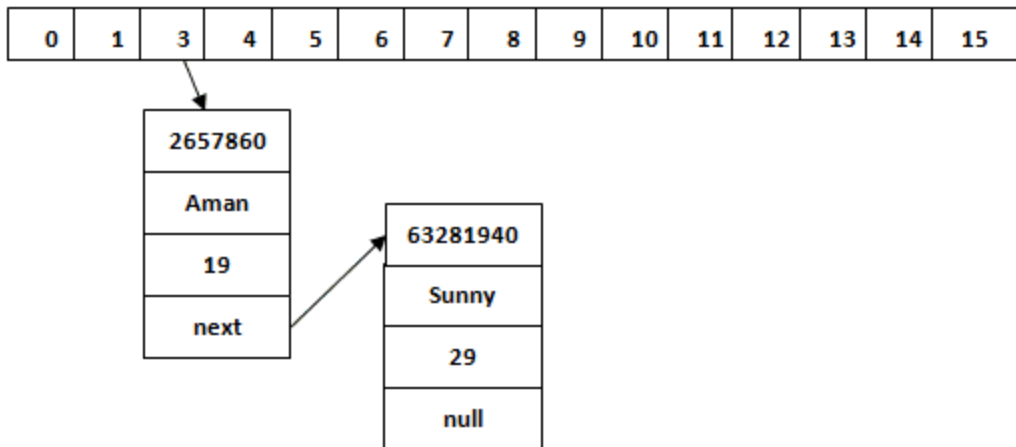


Hash Collision

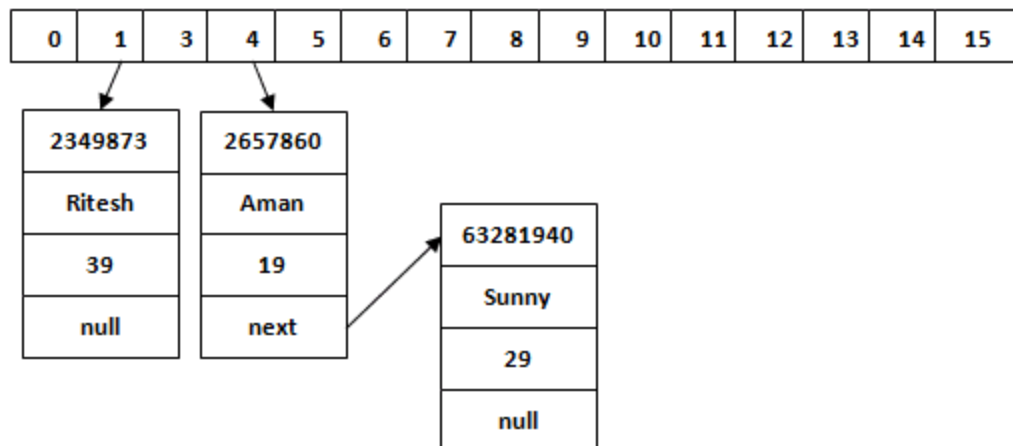
This is the case when the calculated index value is the same for two or more Keys. Let's calculate the hash code for another Key "Sunny." Suppose the hash code for "Sunny" is 63281940. To store the Key in the memory, we have to calculate index by using the index formula.

1. $\text{Index} = 63281940 \ \& \ (16-1) = 4$

The value 4 is the computed index value where the Key will be stored in HashMap. In this case, equals() method check that both Keys are equal or not. If Keys are same, replace the value with the current value. Otherwise, connect this node object to the existing node object through the LinkedList. Hence both Keys will be stored at index 4.



Similarly, we will store the Key "Ritesh." Suppose hash code for the Key is 2349873. The index value will be 1. Hence this Key will be stored at index 1.



get() method in HashMap

get() method is used to get the value by its Key. It will not fetch the value if you don't know the Key. When get(K Key) method is called, it calculates the hash code of the Key.

Suppose we have to fetch the Key "Aman." The following method will be called.

```
1. map.get(new Key("Aman"));
```

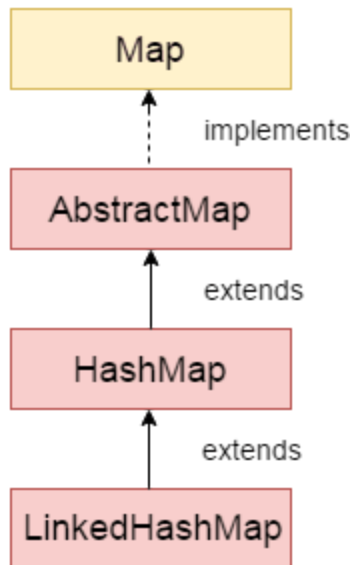
It generates the hash code as 2657860. Now calculate the index value of 2657860 by using index formula.

The index value will be 4, as we have calculated above. get() method search for the index value 4. It compares the first element Key with the given Key. If both keys are equal, then it returns the value else check for the next element in the node if it exists. In our scenario, it is found as the first element of the node and return the value 19.

Let's fetch another Key "Sunny."

The hash code of the Key "Sunny" is 63281940. The calculated index value of 63281940 is 4, as we have calculated for put() method. Go to index 4 of the array and compare the first element's Key with the given Key. It also compares Keys. In our scenario, the given Key is the second element, and the next of the node is null. It compares the second element Key with the specified Key and returns the value 29. It returns null if the next of the node is null.

Java LinkedHashMap class



Java LinkedHashMap class is Hashtable and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface.

Points to remember

- Java LinkedHashMap contains values based on the key.
- Java LinkedHashMap contains unique elements.
- Java LinkedHashMap may have one null key and multiple null values.
- Java LinkedHashMap is non synchronized.
- Java LinkedHashMap maintains insertion order.
- The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

LinkedHashMap class declaration

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

1. **public class** LinkedHashMap<K,V> **extends** HashMap<K,V> **implements** Map<K,V>

LinkedHashMap class Parameters

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

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

Constructors of Java LinkedHashMap class

Constructor	Description
LinkedHashMap()	It is used to construct a default LinkedHashMap.
LinkedHashMap(int capacity)	It is used to initialize a LinkedHashMap with the given capacity.
LinkedHashMap(int capacity, float loadFactor)	It is used to initialize both the capacity and the load factor.
LinkedHashMap(int capacity, float loadFactor, boolean accessOrder)	It is used to initialize both the capacity and the load factor with specified ordering mode.
LinkedHashMap(Map<? extends K,? extends V> m)	It is used to initialize the LinkedHashMap with the elements from the given Map class m.

Methods of Java LinkedHashMap class

Method	Description
V get(Object key)	It returns the value to which the specified key is mapped.
void clear()	It removes all the key-value pairs from a map.
boolean containsValue(Object value)	It returns true if the map maps one or more keys to the specified value.
Set<Map.Entry<K,V>> entrySet()	It returns a Set view of the mappings contained in the map.

<code>void forEach(BiConsumer<? super K, ? super V> action)</code>	It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.
<code>V getOrDefault(Object key, V defaultValue)</code>	It returns the value to which the specified key is mapped or defaultValue if this map contains no mapping for the key.
<code>Set<K> keySet()</code>	It returns a Set view of the keys contained in the map
<code>protected boolean removeEldestEntry(Map.Entry<K,V> eldest)</code>	It returns true on removing its eldest entry.
<code>void replaceAll(BiFunction<? super K, ? super V, ? extends V> function)</code>	It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
<code>Collection<V> values()</code>	It returns a Collection view of the values contained in this map.

Java LinkedHashMap Example

1. `import java.util.*;`
2. `class LinkedHashMap1{`
3. `public static void main(String args[]){`
- 4.
5. `LinkedHashMap<Integer,String> hm=new LinkedHashMap<Integer,String>();`
- 6.
7. `hm.put(100,"Amit");`
8. `hm.put(101,"Vijay");`

```

9.    hm.put(102,"Rahul");
10.
11.   for(Map.Entry m:hm.entrySet()){
12.       System.out.println(m.getKey()+" "+m.getValue());
13.   }
14. }
15. }

```

Output:100 Amit

101 Vijay

102 Rahul

Java LinkedHashMap Example: Key-Value pair

```

1.   import java.util.*;
2.   class LinkedHashMap2{
3.       public static void main(String args[]){
4.           LinkedHashMap<Integer, String> map = new LinkedHashMap<Integer, String>();
5.           map.put(100,"Amit");
6.           map.put(101,"Vijay");
7.           map.put(102,"Rahul");
8.           //Fetching key
9.           System.out.println("Keys: "+map.keySet());
10.          //Fetching value
11.          System.out.println("Values: "+map.values());
12.          //Fetching key-value pair
13.          System.out.println("Key-Value pairs: "+map.entrySet());
14.      }
15.  }

```

Keys: [100, 101, 102]

Values: [Amit, Vijay, Rahul]

Key-Value pairs: [100=Amit, 101=Vijay, 102=Rahul]

Java LinkedHashMap Example:remove()

```
1. import java.util.*;
2. public class LinkedHashMap3 {
3.     public static void main(String args[]) {
4.         Map<Integer,String> map=new LinkedHashMap<Integer,String>();
5.         map.put(101,"Amit");
6.         map.put(102,"Vijay");
7.         map.put(103,"Rahul");
8.         System.out.println("Before invoking remove() method: "+map);
9.         map.remove(102);
10.        System.out.println("After invoking remove() method: "+map);
11.    }
12. }
```

Output:

Before invoking remove() method: {101=Amit, 102=Vijay, 103=Rahul}

After invoking remove() method: {101=Amit, 103=Rahul}

Java LinkedHashMap Example: Book

```
1. import java.util.*;
2. class Book {
3.     int id;
4.     String name,author,publisher;
5.     int quantity;
6.     public Book(int id, String name, String author, String publisher, int quantity) {
```

```

7.     this.id = id;
8.     this.name = name;
9.     this.author = author;
10.    this.publisher = publisher;
11.    this.quantity = quantity;
12. }
13. }
14. public class MapExample {
15.     public static void main(String[] args) {
16.         //Creating map of Books
17.         Map<Integer,Book> map=new LinkedHashMap<Integer,Book>();
18.         //Creating Books
19.         Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPP",8);
20.         Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21.         Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
22.         //Adding Books to map
23.         map.put(2,b2);
24.         map.put(1,b1);
25.         map.put(3,b3);
26.
27.         //Traversing map
28.         for(Map.Entry<Integer, Book> entry:map.entrySet()){
29.             int key=entry.getKey();
30.             Book b=entry.getValue();
31.             System.out.println(key+" Details:");
32.             System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
33.         }
34.     }

```

35. }

Output:

2 Details:

102 Data Communications & Networking Forouzan Mc Graw Hill 4

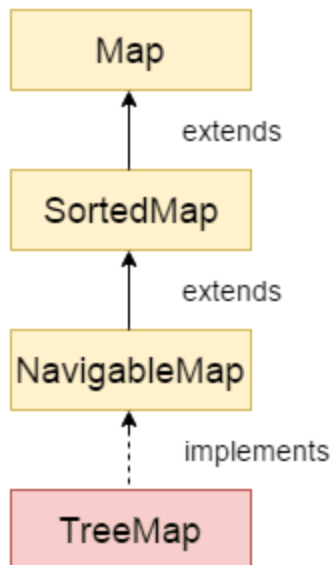
1 Details:

101 Let us C Yashwant Kanetkar BPB 8

3 Details:

103 Operating System Galvin Wiley 6

Java TreeMap class



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

The important points about Java TreeMap class are:

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

TreeMap class declaration

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

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

TreeMap class Parameters

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

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

Constructors of Java TreeMap class

Constructor	Description
TreeMap()	It is used to construct an empty tree map that will be sorted using the natural order of its key.
TreeMap(Comparator<? super K> comparator)	It is used to construct an empty tree-based map that will be sorted using the comparator comp.
TreeMap(Map<? extends K,? extends V> m)	It is used to initialize a treemap with the entries from m , which will be sorted using the natural order of the keys.
TreeMap(SortedMap<K,? extends V> m)	It is used to initialize a treemap with the entries from the SortedMap sm , which will be sorted in the same order as sm .

Methods of Java TreeMap class

Method	Description
--------	-------------

Map.Entry<K,V> ceilingEntry(K key)	It returns the key-value pair having the least key, greater than or equal to the specified key, or null if there is no such key.
K ceilingKey(K key)	It returns the least key, greater than the specified key or null if there is no such key.
void clear()	It removes all the key-value pairs from a map.
Object clone()	It returns a shallow copy of TreeMap instance.
Comparator<? super K> comparator()	It returns the comparator that arranges the key in order, or null if the map uses the natural ordering.
NavigableSet<K> descendingKeySet()	It returns a reverse order NavigableSet view of the keys contained in the map.
NavigableMap<K,V> descendingMap()	It returns the specified key-value pairs in descending order.
Map.Entry firstEntry()	It returns the key-value pair having the least key.
Map.Entry<K,V> floorEntry(K key)	It returns the greatest key, less than or equal to the specified key, or null if there is no such key.
void forEach(BiConsumer<? super K,? super V> action)	It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.
SortedMap<K,V> headMap(K toKey)	It returns the key-value pairs whose keys are strictly less than toKey.
NavigableMap<K,V> headMap(K toKey, boolean inclusive)	It returns the key-value pairs whose keys are less than (or equal to if inclusive is true) toKey.

Map.Entry<K,V> higherEntry(K key)	It returns the least key strictly greater than the given key, or null if there is no such key.
K higherKey(K key)	It is used to return true if this map contains a mapping for the specified key.
Set keySet()	It returns the collection of keys exist in the map.
Map.Entry<K,V> lastEntry()	It returns the key-value pair having the greatest key, or null if there is no such key.
Map.Entry<K,V> lowerEntry(K key)	It returns a key-value mapping associated with the greatest key strictly less than the given key, or null if there is no such key.
K lowerKey(K key)	It returns the greatest key strictly less than the given key, or null if there is no such key.
NavigableSet<K> navigableKeySet()	It returns a NavigableSet view of the keys contained in this map.
Map.Entry<K,V> pollFirstEntry()	It removes and returns a key-value mapping associated with the least key in this map, or null if the map is empty.
Map.Entry<K,V> pollLastEntry()	It removes and returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.
V put(K key, V value)	It inserts the specified value with the specified key in the map.
void putAll(Map<? extends K,? extends V> map)	It is used to copy all the key-value pair from one map to another map.
V replace(K key, V value)	It replaces the specified value for a specified key.

boolean replace(K key, V oldValue, V newValue)	It replaces the old value with the new value for a specified key.
void replaceAll(BiFunction<? super K,? super V,? extends V> function)	It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
NavigableMap<K,V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)	It returns key-value pairs whose keys range from fromKey to toKey.
SortedMap<K,V> subMap(K fromKey, K toKey)	It returns key-value pairs whose keys range from fromKey, inclusive, to toKey, exclusive.
SortedMap<K,V> tailMap(K fromKey)	It returns key-value pairs whose keys are greater than or equal to fromKey.
NavigableMap<K,V> tailMap(K fromKey, boolean inclusive)	It returns key-value pairs whose keys are greater than (or equal to, if inclusive is true) fromKey.
boolean containsKey(Object key)	It returns true if the map contains a mapping for the specified key.
boolean containsValue(Object value)	It returns true if the map maps one or more keys to the specified value.
K firstKey()	It is used to return the first (lowest) key currently in this sorted map.
V get(Object key)	It is used to return the value to which the map maps the specified key.

K lastKey()	It is used to return the last (highest) key currently in the sorted map.
V remove(Object key)	It removes the key-value pair of the specified key from the map.
Set<Map.Entry<K,V>> entrySet()	It returns a set view of the mappings contained in the map.
int size()	It returns the number of key-value pairs exists in the hashtable.
Collection values()	It returns a collection view of the values contained in the map.

Java TreeMap Example

```

1. import java.util.*;
2. class TreeMap1{
3.     public static void main(String args[]){
4.         TreeMap<Integer,String> map=new TreeMap<Integer,String>();
5.         map.put(100,"Amit");
6.         map.put(102,"Ravi");
7.         map.put(101,"Vijay");
8.         map.put(103,"Rahul");
9.
10.        for(Map.Entry m:map.entrySet()){
11.            System.out.println(m.getKey()+" "+m.getValue());
12.        }
13.    }
14. }
```

Output:100 Amit
101 Vijay
102 Ravi
103 Rahul

Java TreeMap Example: remove()

```
1. import java.util.*;
2. public class TreeMap2 {
3.     public static void main(String args[]) {
4.         TreeMap<Integer,String> map=new TreeMap<Integer,String>();
5.         map.put(100,"Amit");
6.         map.put(102,"Ravi");
7.         map.put(101,"Vijay");
8.         map.put(103,"Rahul");
9.         System.out.println("Before invoking remove() method");
10.        for(Map.Entry m:map.entrySet())
11.        {
12.            System.out.println(m.getKey()+" "+m.getValue());
13.        }
14.        map.remove(102);
15.        System.out.println("After invoking remove() method");
16.        for(Map.Entry m:map.entrySet())
17.        {
18.            System.out.println(m.getKey()+" "+m.getValue());
19.        }
20.    }
21. }
```

Output:

Before invoking remove() method

100 Amit

101 Vijay

102 Ravi

103 Rahul

After invoking remove() method

100 Amit
101 Vijay
103 Rahul

Java TreeMap Example: NavigableMap

```
1. import java.util.*;
2. class TreeMap3{
3.     public static void main(String args[]){
4.         NavigableMap<Integer,String> map=new TreeMap<Integer,String>();
5.         map.put(100,"Amit");
6.         map.put(102,"Ravi");
7.         map.put(101,"Vijay");
8.         map.put(103,"Rahul");
9.         //Maintains descending order
10.        System.out.println("descendingMap: "+map.descendingMap());
11.        //Returns key-value pairs whose keys are less than or equal to the specified key.
12.        System.out.println("headMap: "+map.headMap(102,true));
13.        //Returns key-value pairs whose keys are greater than or equal to the specified key.
14.        System.out.println("tailMap: "+map.tailMap(102,true));
15.        //Returns key-value pairs exists in between the specified key.
16.        System.out.println("subMap: "+map.subMap(100, false, 102, true));
17.    }
18. }
```

descendingMap: {103=Rahul, 102=Ravi, 101=Vijay, 100=Amit}

headMap: {100=Amit, 101=Vijay, 102=Ravi}

tailMap: {102=Ravi, 103=Rahul}

subMap: {101=Vijay, 102=Ravi}

Java TreeMap Example: SortedMap

```
1. import java.util.*;
2. class TreeMap4{
3.     public static void main(String args[]){
4.         SortedMap<Integer,String> map=new TreeMap<Integer,String>();
5.         map.put(100,"Amit");
6.         map.put(102,"Ravi");
7.         map.put(101,"Vijay");
8.         map.put(103,"Rahul");
9.         //Returns key-value pairs whose keys are less than the specified key.
10.        System.out.println("headMap: "+map.headMap(102));
11.        //Returns key-value pairs whose keys are greater than or equal to the specified key.
12.        System.out.println("tailMap: "+map.tailMap(102));
13.        //Returns key-value pairs exists in between the specified key.
14.        System.out.println("subMap: "+map.subMap(100, 102));
15.    }
16. }
```

headMap: {100=Amit, 101=Vijay}

tailMap: {102=Ravi, 103=Rahul}

subMap: {100=Amit, 101=Vijay}

What is difference between HashMap and TreeMap?

HashMap	TreeMap
1) HashMap can contain one null key.	TreeMap cannot contain any null key.
2) HashMap maintains no order.	TreeMap maintains ascending order.

Java TreeMap Example: Book


```
1. import java.util.*;
2. class Book {
3.     int id;
4.     String name,author,publisher;
5.     int quantity;
6.     public Book(int id, String name, String author, String publisher, int quantity) {
7.         this.id = id;
8.         this.name = name;
9.         this.author = author;
10.        this.publisher = publisher;
11.        this.quantity = quantity;
12.    }
13. }
14. public class MapExample {
15.     public static void main(String[] args) {
16.         //Creating map of Books
17.         Map<Integer,Book> map=new TreeMap<Integer,Book>();
18.         //Creating Books
19.         Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20.         Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21.         Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
22.         //Adding Books to map
23.         map.put(2,b2);
24.         map.put(1,b1);
25.         map.put(3,b3);
26.
27.         //Traversing map
28.         for(Map.Entry<Integer, Book> entry:map.entrySet()){
```

```
29.     int key=entry.getKey();
30.     Book b=entry.getValue();
31.     System.out.println(key+" Details:");
32.     System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
33. }
34. }
35. }
```

Output:

1 Details:

101 Let us C Yashwant Kanetkar BPB 8

2 Details:

102 Data Communications & Networking Forouzan Mc Graw Hill 4

3 Details:

103 Operating System Galvin Wiley 6

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.

1. **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.

Methods of Java Hashtable class

Method	Description
void clear()	It is used to reset the hash table.

Object clone()	It returns a shallow copy of the Hashtable.
V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)	It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null.
V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null.
Enumeration elements()	It returns an enumeration of the values in the hash table.
Set<Map.Entry<K,V>> entrySet()	It returns a set view of the mappings contained in the map.
boolean equals(Object o)	It is used to compare the specified Object with the Map.
void forEach(BiConsumer<? super K,? super V> action)	It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.
V getOrDefault(Object key, V defaultValue)	It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key.
int hashCode()	It returns the hash code value for the Map
Enumeration<K> keys()	It returns an enumeration of the keys in the hashtable.

Set<K> keySet()	It returns a Set view of the keys contained in the map.
V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)	If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
V put(K key, V value)	It inserts the specified value with the specified key in the hash table.
void putAll(Map<? extends K,? extends V> t))	It is used to copy all the key-value pair from map to hashtable.
V putIfAbsent(K key, V value)	If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.
boolean remove(Object key, Object value)	It removes the specified values with the associated specified keys from the hashtable.
V replace(K key, V value)	It replaces the specified value for a specified key.
boolean replace(K key, V oldValue, V newValue)	It replaces the old value with the new value for a specified key.
void replaceAll(BiFunction<? super K,? super V,? extends V> function)	It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
String toString()	It returns a string representation of the Hashtable object.
Collection values()	It returns a collection view of the values contained in the map.

boolean contains(Object value)	This method returns true if some value equal to the value exists within the hash table, else return false.
boolean containsValue(Object value)	This method returns true if some value equal to the value exists within the hash table, else return false.
boolean containsKey(Object key)	This method return true if some key equal to the key exists within the hash table, else return false.
boolean isEmpty()	This method returns true if the hash table is empty; returns false if it contains at least one key.
protected void rehash()	It is used to increase the size of the hash table and rehashes all of its keys.
V get(Object key)	This method returns the object that contains the value associated with the key.
V remove(Object key)	It is used to remove the key and its value. This method returns the value associated with the key.
int size()	This method returns the number of entries in the hash table.

Java Hashtable Example

1. **import** java.util.*;
2. **class** Hashtable1{
3. **public static void** main(String args[]){
4. Hashtable<Integer,String> hm=**new** Hashtable<Integer,String>();
- 5.
6. hm.put(100,"Amit");
7. hm.put(102,"Ravi");

```
8.  hm.put(101,"Vijay");
9.  hm.put(103,"Rahul");
10.
11. for(Map.Entry m:hm.entrySet()){
12.     System.out.println(m.getKey()+" "+m.getValue());
13. }
14. }
15. }
```

Test it Now

Output:

```
103 Rahul
102 Ravi
101 Vijay
100 Amit
```

Java Hashtable Example: remove()

```
1.  import java.util.*;
2.  public class Hashtable2 {
3.      public static void main(String args[]) {
4.          Hashtable<Integer,String> map=new Hashtable<Integer,String>();
5.          map.put(100,"Amit");
6.          map.put(102,"Ravi");
7.          map.put(101,"Vijay");
8.          map.put(103,"Rahul");
9.          System.out.println("Before remove: "+ map);
10.         // Remove value for key 102
11.         map.remove(102);
12.         System.out.println("After remove: "+ map);
```

13. }

14. }

Output:

Before remove: {103=Rahul, 102=Ravi, 101=Vijay, 100=Amit}

After remove: {103=Rahul, 101=Vijay, 100=Amit}

Java Hashtable Example: getOrDefault()

```
1. import java.util.*;
2. class Hashtable3{
3.     public static void main(String args[]){
4.         Hashtable<Integer,String> map=new Hashtable<Integer,String>();
5.         map.put(100,"Amit");
6.         map.put(102,"Ravi");
7.         map.put(101,"Vijay");
8.         map.put(103,"Rahul");
9.         //Here, we specify the if and else statement as arguments of the method
10.        System.out.println(map.getOrDefault(101, "Not Found"));
11.        System.out.println(map.getOrDefault(105, "Not Found"));
12.    }
13. }
```

Output:

Vijay

Not Found

Java Hashtable Example: putIfAbsent()

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



```

2.  class Hashtable4{
3.      public static void main(String args[]){
4.          Hashtable<Integer,String> map=new Hashtable<Integer,String>();
5.          map.put(100,"Amit");
6.          map.put(102,"Ravi");
7.          map.put(101,"Vijay");
8.          map.put(103,"Rahul");
9.          System.out.println("Initial Map: "+map);
10.         //Inserts, as the specified pair is unique
11.         map.putIfAbsent(104,"Gaurav");
12.         System.out.println("Updated Map: "+map);
13.         //Returns the current value, as the specified pair already exist
14.         map.putIfAbsent(101,"Vijay");
15.         System.out.println("Updated Map: "+map);
16.     }
17. }

```

Output:

Initial Map: {103=Rahul, 102=Ravi, 101=Vijay, 100=Amit}

Updated Map: {104=Gaurav, 103=Rahul, 102=Ravi, 101=Vijay, 100=Amit}

Updated Map: {104=Gaurav, 103=Rahul, 102=Ravi, 101=Vijay, 100=Amit}

Java Hashtable Example: Book

```

1.  import java.util.*;
2.  class Book {
3.      int id;
4.      String name,author,publisher;
5.      int quantity;
6.      public Book(int id, String name, String author, String publisher, int quantity) {

```

```

7.  this.id = id;
8.  this.name = name;
9.  this.author = author;
10. this.publisher = publisher;
11. this.quantity = quantity;
12. }
13. }
14. public class HashtableExample {
15. public static void main(String[] args) {
16.     //Creating map of Books
17.     Map<Integer,Book> map=new Hashtable<Integer,Book>();
18.     //Creating Books
19.     Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPP",8);
20.     Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21.     Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
22.     //Adding Books to map
23.     map.put(1,b1);
24.     map.put(2,b2);
25.     map.put(3,b3);
26.     //Traversing map
27.     for(Map.Entry<Integer, Book> entry:map.entrySet()){
28.         int key=entry.getKey();
29.         Book b=entry.getValue();
30.         System.out.println(key+" Details:");
31.         System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
32.     }
33. }
34. }

```

Output:

3 Details:

103 Operating System Galvin Wiley 6

2 Details:

102 Data Communications & Networking Forouzan Mc Graw Hill 4

1 Details:

101 Let us C Yashwant Kanetkar BPB 8

Difference between HashMap and Hashtable

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

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

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

7) Iterator in HashMap is fail-fast .	Enumerator in Hashtable is not fail-fast .
8) HashMap inherits AbstractMap class.	Hashtable inherits Dictionary class.

Java EnumMap class

Java EnumMap class is the specialized Map implementation for enum keys. It inherits Enum and AbstractMap classes.

EnumMap class hierarchy

The hierarchy of EnumMap class is given in the figure given below.

EnumMap class declaration

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

1. **public class** EnumMap<K **extends** Enum<K>,V> **extends** AbstractMap<K,V> **implements** Serializable, Cloneable

EnumMap class Parameters

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

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

Constructors of Java EnumMap class

Constructor	Description
EnumMap(Class<K> keyType)	It is used to create an empty enum map with the specified key type.

EnumMap(EnumMap<K,? extends V> m)	It is used to create an enum map with the same key type as the specified enum map.
EnumMap(Map<K,? extends V> m)	It is used to create an enum map initialized from the specified map.

Methods of Java EnumMap class

S N	Method	Description
1	<code>clear()</code>	It is used to clear all the mapping from the map.
2	<code>clone()</code>	It is used to copy the mapped value of one map to another map.
3	<code>containsKey())</code>	It is used to check whether a specified key is present in this map or not.
4	<code>containsValue() e()</code>	It is used to check whether one or more key is associated with a given value or not.
5	<code>entrySet()</code>	It is used to create a set of elements contained in the EnumMap.
6	<code>equals()</code>	It is used to compare two maps for equality.
7	<code>get()</code>	It is used to get the mapped value of the specified key.
8	<code>hashCode()</code>	It is used to get the hashCode value of the EnumMap.
9	<code>keySet()</code>	It is used to get the set view of the keys contained in the map.
10	<code>size()</code>	It is used to get the size of the EnumMap.

11	<code>Values()</code>	It is used to create a collection view of the values contained in this map.
12	<code>put()</code>	It is used to associate the given value with the given key in this EnumMap.
13	<code>putAll()</code>	It is used to copy all the mappings from one EnumMap to a new EnumMap.
14	<code>remove()</code>	It is used to remove the mapping for the given key from EnumMap if the given key is present.

Java EnumMap Example

```

1. import java.util.*;
2. public class EnumMapExample {
3.     // create an enum
4.     public enum Days {
5.         Monday, Tuesday, Wednesday, Thursday
6.     };
7.     public static void main(String[] args) {
8.         //create and populate enum map
9.         EnumMap<Days, String> map = new EnumMap<Days, String>(Days.class);
10.        map.put(Days.Monday, "1");
11.        map.put(Days.Tuesday, "2");
12.        map.put(Days.Wednesday, "3");
13.        map.put(Days.Thursday, "4");
14.        // print the map
15.        for(Map.Entry m:map.entrySet()){
16.            System.out.println(m.getKey()+" "+m.getValue());
17.        }
18.    }

```

19. }

Output:

Monday 1

Tuesday 2

Wednesday 3

Thursday 4

Java EnumMap Example: Book

```
1. import java.util.*;
2. class Book {
3.     int id;
4.     String name,author,publisher;
5.     int quantity;
6.     public Book(int id, String name, String author, String publisher, int quantity) {
7.         this.id = id;
8.         this.name = name;
9.         this.author = author;
10.        this.publisher = publisher;
11.        this.quantity = quantity;
12.    }
13. }
14. public class EnumMapExample {
15.     // Creating enum
16.     public enum Key{
17.         One, Two, Three
18.     };
19.     public static void main(String[] args) {
```

```

20. EnumMap<Key, Book> map = new EnumMap<Key, Book>(Key.class);
21. // Creating Books
22. Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
23. Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
24. Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
25. // Adding Books to Map
26. map.put(Key.One, b1);
27. map.put(Key.Two, b2);
28. map.put(Key.Three, b3);
29. // Traversing EnumMap
30. for(Map.Entry<Key, Book> entry:map.entrySet()){
31.     Book b=entry.getValue();
32.     System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
33. }
34. }
35. }

```

Output:

```

101 Let us C Yashwant Kanetkar BPB 8
102 Data Communications & Networking Forouzan Mc Graw Hill 4
103 Operating System Galvin Wiley 6

```

Java Collections class

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

The important points about Java Collections class are:

- Java Collection class supports the **polymorphic algorithms** that operate on collections.
- Java Collection class throws a **NullPointerException** if the collections or class objects provided to them are null.

Collections class declaration

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

1. **public class** Collections **extends** Object

S N	Modifier & Type	Methods	Descriptions
1)	static <T> boolean	<code>addAll()</code>	It is used to adds all of the specified elements to the specified collection.
2)	static <T> Queue<T>	<code>asLifoQueue()</code>	It returns a view of a Deque as a Last-in-first-out (LIFO) Queue.
3)	static <T> int	<code>binarySearch()</code>	It searches the list for the specified object and returns their position in a sorted list.
4)	static <E> Collection<E>	<code>checkedCollection()</code>	It is used to returns a dynamically typesafe view of the specified collection.
5)	static <E> List<E>	<code>checkedList()</code>	It is used to returns a dynamically typesafe view of the specified list.
6)	static <K,V> Map<K,V>	<code>checkedMap()</code>	It is used to returns a dynamically typesafe view of the specified map.
7)	static <K,V> NavigableMap<K, V>	<code>checkedNavigableM ap()</code>	It is used to returns a dynamically typesafe view of the specified navigable map.

8)	static <E> NavigableSet<E>	checkedNavigableSet ()	It is used to returns a dynamically typesafe view of the specified navigable set.
9)	static <E> Queue<E>	checkedQueue()	It is used to returns a dynamically typesafe view of the specified queue.
10)	static <E> Set<E>	checkedSet()	It is used to returns a dynamically typesafe view of the specified set.
11)	static <K,V> SortedMap<K,V>	checkedSortedMap()	It is used to returns a dynamically typesafe view of the specified sorted map.
12)	static <E> SortedSet<E>	checkedSortedSet()	It is used to returns a dynamically typesafe view of the specified sorted set.
13)	static <T> void	copy()	It is used to copy all the elements from one list into another list.
14)	static boolean	disjoint()	It returns true if the two specified collections have no elements in common.
15)	static <T> Enumeration<T>	emptyEnumeration()	It is used to get an enumeration that has no elements.
16)	static <T> Iterator<T>	emptyIterator()	It is used to get an Iterator that has no elements.
17)	static <T> List<T>	emptyList()	It is used to get a List that has no elements.
18)	static <T> ListIterator<T>	emptyListIterator()	It is used to get a List Iterator that has no elements.

19)	static <K,V> Map<K,V>	emptyMap()	It returns an empty map which is immutable.
20)	static <K,V> NavigableMap<K, V>	emptyNavigableMap ()	It returns an empty navigable map which is immutable.
21)	static <E> NavigableSet<E>	emptyNavigableSet()	It is used to get an empty navigable set which is immutable in nature.
22)	static <T> Set<T>	emptySet()	It is used to get the set that has no elements.
23)	static <K,V> SortedMap<K,V>	emptySortedMap()	It returns an empty sorted map which is immutable.
24)	static <E> SortedSet<E>	emptySortedSet()	It is used to get the sorted set that has no elements.
25)	static <T> Enumeration<T>	enumeration()	It is used to get the enumeration over the specified collection.
26)	static <T> void	fill()	It is used to replace all of the elements of the specified list with the specified elements.
27)	static int	frequency()	It is used to get the number of elements in the specified collection equal to the specified object.
28)	static int	indexOfSubList()	It is used to get the starting position of the first occurrence of the specified target list within the

			specified source list. It returns -1 if there is no such occurrence in the specified list.
29)	static int	<code>lastIndexOfSubList()</code>	It is used to get the starting position of the last occurrence of the specified target list within the specified source list. It returns -1 if there is no such occurrence in the specified list.
30)	static <T> ArrayList<T>	<code>list()</code>	It is used to get an array list containing the elements returned by the specified enumeration in the order in which they are returned by the enumeration.
31)	static <T extends Object & Comparable<? super T>> T	<code>max()</code>	It is used to get the maximum value of the given collection, according to the natural ordering of its elements.
32)	static <T extends Object & Comparable<? super T>> T	<code>min()</code>	It is used to get the minimum value of the given collection, according to the natural ordering of its elements.
33)	static <T> List<T>	<code>nCopies()</code>	It is used to get an immutable list consisting of n copies of the specified object.
34)	static <E> Set<E>	<code>newSetFromMap()</code>	It is used to return a set backed by the specified map.
35)	static <T> boolean	<code>replaceAll()</code>	It is used to replace all occurrences of one specified value in a list with the other specified value.

36)	static void	<code>reverse()</code>	It is used to reverse the order of the elements in the specified list.
37)	static <T> Comparator<T>	<code>reverseOrder()</code>	It is used to get the comparator that imposes the reverse of the natural ordering on a collection of objects which implement the Comparable interface.
38)	static void	<code>rotate()</code>	It is used to rotate the elements in the specified list by a given distance.
39)	static void	<code>shuffle()</code>	It is used to randomly reorders the specified list elements using a default randomness.
40)	static <T> Set<T>	<code>singleton()</code>	It is used to get an immutable set which contains only the specified object.
41)	static <T> List<T>	<code>singletonList()</code>	It is used to get an immutable list which contains only the specified object.
42)	static <K,V> Map<K,V>	<code>singletonMap()</code>	It is used to get an immutable map, mapping only the specified key to the specified value.
43)	static <T extends Comparable<? super T>>void	<code>sort()</code>	It is used to sort the elements presents in the specified list of collection in ascending order.
44)	static void	<code>swap()</code>	It is used to swap the elements at the specified positions in the specified list.
45)	static <T> Collection<T>	<code>synchronizedCollecti on()</code>	It is used to get a synchronized (thread-safe) collection backed by the specified collection.

46)	static <T> List<T>	synchronizedList()	It is used to get a synchronized (thread-safe) collection backed by the specified list.
47)	static <K,V> Map<K,V>	synchronizedMap()	It is used to get a synchronized (thread-safe) map backed by the specified map.
48)	static <K,V> NavigableMap<K, V>	synchronizedNavigableMap()	It is used to get a synchronized (thread-safe) navigable map backed by the specified navigable map.
49)	static <T> NavigableSet<T>	synchronizedNavigableSet()	It is used to get a synchronized (thread-safe) navigable set backed by the specified navigable set.
50)	static <T> Set<T>	synchronizedSet()	It is used to get a synchronized (thread-safe) set backed by the specified set.
51)	static <K,V> SortedMap<K,V>	synchronizedSortedMap()	It is used to get a synchronized (thread-safe) sorted map backed by the specified sorted map.
52)	static <T> SortedSet<T>	synchronizedSortedSet()	It is used to get a synchronized (thread-safe) sorted set backed by the specified sorted set.
53)	static <T> Collection<T>	unmodifiableCollection()	It is used to get an unmodifiable view of the specified collection.
54)	static <T> List<T>	unmodifiableList()	It is used to get an unmodifiable view of the specified list.
55)	static <K,V> Map<K,V>	unmodifiableMap()	It is used to get an unmodifiable view of the specified map.

56)	static <K,V> NavigableMap<K, V>	unmodifiableNaviga bleMap()	It is used to get an unmodifiable view of the specified navigable map.
57)	static <T> NavigableSet<T>	unmodifiableNaviga bleSet()	It is used to get an unmodifiable view of the specified navigable set.
58)	static <T> Set<T>	unmodifiableSet()	It is used to get an unmodifiable view of the specified set.
59)	static <K,V> SortedMap<K,V>	unmodifiableSorted Map()	It is used to get an unmodifiable view of the specified sorted map.
60	static <T> SortedSet<T>	unmodifiableSortedS et()	It is used to get an unmodifiable view of the specified sorted set.

Java Collections Example

1. **import** java.util.*;
2. **public class** CollectionsExample {
3. **public static void** main(String a[]){
4. List<String> list = **new** ArrayList<String>();
5. list.add("C");
6. list.add("Core Java");
7. list.add("Advance Java");
8. System.out.println("Initial collection value:"+list);
9. Collections.addAll(list, "Servlet","JSP");
10. System.out.println("After adding elements collection value:"+list);
11. String[] strArr = {"C#", ".Net"};
12. Collections.addAll(list, strArr);

```
13.     System.out.println("After adding array collection value:"+list);
14. }
15. }
```

Output:

Initial collection value:[C, Core Java, Advance Java]

After adding elements collection value:[C, Core Java, Advance Java, Servlet, JSP]

After adding array collection value:[C, Core Java, Advance Java, Servlet, JSP, C#, .Net]

Java Collections Example: max()

```
1.  import java.util.*;
2.  public class CollectionsExample {
3.      public static void main(String a[]){
4.          List<Integer> list = new ArrayList<Integer>();
5.          list.add(46);
6.          list.add(67);
7.          list.add(24);
8.          list.add(16);
9.          list.add(8);
10.         list.add(12);
11.         System.out.println("Value of maximum element from the collection: "+Collections.max(list));
12.     }
13. }
```

Output:

Value of maximum element from the collection: 67

Java Collections Example: min()

```
1. import java.util.*;
2. public class CollectionsExample {
3.     public static void main(String a[]){
4.         List<Integer> list = new ArrayList<Integer>();
5.         list.add(46);
6.         list.add(67);
7.         list.add(24);
8.         list.add(16);
9.         list.add(8);
10.        list.add(12);
11.        System.out.println("Value of minimum element from the collection: "+Collections.min(list));
12.    }
13. }
```

Output:

Value of minimum element from the collection: 8

Sorting in Collection

We can sort the elements of:

1. String objects
2. Wrapper class objects
3. User-defined class objects

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

Method of Collections class for sorting List elements

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

Note: String class and Wrapper classes implement the Comparable interface. So if you store the objects of string or wrapper classes, it will be Comparable.

Example to sort string objects

```
1. import java.util.*;
2. class TestSort1{
3.     public static void main(String args[]){
4.
5.         ArrayList<String> al=new ArrayList<String>();
6.         al.add("Viru");
7.         al.add("Saurav");
8.         al.add("Mukesh");
9.         al.add("Tahir");
10.
11.        Collections.sort(al);
12.        Iterator itr=al.iterator();
13.        while(itr.hasNext()){
14.            System.out.println(itr.next());
15.        }
16.    }
17. }
```

Test it Now

Mukesh
Saurav
Tahir
Viru

Example to sort string objects in reverse order

```
1. import java.util.*;
2. class TestSort2{
3.     public static void main(String args[]){
4.
5.         ArrayList<String> al=new ArrayList<String>();
6.         al.add("Viru");
7.         al.add("Saurav");
8.         al.add("Mukesh");
9.         al.add("Tahir");
10.
11.         Collections.sort(al,Collections.reverseOrder());
12.         Iterator i=al.iterator();
13.         while(i.hasNext())
14.         {
15.             System.out.println(i.next());
16.         }
17. }
18. }
```

Viru
Tahir
Saurav
Mukesh

Example to sort Wrapper class objects

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

```

2.  class TestSort3{
3.      public static void main(String args[]){
4.
5.      ArrayList al=new ArrayList();
6.      al.add(Integer.valueOf(201));
7.      al.add(Integer.valueOf(101));
8.      al.add(230);//internally will be converted into objects as Integer.valueOf(230)
9.
10.     Collections.sort(al);
11.
12.     Iterator itr=al.iterator();
13.     while(itr.hasNext()){
14.         System.out.println(itr.next());
15.     }
16. }
17. }

```

101

201

230

Example to sort user-defined class objects

```

1.  import java.util.*;
2.
3.  class Student implements Comparable<Student> {
4.      public String name;
5.      public Student(String name) {
6.          this.name = name;
7.      }

```

```
8.  public int compareTo(Student person) {
9.      return name.compareTo(person.name);
10.
11. }
12. }
13. public class TestSort4 {
14.     public static void main(String[] args) {
15.         ArrayList<Student> al=new ArrayList<Student>();
16.         al.add(new Student("Viru"));
17.         al.add(new Student("Saurav"));
18.         al.add(new Student("Mukesh"));
19.         al.add(new Student("Tahir"));
20.
21.         Collections.sort(al);
22.         for (Student s : al) {
23.             System.out.println(s.name);
24.         }
25.     }
26. }
```

Mukesh

Saurav

Tahir

Viru

Java Comparable interface

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

compareTo(Object obj) method

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

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

We can sort the elements of:

1. String objects
2. Wrapper class objects
3. User-defined class objects

Collections class

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

Method of Collections class for sorting List elements

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

Note: String class and Wrapper classes implement the Comparable interface by default. So if you store the objects of string or wrapper classes in a list, set or map, it will be Comparable by default.

Java Comparable Example

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

File: Student.java

```
1.  class Student implements Comparable<Student>{
2.      int rollno;
3.      String name;
4.      int age;
5.      Student(int rollno,String name,int age){
6.          this.rollno=rollno;
7.          this.name=name;
8.          this.age=age;
9.      }
10.
11.     public int compareTo(Student st){
12.         if(age==st.age)
13.             return 0;
14.         else if(age>st.age)
15.             return 1;
16.         else
17.             return -1;
18.     }
19. }
```

File: TestSort1.java

```
1.  import java.util.*;
2.  public class TestSort1{
3.      public static void main(String args[]){
4.          ArrayList<Student> al=new ArrayList<Student>();
5.          al.add(new Student(101,"Vijay",23));
6.          al.add(new Student(106,"Ajay",27));
7.          al.add(new Student(105,"Jai",21));
```

```
8.  
9. Collections.sort(al);  
10. for(Student st:al){  
11. System.out.println(st.rollno+" "+st.name+" "+st.age);  
12. }  
13. }  
14. }
```

```
105 Jai 21  
101 Vijay 23  
106 Ajay 27
```

Java Comparable Example: reverse order

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

File: Student.java

```
1. class Student implements Comparable<Student>{  
2.     int rollno;  
3.     String name;  
4.     int age;  
5.     Student(int rollno,String name,int age){  
6.         this.rollno=rollno;  
7.         this.name=name;  
8.         this.age=age;  
9.     }  
10.  
11.     public int compareTo(Student st){  
12.         if(age==st.age)  
13.             return 0;
```



```
14. else if(age<st.age)
15. return 1;
16. else
17. return -1;
18. }
19. }
```

File: TestSort2.java

```
1. import java.util.*;
2. public class TestSort2{
3.     public static void main(String args[]){
4.         ArrayList<Student> al=new ArrayList<Student>();
5.         al.add(new Student(101,"Vijay",23));
6.         al.add(new Student(106,"Ajay",27));
7.         al.add(new Student(105,"Jai",21));
8.
9.         Collections.sort(al);
10.        for(Student st:al){
11.            System.out.println(st.rollno+" "+st.name+" "+st.age);
12.        }
13.    }
14. }
```

```
106 Ajay 27
101 Vijay 23
105 Jai 21
```

Java Comparator interface

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

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

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

Methods of Java Comparator Interface

Method	Description
public int compare(Object obj1, Object obj2)	It compares the first object with the second object.
public boolean equals(Object obj)	It is used to compare the current object with the specified object.
public boolean equals(Object obj)	It is used to compare the current object with the specified object.

Collections class

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

Method of Collections class for sorting List elements

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

Java Comparator Example (Non-generic Old Style)

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

1. Student.java
2. AgeComparator.java
3. NameComparator.java
4. Simple.java

Student.java

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

```
1. class Student{
2.     int rollno;
3.     String name;
4.     int age;
5.     Student(int rollno,String name,int age){
6.         this.rollno=rollno;
7.         this.name=name;
8.         this.age=age;
9.     }
10. }
```

AgeComparator.java

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

```
1. import java.util.*;
2. class AgeComparator implements Comparator{
3.     public int compare(Object o1,Object o2){
4.         Student s1=(Student)o1;
5.         Student s2=(Student)o2;
6.         
```

```
7.  if(s1.age==s2.age)
8.      return 0;
9.  else if(s1.age>s2.age)
10.     return 1;
11. else
12.     return -1;
13. }
14. }
```

NameComparator.java

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

```
1.  import java.util.*;
2.  class NameComparator implements Comparator{
3.      public int compare(Object o1,Object o2){
4.          Student s1=(Student)o1;
5.          Student s2=(Student)o2;
6.
7.          return s1.name.compareTo(s2.name);
8.      }
9.  }
```

Simple.java

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

```
1.  import java.util.*;
2.  import java.io.*;
3.
4.  class Simple{
5.      public static void main(String args[]){
6.
```

```
7. ArrayList al=new ArrayList();
8. al.add(new Student(101,"Vijay",23));
9. al.add(new Student(106,"Ajay",27));
10. al.add(new Student(105,"Jai",21));
11.
12. System.out.println("Sorting by Name");
13.
14. Collections.sort(al,new NameComparator());
15. Iterator itr=al.iterator();
16. while(itr.hasNext()){
17. Student st=(Student)itr.next();
18. System.out.println(st.rollno+" "+st.name+" "+st.age);
19. }
20.
21. System.out.println("Sorting by age");
22.
23. Collections.sort(al,new AgeComparator());
24. Iterator itr2=al.iterator();
25. while(itr2.hasNext()){
26. Student st=(Student)itr2.next();
27. System.out.println(st.rollno+" "+st.name+" "+st.age);
28. }
29.
30.
31. }
32. }
```

Sorting by Name

106 Ajay 27

105 Jai 21
101 Vijay 23

Sorting by age

105 Jai 21
101 Vijay 23
106 Ajay 27

Java Comparator Example (Generic)

Student.java

```
1. class Student{
2.     int rollno;
3.     String name;
4.     int age;
5.     Student(int rollno,String name,int age){
6.         this.rollno=rollno;
7.         this.name=name;
8.         this.age=age;
9.     }
10. }
```

AgeComparator.java

```
1. import java.util.*;
2. class AgeComparator implements Comparator<Student>{
3.     public int compare(Student s1,Student s2){
4.         if(s1.age==s2.age)
5.             return 0;
6.         else if(s1.age>s2.age)
```

```
7. return 1;
8. else
9. return -1;
10. }
11. }
```

NameComparator.java

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

```
1. import java.util.*;
2. class NameComparator implements Comparator<Student>{
3. public int compare(Student s1,Student s2){
4. return s1.name.compareTo(s2.name);
5. }
6. }
```

Simple.java

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

```
1. import java.util.*;
2. import java.io.*;
3. class Simple{
4. public static void main(String args[]){
5.
6. ArrayList<Student> al=new ArrayList<Student>();
7. al.add(new Student(101,"Vijay",23));
8. al.add(new Student(106,"Ajay",27));
9. al.add(new Student(105,"Jai",21));
10.
11. System.out.println("Sorting by Name");
12. }
```

```
13. Collections.sort(al,new NameComparator());
14. for(Student st: al){
15. System.out.println(st.rollno+" "+st.name+" "+st.age);
16. }
17.
18. System.out.println("Sorting by age");
19.
20. Collections.sort(al,new AgeComparator());
21. for(Student st: al){
22. System.out.println(st.rollno+" "+st.name+" "+st.age);
23. }
24. }
25. }
```

Sorting by Name

```
106 Ajay 27
105 Jai 21
101 Vijay 23
```

Sorting by age

```
105 Jai 21
101 Vijay 23
106 Ajay 27
```

Java 8 Comparator interface

Java 8 Comparator interface is a functional interface that contains only one abstract method. Now, we can use the Comparator interface as the assignment target for a lambda expression or method reference.

Methods of Java 8 Comparator Interface

Method	Description
int compare(T o1, T o2)	It compares the first object with second object.
static <T,U extends Comparable<? super U>> Comparator<T> comparing(Function<? super T,? extends U> keyExtractor)	It accepts a function that extracts a Comparable sort key from a type T, and returns a Comparator that compares by that sort key.
static <T,U> Comparator<T> comparing(Function<? super T,? extends U> keyExtractor, Comparator<? super U> keyComparator)	It accepts a function that extracts a sort key from a type T, and returns a Comparator that compares by that sort key using the specified Comparator.
static <T> Comparator<T> comparingDouble(ToDoubleFunction<? super T> keyExtractor)	It accepts a function that extracts a double sort key from a type T, and returns a Comparator that compares by that sort key.
static <T> Comparator<T> comparingInt(ToIntFunction<? super T> keyExtractor)	It accepts a function that extracts an int sort key from a type T, and returns a Comparator that compares by that sort key.
static <T> Comparator<T> comparingLong(ToLongFunction<? super T> keyExtractor)	It accepts a function that extracts a long sort key from a type T, and returns a Comparator that compares by that sort key.
boolean equals(Object obj)	It is used to compare the current object with the specified object.

static <T extends Comparable<? super T>> Comparator<T> naturalOrder()	It returns a comparator that compares Comparable objects in natural order.
static <T> Comparator<T> nullsFirst(Comparator<? super T> comparator)	It returns a comparator that treats null to be less than non-null elements.
static <T> Comparator<T> nullsLast(Comparator<? super T> comparator)	It returns a comparator that treats null to be greater than non-null elements.
default Comparator<T> reversed()	It returns comparator that contains reverse ordering of the provided comparator.
static <T extends Comparable<? super T>> Comparator<T> reverseOrder()	It returns comparator that contains reverse of natural ordering.
default Comparator<T> thenComparing(Comparator<? super T> other)	It returns a lexicographic-order comparator with another comparator.
default <U extends Comparable<? super U>> Comparator<T> thenComparing(Function<? super T,? extends U> keyExtractor)	It returns a lexicographic-order comparator with a function that extracts a Comparable sort key.
default <U> Comparator<T> thenComparing(Function<? super T,? extends U> keyExtractor, Comparator<? super U> keyComparator)	It returns a lexicographic-order comparator with a function that extracts a key to be compared with the given Comparator.

<pre>default Comparator<T> thenComparingDouble(ToDoubleFunction<? super T> keyExtractor)</pre>	<p>It returns a lexicographic-order comparator with a function that extracts a double sort key.</p>
<pre>default Comparator<T> thenComparingInt(ToIntFunction<? super T> keyExtractor)</pre>	<p>It returns a lexicographic-order comparator with a function that extracts a int sort key.</p>
<pre>default Comparator<T> thenComparingLong(ToLongFunction<? super T> keyExtractor)</pre>	<p>It returns a lexicographic-order comparator with a function that extracts a long sort key.</p>

Java 8 Comparator Example

Let's see the example of sorting the elements of List on the basis of age and name.

File: Student.java

```

1.  class Student {
2.      int rollno;
3.      String name;
4.      int age;
5.      Student(int rollno,String name,int age){
6.          this.rollno=rollno;
7.          this.name=name;
8.          this.age=age;
9.      }
10.
11.     public int getRollno() {
```

```
12.     return rollno;
13. }
14.
15. public void setRollno(int rollno) {
16.     this.rollno = rollno;
17. }
18.
19. public String getName() {
20.     return name;
21. }
22.
23. public void setName(String name) {
24.     this.name = name;
25. }
26.
27. public int getAge() {
28.     return age;
29. }
30.
31. public void setAge(int age) {
32.     this.age = age;
33. }
34.
35. }
```

File: TestSort1.java

```
1. import java.util.*;
2. public class TestSort1{
3. public static void main(String args[]){
```

```

4.  ArrayList<Student> al=new ArrayList<Student>();
5.  al.add(new Student(101,"Vijay",23));
6.  al.add(new Student(106,"Ajay",27));
7.  al.add(new Student(105,"Jai",21));
8.  /Sorting elements on the basis of name
9.  Comparator<Student> cm1=Comparator.comparing(Student::getName);
10. Collections.sort(al,cm1);
11. System.out.println("Sorting by Name");
12. for(Student st: al){
13.     System.out.println(st.rollno+" "+st.name+" "+st.age);
14. }
15. //Sorting elements on the basis of age
16. Comparator<Student> cm2=Comparator.comparing(Student::getAge);
17. Collections.sort(al,cm2);
18. System.out.println("Sorting by Age");
19. for(Student st: al){
20.     System.out.println(st.rollno+" "+st.name+" "+st.age);
21. }
22. }
23. }

```

Sorting by Name

106 Ajay 27

105 Jai 21

101 Vijay 23

Sorting by Age

105 Jai 21

101 Vijay 23

106 Ajay 27

Java 8 Comparator Example: nullsFirst() and nullsLast() method

Here, we sort the list of elements that also contains null.

File: Student.java

```
1.  class Student {
2.      int rollno;
3.      String name;
4.      int age;
5.      Student(int rollno,String name,int age){
6.          this.rollno=rollno;
7.          this.name=name;
8.          this.age=age;
9.      }
10.     public int getRollno() {
11.         return rollno;
12.     }
13.     public void setRollno(int rollno) {
14.         this.rollno = rollno;
15.     }
16.     public String getName() {
17.         return name;
18.     }
19.
20.     public void setName(String name) {
21.         this.name = name;
22.     }
23.
```

```
24. public int getAge() {
25.     return age;
26. }
27. public void setAge(int age) {
28.     this.age = age;
29. }
30. }
```

File: TestSort2.java

```
1. import java.util.*;
2. public class TestSort2{
3.     public static void main(String args[]){
4.         ArrayList<Student> al=new ArrayList<Student>();
5.         al.add(new Student(101,"Vijay",23));
6.         al.add(new Student(106,"Ajay",27));
7.         al.add(new Student(105,null,21));
8.         Comparator<Student>
           cm1=Comparator.comparing(Student::getName,Comparator.nullsFirst(String::compareTo));
9.         Collections.sort(al,cm1);
10.        System.out.println("Considers null to be less than non-null");
11.        for(Student st: al){
12.            System.out.println(st.rollno+" "+st.name+" "+st.age);
13.        }
14.        Comparator<Student>
           cm2=Comparator.comparing(Student::getName,Comparator.nullsLast(String::compareTo));
15.        Collections.sort(al,cm2);
16.        System.out.println("Considers null to be greater than non-null");
17.        for(Student st: al){
18.            System.out.println(st.rollno+" "+st.name+" "+st.age);
```

```
19.    }  
20.  }  
21.  }
```

Considers null to be less than non-null

105 null 21

106 Ajay 27

101 Vijay 23

Considers null to be greater than non-null

106 Ajay 27

101 Vijay 23

105 null 21

Difference between Comparable and Comparator

Comparable and Comparator both are interfaces and can be used to sort collection elements.

However, there are many differences between Comparable and Comparator interfaces that are given below.

Comparable	Comparator
1) Comparable provides a single sorting sequence . In other words, we can sort the collection on the basis of a single element such as id, name, and price.	The Comparator provides multiple sorting sequences . In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc.
2) Comparable affects the original class , i.e., the actual class is modified.	Comparator doesn't affect the original class , i.e., the actual class is not modified.
3) Comparable provides compareTo() method to sort elements.	Comparator provides compare() method to sort elements.

4) Comparable is present in java.lang package.	A Comparator is present in the java.util package.
5) We can sort the list elements of Comparable type by Collections.sort(List) method.	We can sort the list elements of Comparator type by Collections.sort(List, Comparator) method.

Java Comparable Example

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

File: TestSort3.java

```

1. //Java Program to demonstrate the use of Java Comparable.
2. //Creating a class which implements Comparable Interface
3. import java.util.*;
4. import java.io.*;
5. class Student implements Comparable<Student>{
6.     int rollno;
7.     String name;
8.     int age;
9.     Student(int rollno,String name,int age){
10.         this.rollno=rollno;
11.         this.name=name;
12.         this.age=age;
13.     }
14.     public int compareTo(Student st){
15.         if(age==st.age)
16.             return 0;
17.         else if(age>st.age)
18.             return 1;
19.         else

```

```
20. return -1;
21. }
22. }
23. //Creating a test class to sort the elements
24. public class TestSort3{
25. public static void main(String args[]){
26. ArrayList<Student> al=new ArrayList<Student>();
27. al.add(new Student(101,"Vijay",23));
28. al.add(new Student(106,"Ajay",27));
29. al.add(new Student(105,"Jai",21));
30.
31. Collections.sort(al);
32. for(Student st:al){
33. System.out.println(st.rollno+" "+st.name+" "+st.age);
34. }
35. }
36. }
```

Test it Now

Output:

```
105 Jai 21
101 Vijay 23
106 Ajay 27
```

Java Comparator Example

Let's see an example of the Java Comparator interface where we are sorting the elements of a list using different comparators.

Student.java

```
1. class Student{
```

```
2.  int rollno;
3.  String name;
4.  int age;
5.  Student(int rollno,String name,int age){
6.  this.rollno=rollno;
7.  this.name=name;
8.  this.age=age;
9.  }
10. }
```

AgeComparator.java

```
1.  import java.util.*;
2.  class AgeComparator implements Comparator<Student>{
3.  public int compare(Student s1,Student s2){
4.  if(s1.age==s2.age)
5.  return 0;
6.  else if(s1.age>s2.age)
7.  return 1;
8.  else
9.  return -1;
10. }
11. }
```

NameComparator.java

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

```
1.  import java.util.*;
2.  class NameComparator implements Comparator<Student>{
3.  public int compare(Student s1,Student s2){
4.  return s1.name.compareTo(s2.name);
```

5. }
6. }

TestComparator.java

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

1. *//Java Program to demonstrate the use of Java Comparator*
2. **import** java.util.*;
3. **import** java.io.*;
4. **class** TestComparator{
5. **public static void** main(String args[]){
6. *//Creating a list of students*
7. ArrayList<Student> al=**new** ArrayList<Student>();
8. al.add(**new** Student(101,"Vijay",23));
9. al.add(**new** Student(106,"Ajay",27));
10. al.add(**new** Student(105,"Jai",21));
- 11.
12. System.out.println("Sorting by Name");
13. *//Using NameComparator to sort the elements*
14. Collections.sort(al,**new** NameComparator());
15. *//Traversing the elements of list*
16. **for**(Student st: al){
17. System.out.println(st.rollno+" "+st.name+" "+st.age);
18. }
- 19.
20. System.out.println("sorting by Age");
21. *//Using AgeComparator to sort the elements*
22. Collections.sort(al,**new** AgeComparator());
23. *//Travering the list again*
24. **for**(Student st: al){

```
25. System.out.println(st.rollno+" "+st.name+" "+st.age);  
26. }  
27.  
28. }  
29. }
```

Output:

Sorting by Name

106 Ajay 27

105 Jai 21

101 Vijay 23

Sorting by Age

105 Jai 21

101 Vijay 23

106 Ajay 27

Properties class in Java

The **properties** object contains key and value pair both as a string. The `java.util.Properties` class is the subclass of `Hashtable`.

It can be used to get property value based on the property key. The `Properties` class provides methods to get data from the properties file and store data into the properties file. Moreover, it can be used to get the properties of a system.

An Advantage of the properties file

Recompilation is not required if the information is changed from a properties file: If any information is changed from the properties file, you don't need to recompile the java class. It is used to store information which is to be changed frequently.

Constructors of Properties class

Method	Description
<code>Properties()</code>	It creates an empty property list with no default values.

Properties(Properties defaults)	It creates an empty property list with the specified defaults.
---------------------------------	--

Methods of Properties class

The commonly used methods of Properties class are given below.

Method	Description
public void load(Reader r)	It loads data from the Reader object.
public void load(InputStream is)	It loads data from the InputStream object
public void loadFromXML(InputStream in)	It is used to load all of the properties represented by the XML document on the specified input stream into this properties table.
public String getProperty(String key)	It returns value based on the key.
public String getProperty(String key, String defaultValue)	It searches for the property with the specified key.
public void setProperty(String key, String value)	It calls the put method of Hashtable.
public void list(PrintStream out)	It is used to print the property list out to the specified output stream.
public void list(PrintWriter out))	It is used to print the property list out to the specified output stream.

<code>public Enumeration<?> propertyNames()</code>	It returns an enumeration of all the keys from the property list.
<code>public Set<String> stringPropertyNames()</code>	It returns a set of keys in from property list where the key and its corresponding value are strings.
<code>public void store(Writer w, String comment)</code>	It writes the properties in the writer object.
<code>public void store(OutputStream os, String comment)</code>	It writes the properties in the OutputStream object.
<code>public void storeToXML(OutputStream os, String comment)</code>	It writes the properties in the writer object for generating XML document.
<code>public void storeToXML(Writer w, String comment, String encoding)</code>	It writes the properties in the writer object for generating XML document with the specified encoding.

Example of Properties class to get information from the properties file

To get information from the properties file, create the properties file first.

db.properties

1. user=system
2. password=oracle

Now, let's create the java class to read the data from the properties file.

Test.java

1. **import** java.util.*;

```
2. import java.io.*;
3. public class Test {
4.     public static void main(String[] args)throws Exception{
5.         FileReader reader=new FileReader("db.properties");
6.
7.         Properties p=new Properties();
8.         p.load(reader);
9.
10.        System.out.println(p.getProperty("user"));
11.        System.out.println(p.getProperty("password"));
12.    }
13. }
```

Output:system
oracle

Now if you change the value of the properties file, you don't need to recompile the java class. That means no maintenance problem.

Example of Properties class to get all the system properties

By System.getProperties() method we can get all the properties of the system. Let's create the class that gets information from the system properties.

Test.java

```
1. import java.util.*;
2. import java.io.*;
3. public class Test {
4.     public static void main(String[] args)throws Exception{
5.
6.         Properties p=System.getProperties();
```



```
7. Set set=p.entrySet();
8.
9. Iterator itr=set.iterator();
10. while(itr.hasNext()){
11. Map.Entry entry=(Map.Entry)itr.next();
12. System.out.println(entry.getKey()+" = "+entry.getValue());
13. }
14.
15. }
16. }
```

Output:

```
java.runtime.name = Java(TM) SE Runtime Environment
sun.boot.library.path = C:\Program Files\Java\jdk1.7.0_01\jre\bin
java.vm.version = 21.1-b02
java.vm.vendor = Oracle Corporation
java.vendor.url = http://java.oracle.com/
path.separator = ;
java.vm.name = Java HotSpot(TM) Client VM
file.encoding.pkg = sun.io
user.country = US
user.script =
sun.java.launcher = SUN_STANDARD
.....
```

Example of Properties class to create the properties file

Now let's write the code to create the properties file.

Test.java

```
1. import java.util.*;
2. import java.io.*;
3. public class Test {
4.     public static void main(String[] args)throws Exception{
```

```
5.  
6. Properties p=new Properties();  
7. p.setProperty("name","Sonoo Jaiswal");  
8. p.setProperty("email","sonoojaiswal@javatpoint.com");  
9.  
10. p.store(new FileWriter("info.properties"),"Javatpoint Properties Example");  
11.  
12. }  
13. }
```

Let's see the generated properties file.

info.properties

```
1. #Javatpoint Properties Example  
2. #Thu Oct 03 22:35:53 IST 2013  
3. email=sonoojaiswal@javatpoint.com  
4. name=Sonoo Jaiswal
```

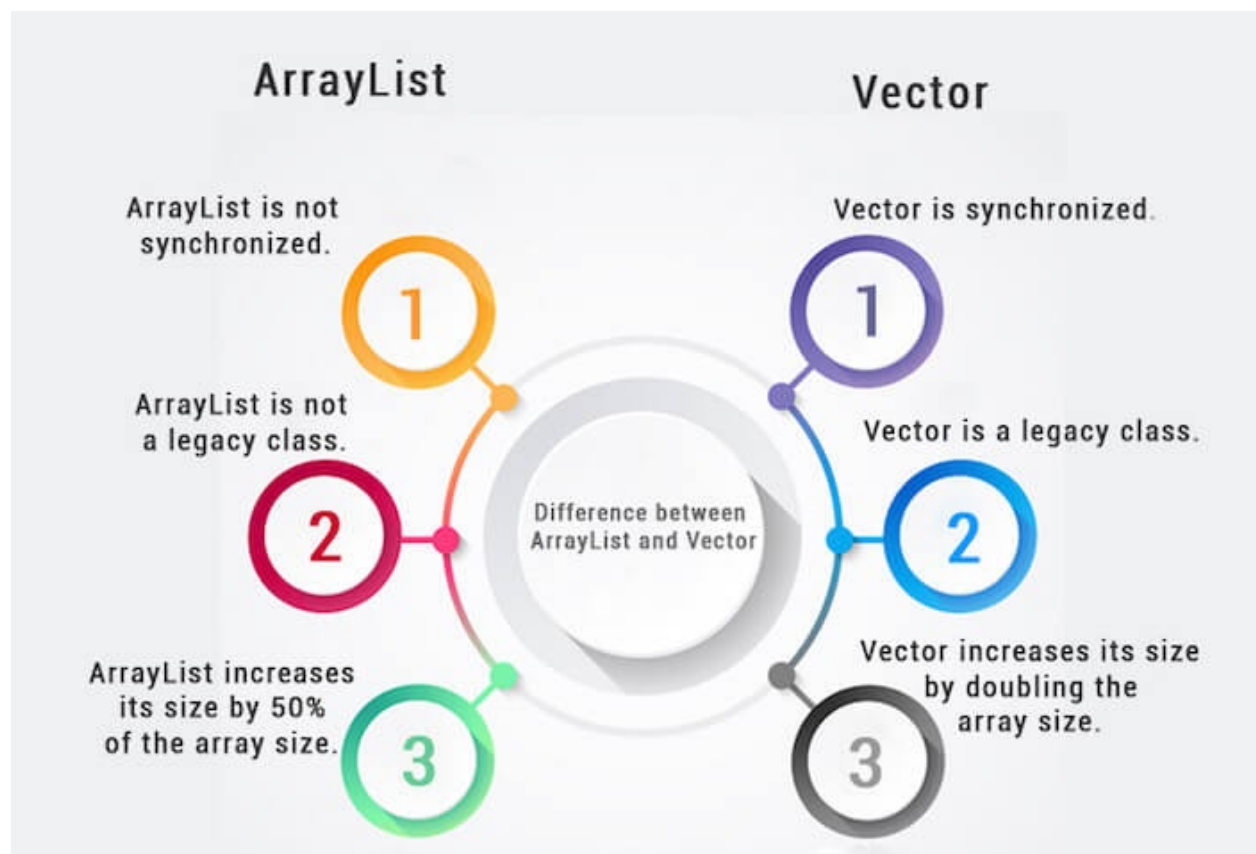
Difference between ArrayList and Vector

ArrayList and Vector both implements List interface and maintains insertion order.

However, there are many differences between ArrayList and Vector classes that are given below.

ArrayList	Vector
1) ArrayList is not synchronized .	Vector is synchronized .
2) ArrayList increments 50% of current array size if the number of elements exceeds from its capacity.	Vector increments 100% means doubles the array size if the total number of elements exceeds than its capacity.

3) ArrayList is not a legacy class. It is introduced in JDK 1.2.	Vector is a legacy class.
4) ArrayList is fast because it is non-synchronized.	Vector is slow because it is synchronized, i.e., in a multithreading environment, it holds the other threads in runnable or non-runnable state until current thread releases the lock of the object.
5) ArrayList uses the Iterator interface to traverse the elements.	A Vector can use the Iterator interface or Enumeration interface to traverse the elements.



Example of Java ArrayList

Let's see a simple example where we are using ArrayList to store and traverse the elements.

```
1. import java.util.*;
2. class TestArrayList21{
3.     public static void main(String args[]){
4.
5.         List<String> al=new ArrayList<String>();//creating arraylist
6.         al.add("Sonoo");//adding object in arraylist
7.         al.add("Michael");
8.         al.add("James");
9.         al.add("Andy");
10.        //traversing elements using Iterator
11.        Iterator itr=al.iterator();
12.        while(itr.hasNext()){
13.            System.out.println(itr.next());
14.        }
15.    }
16. }
```

Test it Now

Output:

```
Sonoo
Michael
James
Andy
```

Example of Java Vector

Let's see a simple example of a Java Vector class that uses the Enumeration interface.

```
1. import java.util.*;
2. class TestVector1{
3.     public static void main(String args[]){
```

```
4. Vector<String> v=new Vector<String>();//creating vector
5. v.add("umesh");//method of Collection
6. v.addElement("irfan");//method of Vector
7. v.addElement("kumar");
8. //traversing elements using Enumeration
9. Enumeration e=v.elements();
10. while(e.hasMoreElements()){
11.     System.out.println(e.nextElement());
12. }
13. }
14. }
```

Test it Now

Output:

umesh

irfan

kumar

Java Vector

Vector is like the *dynamic array* which can grow or shrink its size. Unlike array, we can store n-number of elements in it as there is no size limit. It is a part of Java Collection framework since Java 1.2. It is found in the `java.util` package and implements the *List* interface, so we can use all the methods of List interface here.

It is recommended to use the Vector class in the thread-safe implementation only. If you don't need to use the thread-safe implementation, you should use the ArrayList, the ArrayList will perform better in such case.

The Iterators returned by the Vector class are *fail-fast*. In case of concurrent modification, it fails and throws the `ConcurrentModificationException`.

It is similar to the ArrayList, but with two differences-

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

Java Vector class Declaration

1. `public class` Vector<E>
2. `extends` Object<E>
3. `implements` List<E>, Cloneable, Serializable

Java Vector Constructors

Vector class supports four types of constructors. These are given below:

S	Constructor	Description
N		

1	<code>vector()</code>)	It constructs an empty vector with the default size as 10.
2	<code>vector(int initialCapacity)</code>)	It constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.
3	<code>vector(int initialCapacity, int capacityIncrement)</code>	It constructs an empty vector with the specified initial capacity and capacity increment.
4	<code>Vector(Collection<? extends E> c)</code>	It constructs a vector that contains the elements of a collection c.

Java Vector Methods

The following are the list of Vector class methods:

S	M	Description
N	et	
h	o	
d	d	
1	<code>add()</code>	It is used to append the specified element in the given vector.
2	<code>add) All()</code>	It is used to append all of the elements in the specified

		collection to the end of this Vector.
3)	<code>add Element() ()</code>	It is used to append the specified component to the end of this vector. It increases the vector size by one.
4)	<code>capacity() ()</code>	It is used to get the current capacity of this vector.
5)	<code>clear() ()</code>	It is used to delete all of the elements from this vector.
6)	<code>clone() ()</code>	It returns a clone of this vector.
7)	<code>contains() ()</code>	It returns true if the vector contains the specified element.
8)	<code>containsAll(Collection)</code>	It returns true if the vector contains all of the elements in the specified collection.
9)	<code>copyInto(int array) ()</code>	It is used to copy the components of the vector into the specified array.

1 0)	ele men tAt()	It is used to get the component at the specified index.
1 1)	ele men ts()	It returns an enumeration of the components of a vector.
1 2)	ensu reCa paci ty()	It is used to increase the capacity of the vector which is in use, if necessary. It ensures that the vector can hold at least the number of components specified by the minimum capacity argument.
1 3)	equa ls()	It is used to compare the specified object with the vector for equality.
1 4)	first Ele men t()	It is used to get the first component of the vector.
1 5)	forE ach()	It is used to perform the given action for each element of the Iterable until all elements have

		been processed or the action throws an exception.
1 6)	<code>get()</code>	It is used to get an element at the specified position in the vector.
1 7)	<code>hashCode()</code>	It is used to get the hash code value of a vector.
1 8)	<code>indexOf()</code>	It is used to get the index of the first occurrence of the specified element in the vector. It returns -1 if the vector does not contain the element.
1 9)	<code>insertAt()</code>	It is used to insert the specified object as a component in the given vector at the specified index.
2 0)	<code>isEmpty()</code>	It is used to check if this vector has no components.
2 1)	<code>iterator()</code>	It is used to get an iterator over the elements in the list in proper sequence.

2 2)	last Ele men t()	It is used to get the last component of the vector.
2 3)	lastI nde xOf()	It is used to get the index of the last occurrence of the specified element in the vector. It returns -1 if the vector does not contain the element.
2 4)	listIt erat or()	It is used to get a list iterator over the elements in the list in proper sequence.
2 5)	rem ove()	It is used to remove the specified element from the vector. If the vector does not contain the element, it is unchanged.
2 6)	rem ove All()	It is used to delete all the elements from the vector that are present in the specified collection.
2 7)	rem ove AllE lem	It is used to remove all elements from the vector and set the size of the vector to zero.

	ents ()	
2 8)	rem ove Ele men t()	It is used to remove the first (lowest-indexed) occurrence of the argument from the vector.
2 9)	rem ove Ele men tAt()	It is used to delete the component at the specified index.
3 0)	rem oveI f()	It is used to remove all of the elements of the collection that satisfy the given predicate.
3 1)	rem ove Ran ge()	It is used to delete all of the elements from the vector whose index is between fromIndex, inclusive and toIndex, exclusive.
3 2)	repl ace All()	It is used to replace each element of the list with the result of applying the operator to that element.

3 3)	retainAll() 0	It is used to retain only that element in the vector which is contained in the specified collection.
3 4)	set()	It is used to replace the element at the specified position in the vector with the specified element.
3 5)	setElementAt() At()	It is used to set the component at the specified index of the vector to the specified object.
3 6)	setSize()	It is used to set the size of the given vector.
3 7)	size()	It is used to get the number of components in the given vector.
3 8)	sort()	It is used to sort the list according to the order induced by the specified Comparator.
3 9)	splitErat() or()	It is used to create a late-binding and fail-fast Spliterator over the elements in the list.

4 0)	sub List()	It is used to get a view of the portion of the list between fromIndex, inclusive, and toIndex, exclusive.
4 1)	toAr ray()	It is used to get an array containing all of the elements in this vector in correct order.
4 2)	toSt ring) 0	It is used to get a string representation of the vector.
4 3)	trim ToSi ze()	It is used to trim the capacity of the vector to the vector's current size.

Java Vector Example

1. **import** java.util.*;
2. **public class** VectorExample {
3. **public static void** main(String args[]) {
4. //Create a vector
5. Vector<String> vec = **new** Vector<String>();
6. //Adding elements using add() method of List
7. vec.add("Tiger");
8. vec.add("Lion");
9. vec.add("Dog");
10. vec.add("Elephant");

```
11. //Adding elements using addElement() method of Vector
12. vec.addElement("Rat");
13. vec.addElement("Cat");
14. vec.addElement("Deer");
15.
16. System.out.println("Elements are: "+vec);
17. }
18. }
```

Test it Now

Output:

Elements are: [Tiger, Lion, Dog, Elephant, Rat, Cat, Deer]

Java Vector Example 2

```
1. import java.util.*;
2. public class VectorExample1 {
3.     public static void main(String args[]) {
4.         //Create an empty vector with initial capacity 4
5.         Vector<String> vec = new Vector<String>(4);
6.         //Adding elements to a vector
7.         vec.add("Tiger");
8.         vec.add("Lion");
9.         vec.add("Dog");
10.        vec.add("Elephant");
11.        //Check size and capacity
12.        System.out.println("Size is: "+vec.size());
```

```

13.    System.out.println("Default capacity is: "+vec.capacity());
14.    //Display Vector elements
15.    System.out.println("Vector element is: "+vec);
16.    vec.addElement("Rat");
17.    vec.addElement("Cat");
18.    vec.addElement("Deer");
19.    //Again check size and capacity after two insertions
20.    System.out.println("Size after addition: "+vec.size());
21.    System.out.println("Capacity after addition is: "+vec.capacity());
22.    //Display Vector elements again
23.    System.out.println("Elements are: "+vec);
24.    //Checking if Tiger is present or not in this vector
25.    if(vec.contains("Tiger"))
26.    {
27.        System.out.println("Tiger is present at the index " +vec.indexOf("Tiger"));
28.    }
29.    else
30.    {
31.        System.out.println("Tiger is not present in the list.");
32.    }
33.    //Get the first element
34.    System.out.println("The first animal of the vector is = "+vec.firstElement());
35.    //Get the last element
36.    System.out.println("The last animal of the vector is = "+vec.lastElement());
37.    }
38. }

```

Test it Now

Output:

Size is: 4
Default capacity is: 4
Vector element is: [Tiger, Lion, Dog, Elephant]
Size after addition: 7
Capacity after addition is: 8
Elements are: [Tiger, Lion, Dog, Elephant, Rat, Cat, Deer]
Tiger is present at the index 0
The first animal of the vector is = Tiger
The last animal of the vector is = Deer

Java Vector Example 3

```
1.  import java.util.*;
2.  public class VectorExample2 {
3.      public static void main(String args[]) {
4.          //Create an empty Vector
5.          Vector<Integer> in = new Vector<>();
6.          //Add elements in the vector
7.          in.add(100);
8.          in.add(200);
9.          in.add(300);
10.         in.add(200);
11.         in.add(400);
12.         in.add(500);
13.         in.add(600);
14.         in.add(700);
15.         //Display the vector elements
16.         System.out.println("Values in vector: " +in);
17.         //use remove() method to delete the first occurrence of an element
18.         System.out.println("Remove first occurrence of element 200: "+in.remove((Integer)200));
```

```

19.    //Display the vector elements afre remove() method
20.    System.out.println("Values in vector: " +in);
21.    //Remove the element at index 4
22.    System.out.println("Remove element at index 4: " +in.remove(4));
23.    System.out.println("New Value list in vector: " +in);
24.    //Remove an element
25.    in.removeElementAt(5);
26.    //Checking vector and displays the element
27.    System.out.println("Vector element after removal: " +in);
28.    //Get the hashCode for this vector
29.    System.out.println("Hash code of this vector = "+in.hashCode());
30.    //Get the element at specified index
31.    System.out.println("Element at index 1 is = "+in.get(1));
32.    }
33. }

```

Test it Now

Output:

```

Values in vector: [100, 200, 300, 200, 400, 500, 600, 700]
Remove first occurence of element 200: true
Values in vector: [100, 300, 200, 400, 500, 600, 700]
Remove element at index 4: 500
New Value list in vector: [100, 300, 200, 400, 600, 700]
Vector element after removal: [100, 300, 200, 400, 600]
Hash code of this vector = 130123751
Element at index 1 is = 300

```

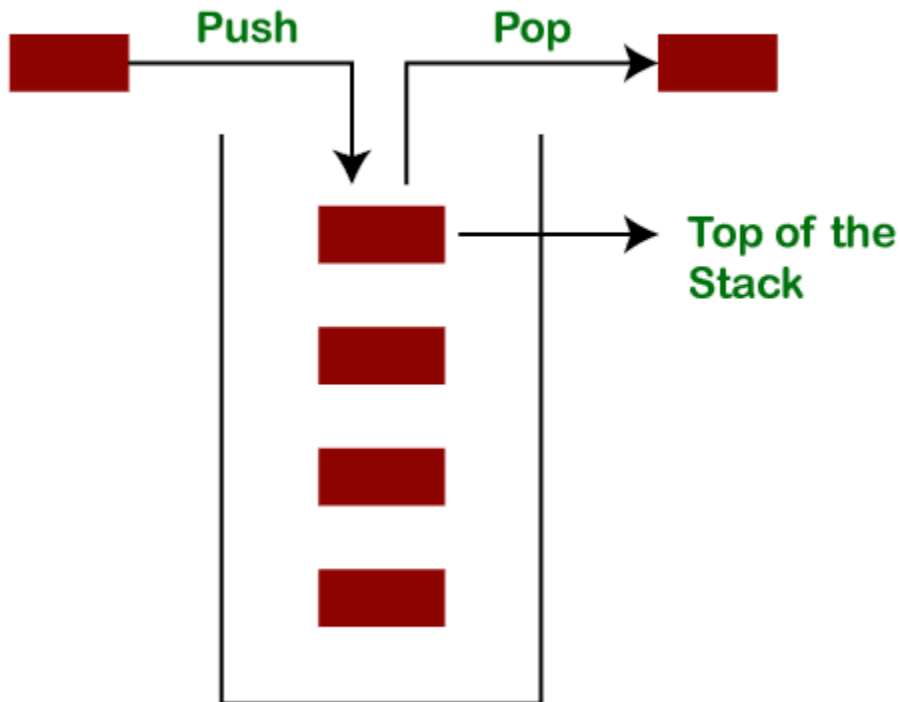
Java Stack

The **stack** is a linear data structure that is used to store the collection of objects. It is based on **Last-In-First-Out (LIFO)**. **Java collection** framework provides many interfaces and classes to store the

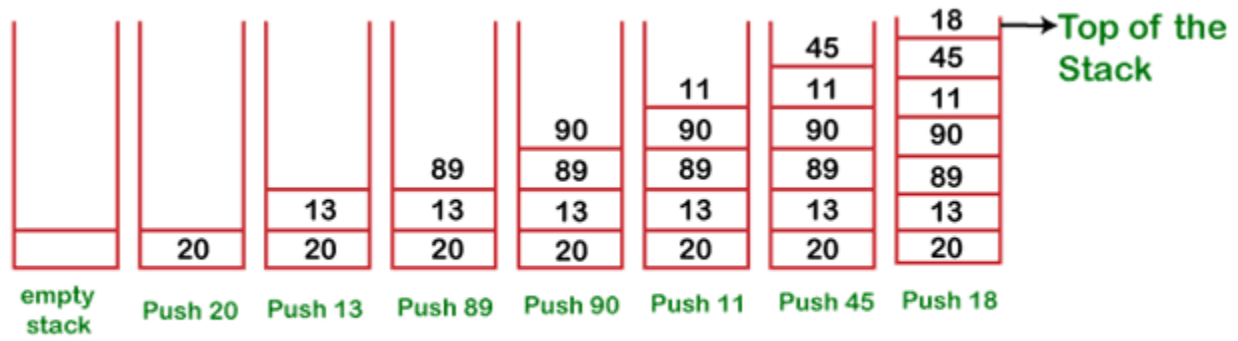
collection of objects. One of them is the **Stack class** that provides different operations such as push, pop, search, etc.

In this section, we will discuss the **Java Stack class**, its **methods**, and **implement** the stack data structure in a **Java program**. But before moving to the Java Stack class have a quick view of how the stack works.

The stack data structure has the two most important operations that are **push** and **pop**. The push operation inserts an element into the stack and pop operation removes an element from the top of the stack. Let's see how they work on stack.

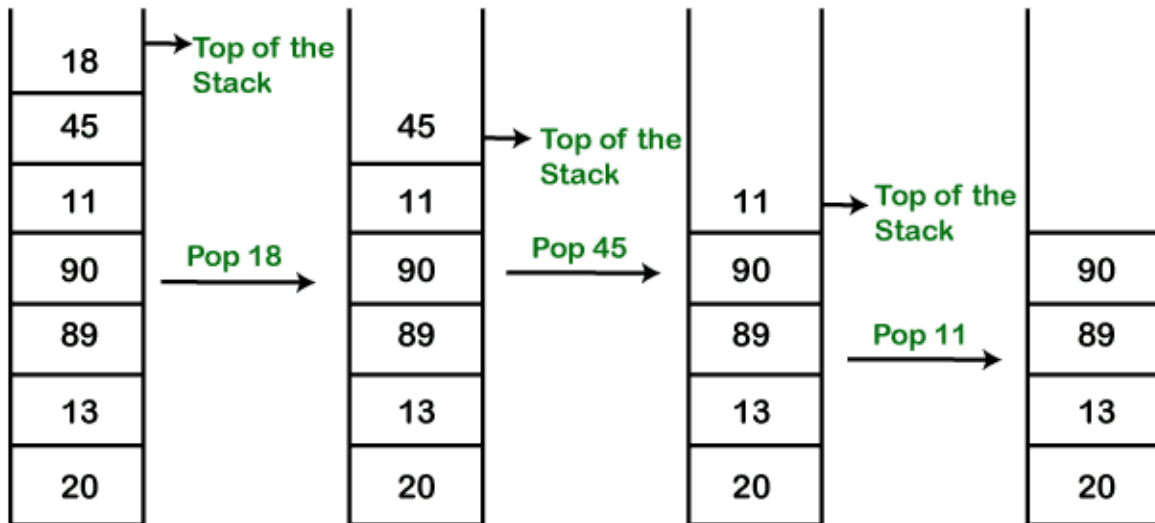


Let's push 20, 13, 89, 90, 11, 45, 18, respectively into the stack.



Push operation

Let's remove (pop) 18, 45, and 11 from the stack.



Pop operation

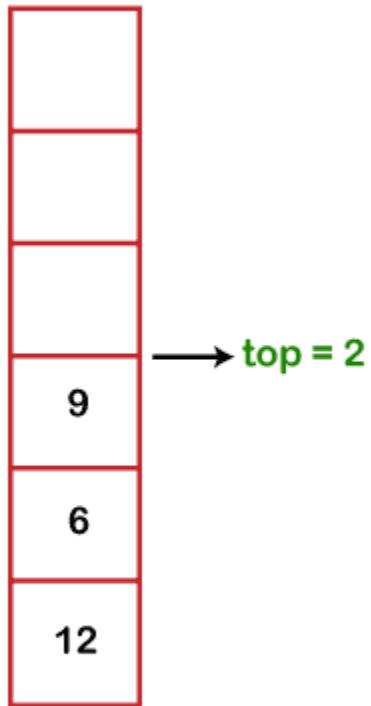
Empty Stack: If the stack has no element is known as an **empty stack**. When the stack is empty the value of the top variable is -1.

Empty Stack

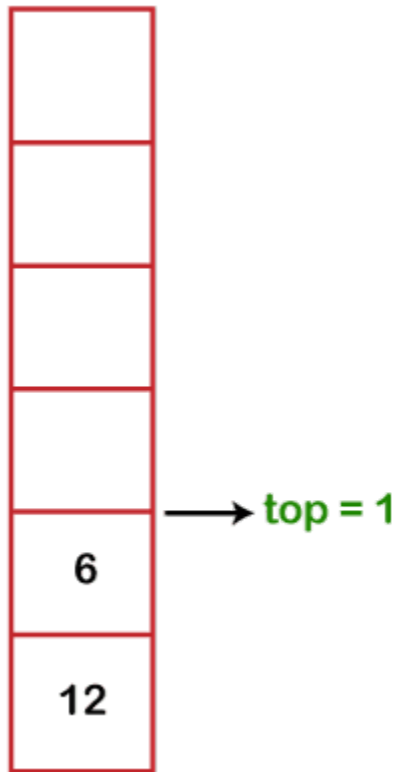


When we push an element into the stack the top is **increased by 1**. In the following figure,

- Push 12, top=0
- Push 6, top=1
- Push 9, top=2



When we pop an element from the stack the value of top is **decreased by 1**. In the following figure, we have popped 9.

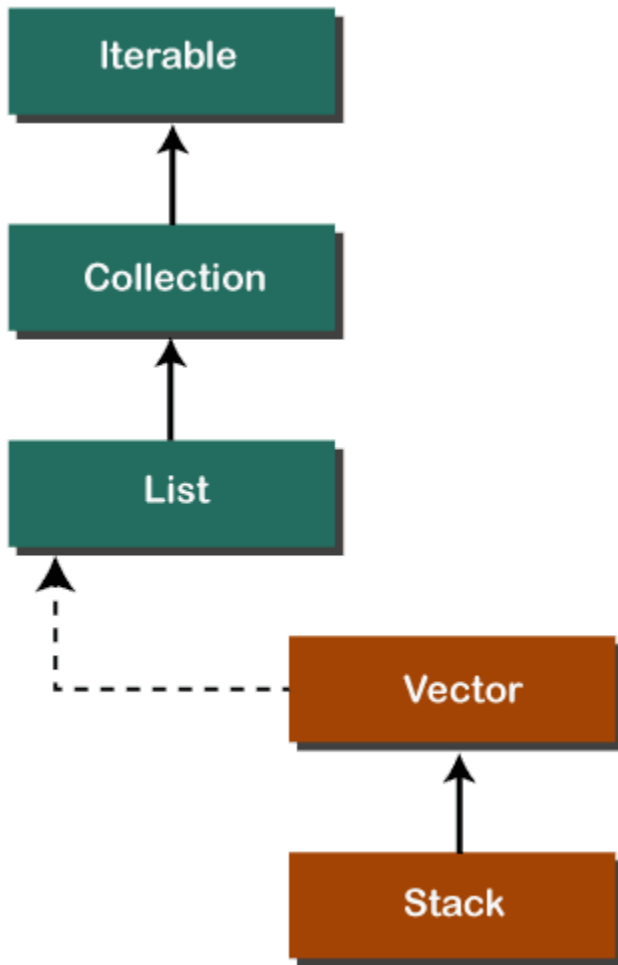


The following table shows the different values of the top.

Top value	Meaning
-1	It shows the stack is empty.
0	The stack has only an element.
N-1	The stack is full.
N	The stack is overflow.

Java Stack Class

In Java, **Stack** is a class that falls under the Collection framework that extends the **Vector** class. It also implements interfaces **List**, **Collection**, **Iterable**, **Cloneable**, **Serializable**. It represents the LIFO stack of objects. Before using the Stack class, we must import the `java.util` package. The stack class arranged in the Collections framework hierarchy, as shown below.



Stack Class Constructor

The Stack class contains only the **default constructor** that creates an empty stack.

1. `public Stack()`

Creating a Stack

If we want to create a stack, first, import the `java.util` package and create an object of the Stack class.

1. `Stack stk = new Stack();`

Or

1. `Stack<type> stk = new Stack<>();`

Where type denotes the type of stack like Integer, String, etc.

Methods of the Stack Class

We can perform push, pop, peek and search operation on the stack. The Java Stack class provides mainly five methods to perform these operations. Along with this, it also provides all the methods of the [Java Vector class](#).

Method	Modifier and Type	Method Description
empty()	boolean	The method checks the stack is empty or not.
push(E item)	E	The method pushes (insert) an element onto the top of the stack.
pop()	E	The method removes an element from the top of the stack and returns the same element as the value of that function.
peek()	E	The method looks at the top element of the stack without removing it.
search(Object o)	int	The method searches the specified object and returns the position of the object.

Stack Class empty() Method

The **empty()** method of the Stack class check the stack is empty or not. If the stack is empty, it returns true, else returns false. We can also use the [isEmpty\(\) method of the Vector class](#).

Syntax

1. **public boolean** empty()

Returns: The method returns true if the stack is empty, else returns false.

In the following example, we have created an instance of the Stack class. After that, we have invoked the empty() method two times. The first time it returns **true** because we have not pushed any element into the stack. After that, we have pushed elements into the stack. Again we have invoked the empty() method that returns **false** because the stack is not empty.

StackEmptyMethodExample.java

```
1. import java.util.Stack;
2. public class StackEmptyMethodExample
3. {
4.     public static void main(String[] args)
5.     {
6.         //creating an instance of Stack class
7.         Stack<Integer> stk= new Stack<>();
8.         // checking stack is empty or not
9.         boolean result = stk.empty();
10.        System.out.println("Is the stack empty? " + result);
11.        // pushing elements into stack
12.        stk.push(78);
13.        stk.push(113);
14.        stk.push(90);
15.        stk.push(120);
16.        //prints elements of the stack
17.        System.out.println("Elements in Stack: " + stk);
18.        result = stk.empty();
19.        System.out.println("Is the stack empty? " + result);
20.    }
21. }
```

Output:

```
Is the stack empty? true
Elements in Stack: [78, 113, 90, 120]
Is the stack empty? false
```

Stack Class push() Method

The method inserts an item onto the top of the stack. It works the same as the method `addElement(item)` `method` of the Vector class. It passes a parameter **item** to be pushed into the stack.

Syntax

1. `public E push(E item)`

Parameter: An item to be pushed onto the top of the stack.

Returns: The method returns the argument that we have passed as a parameter.

Stack Class pop() Method

The method removes an object at the top of the stack and returns the same object. It throws **EmptyStackException** if the stack is empty.

Syntax

1. `public E pop()`

Returns: It returns an object that is at the top of the stack.

Let's implement the stack in a Java program and perform push and pop operations.

StackPushPopExample.java

1. `import java.util.*;`
2. `public class StackPushPopExample`
3. `{`
4. `public static void main(String args[])`
5. `{`
6. `//creating an object of Stack class`
7. `Stack <Integer> stk = new Stack<>();`
8. `System.out.println("stack: " + stk);`
9. `//pushing elements into the stack`
10. `pushelmnt(stk, 20);`
11. `pushelmnt(stk, 13);`
12. `pushelmnt(stk, 89);`

```
13. pushelmnt(stk, 90);
14. pushelmnt(stk, 11);
15. pushelmnt(stk, 45);
16. pushelmnt(stk, 18);
17. //popping elements from the stack
18. popelmnt(stk);
19. popelmnt(stk);
20. //throws exception if the stack is empty
21. try
22. {
23. popelmnt(stk);
24. }
25. catch (EmptyStackException e)
26. {
27. System.out.println("empty stack");
28. }
29. }
30. //performing push operation
31. static void pushelmnt(Stack stk, int x)
32. {
33. //invoking push() method
34. stk.push(new Integer(x));
35. System.out.println("push -> " + x);
36. //prints modified stack
37. System.out.println("stack: " + stk);
38. }
39. //performing pop operation
40. static void popelmnt(Stack stk)
```

```
41. {  
42. System.out.print("pop -> ");  
43. //invoking pop() method  
44. Integer x = (Integer) stk.pop();  
45. System.out.println(x);  
46. //prints modified stack  
47. System.out.println("stack: " + stk);  
48. }  
49. }
```

Output:

```
stack: []  
push -> 20  
stack: [20]  
push -> 13  
stack: [20, 13]  
push -> 89  
stack: [20, 13, 89]  
push -> 90  
stack: [20, 13, 89, 90]  
push -> 11  
stack: [20, 13, 89, 90, 11]  
push -> 45  
stack: [20, 13, 89, 90, 11, 45]  
push -> 18  
stack: [20, 13, 89, 90, 11, 45, 18]  
pop -> 18  
stack: [20, 13, 89, 90, 11, 45]  
pop -> 45  
stack: [20, 13, 89, 90, 11]  
pop -> 11  
stack: [20, 13, 89, 90]
```

Stack Class peek() Method

It looks at the element that is at the top in the stack. It also throws **EmptyStackException** if the stack is empty.

Syntax

1. **public** E peek()

Returns: It returns the top elements of the stack.

Let's see an example of the peek() method.

StackPeekMethodExample.java

```
1. import java.util.Stack;
2. public class StackPeekMethodExample
3. {
4.     public static void main(String[] args)
5.     {
6.         Stack<String> stk= new Stack<>();
7.         // pushing elements into Stack
8.         stk.push("Apple");
9.         stk.push("Grapes");
10.        stk.push("Mango");
11.        stk.push("Orange");
12.        System.out.println("Stack: " + stk);
13.        // Access element from the top of the stack
14.        String fruits = stk.peek();
15.        //prints stack
16.        System.out.println("Element at top: " + fruits);
17.    }
18. }
```

Output:

Stack: [Apple, Grapes, Mango, Orange]

Element at the top of the stack: Orange

Stack Class search() Method

The method searches the object in the stack from the top. It parses a parameter that we want to search for. It returns the 1-based location of the object in the stack. The topmost object of the stack is considered at distance 1.

Suppose, o is an object in the stack that we want to search for. The method returns the distance from the top of the stack of the occurrence nearest the top of the stack. It uses **equals()** method to search an object in the stack.

Syntax

1. **public int** search(Object o)

Parameter: o is the desired object to be searched.

Returns: It returns the object location from the top of the stack. If it returns -1, it means that the object is not on the stack.

Let's see an example of the search() method.

StackSearchMethodExample.java

1. **import** java.util.Stack;
2. **public class** StackSearchMethodExample
3. {
4. **public static void** main(String[] args)
5. {
6. Stack<String> stk= **new** Stack<>();
7. *//pushing elements into Stack*
8. stk.push("Mac Book");
9. stk.push("HP");
10. stk.push("DELL");
11. stk.push("Asus");
12. System.out.println("Stack: " + stk);
13. *// Search an element*
14. **int** location = stk.search("HP");
15. System.out.println("Location of Dell: " + location);

16. }

17. }

Java Stack Operations

Size of the Stack

We can also find the size of the stack using the `size()` method of the `Vector` class. It returns the total number of elements (size of the stack) in the stack.

Syntax

1. **public int** size()

Let's see an example of the `size()` method of the `Vector` class.

StackSizeExample.java

```
1. import java.util.Stack;
2. public class StackSizeExample
3. {
4.     public static void main (String[] args)
5.     {
6.         Stack stk = new Stack();
7.         stk.push(22);
8.         stk.push(33);
9.         stk.push(44);
10.        stk.push(55);
11.        stk.push(66);
12.        // Checks the Stack is empty or not
13.        boolean rslt=stk.empty();
14.        System.out.println("Is the stack empty or not? " +rslt);
15.        // Find the size of the Stack
16.        int x=stk.size();
```



```
17. System.out.println("The stack size is: "+x);  
18. }  
19. }
```

Output:

```
Is the stack empty or not? false  
The stack size is: 5
```

Iterate Elements

Iterate means to fetch the elements of the stack. We can fetch elements of the stack using three different methods are as follows:

- Using **iterator()** Method
- Using **forEach()** Method
- Using **listIterator()** Method

Using the iterator() Method

It is the method of the Iterator interface. It returns an iterator over the elements in the stack. Before using the iterator() method import the `java.util.Iterator` package.

Syntax

```
1. Iterator<T> iterator()
```

Let's perform an iteration over the stack.

StackIterationExample1.java

```
1. import java.util.Iterator;  
2. import java.util.Stack;  
3. public class StackIterationExample1  
4. {  
5. public static void main (String[] args)
```

```
6. {  
7. //creating an object of Stack class  
8. Stack stk = new Stack();  
9. //pushing elements into stack  
10. stk.push("BMW");  
11. stk.push("Audi");  
12. stk.push("Ferrari");  
13. stk.push("Bugatti");  
14. stk.push("Jaguar");  
15. //iteration over the stack  
16. Iterator iterator = stk.iterator();  
17. while(iterator.hasNext())  
18. {  
19. Object values = iterator.next();  
20. System.out.println(values);  
21. }  
22. }  
23. }
```

Output:

```
BMW  
Audi  
Ferrari  
Bugatti  
Jaguar
```

Using the forEach() Method

Java provides a `forEach()` method to iterate over the elements. The method is defined in the **Iterable** and **Stream** interface.

Syntax

1. **default void** forEach(Consumer<**super** T>action)

Let's iterate over the stack using the forEach() method.

StackIterationExample2.java

1. **import** java.util.*;
2. **public class** StackIterationExample2
3. {
4. **public static void** main (String[] args)
5. {
6. *//creating an instance of Stack class*
7. Stack <Integer> stk = **new** Stack<>();
8. *//pushing elements into stack*
9. stk.push(**119**);
10. stk.push(**203**);
11. stk.push(**988**);
12. System.out.println("Iteration over the stack using forEach() Method:");
13. *//invoking forEach() method for iteration over the stack*
14. stk.forEach(n ->
15. {
16. System.out.println(n);
17. });
18. }
19. }

Output:

Iteration over the stack using forEach() Method:

119

203

988

Using listIterator() Method

This method returns a list iterator over the elements in the mentioned list (in sequence), starting at the specified position in the list. It iterates the stack from top to bottom.

Syntax

1. ListIterator listIterator(**int** index)

Parameter: The method parses a parameter named **index**.

Returns: This method returns a list iterator over the elements, in sequence.

Exception: It throws **IndexOutOfBoundsException** if the index is out of range.

Let's iterate over the stack using the listIterator() method.

StackIterationExample3.java

1. **import** java.util.Iterator;
2. **import** java.util.ListIterator;
3. **import** java.util.Stack;
- 4.
5. **public class** StackIterationExample3
6. {
7. **public static void** main (String[] args)
8. {
9. Stack <Integer> stk = **new** Stack<>();
10. stk.push(**119**);
11. stk.push(**203**);
12. stk.push(**988**);
13. ListIterator<Integer> ListIterator = stk.listIterator(stk.size());
14. System.out.println("Iteration over the Stack from top to bottom:");
15. **while** (ListIterator.hasPrevious())
16. {
17. Integer avg = ListIterator.previous();

```
18. System.out.println(avg);
```

```
19. }
```

```
20. }
```

```
21. }
```

Output:

Iteration over the Stack from top to bottom:

988

203

119