Course Overview

Module	Topic	Technology/Framework Learn
1	Introduction	Core Java
2	Web app Development	Servlet,JSP,MVC
3	Java Persistence using JPA and Hibernate	Hibernate,ORM,HQL,HPQL,JPA
4	Spring core	Spring,Spring Boot,Spring REST,Spring MVC,Spring AOP
5	Automation Tools	Junit,Maven,Selinium

Module 1

Introduction

Module1 - Content

- Review of Java
- Advanced concepts of Java
- Java generics
- Java IO
- New Features of Java
- Unit Testing

Module 1 Overview

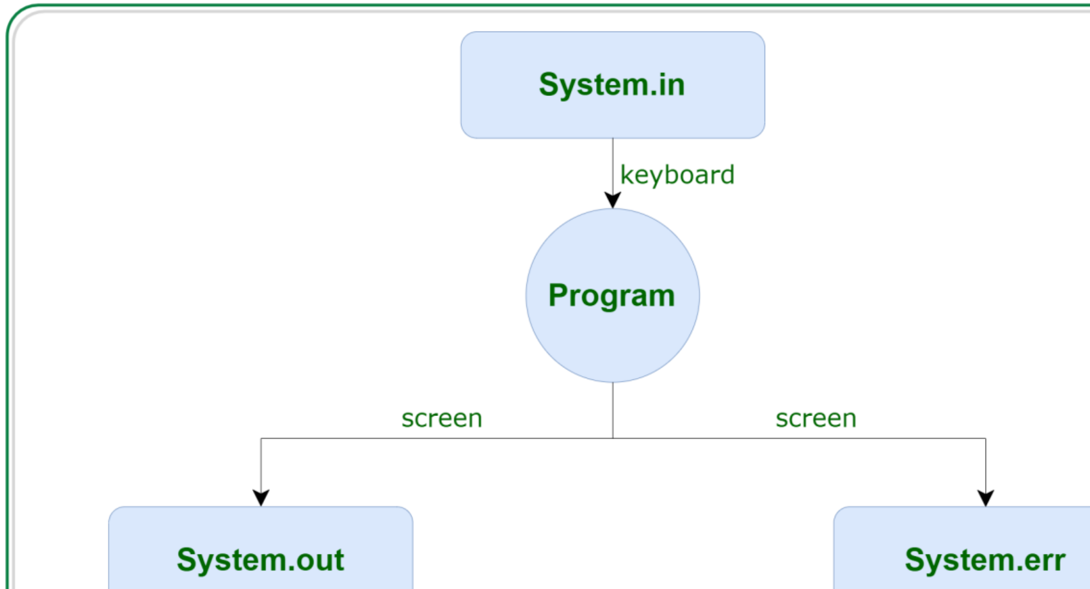
Sdesl. No	Topic	Sub Topics
1	Review of Java	File
2	Java IO	Serialization and Deserialization
3	Advanced concepts of Java	Collection Framework
4	Java generics	Generic Method and Generic class
5	New Features of Java	Annotation, Lambda Expression
6	JDBC	Drivers

Java I/O

- **Java I/O** (Input and Output) is used *to process the input* and *produce the output*.
- Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.
- Perform **file handling in Java** by Java I/O API.

Stream

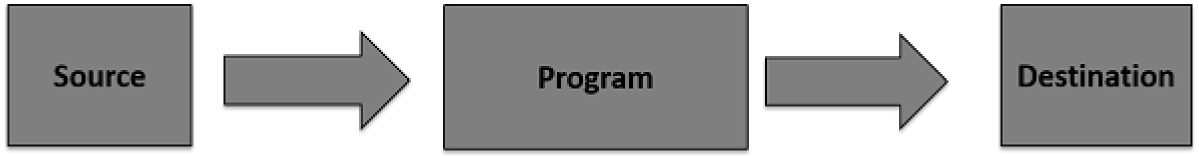
- stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow
- In Java, 3 streams are created for us automatically. All these streams are attached with the console.
- **1) System.out:** standard output stream
- **2) System.in:** standard input stream
- **3) System.err:** standard error stream
- the code to print **output and an error** message to the console.



PRESIDENCY UNIVERSITY
GAIN MORE KNOWLEDGE REACH GREATER HEIGHTS
Private University Estd. in Karnataka State by Act No. 41 of 2013
PRESIDENCY GROUP OVER 40 YEARS OF ACADEMIC WISDOM

Stream

- A stream can be defined as a sequence of data. There are two kinds of Streams −
- **InPutStream** − The InputStream is used to read data from a source.
- **OutPutStream** − The OutputStream is used for writing data to a destination.



- **Byte Streams**
- Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**.
- **Character Streams**
- Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time

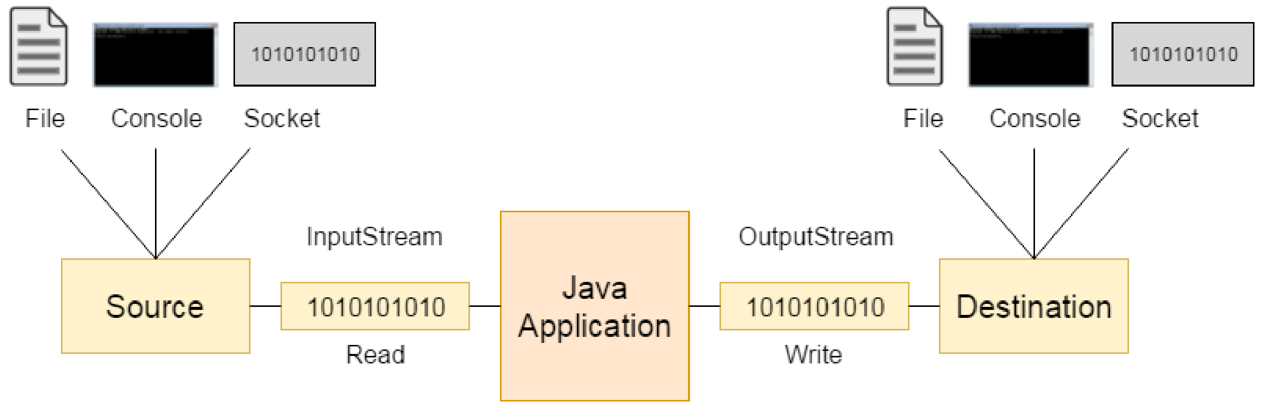
OutputStream vs InputStream

OutputStream

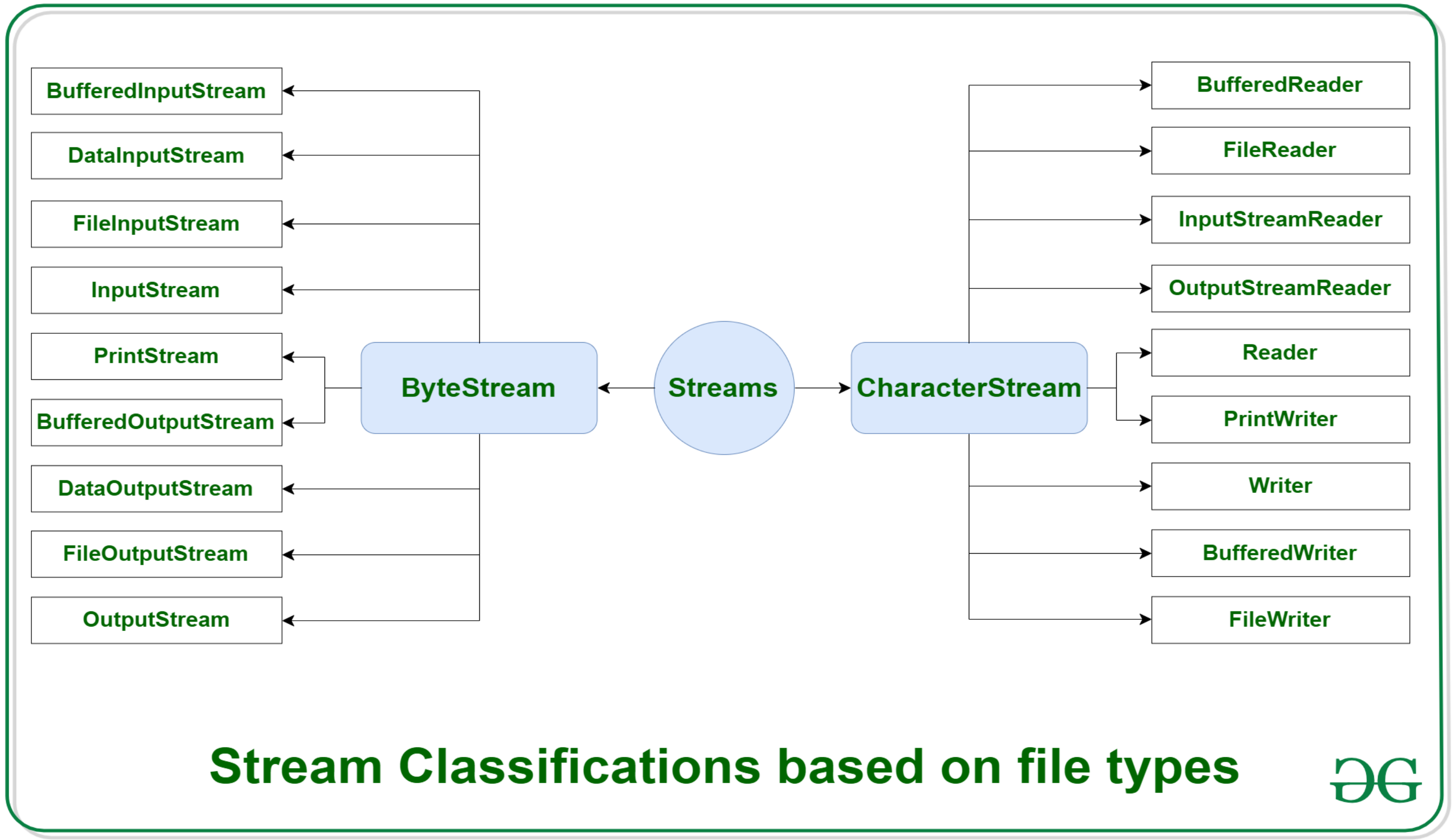
- Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

InputStream

- Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.



Byte Stream and Character Stream

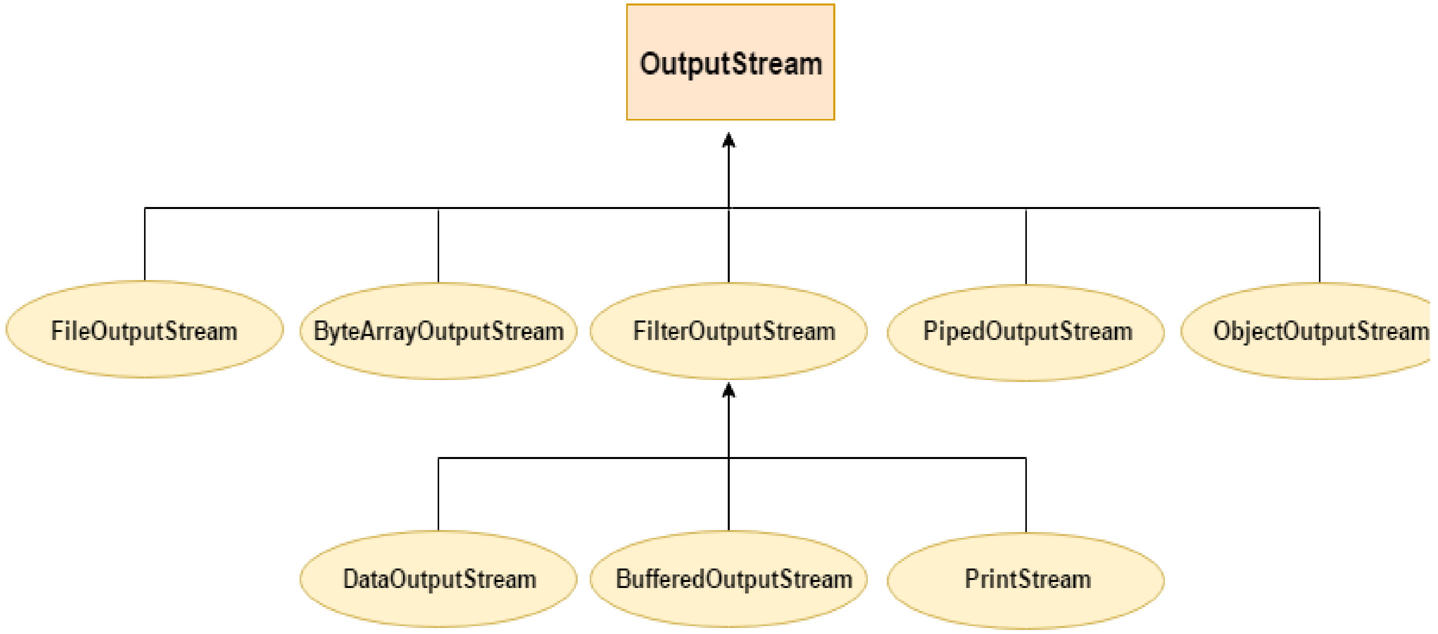


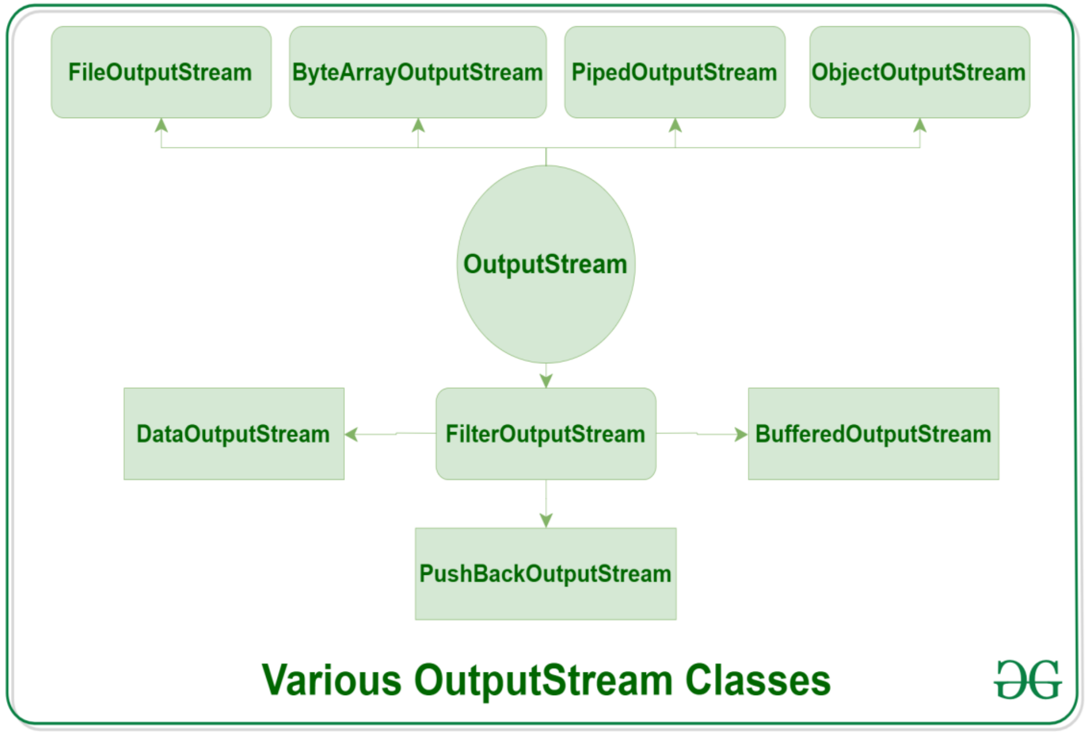
OutputStream class

- OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Method	Description
1) public void write(int)throws IOException	is used to write a byte to the current output stream.
2)public void write(byte[])throws IOException	is used to write an array of byte to the current output stream.
3) public void flush()throws IOException	flushes the current output stream.
4) public void close()throws IOException	is used to close the current output stream.

OutputStream Hierarchy





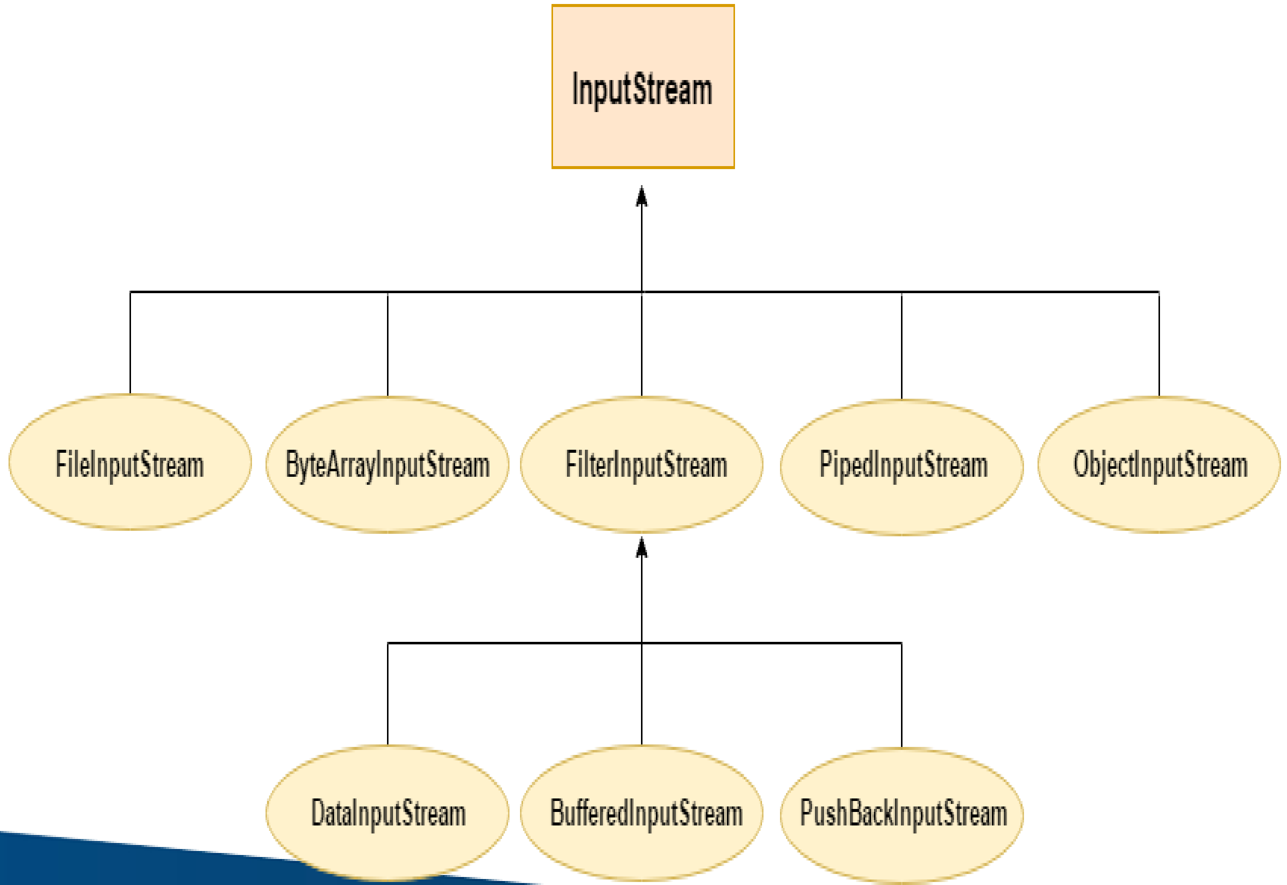
PRESIDENCY UNIVERSITY
Private University Estd. in Karnataka State by Act No. 41 of 2013

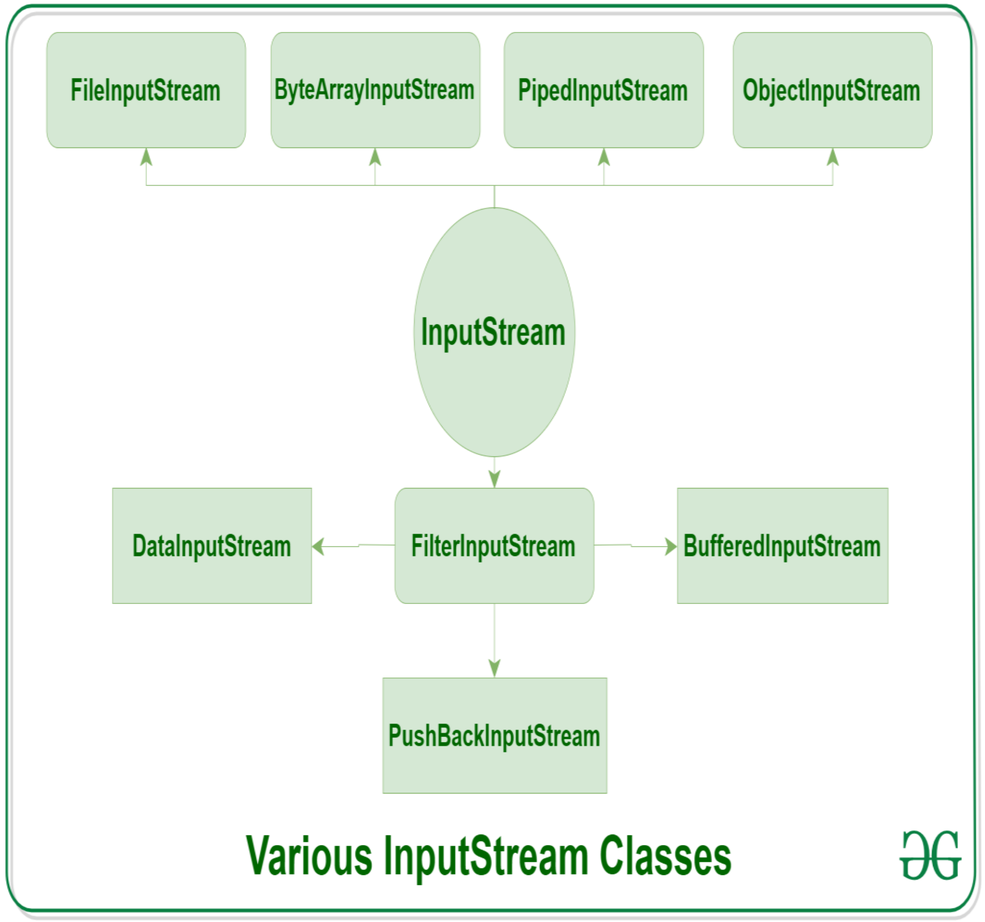
InputStream class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

Method	**Description**
1) public abstract int read()throws IOException	reads the next byte of data from the input stream. It returns -1 at the end of the file.
2) public int available()throws IOException	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException	is used to close the current input stream.

InputStream Hierarchy





FileOutputStream

- Java FileOutputStream is an output stream used for writing data to a file.
- If you have to write primitive values into a file, use FileOutputStream class. You can write byte-oriented as well as character-oriented data through FileOutputStream class.

Methods in FileOutputStream

Method	**Description**
protected void finalize()	It is used to clean up the connection with the file output stream.
void write(byte[] ary)	It is used to write **ary.length** bytes from the byte array to the file output stream.
void write(byte[] ary, int off, int len)	It is used to write **len** bytes from the byte array starting at offset **off** to the file output stream.
void write(int b)	It is used to write the specified byte to the file output stream.
FileChannel getChannel()	It is used to return the file channel object associated with the file output stream.
FileDescriptor getFD()	It is used to return the file descriptor associated with the stream.
void close()	It is used to closes the file output stream.

FileInputStream Class

Java FileInputStream class obtains input bytes from a file. It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc. You can also read character-stream data.

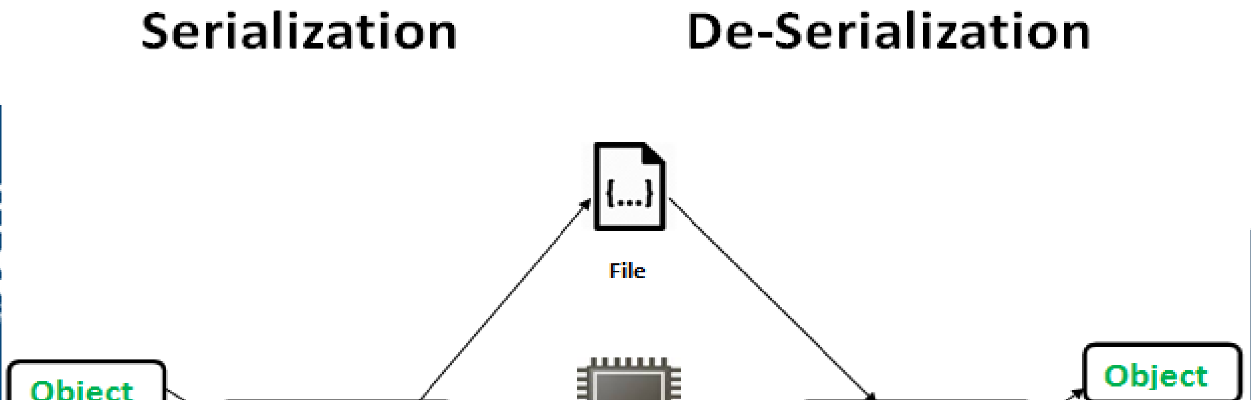
Method	Description
int available()	It is used to return the estimated number of bytes that can be read from the input stream.
int read()	It is used to read the byte of data from the input stream.
int read(byte[] b)	It is used to read up to **b.length** bytes of data from the input stream.
int read(byte[] b, int off, int len)	It is used to read up to **len** bytes of data from the input stream.
long skip(long x)	It is used to skip over and discards x bytes of data from the input stream.
FileChannel getChannel()	It is used to return the unique FileChannel object associated with the file input stream.
FileDescriptor getFD()	It is used to return the FileDescriptor object.

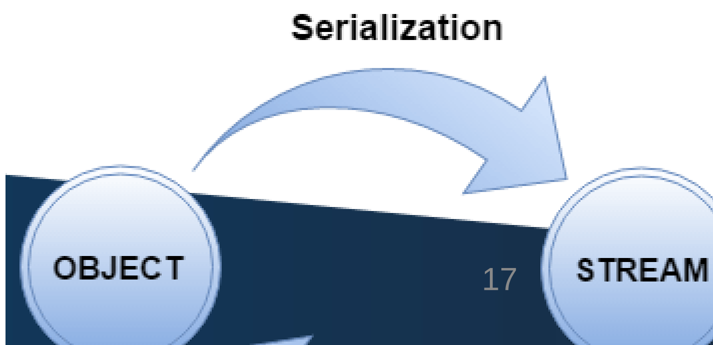
Example

- Refer EXP1

Serialization and Deserialization

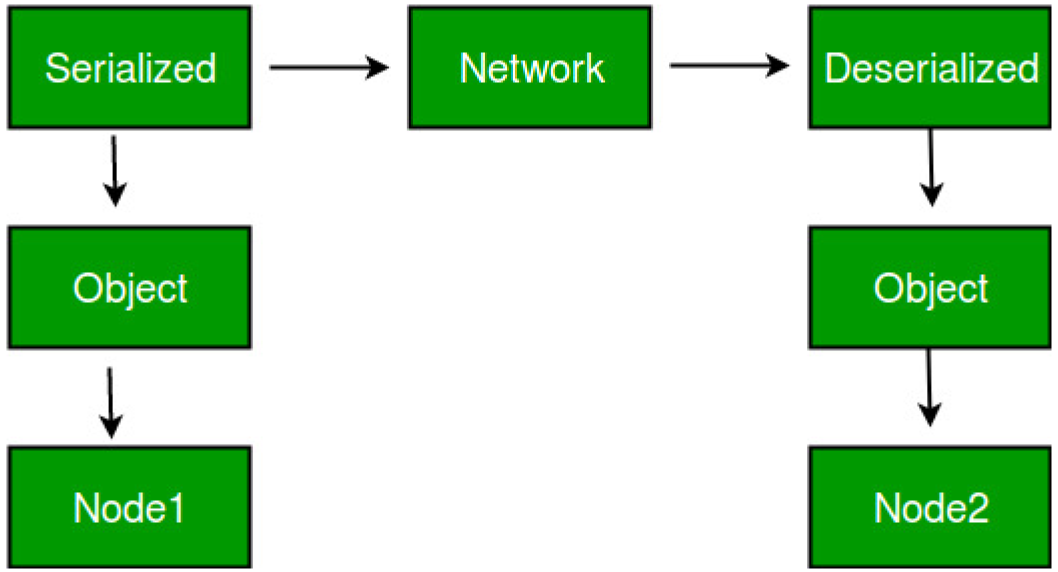
- **Serialization in Java** is a mechanism of *writing the state of an object into a byte-stream*. It is mainly used in Hibernate, RMI, JPA, EJB and JMS technologies.
- The reverse operation of serialization is called ***deserialization*** where byte-stream is converted into an object. The serialization and deserialization process is platform-independent, it means you can serialize an object on one platform and deserialize it on a different platform.
-





Advantages of Java Serialization

- It is mainly used to travel object's state on the network (that is known as marshalling).
- To save/persist state of an object.
- To travel an object across a network.



Methods used for Serialization and Deserialization

- For serializing the object, call the **writeObject()** method of *ObjectOutputStream* class
- for deserialization call the **readObject()** method of *ObjectInputStream* class.
- To implement the *Serializable* interface for serializing the object

Example – Serialization & Deserialization

Refer Exp2

Collection Framework

- provides an architecture to store and manipulate the group of objects.
- 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).

Terms used

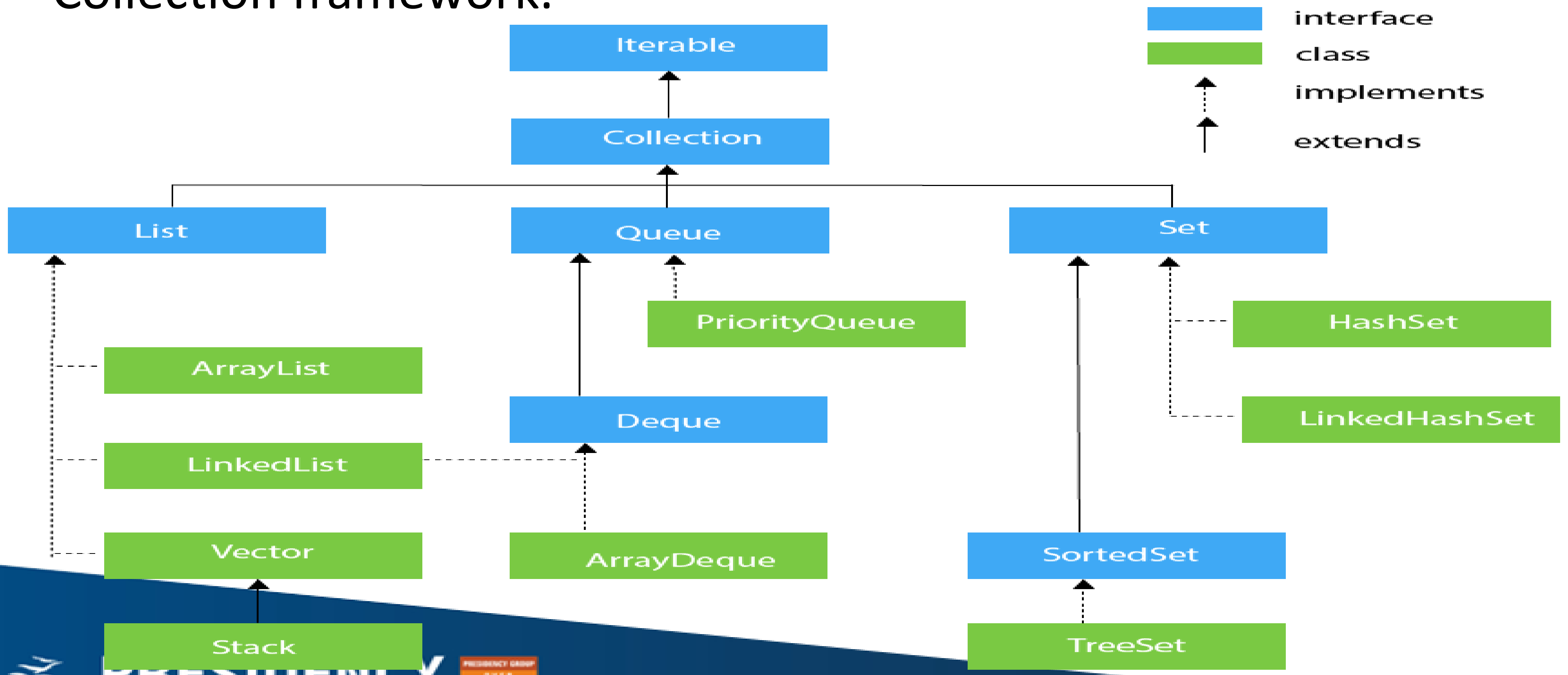
- Collection represents a single unit of objects, i.e., a group.
- framework - provides readymade architecture.

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

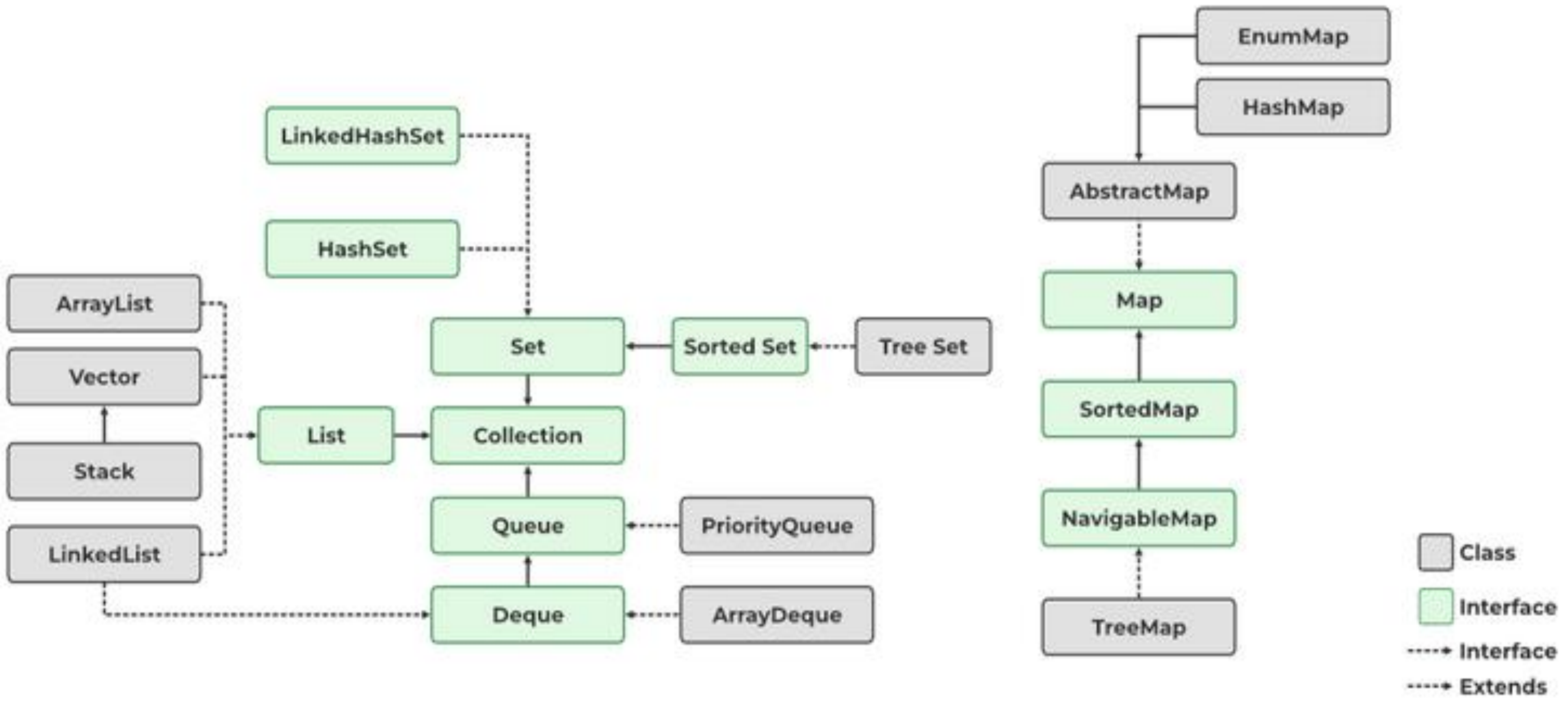
1. Interfaces and its implementations, i.e., classes
2. Algorithm

Hierarchy of Collection Framework

- The **java.util** package contains all the classes and interfaces for the Collection framework.



List,Set,Map


EnumMap
HashMap
AbstractMap
Map
SortedMap
NavigableMap
TreeMap
LinkedHashSet
HashSet
ArrayList
Vector
Stack
LinkedList
List
Set
Sorted Set
Tree Set
Collection
Queue
PriorityQueue
Deque
ArrayDeque
Class
Interface
Interface
Extends

Iterable Interface

- The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.
- It contains only one abstract method. i.e.,
- Iterator<T> iterator()
- It returns the iterator over the elements of type T.

Collection Interface

- The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.
- Some of the methods of Collection interface are Boolean add (Object obj), Boolean addAll (Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

Methods of Collection interface

No.	Method	Description
1	public boolean add(E e)	It is used to insert an element in this collection.
2	public boolean addAll(Collection<? extends E> c)	It is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	It is used to delete an element from the collection.
4	public boolean removeAll(Collection<?> c)	It is used to delete all the elements of the specified collection from the invoking collection.
5	default boolean removeIf(Predicate<? super E> filter)	It is used to delete all the elements of the collection that satisfy the specified predicate.
6	public boolean retainAll(Collection<?> c)	It is used to delete all the elements of invoking collection except the specified collection.
7	public int size()	It returns the total number of elements in

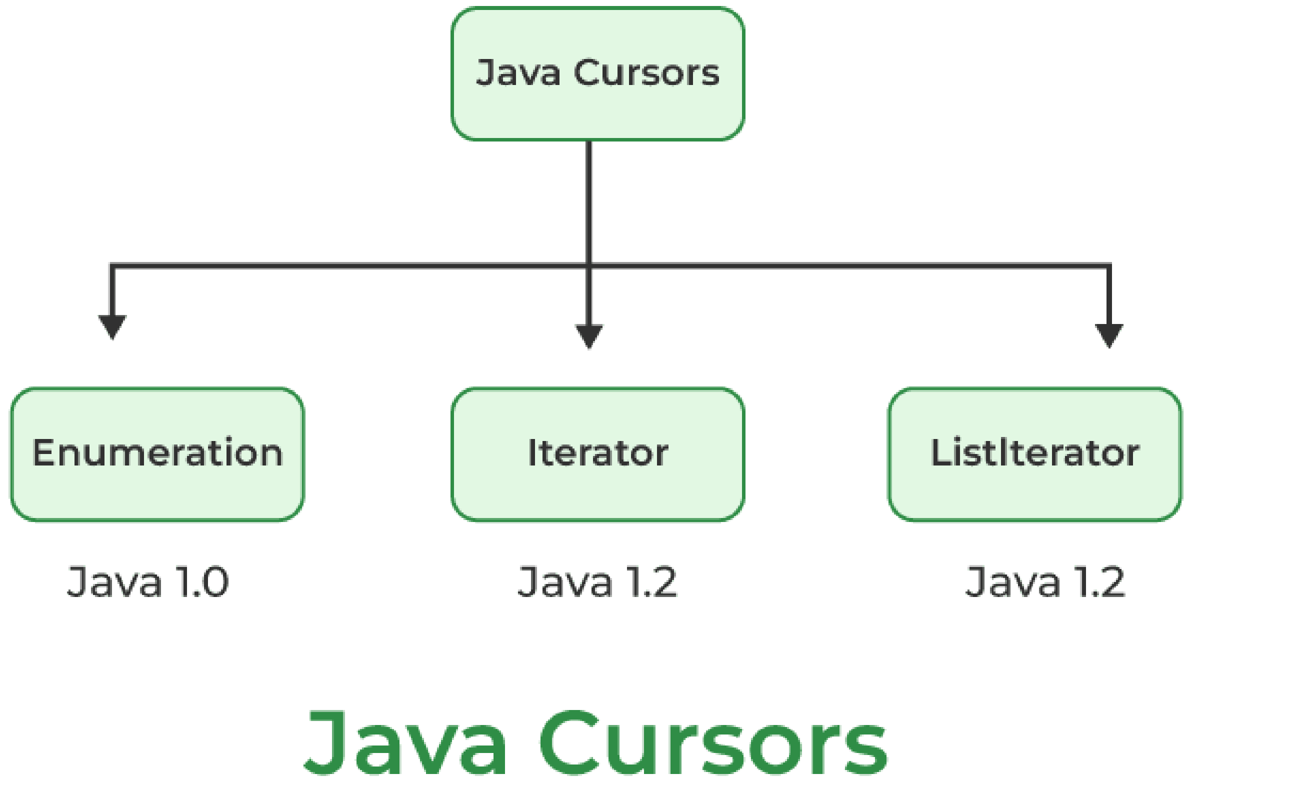
Iterator interface

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

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

Java Cursor

used to iterate or traverse or retrieve a Collection or
Stream object’s elements one by one



Access collection by

- Using **iterator()** Method
- Using **forEach()** Method
- Using **listIterator()** Method

Enumerator

- It is an interface used to get elements of legacy collections(Vector, Hashtable). Enumeration is the first iterator present from JDK 1.0, rests are included in JDK 1.2 with more functionality.
- There are **two** methods in the Enumeration interface namely :
- **1. public boolean hasMoreElements():** This method tests if this enumeration contains more elements or not.
- **2. public Object nextElement():** This method returns the next element of this enumeration. It throws NoSuchElementException if no more element is present

Example

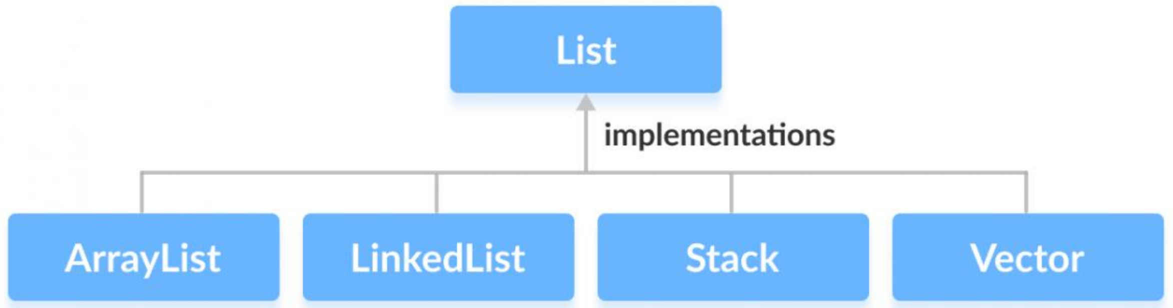
```
public class Test {

  public static void main(String[] args)
  {
      Vector v = new Vector();
            for (int i = 0; i < 10; i++)
                v.addElement(i);

      // Printing elements in vector object
      System.out.println(v);
       Enumeration e = v.elements();
      while (e.hasMoreElements()) {
              int i = (Integer)e.nextElement();
         System.out.print(i + " ");
      }
  }
}
```

Distinguish between Iterator and Enumeration.

Sr. No.	Key	Iterator	Enumeration
1	Basic	In Iterator, we can read and remove element while traversing element in the collections.	Using Enumeration, we can only read element during traversing element in the collections.
2.	Access	It can be used with any class of the collection framework.	It can be used only with legacy class of the collection framework such as a Vector and HashTable.
3.	Fail-Fast and Fail -Safe	Any changes in the collection, such as removing element from the collection during a thread is iterating collection then it throw concurrent modification exception.	Enumeration is Fail safe in nature. It doesn’t throw concurrent modification exception
		Only forward direction iterating is	Remove operations can not be performed

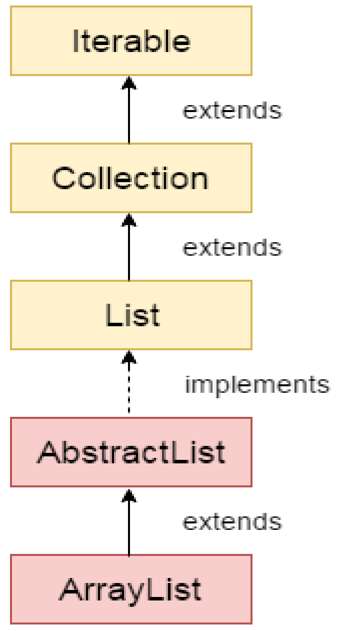
List Interface

- List interface is the child interface of Collection interface.
- store the ordered collection of objects.
- It can have duplicate values.
- can be used to insert, delete, and access the elements from the list.
- List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.
- To instantiate the List interface, we must use :
- List <data-type> list1= **new** ArrayList();
- List <data-type> list2 = **new** LinkedList();
- List <data-type> list3 = **new** Vector();
- List <data-type> list4 = **new** Stack();



ArrayList

- The ArrayList class implements the List interface.
- It uses a dynamic array to store the duplicate element of different data types.
- Java **ArrayList** class uses a *dynamic* *array* for storing the elements. It is like an array, but there is *no size limit* .The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed.
- . We can add or remove elements anytime. So, it is much more flexible than the traditional array



Example1

```
import java.util.*;
public class ArrayListExample2{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();
  list.add("Mango");//Adding object in arraylist
  list.add("Apple");
  list.add("Banana");
  list.add("Grapes");
  //Traversing list through Iterator
  Iterator itr=list.iterator();//getting the Iterator
  while(itr.hasNext()){//check if iterator has the elements
   System.out.println(itr.next());//printing the element and move to next
  }
 }
}
```

Example2

```
import java.util.*;
class SortArrayList{
 public static void main(String args[]){
  //Creating a list of fruits
  List<String> list1=new ArrayList<String>();
  list1.add("Mango");
  list1.add("Apple");
  list1.add("Banana");
  list1.add("Grapes");
  //Sorting the list
  Collections.sort(list1);
   //Traversing list through the for-each loop
  for(String fruit:list1)
    System.out.println(fruit);

 System.out.println("Sorting numbers...");
  //Creating a list of numbers
  List<Integer> list2=new ArrayList<Integer>();
  list2.add(21);
  list2.add(11);
  list2.add(51);
  list2.add(1);
  //Sorting the list
```

User-defined class objects in Java ArrayList

```
class Student{
  int rollno;
  String name;
  int age;
  Student(int rollno,String name,int age){
   this.rollno=rollno;
   this.name=name;
   this.age=age;
  }
}
import java.util.*;
 class ArrayList5{
 public static void main(String args[]){
  //Creating user-defined class objects
  Student s1=new Student(101,"Sonoo",23);
  Student s2=new Student(102,"Ravi",21);
  Student s2=new Student(103,"Hanumat",25);
  //creating arraylist
  ArrayList<Student> al=new ArrayList<Student>();
  al.add(s1);//adding Student class object
  al.add(s2);
  al.add(s3);
  //Getting Iterator
```

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 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.

Example

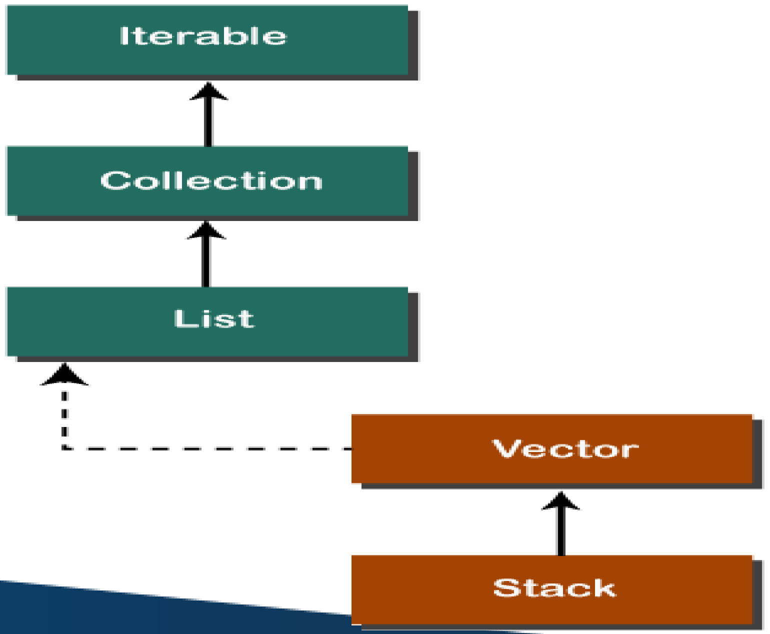
```
import java.util.*;
public class VectorExample1 {
      public static void main(String args[]) {
        //Create an empty vector with initial capacity 4
        Vector<String> vec = new Vector<String>(4);
        //Adding elements to a vector
        vec.add("Tiger");
        vec.add("Lion");
        vec.add("Dog");
        vec.add("Elephant");
        //Check size and capacity
        System.out.println("Size is: "+vec.size());
        System.out.println("Default capacity is: "+vec.capacity());
        //Display Vector elements
        System.out.println("Vector element is: "+vec);
        vec.addElement("Rat");
        vec.addElement("Cat");
        vec.addElement("Deer");
        //Again check size and capacity after two insertions
        System.out.println("Size after addition: "+vec.size());
```

Difference between ArrayList and Vector

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

Stack

- 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.

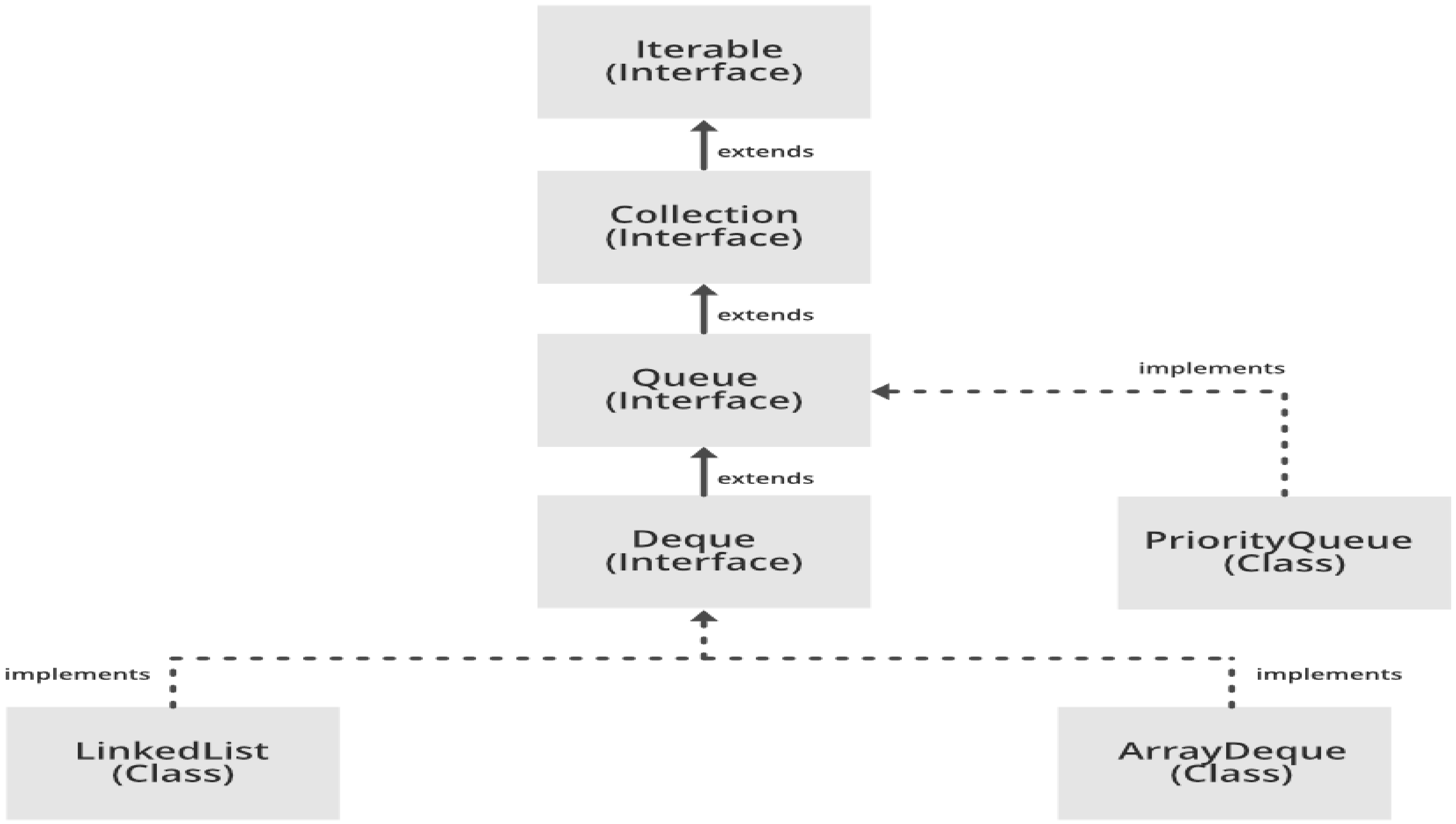


Example

```
import java.util.Stack;
class Main { public static void main(String[] args) {
        Stack<String> animals= new Stack<>();
        animals.push("Dog");
         animals.push("Horse");
       animals.push("Cat");
        System.out.println("Initial Stack: " + animals);
      String element = animals.pop();
System.out.println("Removed Element: " + element);
 }
 }
```

Queue

- is used to hold the elements about to be processed in FIFO(First In First Out) order. It is an [illegible] and deleting rinciple.

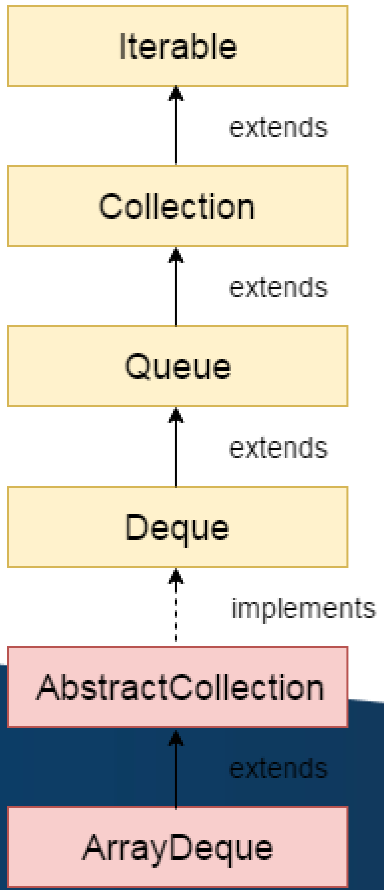


Example

```
import java.util.LinkedList;
import java.util.Queue;

public class QueueExample {
  public static void main(String[] args) {
    Queue<String> queue = new LinkedList<>();

    // add elements to the queue
    queue.add("apple");
    queue.add("banana");
    queue.add("cherry");

    // print the queue
    System.out.println("Queue: " + queue);

    // remove the element at the front of the queue
    String front = queue.remove();
    System.out.println("Removed element: " + front);

    // print the updated queue
```

Java Deque Interface

- The interface called Deque is present in java.util package. It is the subtype of the interface queue. The Deque supports the addition as well as the removal of elements from both ends of the data structure. Therefore, a deque can be used as a stack or a queue. We know that the stack supports the Last In First Out (LIFO) operation, and the operation First In First Out is supported by a queue. As a deque supports both, either of the mentioned operations can be performed on it. Deque is an acronym for **"double ended queue".**

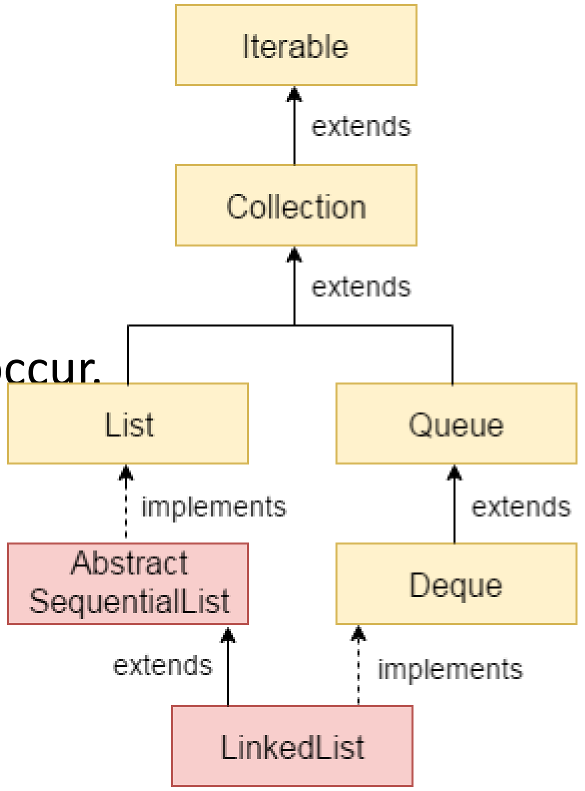


Example

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

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.



PRESIDENCY UNIVERSITY
Private University Estd. in Karnataka State by Act No. 41 of 2013

Example

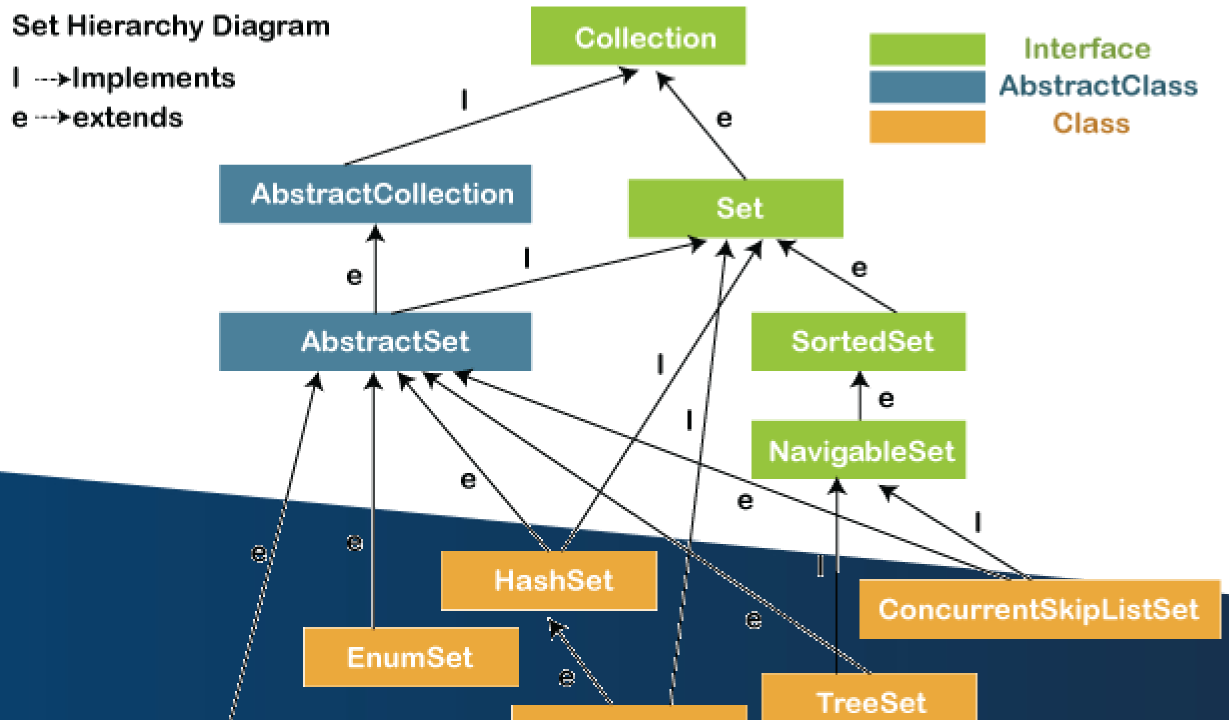
```
public class LinkedList2{
public static void main(String args[]){
LinkedList<String> ll=new LinkedList<String>();
    System.out.println("Initial list of elements: "+ll);
    ll.add("Ravi");
    ll.add("Vijay");
    ll.add("Ajay");
    System.out.println("After invoking add(E e) method: "+ll);
    //Adding an element at the specific position
    ll.add(1, "Gaurav");
    System.out.println("After invoking add(int index, E element) method: "+ll);
    LinkedList<String> ll2=new LinkedList<String>();
    ll2.add("Sonoo");
    ll2.add("Hanumat");
    //Adding second list elements to the first list
    ll.addAll(ll2);
    System.out.println("After invoking addAll(Collection<? extends E> c) method: "+ll);
    LinkedList<String> ll3=new LinkedList<String>();
    ll3.add("John");
    ll3.add("Rahul");
    //Adding second list elements to the first list at specific position
    ll.addAll(1, ll3);
    System.out.println("After invoking addAll(int index, Collection<? extends E> c) method: "+ll);
    //Adding an element at the first position
    ll.addFirst("Lokesh");
    System.out.println("After invoking addFirst(E e) method: "+ll);
```

Example

```
import java.util.*;
public class LinkedList4{
 public static void main(String args[]){

  LinkedList<String> ll=new LinkedList<String>();
           ll.add("Ravi");
           ll.add("Vijay");
           ll.add("Ajay");
           //Traversing the list of elements in reverse order
           Iterator i=ll.descendingIterator();
           while(i.hasNext())
           {
               System.out.println(i.next());
           }

 }
}
```

Set

- The **set** is an interface available in the **java.util** package. The **set** interface extends the Collection interface. An unordered collection or list in which duplicates are not allowed is referred to as a **collection interface**.
- The set interface is used to create the mathematical set. The set interface use collection interface's methods to avoid the insertion of the same elements.
- **SortedSet** and **NavigableSet** are two interfaces that extend the set implementation.

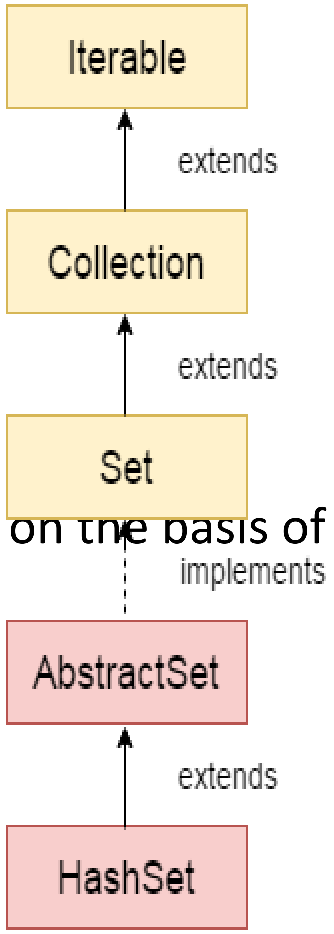


Difference between List and Set

```
public static void main(String args[])
{
    Integer[] A = {22, 45,33, 66, 55, 34, 77};
    Integer[] B = {33, 2, 83, 45, 3, 12, 55};
    Set<Integer> set1 = new HashSet<Integer>();
    set1.addAll(Arrays.asList(A));
    Set<Integer> set2 = new HashSet<Integer>();
    set2.addAll(Arrays.asList(B));

    // Finding Union of set1 and set2
    Set<Integer>  set3= new HashSet<Integer>(set1);
    set3.addAll(set2);
    System.out.print("Union of set1 and set2 is:");
    System.out.println(set3);


    // Finding Intersection of set1 and set2
    Set<Integer> set4 = new HashSet<Integer>(set1);
    set4.retainAll(set2);
```

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.



Java HashSet example to remove elements

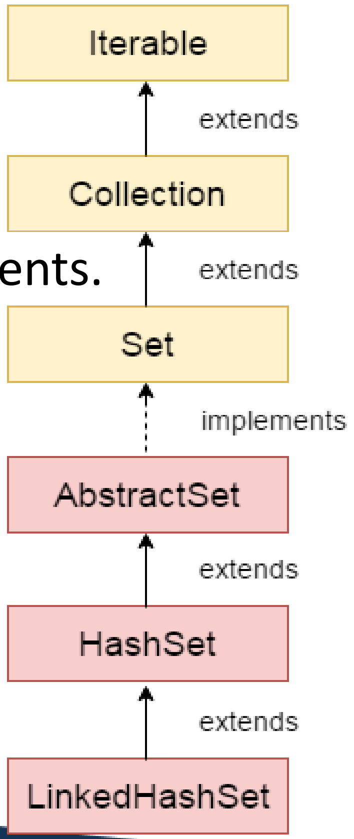
```
import java.util.*;
class HashSet3{
 public static void main(String args[]){
  HashSet<String> set=new HashSet<String>();
  set.add("Ravi");
  set.add("Vijay");
   set.add("Arun");
  set.add("Sumit");
  System.out.println("An initial list of elements: "+set);
   //Removing specific element from HashSet
           set.remove("Ravi");
           System.out.println("After invoking remove(object) method: "+set);
           HashSet<String> set1=new HashSet<String>();
           set1.add("Ajay");
           set1.add("Gaurav");
           set.addAll(set1);
           System.out.println("Updated List: "+set);
           //Removing all the new elements from HashSet
           set.removeAll(set1);
           System.out.println("After invoking removeAll() method: "+set);
```

Java HashSet from another Collection

```
import java.util.*;
class HashSet4{
 public static void main(String args[]){
   ArrayList<String> list=new ArrayList<String>();
          list.add("Ravi");
          list.add("Vijay");
          list.add("Ajay");

          HashSet<String> set=new HashSet(list);
          set.add("Gaurav");
          Iterator<String> i=set.iterator();
          while(i.hasNext())
          {
          System.out.println(i.next());
          }
 }
}
```

LinkedHashSet

- Java LinkedHashSet class is a Hashtable and Linked list implementation of the Set interface. It inherits the HashSet class and implements the Set interface.
- The important points about the Java LinkedHashSet class are:
- Java LinkedHashSet class contains unique elements only like HashSet.
- Java LinkedHashSet class provides all optional set operations and permits null elements.
- Java LinkedHashSet class is non-synchronized.
- Java LinkedHashSet class maintains insertion order.



Example

```
import java.util.*;
class LinkedHashSet1{
 public static void main(String args[]){
 //Creating HashSet and adding elements
       LinkedHashSet<String> set=new LinkedHashSet();
              set.add("One");
              set.add("Two");
              set.add("Three");
              set.add("Four");
              set.add("Five");
              Iterator<String> i=set.iterator();
              while(i.hasNext())
              {
              System.out.println(i.next());
              }
 }
}
```

Example LinkedHashSet

```
import java.util.*;
class LinkedHashSet2{
 public static void main(String args[]){
  LinkedHashSet<String> al=new LinkedHashSet<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ravi");
  al.add("Ajay");
  Iterator<String> itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
```

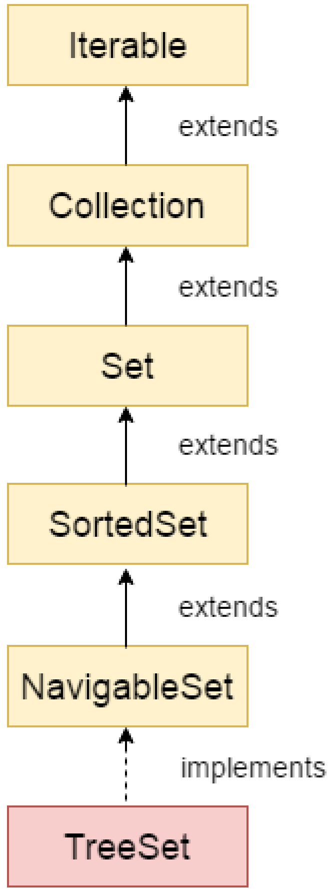
Example3

```
import  java.util.*;

public class LinkedHashSet3
{

// main method
public static void main(String argvs[])
{

// Creating an empty LinekdhashSet of string type
LinkedHashSet<String> lhs = new LinkedHashSet<String>();

// Adding elements to the above Set
// by invoking the add() method
lhs.add("Java");
lhs.add("T");
lhs.add("Point");
lhs.add("Good");
lhs.add("Website");
```

```
import java.util.*;
class Book {
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publisher, int quantity) {
    this.id = id;
    this.name = name;
    this.author = author;
    this.publisher = publisher;
    this.quantity = quantity;
}
}
public class LinkedHashSetExample {
public static void main(String[] args) {
    LinkedHashSet<Book> hs=new LinkedHashSet<Book>();
    //Creating Books
    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
```

TreeSet

- Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order.
- The important points about the Java TreeSet class are:
- Java TreeSet class contains unique elements only like HashSet.
- Java TreeSet class access and retrieval times are quiet fast.
- Java TreeSet class doesn't allow null element.
- Java TreeSet class is non synchronized.
- Java TreeSet class maintains ascending order.
- Java TreeSet class contains unique elements only like HashSet.
- Java TreeSet class access and retrieval times are quite fast.
- Java TreeSet class doesn't allow null elements.
- Java TreeSet class is non-synchronized.
- Java TreeSet class maintains ascending order.
- The TreeSet can only allow those generic types that are comparable. For example The Comparable interface is being implemented by the StringBuffer class.
- Internal Working of The TreeSet Class



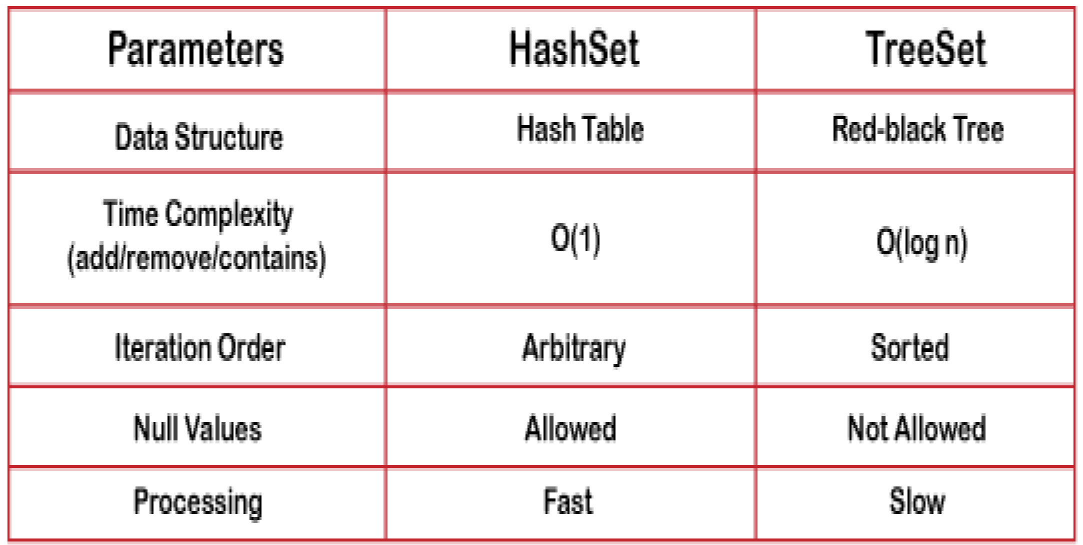
Example

```
import java.util.*;
class TreeSet1{
 public static void main(String args[]){
  //Creating and adding elements
  TreeSet<String> al=new TreeSet<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ravi");
  al.add("Ajay");
  //Traversing elements
  Iterator<String> itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

```
import java.util.*;
class TreeSet4{
 public static void main(String args[]){
  TreeSet<String> set=new TreeSet<String>();
        set.add("A");
        set.add("B");
        set.add("C");
        set.add("D");
        set.add("E");
        System.out.println("Initial Set: "+set);

        System.out.println("Reverse Set: "+set.descendingSet());

        System.out.println("Head Set: "+set.headSet("C", true));

        System.out.println("SubSet: "+set.subSet("A", false, "E", true));

        System.out.println("TailSet: "+set.tailSet("C", false));
```

```
// important import statement
import java.util.*;

class Employee
{

int empId;
String name;

// getting the name of the employee
String getName()
{
   return this.name;
}

// setting the name of the employee
void setName(String name)
{
```

differences and similarities between HashSet and TreeSet.

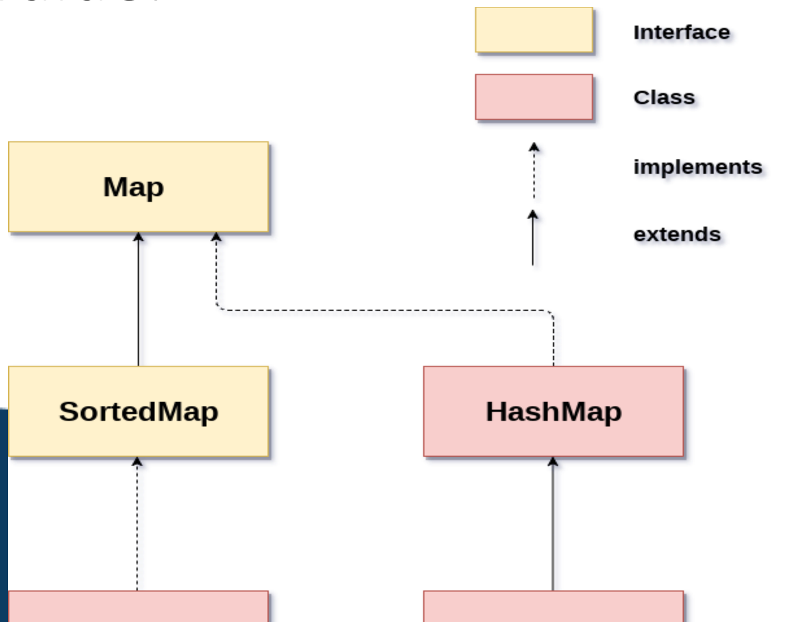


Parameters	HashSet	TreeSet
Data Structure	Hash Table	Red-black Tree
Time Complexity (add/remove/contains)	O(1)	O(log n)
Iteration Order	Arbitrary	Sorted
Null Values	Allowed	Not Allowed
Processing	Fast	Slow

HashSet Vs TreeSet

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.
- 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.



Java Map Example: Non-Generic (Old Style)

```
//Non-generic
import java.util.*;
public class MapExample1 {
public static void main(String[] args) {
    Map map=new HashMap();
    //Adding elements to map
    map.put(1,"Amit");
    map.put(5,"Rahul");
    map.put(2,"Jai");
    map.put(6,"Amit");
    //Traversing Map
    Set set=map.entrySet();//Converting to Set so that we can traverse
    Iterator itr=set.iterator();
    while(itr.hasNext()){
        //Converting to Map.Entry so that we can get key and value separately
        Map.Entry entry=(Map.Entry)itr.next();
        System.out.println(entry.getKey()+" "+entry.getValue());
    }
```

Java Map Example: Generic (New Style)

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

Java Map Example: comparingByKey()

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

Java Map Example: comparingByKey() in Descending Order

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

Java Map Example: comparingByValue()

```
import java.util.*;
class MapExample5{
 public static void main(String args[]){
Map<Integer,String> map=new HashMap<Integer,String>();
      map.put(100,"Amit");
      map.put(101,"Vijay");
      map.put(102,"Rahul");
      //Returns a Set view of the mappings contained in this map
      map.entrySet()
      //Returns a sequential Stream with this collection as its source
      .stream()
      //Sorted according to the provided Comparator
      .sorted(Map.Entry.comparingByValue())
      //Performs an action for each element of this stream
      .forEach(System.out::println);
  }
}
```

Java Map Example: comparingByValue() in Descending Order

```
import java.util.*;
class MapExample6{
 public static void main(String args[]){
Map<Integer,String> map=new HashMap<Integer,String>();
      map.put(100,"Amit");
      map.put(101,"Vijay");
      map.put(102,"Rahul");
     //Returns a Set view of the mappings contained in this map
     map.entrySet()
     //Returns a sequential Stream with this collection as its source
     .stream()
     //Sorted according to the provided Comparator
     .sorted(Map.Entry.comparingByValue(Comparator.reverseOrder()))
     //Performs an action for each element of this stream
     .forEach(System.out::println);
  }
 }
```

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.

Example

```
import java.util.*;
public class HashMapExample1{
 public static void main(String args[]){
   HashMap<Integer,String> map=new HashMap<Integer,String>();//Creating HashMap
   map.put(1,"Mango");  //Put elements in Map
   map.put(2,"Apple");
   map.put(3,"Banana");
   map.put(4,"Grapes");

   System.out.println("Iterating Hashmap...");
   for(Map.Entry m : map.entrySet()){
    System.out.println(m.getKey()+" "+m.getValue());
   }
}
}
```

No Duplicate Key on HashMap

```
 import java.util.*;
public class HashMapExample2{
 public static void main(String args[]){
   HashMap<Integer,String> map=new HashMap<Integer,String>();//Creating HashMap
   map.put(1,"Mango");  //Put elements in Map
   map.put(2,"Apple");
   map.put(3,"Banana");
   map.put(1,"Grapes"); //trying duplicate key

   System.out.println("Iterating Hashmap...");
   for(Map.Entry m : map.entrySet()){
    System.out.println(m.getKey()+" "+m.getValue());
   }
}
}
```

Java HashMap example to add() elements

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

      System.out.println("After invoking put() method ");
      for(Map.Entry m:hm.entrySet()){
       System.out.println(m.getKey()+" "+m.getValue());
      }

      hm.putIfAbsent(103, "Gaurav");
      System.out.println("After invoking putIfAbsent() method ");
      for(Map.Entry m:hm.entrySet()){
        System.out.println(m.getKey()+" "+m.getValue());
       }
      HashMap<Integer,String> map=new HashMap<Integer,String>();
```

Java HashMap example to remove() elements

```
import java.util.*;
public class HashMap2 {
  public static void main(String args[]) {
   HashMap<Integer,String> map=new HashMap<Integer,String>();
     map.put(100,"Amit");
     map.put(101,"Vijay");
     map.put(102,"Rahul");
     map.put(103, "Gaurav");
   System.out.println("Initial list of elements: "+map);
   //key-based removal
   map.remove(100);
   System.out.println("Updated list of elements: "+map);
   //value-based removal
   map.remove(101);
   System.out.println("Updated list of elements: "+map);
   //key-value pair based removal
   map.remove(102, "Rahul");
   System.out.println("Updated list of elements: "+map);
  }
}
```

Java HashMap example to replace() elements

```
import java.util.*;
class HashMap3{
 public static void main(String args[]){
   HashMap<Integer,String> hm=new HashMap<Integer,String>();
      hm.put(100,"Amit");
      hm.put(101,"Vijay");
      hm.put(102,"Rahul");
      System.out.println("Initial list of elements:");
     for(Map.Entry m:hm.entrySet())
     {
        System.out.println(m.getKey()+" "+m.getValue());
     }
     System.out.println("Updated list of elements:");
     hm.replace(102, "Gaurav");
     for(Map.Entry m:hm.entrySet())
     {
        System.out.println(m.getKey()+" "+m.getValue());
     }
```

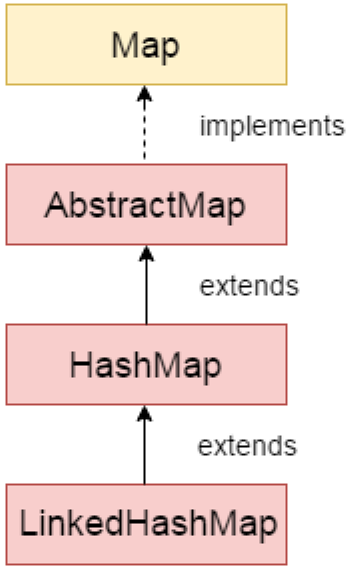
Java HashMap Example: Book

```
import java.util.*;
class Book {
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publisher, int quantity) {
    this.id = id;
    this.name = name;
    this.author = author;
    this.publisher = publisher;
    this.quantity = quantity;
}
}
public class MapExample {
public static void main(String[] args) {
    //Creating map of Books
    Map<Integer,Book> map=new HashMap<Integer,Book>();
    //Creating Books
```

LinkedHashMap

- 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.

```
import java.util.*;
class LinkedHashMap1{
 public static void main(String args[]){

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

  hm.put(100,"Amit");
  hm.put(101,"Vijay");
  hm.put(102,"Rahul");

for(Map.Entry m:hm.entrySet()){
  System.out.println(m.getKey()+" "+m.getValue());
 }
 }
```



Java LinkedHashMap Example: Key-Value pair

```
import java.util.*;
class LinkedHashMap2{
 public static void main(String args[]){
   LinkedHashMap<Integer, String> map = new LinkedHashMap<Integer, String>();
      map.put(100,"Amit");
     map.put(101,"Vijay");
     map.put(102,"Rahul");
       //Fetching key
       System.out.println("Keys: "+map.keySet());
       //Fetching value
       System.out.println("Values: "+map.values());
       //Fetching key-value pair
       System.out.println("Key-Value pairs: "+map.entrySet());
 }
}
```

Java LinkedHashMap Example:remove()

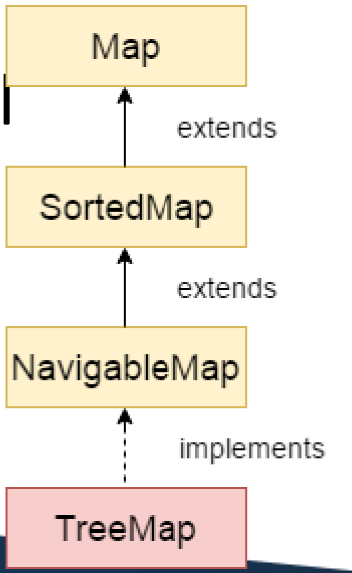
```
import java.util.*;
public class LinkedHashMap3 {
   public static void main(String args[]) {
    Map<Integer,String> map=new LinkedHashMap<Integer,String>();
     map.put(101,"Amit");
     map.put(102,"Vijay");
     map.put(103,"Rahul");
     System.out.println("Before invoking remove() method: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

Java LinkedHashMap Example: Book

```
import java.util.*;
class Book {
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publisher, int quantity) {
    this.id = id;
    this.name = name;
    this.author = author;
    this.publisher = publisher;
    this.quantity = quantity;
}
}
public class MapExample {
public static void main(String[] args) {
    //Creating map of Books
    Map<Integer,Book> map=new LinkedHashMap<Integer,Book>();
    //Creating Books
```

TreeMap

- 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.



Java TreeMap Example

```
import java.util.*;
class TreeMap1{
 public static void main(String args[]){
   TreeMap<Integer,String> map=new TreeMap<Integer,String>();
      map.put(100,"Amit");
      map.put(102,"Ravi");
      map.put(101,"Vijay");
      map.put(103,"Rahul");

      for(Map.Entry m:map.entrySet()){
       System.out.println(m.getKey()+" "+m.getValue());
      }
 }
}
```

Java TreeMap Example: remove()

```
import java.util.*;
public class TreeMap2 {
   public static void main(String args[]) {
    TreeMap<Integer,String> map=new TreeMap<Integer,String>();
      map.put(100,"Amit");
      map.put(102,"Ravi");
      map.put(101,"Vijay");
      map.put(103,"Rahul");
      System.out.println("Before invoking remove() method");
      for(Map.Entry m:map.entrySet())
      {
          System.out.println(m.getKey()+" "+m.getValue());
      }
      map.remove(102);
      System.out.println("After invoking remove() method");
      for(Map.Entry m:map.entrySet())
      {
          System.out.println(m.getKey()+" "+m.getValue());
```

Java TreeMap Example: NavigableMap

- **import** java.util.*;
- **class** TreeMap3{
- **public static void** main(String args[]){
-    NavigableMap<Integer,String> map=**new** TreeMap<Integer,String>();
-       map.put(100,"Amit");
-       map.put(102,"Ravi");
-       map.put(101,"Vijay");
-       map.put(103,"Rahul");
-       //Maintains descending order
-       System.out.println("descendingMap: "+map.descendingMap());
-       //Returns key-value pairs whose keys are less than or equal to the specified key.
-       System.out.println("headMap: "+map.headMap(102,**true**));
-       //Returns key-value pairs whose keys are greater than or equal to the specified key.
-       System.out.println("tailMap: "+map.tailMap(102,**true**));
-       //Returns key-value pairs exists in between the specified key.
-       System.out.println("subMap: "+map.subMap(100, **false**, 102, **true**));
- }
- }

Java TreeMap Example: SortedMap

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

Java TreeMap Example: Book

```
import java.util.*;
class Book {
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publisher, int quantity) {
    this.id = id;
    this.name = name;
    this.author = author;
    this.publisher = publisher;
    this.quantity = quantity;
}
}
public class MapExample {
public static void main(String[] args) {
    //Creating map of Books
    Map<Integer,Book> map=new TreeMap<Integer,Book>();
    //Creating Books
```

Hashtable

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

```
import java.util.*;
class Hashtable1{
 public static void main(String args[]){
  Hashtable<Integer,String> hm=new Hashtable<Integer,String>();

  hm.put(100,"Amit");
  hm.put(102,"Ravi");
  hm.put(101,"Vijay");
  hm.put(103,"Rahul");

  for(Map.Entry m:hm.entrySet()){
   System.out.println(m.getKey()+" "+m.getValue());
  }
 }
}
```

Java Hashtable Example: remove()

```
import java.util.*;
public class Hashtable2 {
   public static void main(String args[]) {
  Hashtable<Integer,String> map=new Hashtable<Integer,String>();
     map.put(100,"Amit");
     map.put(102,"Ravi");
     map.put(101,"Vijay");
     map.put(103,"Rahul");
     System.out.println("Before remove: "+ map);
       // Remove value for key 102
       map.remove(102);
       System.out.println("After remove: "+ map);
   }
}
```

Java Hashtable Example: getOrDefault()

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

Java Hashtable Example: putIfAbsent()

```
import java.util.*;
class Hashtable4{
 public static void main(String args[]){
    Hashtable<Integer,String> map=new Hashtable<Integer,String>();
     map.put(100,"Amit");
     map.put(102,"Ravi");
     map.put(101,"Vijay");
     map.put(103,"Rahul");
     System.out.println("Initial Map: "+map);
     //Inserts, as the specified pair is unique
     map.putIfAbsent(104,"Gaurav");
     System.out.println("Updated Map: "+map);
     //Returns the current value, as the specified pair already exist
     map.putIfAbsent(101,"Vijay");
     System.out.println("Updated Map: "+map);
 }
}
```

Java Hashtable Example: Book

```
import java.util.*;
class Book {
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publisher, int quantity) {
    this.id = id;
    this.name = name;
    this.author = author;
    this.publisher = publisher;
    this.quantity = quantity;    }
}
public class HashtableExample {
public static void main(String[] args) {
    //Creating map of Books
    Map<Integer,Book> map=new Hashtable<Integer,Book>();
    //Creating Books
    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
    Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
    Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
```

EnumSet

Java EnumSet class is the specialized Set implementation for use with enum types. It inherits AbstractSet class and implements the Set interface.

```
import java.util.*;
enum days {
  SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
public class EnumSetExample {
  public static void main(String[] args) {
    Set<days> set = EnumSet.of(days.TUESDAY, days.WEDNESDAY);
    // Traversing elements
    Iterator<days> iter = set.iterator();
    while (iter.hasNext())
      System.out.println(iter.next());
  }
}
```

Java EnumSet Example: allOf() and noneOf()

```
import java.util.*;
enum days {
  SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
public class EnumSetExample {
  public static void main(String[] args) {
    Set<days> set1 = EnumSet.allOf(days.class);
      System.out.println("Week Days:"+set1);
      Set<days> set2 = EnumSet.noneOf(days.class);
      System.out.println("Week Days:"+set2);
  }
}
```

EnumMap

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

```
import java.util.*;
public class EnumMapExample {
   // create an enum
   public enum Days {
   Monday, Tuesday, Wednesday, Thursday
   };
   public static void main(String[] args) {
   //create and populate enum map
   EnumMap<Days, String> map = new EnumMap<Days, String>(Days.class);
   map.put(Days.Monday, "1");
   map.put(Days.Tuesday, "2");
   map.put(Days.Wednesday, "3");
   map.put(Days.Thursday, "4");
   // print the map
   for(Map.Entry m:map.entrySet()){
```

```
import java.util.*;
class Book {
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publisher, int quantity) {
    this.id = id;
    this.name = name;
    this.author = author;
    this.publisher = publisher;
    this.quantity = quantity;
}
}
public class EnumMapExample {
// Creating enum
    public enum Key{
        One, Two, Three
        };
```

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,

```
class Student implements Comparable<Student>{
int rollno;
String name;
int age;
Student(int rollno,String name,int age){
this.rollno=rollno;
this.name=name;
this.age=age;
}

public int compareTo(Student st){
if(age==st.age)
return 0;
else if(age>st.age)
return 1;
else
return -1;
}
```

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

Collections.sort(al);
for(Student st:al){
System.out.println(st.rollno+" "+st.name+" "+st.age);
}
}
}
```

Comparable Example: reverse order

```
class Student implements Comparable<Student>{
 int rollno;
 String name;
 int age;
 Student(int rollno,String name,int age){
 this.rollno=rollno;
 this.name=name;
 this.age=age;
 }

 public int compareTo(Student st){
 if(age==st.age)
 return 0;
 else if(age<st.age)
 return 1;
 else
 return -1;
 }
```

```
import java.util.*;
public class TestSort2{
public static void main(String args[]){
ArrayList<Student> al=new ArrayList<Student>();
al.add(new Student(101,"Vijay",23));
al.add(new Student(106,"Ajay",27));
al.add(new Student(105,"Jai",21));

Collections.sort(al);
for(Student st:al){
System.out.println(st.rollno+" "+st.name+" "+st.age);
}
}
}
```

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.

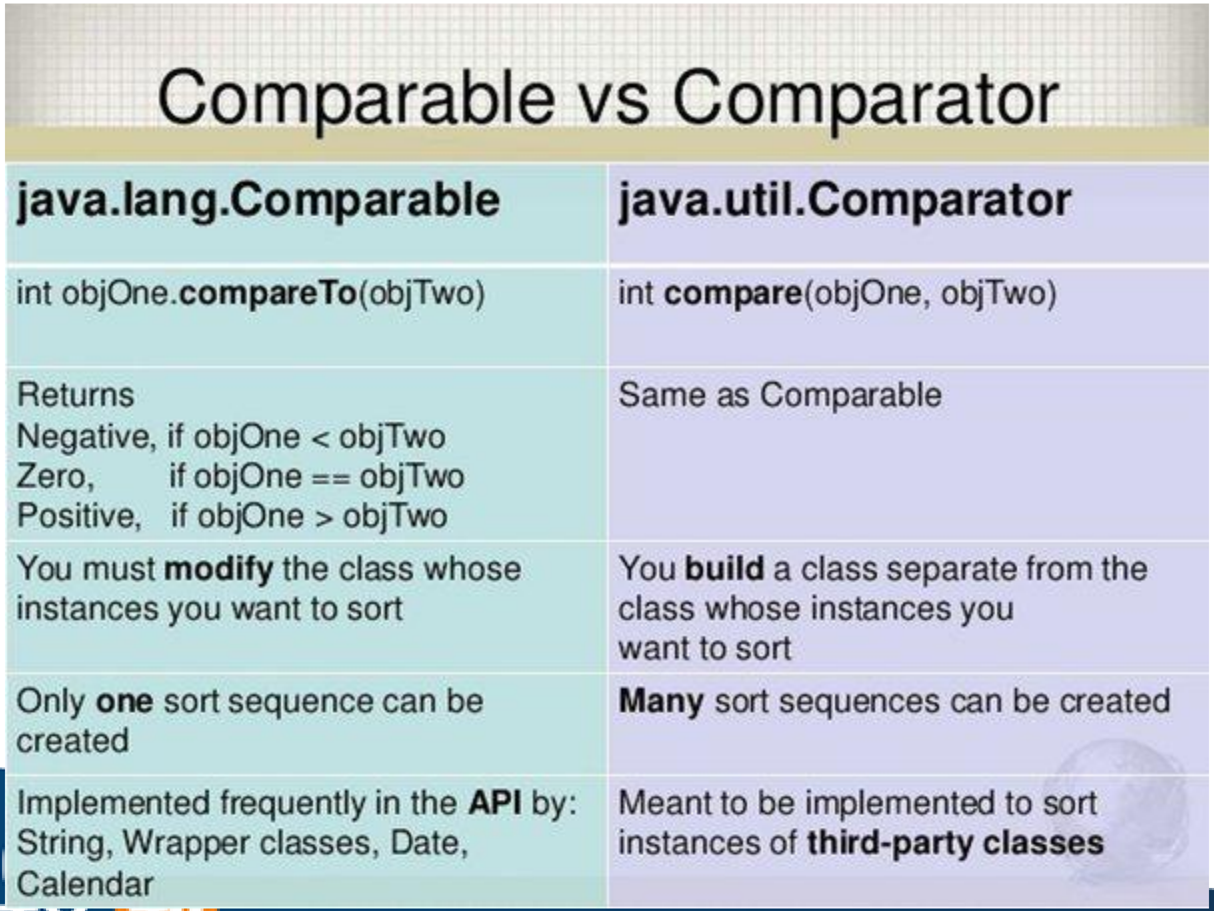
```
class Student {
   int rollno;
   String name;
  int age;
    Student(int rollno,String name,int age){
    this.rollno=rollno;
    this.name=name;
    this.age=age;
    }

    public int getRollno() {
        return rollno;
    }

    public void setRollno(int rollno) {
        this.rollno = rollno;
    }
```

```
import java.util.*;
  public class TestSort1{
  public static void main(String args[]){
  ArrayList<Student> al=new ArrayList<Student>();
  al.add(new Student(101,"Vijay",23));
  al.add(new Student(106,"Ajay",27));
  al.add(new Student(105,"Jai",21));
/Sorting elements on the basis of name
  Comparator<Student> cm1=Comparator.comparing(Student::getName);
   Collections.sort(al,cm1);
   System.out.println("Sorting by Name");
   for(Student st: al){
     System.out.println(st.rollno+" "+st.name+" "+st.age);
     }
   //Sorting elements on the basis of age
    Comparator<Student> cm2=Comparator.comparing(Student::getAge);
Collections.sort(al,cm2);
   System.out.println("Sorting by Age");
```

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

```
class Student {
   int rollno;
   String name;
  int age;
    Student(int rollno,String name,int age){
    this.rollno=rollno;
    this.name=name;
    this.age=age;
    }
    public int getRollno() {
        return rollno;
    }
    public void setRollno(int rollno) {
        this.rollno = rollno;
    }
    public String getName() {
        return name;
    }
```

Distinguish between Comparable Versus Comparator



Comparable vs Comparator

java.lang.Comparable	**java.util.Comparator**
int objOne.**compareTo**(objTwo)	int **compare**(objOne, objTwo)
Returns Negative, if objOne < objTwo Zero, if objOne == objTwo Positive, if objOne > objTwo	Same as Comparable
You must **modify** the class whose instances you want to sort	You **build** a class separate from the class whose instances you want to sort
Only **one** sort sequence can be created	**Many** sort sequences can be created
Implemented frequently in the **API** by: String, Wrapper classes, Date, Calendar	Meant to be implemented to sort instances of **third-party classes**

Generics

- The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects. It makes the code stable by detecting the bugs at compile time.
- Before generics, we can store any type of objects in the collection, i.e., non-generic. Now generics force the java programmer to store a specific type of objects.

Advantage of Java Generics

1. **Type-safety:** We can hold only a single type of objects in generics. It doesnot allow to store other objects.
2. **Type casting is not required:** There is no need to typecast the object.
3. **Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

Type Safety

- Without Generics, we can store any type of objects.

```
List list = new ArrayList();
list.add(10);
list.add("10");
```

- With Generics, it is required to specify the type of object we need to s tore.

```
List<Integer> list = new ArrayList<Integer>();
list.add(10);
list.add("10");// compile-time error
```

Type casting is not required

- There is no need to typecast the object.
- Before Generics, we need to type cast.

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);//typecasting
```

- After Generics, we don't need to typecast the object.

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

Example1

```
import java.util.*;
class TestGenerics1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();
list.add("rahul");
list.add("jai");
//list.add(32);//compile time error

String s=list.get(1);//type casting is not required
System.out.println("element is: "+s);

Iterator<String> itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

```
import java.util.*;
class TestGenerics1{
```

Example2

```
import java.util.*;
class TestGenerics2{
public static void main(String args[]){
Map<Integer,String> map=new HashMap<Integer,String>();
map.put(1,"vijay");
map.put(4,"umesh");
map.put(2,"ankit");

//Now use Map.Entry for Set and Iterator
Set<Map.Entry<Integer,String>> set=map.entrySet();

Iterator<Map.Entry<Integer,String>> itr=set.iterator();
while(itr.hasNext()){
Map.Entry e=itr.next();//no need to typecast
System.out.println(e.getKey()+" "+e.getValue());
}

}
```

Generic class

- A class that can refer to any type is known as a generic class. Here, we are using the T type parameter to create the generic class of specific type.
- **Creating a generic class:**

```
class MyGen<T>{
        T obj;
        void set(T obj){this.obj=obj;}
        T get(){return obj;}
}
```

- The T type indicates that it can refer to any type (like String, Integer, and Employee). The type you specify for the class will be used to store and retrieve the data.

Using generic class

```
class TestGenerics3{
    public static void main(String args[]){
        MyGen<Integer> m=new MyGen<Integer>();
        m.add(2);
        //m.add("vivek");//Compile time error
        System.out.println(m.get());
    }
}
```

Contd..

```
class TestGenerics3 {
public static void main(String args[] )   {
     MyGen<Integer> m=new MyGen<Integer>();
     m.add(2);
    //m.add("vivek");//Compile time error
    System.out.println(m.get());
}
}
```

Example 2

```
class Pair<K, V>  {
 private K key;
 private V value;
 public void setKey(K key) {
        this.key = key;
  }
 public void setValue(V value) {
         this.value = value;
 }
public K getKey() {
      return key;
}
public V getValue() {
          return value;
}
 }
```

Contd...

```
class HelloWorld {
        public static void main(String[] args) {
            Pair<Integer,String> p = new Pair<Integer,String>();
            p.setKey(1);
           p.setValue("abc");
          System.out.println(p.getKey() + p.getValue());
     }
}
```

Type Parameters

- The type parameters naming conventions are important to learn generics thoroughly. The common type parameters are as follows:
- T - Type
- E - Element
- K - Key
- N - Number
- V - Value

Generic Method

- Like the generic class, we can create a generic method that can accept any type of arguments. Here, the scope of arguments is limited to the method where it is declared. It allows static as well as non-static methods.
- Let's see a simple example of java generic method to print array elements. We are using here **E** to denote the element.

```
public class TestGenerics4{
  public static < E > void printArray(E[] elements) {
      for ( E element : elements){
         System.out.println(element );
       }
       System.out.println();
   }
   public static void main( String args[] ) {
      Integer[] intArray = { 10, 20, 30, 40, 50 };
      Character[] charArray = { 'J', 'A', 'V', 'A', 'T','P','O','I','N','T' };

      System.out.println( "Printing Integer Array" );
```

Wildcard in Java Generics

- The ? (question mark) symbol represents the wildcard element. It means any type. If we write <? extends Number>, it means any child class of Number, e.g., Integer, Float, and double. Now we can call the method of Number class through any child class object.
- We can use a wildcard as a type of a parameter, field, return type, or local variable. However, it is not allowed to use a wildcard as a type argument for a generic method invocation, a generic class instance creation, or a supertype.

```
import java.util.*;
    abstract void draw();
}
abstract class Shape{
class Rectangle extends Shape{
     void draw(){System.out.println("drawing rectangle");}
}
class Circle extends Shape{
      void draw(){System.out.println("drawing circle");}
}
```

Example

```
import java.util.ArrayList;
  public class UpperBoundWildcard {

        private static Double add(ArrayList<? extends Number> num) {
              double sum=0.0;
              for(Number n:num)    {
                       sum = sum+n.doubleValue();
              }
              return sum;
        }
      public static void main(String[] args) {
        ArrayList<Integer> l1=new ArrayList<Integer>();
        l1.add(10);
        l1.add(20);
        System.out.println("displaying the sum= "+add(l1));

        ArrayList<Double> l2=new ArrayList<Double>();
        l2.add(30.0);
        l2.add(40.0);
        System.out.println("displaying the sum= "+add(l2));
      }
}
```

Upper Bounded Wildcards

- The purpose of upper bounded wildcards is to decrease the restrictions on a variable. It restricts the unknown type to be a specific type or a subtype of that type. It is used by declaring wildcard character ("?") followed by the extends (in case of, class) or implements (in case of, interface) keyword, followed by its upper bound.
- Syntax
- List<? **extends** Number>
- **?** is a wildcard character.
- **extends**, is a keyword.
- **Number**, is a class present in java.lang package
- Suppose, we want to write the method for the list of Number and its subtypes (like Integer, Double). Using **List<? extends Number>** is suitable for a list of type Number or any of its subclasses

```
import java.util.ArrayList;
 public class UpperBoundWildcard {
     private static Double add(ArrayList<? extends Number> num) {
         double sum=0.0;
         for(Number n:num)   {
           sum = sum+n.doubleValue();
         }
         return sum;
    }
     public static void main(String[] args) {
         ArrayList<Integer> l1=new ArrayList<Integer>();
```

Unbounded Wildcards

The unbounded wildcard type represents the list of an unknown type such as List<?>. This approach can be useful in the following scenarios: -

When the given method is implemented by using the functionality provided in the Object class.

When the generic class contains the methods that don't depend on the type parameter.

Example of Unbounded Wildcards

```
import java.util.Arrays;
import java.util.List;

public class UnboundedWildcard {

  public static void display(List<?> list)
```

Lower Bounded Wildcards

- The purpose of lower bounded wildcards is to restrict the unknown type to be a specific type or a supertype of that type. It is used by declaring wildcard character ("?") followed by the super keyword, followed by its lower bound.
- Syntax
- List<? **super** Integer>
- Here,
- **?** is a wildcard character.
- **super**, is a keyword.
- **Integer**, is a wrapper class.
- Suppose, we want to write the method for the list of Integer and its supertype (like Number, Object). Using **List<? super Integer>** is suitable for a list of type Integer or any of its superclasses

Example

```
import java.util.Arrays;
import java.util.List;

public class LowerBoundWildcard {

    public static void addNumbers(List<? super Integer> list) {

        for(Object n:list)
        {
            System.out.println(n);
        }
```

PRESIDENCY UNIVERSITY

GAIN MORE KNOWLEDGE REACH GREATER HEIGHTS

Private University Estd. in Karnataka State by Act No. 41 of 2013

PRESIDENCY GROUP OVER 40 YEARS OF ACADEMIC WISDOM

Annotation

- Java **Annotation** is a tag that represents the *metadata* i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.
- Annotations in Java are used to provide additional information, so it is an alternative option for XML and Java marker interfaces.

Built-In Java Annotations

- Built-In Java Annotations used in Java code

@Override

@SuppressWarnings

@Deprecated

- Built-In Java Annotations used in other annotations

@Target

@Retention

@Inherited

@Documented

@Override

- @Override annotation assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.
- Sometimes, we does the silly mistake such as spelling mistakes etc. So, it is better to mark @Override annotation that provides assurity that method is overridden.

```
class Animal{
    void eatSomething() { System.out.println("eating something"); }
}
 class Dog extends Animal{
   @Override
   void eatsomething() { System.out.println("eating foods"); }//should be eatSomething
}
 class TestAnnotation1{
        public static void main(String args[]) {
             Animal a=new Dog();
             a.eatSomething();
      }
}
```

PRESIDENCY UNIVERSITY

@SuppressWarnings

- @SuppressWarnings annotation: is used to suppress warnings issued by the compiler.
- If you remove the @SuppressWarnings("unchecked") annotation, it will show warning at compile time because we are using non-generic collection.

```
import java.util.*;
class TestAnnotation2{
    @SuppressWarnings("unchecked")
    public static void main(String args[]){
        ArrayList list=new ArrayList();
        list.add("sonoo");
        list.add("vimal");
        list.add("ratan");
       for(Object obj:list)
           System.out.println(obj);
  }
}
```

@Deprecated

@Deprecated annoation marks that this method is deprecated so compiler prints warning. It informs user that it may be removed in the future versions. So, it is better not to use such methods.

```
class A{
    void m(){  System.out.println("hello m"); }
   @Deprecated
    void n(){System.out.println("hello n");}
}

class TestAnnotation3{
    public static void main(String args[]){
            A a=new A();
            a.n();
    }
}
```

PRESIDENCY UNIVERSITY
Private University Estd. in Karnataka State by Act No. 41 of 2013

Java Custom Annotations/ Java User-defined annotations

- are easy to create and use.
- The *@interface* element is used to declare an annotation. For example:

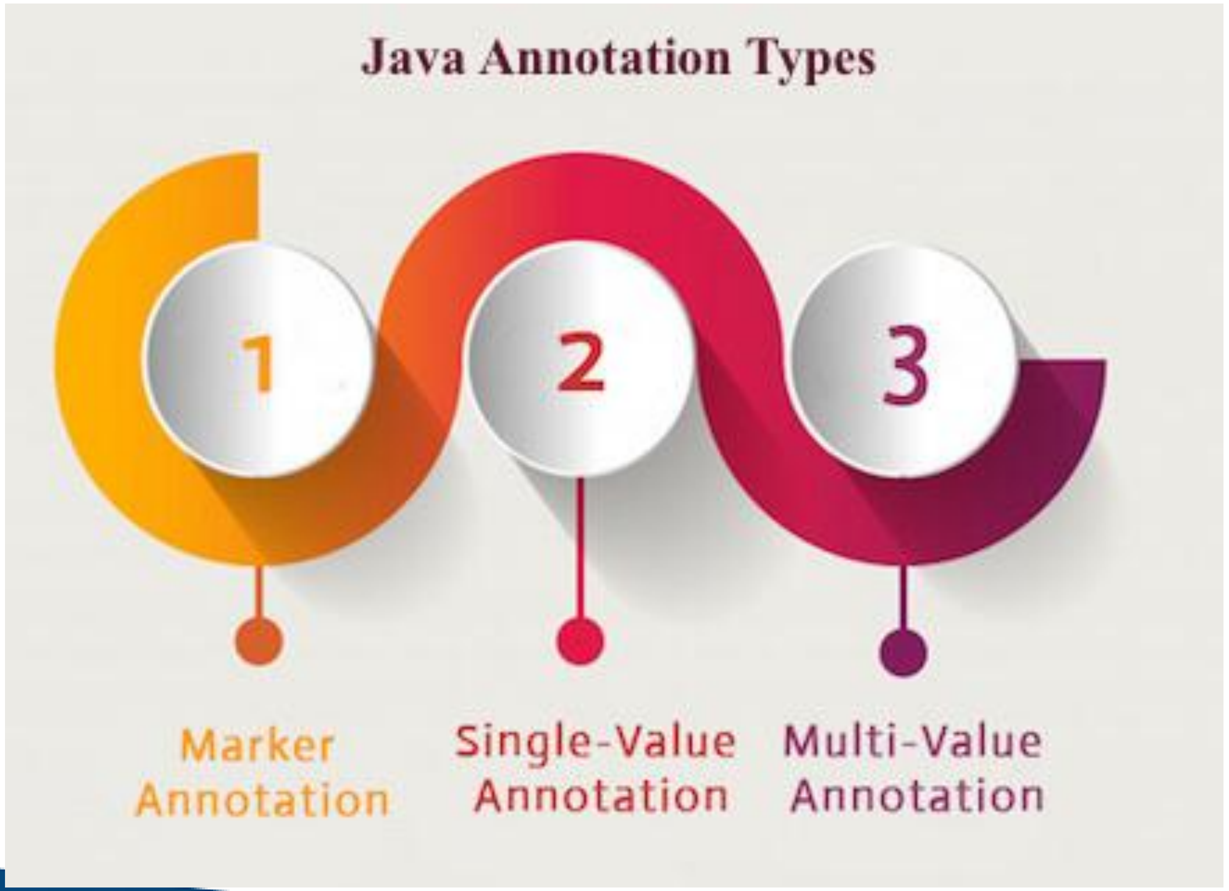
  @interface MyAnnotation{}

Here, MyAnnotation is the custom annotation name.

- There are few points that should be remembered by the programmer.

1. Method should not have any throws clauses
2. Method should return one of the following: primitive data types, String, Class, enum or array of these data types.
3. Method should not have any parameter.
4. We should attach @ just before interface keyword to define annotation

Types of Annotation



Marker Annotation

- An annotation that has no method, is called marker annotation. For example:
- **@interface** MyAnnotation{}
- The @Override and @Deprecated are marker annotations.

Single-Value Annotation

- An annotation that has one method, is called single-value annotation. For example

```
@interface MyAnnotation{
int value();
}
```

- We can provide the default value also. For example:

```
@interface MyAnnotation{
int value() default 0;
}
```

Multi-Value Annotation

An annotation that has more than one method, is called Multi-Value annotation. For example:

```
@interface MyAnnotation {
    int value1();
   String value2();
   String value3();
}
```

We can provide the default value also. For example:

```
@interface MyAnnotation {
    int value1() default 1;
    String value2() default "";
    String value3() default "xyz";
}
```

How to apply Multi-Value Annotation

Let's see the code to apply the multi-value annotation.

Built-in Annotations used in custom annotations in java

- @Target
- @Retention
- @Inherited
- @Documented

@Target

- **@Target** tag is used to specify at which type, the annotation is used.
- The java.lang.annotation. **ElementType** enum declares many constants to specify the type of element where annotation is to be applied such as TYPE, METHOD, FIELD etc. Let's see the constants of ElementType enum:

Element Types	**Where the annotation can be applied**
TYPE	class, interface or enumeration
FIELD	fields
METHOD	methods
CONSTRUCTOR	constructors
LOCAL_VARIABLE	local variables
ANNOTATION_TYPE	annotation type
PARAMETER	parameter

Example

- Example to specify annoation for a class

```
@Target(ElementType.TYPE)
@interface MyAnnotation{
    int value1();
    String value2();
}
```

- Example to specify annotation for a class, methods or fields

```
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.METHOD})
@interface MyAnnotation{
   int value1();
   String value2();
}
```

@Retention

@Retention annotation is used to specify to what level annotation will be available.

RetentionPolicy	**Availability**
RetentionPolicy.SOURCE	refers to the source code, discarded during compilation. It will not be available in the compiled class.
RetentionPolicy.CLASS	refers to the .class file, available to java compiler but not to JVM . It is included in the class file.
RetentionPolicy.RUNTIME	refers to the runtime, available to java compiler and JVM .

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface MyAnnotation{
int value1();
String value2();
}
```

Example

- *File: Test.java*

```
//Creating annotation
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface MyAnnotation{
int value();
}

//Applying annotation
class Hello{
@MyAnnotation(value=10)
public void sayHello(){System.out.println("hello annotation");}
}

//Accessing annotation
```

@Inherited

- By default, annotations are not inherited to subclasses. The @Inherited annotation marks the annotation to be inherited to subclasses.

```
@Inherited
@interface ForEveryone { }//Now it will be available to subclass also

@interface ForEveryone { }
class Superclass{}

class Subclass extends Superclass{}
```

@Documented

- The @Documented Marks the annotation for inclusion in the documentation.

Lambda Expression

- Lambda expression is a new and important feature of Java which was included in Java SE 8.
- It provides a clear and concise way to represent one method interface using an expression.
- It is very useful in collection library. It helps to iterate, filter and extract data from collection.
- The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code.
- In case of lambda expression, we don't need to define the method again for providing the implementation.
- Java lambda expression is treated as a function, so compiler does not create .class file.

Functional Interface

- Lambda expression provides implementation of *functional interface*.
- An interface which has only one abstract method is called functional interface.
- Java provides an anotation *@FunctionalInterface*, which is used to declare an interface as functional interface.

Why use Lambda Expression

1. To provide the implementation of Functional interface.
2. Less coding.

- <u>Java Lambda Expression Syntax</u>

  (argument-list) -> {body}

- Java lambda expression is consisted of three components.

1) Argument-list: It can be empty or non-empty as well.

2) Arrow-token: It is used to link arguments-list and body of expression.

3) Body: It contains expressions and statements for lambda expression.

Parameter Syntax

- **No Parameter Syntax**

() -> {

//Body of no parameter lambda

}

- **One Parameter Syntax**

(p1) -> {

//Body of single parameter lambda

}

- **Two Parameter Syntax**

(p1,p2) -> {

//Body of multiple parameter lambda

Without Lambda Expression

```
interface Drawable{
    public void draw();
}
public class LambdaExpressionExample {
    public static void main(String[] args) {
        int width=10;

        //without lambda, Drawable implementation using anonymous class
        Drawable d=new Drawable(){
            public void draw(){System.out.println("Drawing "+width);}
        };
        d.draw();
    }
}
```

Java Lambda Expression Example

```
@FunctionalInterface  //It is optional
interface Drawable{
   public void draw();
}

public class LambdaExpressionExample2 {
   public static void main(String[] args) {
      int width=10;

      //with lambda
      Drawable d2=()->{
         System.out.println("Drawing "+width);
      };
      d2.draw();
   }
}
```

Java Lambda Expression Example: No Parameter

```
interface Sayable{
    public String say();
}
public class LambdaExpressionExample3{
public static void main(String[] args) {
    Sayable s=()->{
        return "I have nothing to say.";
    };
    System.out.println(s.say());
}
}
```

PRESIDENCY UNIVERSITY

Private University Estd. in Karnataka State by Act No. 41 of 2013

Java Lambda Expression Example: Single Parameter

```
interface Sayable{
   public String say(String name);
}
public class LambdaExpressionExample4{
   public static void main(String[] args) {
      // Lambda expression with single parameter.
       Sayable s1=(name)->{
          return "Hello, "+name;
       };
       System.out.println(s1.say("Sonoo"));

       // You can omit function parentheses
       Sayable s2= name ->{
          return "Hello, "+name;
       };
       System.out.println(s2.say("Sonoo"));
   }
}
```

Java Lambda Expression Example: Multiple Parameters

```
interface Addable{
    int add(int a,int b);
}

public class LambdaExpressionExample5{
    public static void main(String[] args) {
        // Multiple parameters in lambda expression
        Addable ad1=(a,b)->(a+b);
        System.out.println(ad1.add(10,20));
        // Multiple parameters with data type in lambda expression
        Addable ad2=(int a,int b)->(a+b);
        System.out.println(ad2.add(100,200));
    }
}
```

Java Lambda Expression Example: with or without return keyword

```
interface Addable{
    int add(int a,int b);
}

public class LambdaExpressionExample6 {
    public static void main(String[] args) {

        // Lambda expression without return keyword.
        Addable ad1=(a,b)->(a+b);
        System.out.println(ad1.add(10,20));

        // Lambda expression with return keyword.
        Addable ad2=(int a,int b)->{
                    return (a+b);
                    };
```

Java Lambda Expression Example: Foreach Loop

```
 import java.util.*;
public class LambdaExpressionExample7{
   public static void main(String[] args) {

        List<String> list=new ArrayList<String>();
        list.add("ankit");
        list.add("mayank");
        list.add("irfan");
        list.add("jai");

        list.forEach(
           (n)->System.out.println(n)
        );
   }
}
```

Java Lambda Expression Example: Multiple Statements

```
 @FunctionalInterface
interface Sayable{
   String say(String message);
}

public class LambdaExpressionExample8{
   public static void main(String[] args) {

      // You can pass multiple statements in lambda expression
      Sayable person = (message)-> {
         String str1 = "I would like to say, ";
         String str2 = str1 + message;
         return str2;
      };
```

Java Lambda Expression Example: Creating Thread

```
public class LambdaExpressionExample9{
    public static void main(String[] args) {

        //Thread Example without lambda
        Runnable r1=new Runnable(){
            public void run(){
                System.out.println("Thread1 is running...");
            }
        };
        Thread t1=new Thread(r1);
        t1.start();
        //Thread Example with lambda
        Runnable r2=()->{
                System.out.println("Thread2 is running...");
        };
        Thread t2=new Thread(r2);
        t2.start();
    }
}
```

Java Lambda Expression Example: Comparator

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
class Product{
    int id;
    String name;
    float price;
    public Product(int id, String name, float price) {
        super();
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
public class LambdaExpressionExample10{
    public static void main(String[] args) {
        List<Product> list=new ArrayList<Product>();

        //Adding Products
        list.add(new Product(1,"HP Laptop",25000f));
        list.add(new Product(3,"Keyboard",300f));
```

Java Lambda Expression Example: Filter Collection Data

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;
class Product{
   int id;
   String name;
   float price;
   public Product(int id, String name, float price) {
      super();
      this.id = id;
      this.name = name;
      this.price = price;
   }
}
public class LambdaExpressionExample11{
   public static void main(String[] args) {
      List<Product> list=new ArrayList<Product>();
      list.add(new Product(1,"Samsung A5",17000f));
      list.add(new Product(3,"Iphone 6S",65000f));
      list.add(new Product(2,"Sony Xperia",25000f));
      list.add(new Product(4,"Nokia Lumia",15000f));
      list.add(new Product(5,"Redmi4 ",26000f));
      list.add(new Product(6,"Lenevo Vibe",19000f));
```

Java Lambda Expression Example: Event Listener

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JTextField;
public class LambdaEventListenerExample {
    public static void main(String[] args) {
        JTextField tf=new JTextField();
        tf.setBounds(50, 50,150,20);
        JButton b=new JButton("click");
        b.setBounds(80,100,70,30);

        // lambda expression implementing here.
        b.addActionListener(e-> {tf.setText("hello swing");});

        JFrame f=new JFrame();
        f.add(tf);f.add(b);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setLayout(null);
        f.setSize(300, 200);
        f.setVisible(true);
```

API

- API (Application programming interface) is a document that contains a description of all the features of a product or software. It represents classes and interfaces that software programs can follow to communicate with each other. An API can be created for applications, libraries, operating systems, etc

JDBC

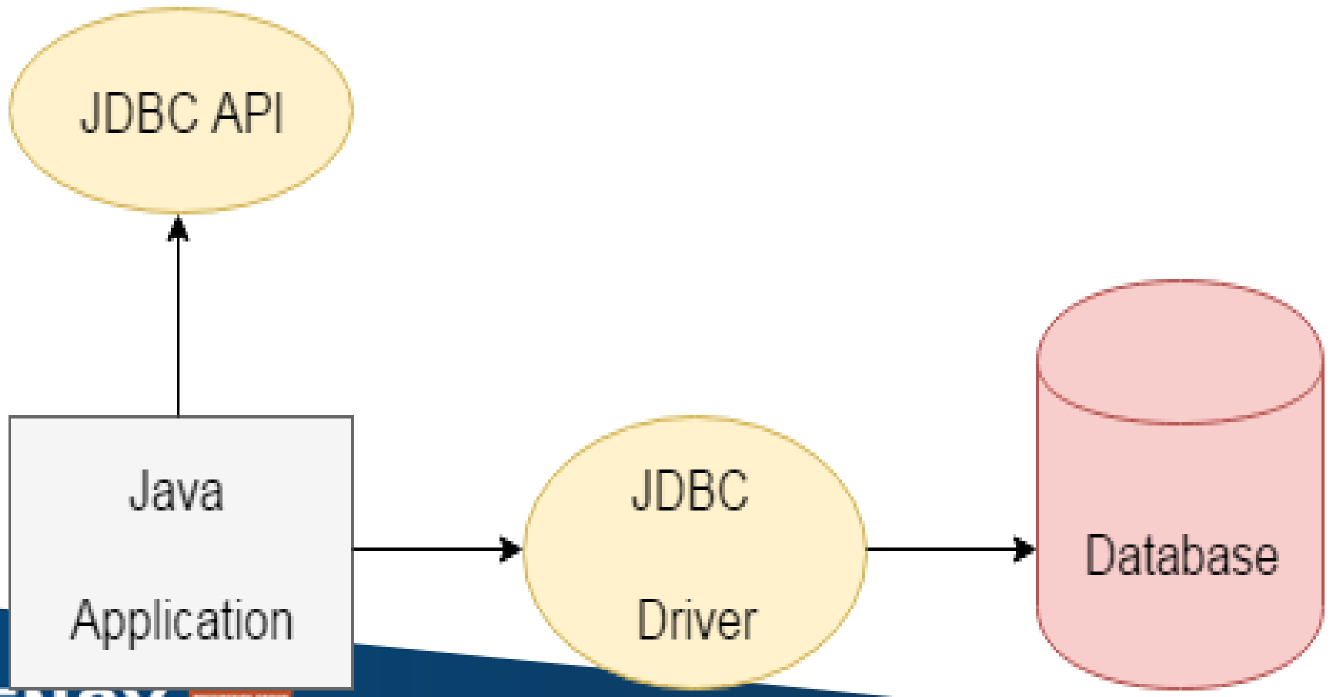
- JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database.

There are four types of JDBC drivers:

- JDBC-ODBC Bridge Driver,
- Native Driver,
- Network Protocol Driver, and
- Thin Driver
- use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database. It is like Open Database Connectivity (ODBC)

JDBC

The current version of JDBC is 4.3. It is the stable release since 21st September, 2017. It is based on the X/Open SQL Call Level Interface. The **java.sql** package contains classes and interfaces for JDBC API.



interfaces of JDBC API

- Driver interface
- Connection interface
- Statement interface
- PreparedStatement interface
- CallableStatement interface
- ResultSet interface
- ResultSetMetaData interface
- DatabaseMetaData interface
- RowSet interface

classes of JDBC API

DriverManager class

- Blob class
- Clob class
- Types class

- Before JDBC, ODBC API was the database API to connect and execute the query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).
- We can use JDBC API to handle database using Java program and can perform the following activities:

1. Connect to the database
2. Execute queries and update statements to the database
3. Retrieve the result received from the database.

JDBC Driver

- JDBC Driver is a software component that enables java application to interact with the database.
- There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)
4. Thin driver (fully java driver)

JDBC-ODBC bridge driver

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver.

- Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

Advantages:

- easy to use.
- can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC meth
the ODBC function calls.

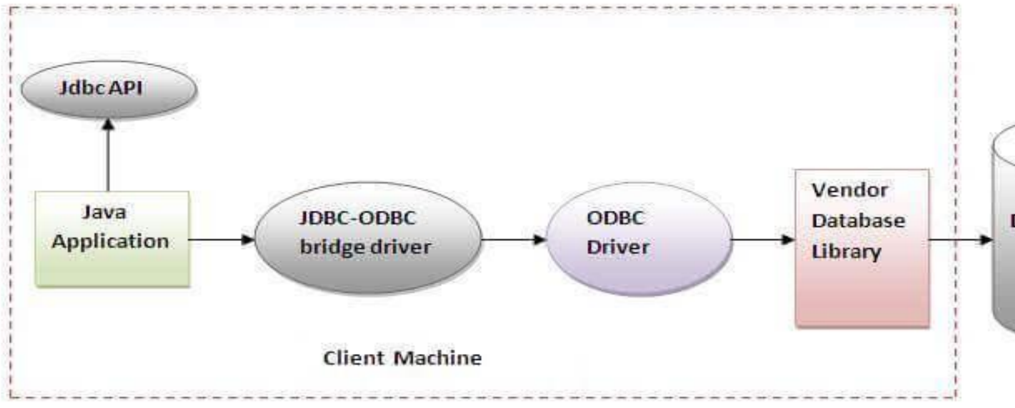


Figure- JDBC-ODBC Bridge Driver

Native-API driver

- The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.
- Advantage:
- performance upgraded than JDBC-ODBC bridge driver.
- Disadvantage:
- The Native driver needs to be installed on the each
- The Vendor client library needs to be installed on cli

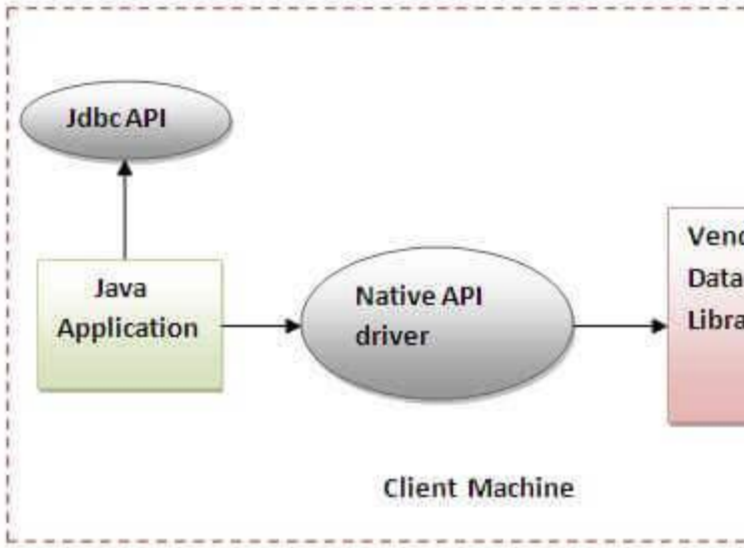


Figure- Native API Driv

Network Protocol driver

- The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.
- Advantage:
- No client side library is required because of can perform many tasks like auditing, load
- Disadvantages:
- Network support is required on client mach
- Requires database-specific coding to be dor
- Maintenance of Network Protocol driver be requires database-specific coding to be done in the middle tier.
-

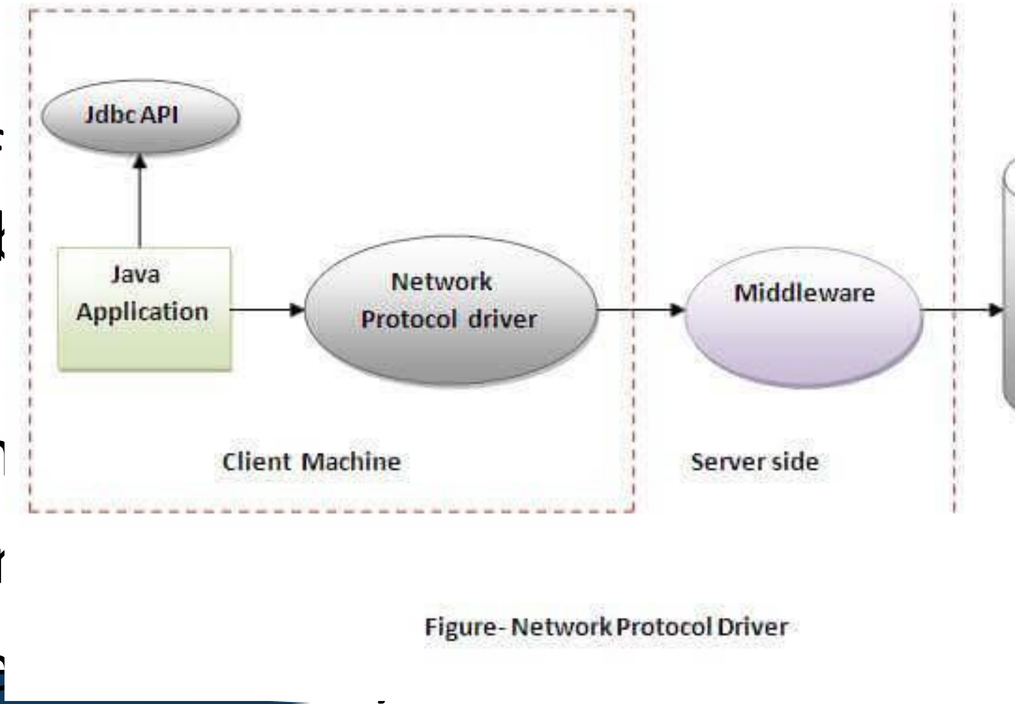


Figure- Network Protocol Driver

Thin driver

- The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.
  Advantage:
- Better performance than all other driv
- No software is required at client side c

Disadvantage:

- Drivers depend on the Database.

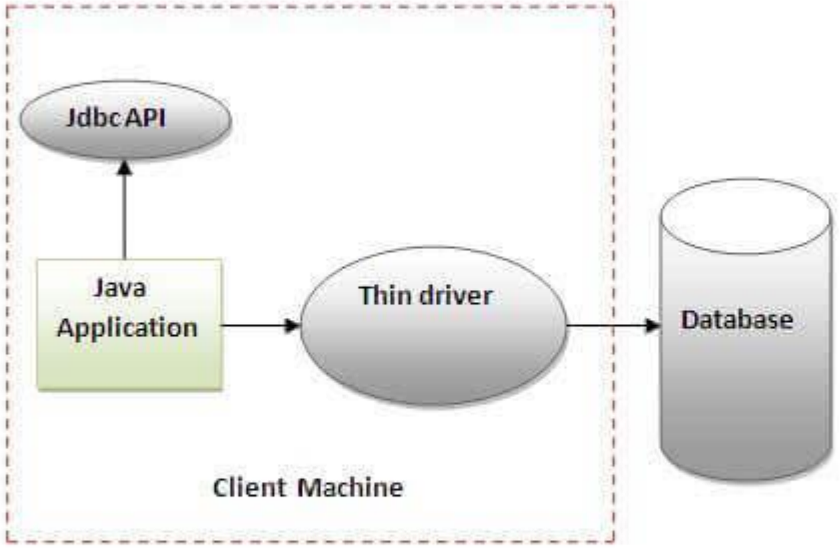
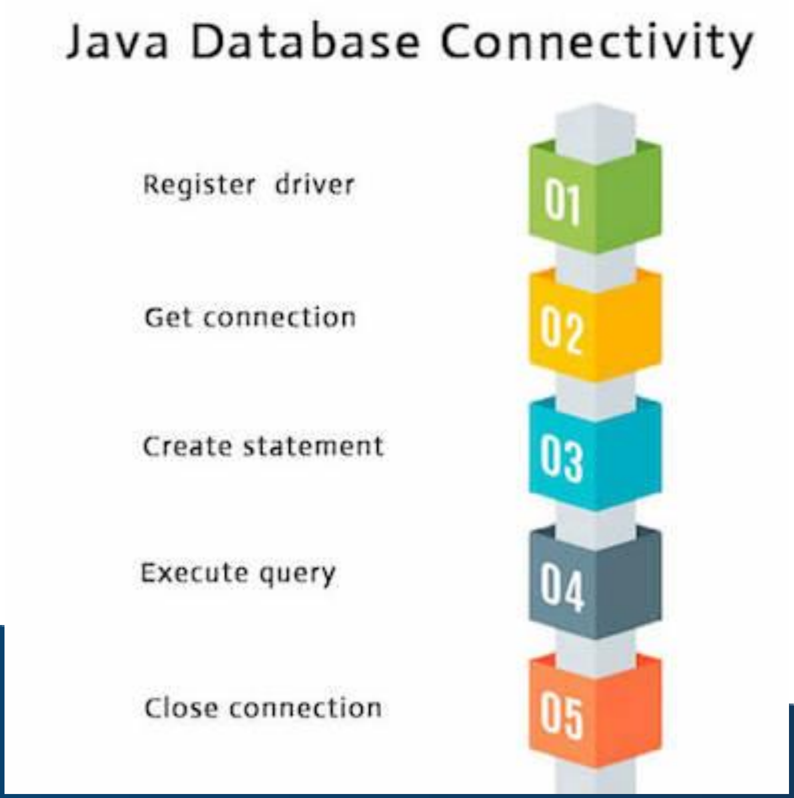


Figure- Thin Driver

Database Connectivity with 5 Steps

- There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:



1) Register the driver class

- The **forName()** method of Class class is used to register the driver class. This method is used to dynamically load the driver class.Syntax of forName() method
- **public static void** forName(String className)**throws** ClassNotFoundException
- **Driver class:** The driver class for the mysql database is **com.mysql.jdbc.Driver**.
- Class.forName("com.mysql.jdbc.Driver");

2) Create the connection object

- The **getConnection()** method of DriverManager class is used to establish connection with the database.
- Syntax of getConnection() method

1) **public static** Connection getConnection(String url)**throws** SQLException

2) **public static** Connection getConnection(String url,String name,String password) **throws** SQLExceptio n

- **Connection URL:** The connection URL for the mysql database is **jdbc:mysql://localhost:3306/god** where jdbc is the API, mysql is the database, localhost is the server name on which mysql is running, we may also use IP address, 3306 is the port number and god is the database name. We may use any database, in such case, we need to replace the god with our database name.
- Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/god","root","root");

3) Create the Statement object

- The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.Syntax of createStatement() method
- **public** Statement createStatement()**throws** SQLException
- Example to create the statement object
- Statement stmt=con.createStatement();

4) Execute the query

- The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.
- Syntax of executeQuery() method
- **public** ResultSet executeQuery(String sql)**throws** SQLException
- Example to execute query
- ResultSet rs=stmt.executeQuery("select * from emp");

```
  while(rs.next()){
System.out.println(rs.getInt(1)+" "+rs.getString(2) + “ “ + rs.getInt(3));
}
```

5) Close the connection object

- By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.Syntax of close() method
- **public void** close()**throws** SQLException
- Example to close connection
- con.close();

PRESIDENCY UNIVERSITY
Private University Estd. in Karnataka State by Act No. 41 of 2013

Java Database Connectivity with Oracle

- **Driver class:** The driver class for the oracle database is **oracle.jdbc.driver.OracleDriver**.
- **Connection URL:** The connection URL for the oracle10G database is **jdbc:oracle:thin:@localhost:1521:xe** where jdbc is the API, oracle is the database, thin is the driver, localhost is the server name on which oracle is running, we may also use IP address, 1521 is the port number and XE is the Oracle service name. You may get all these information from the tnsnames.ora file.
- **Username:** The default username for the oracle database is **system**.
- **Password:** It is the password given by the user at the time of installing the oracle database.

Example

Install MySQL
Create database god;
Use god;
create table emp(rno int,name varchar(30),age int);

Statement interface to insert, update and delete the record.

```
import java.sql.*;
class FetchRecord{
public static void main(String args[])throws Exception{
      Class.forName(" com.mysql.jdbc.Driver ");
      Connection con=DriverManager.getConnection("
   jdbc:mysql://localhost:3306/god","root","root");
        Statement stmt=con.createStatement();

       stmt.executeUpdate("insert into emp values(1,'Irfan',50000)");
       int result=stmt.executeUpdate("update emp set name='Vimal',salary=10000 where rno=1");
        int result=stmt.executeUpdate("delete from emp  where rno=1");
        System.out.println(result+" records affected");
con.close();
}
}
```

ResultSet

```
Class.forName(" com.mysql.jdbc.Driver ");
Connection con=DriverManager.getConnection(" jdbc:mysql://localhost:3306/god","root","root");
Statement stmt=con.createStatement();
ResultSet rs=stmt.executeQuery("select * from emp”);
  while(rs.next()) {
       System.out.println(rs.getInt(1)+" "+rs.getString(2)+" "+rs.getInt(3));
  }
con.close();
```

PreparedStatement

```
import java.sql.*;
class InsertPrepared{
public static void main(String args[]){
try{
 Class.forName(" com.mysql.jdbc.Driver ");
Connection con=DriverManager.getConnection(" jdbc:mysql://localhost:3306/god","root","root");

PreparedStatement stmt=con.prepareStatement("insert into emp values(?,?,?)");
stmt.setInt(1,1);//1 specifies the first parameter in the query
stmt.setString(2,"Ratan");
 stmt.setInt(3,100000);
int i=stmt.executeUpdate();
System.out.println(i+" records inserted");

con.close();

}catch(Exception e){ System.out.println(e);}

}
```

Contd..

```
PreparedStatement stmt=con.prepareStatement("delete from emp where  rno=?");
stmt.setInt(1,101);
  int i=stmt.executeUpdate();
System.out.println(i+" records deleted");
PreparedStatement stmt=con.prepareStatement("select * from emp");
ResultSet rs=stmt.executeQuery();
while(rs.next()){
System.out.println(rs.getInt(1)+" "+rs.getString(2) + “ “ + getInt(3));
}
```

ResultSetMetaData

```
import java.sql.*;
class Rsmd{
public static void main(String args[]){
try{
  Class.forName(" com.mysql.jdbc.Driver ");
Connection con=DriverManager.getConnection(" jdbc:mysql://localhost:3306/god","root","root");


PreparedStatement ps=con.prepareStatement("select * from emp");
ResultSet rs=ps.executeQuery();
ResultSetMetaData rsmd=rs.getMetaData();

System.out.println("Total columns: "+rsmd.getColumnCount());
System.out.println("Column Name of 1st column: "+rsmd.getColumnName(1));
System.out.println("Column Type Name of 1st column: "+rsmd.getColumnTypeName(1));


con.close();
}catch(Exception e){ System.out.println(e);}
}
```

Configuring

- To connect java application with the mysql database, **mysqlconnector.jar** file is required to be loaded.
- download the jar file mysql-connector.jar
- Two ways to load the jar file:
- Paste the mysqlconnector.jar file in jre/lib/ext folder
- Set classpath
- 1) Paste the mysqlconnector.jar file in JRE/lib/ext folder:
- Download the mysqlconnector.jar file. Go to jre/lib/ext folder and paste the jar file here.2) Set classpath:
- There are two ways to set the classpath:temporary
- permanent