Collections and Framework

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

What is a framework in Java?

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

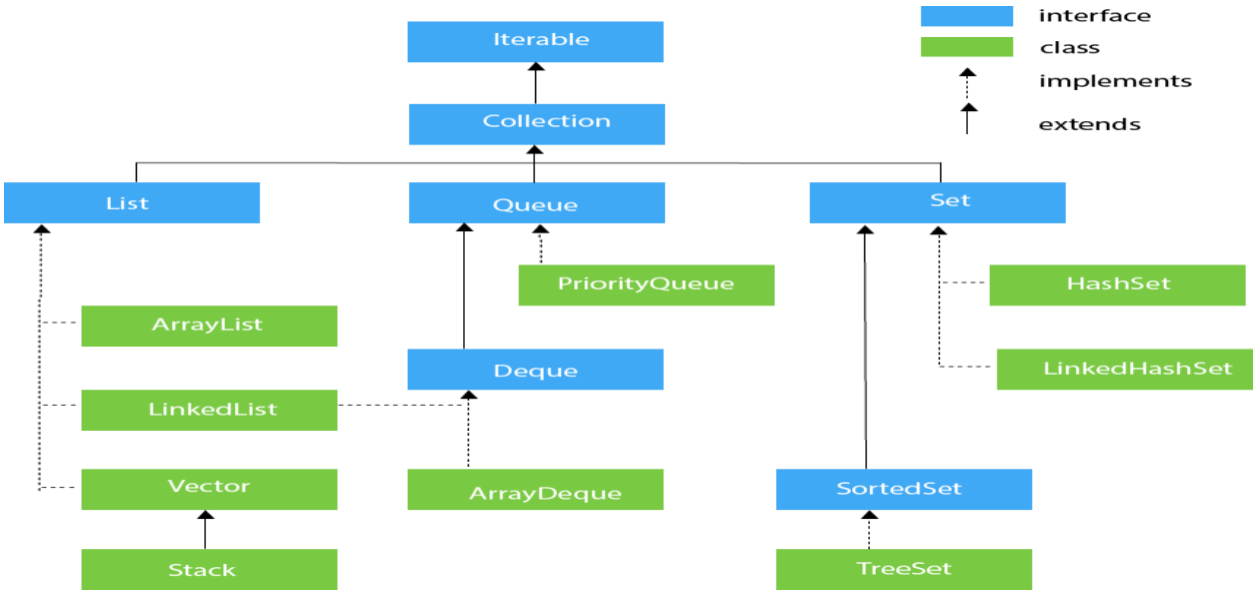
What is Collection framework?

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

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

Hierarchy of Collection Framework

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



Iterator interface

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

Methods of Iterator interface:

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.

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.

methods of Collection interface are

Boolean add (Object obj),

Boolean addAll (Collection c),

void clear(), etc. which are implemented by all the subclasses of Collection interface.

List Interface

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

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

To instantiate the List interface, we must use :

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

3. List <data-type> list3 = **new** Vector();
4. List <data-type> list4 = **new** Stack();

Java ArrayList

Java **ArrayList** class uses a *dynamic array* for storing the elements. It is like an array, but there is *no size limit*. We can add or remove elements anytime.

It implements the List interface so we can use all the methods of the List interface here. The ArrayList maintains the insertion order internally.

It inherits the AbstractList class and implements List interface.

The important points about the Java ArrayList class are:

Java ArrayList class maintains insertion order.

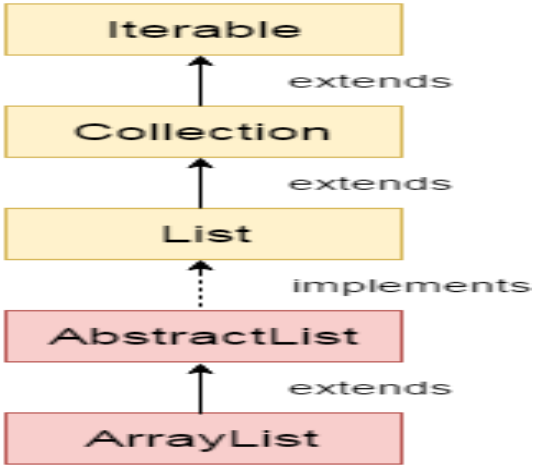
- Java ArrayList allows random access because the array works on an index basis.
- In ArrayList, manipulation is a little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.
- We can not create an array list of the primitive types, such as int, float, char, etc. It is required to use the required wrapper class in such cases. For example:

1. ArrayList<**int**> al = ArrayList<**int**>(); // does not work
2. ArrayList<Integer> al = **new** ArrayList<Integer>(); // works fine

- Java ArrayList gets initialized by the size. The size is dynamic in the array list, which varies according to the elements getting added or removed from the list.

Hierarchy of ArrayList class

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



ArrayList class declaration

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

Constructors of ArrayList

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

Methods of ArrayList

Method	Description
void add(int index, E element)	It is used to insert the specified element at the specified position in a list.

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

Java ArrayList Example1:

```
import java.util.*;
 public class ArrayListExample1{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();//Creating arraylist
     list.add("Mango");//Adding object in arraylist
     list.add("Apple");
     list.add("Banana");
     list.add("Grapes");
     //Printing the arraylist object
     System.out.println(list);
 }
}
```

Output:

```
[Mango, Apple, Banana, Grapes]
```

2.Iterating ArrayList using Iterator

```
import java.util.*;
public class ArrayListExample2{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();//Creating arraylist
  list.add("C");//Adding object in arraylist
```

```
  list.add("C++");
  list.add("PYTHON");
  list.add("JAVA");
  //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
  }
 }
}
```

Output:

C
C++
PYTHON
JAVA

3.Iterating ArrayList using For-each loop

```
import java.util.*;
public class ArrayListExample3{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();//Creating arraylist
  list.add("C");//Adding object in arraylist
  list.add("C++");
  list.add("PYTHON");
  list.add("JAVA");
  //Traversing list through for-each loop
  for(String fruit:list)
    System.out.println(fruit);

 }
}
```

Output:

C
C++
PYTHON
JAVA

4.Get and Set ArrayList

get() *method* returns the element at the specified index.

set() *method* changes the element.

```
1.  import java.util.*;
public class ArrayListExample4{
 public static void main(String args[]){
  ArrayList<String> al=new ArrayList<String>();
  al.add("Mango");
  al.add("Apple");
  al.add("Banana");
  al.add("Grapes");
  //accessing the element
  System.out.println("Returning element: "+al.get(1));//it will return the 2nd element, because index s
tarts from 0
  //changing the element
  al.set(1,"Dates");
  //Traversing list
  for(String fruit:al)
    System.out.println(fruit);

 }
}
```

Output

Returning element: Apple

Mango

Dates

Banana

2. Grapes

How to Sort ArrayList

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

```
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

  Collections.sort(list2);

   //Traversing list through the for-each loop
```

```
  for(Integer number:list2)
    System.out.println(number);
  }

}
```

Output:

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

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
Iterator itr=al.iterator();
//traversing elements of ArrayList object
while(itr.hasNext()){
  Student st=(Student)itr.next();
  System.out.println(st.rollno+" "+st.name+" "+st.age);
 }
}
}
```

Output:

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

Java LinkedList class

Java LinkedList class uses a doubly linked list to store the elements. It provides a linked-list data structure.

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.

LinkedList class declaration

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

Constructors of Java LinkedList

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

Methods of Java LinkedList

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

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

Java LinkedList Example1:

```
import java.util.*;
        public class LinkedList1{
         public static void main(String args[]){

          LinkedList<String> al=new LinkedList<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());
      }
     }
    }
```

Output:

```
Ravi
 Vijay
     Ravi
     Ajay
```

Java LinkedList Example 2: 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 LinkedListExample {
public static void main(String[] args) {
   //Creating list of Books
   List<Book> list=new LinkedList<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);
    //Adding Books to list
    list.add(b1);
    list.add(b2);
    list.add(b3);
    //Traversing list
    for(Book b:list){
    System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
    }
}
}
```

Output:

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

Difference Between ArrayList and LinkedList

ArrayList	LinkedList
1) ArrayList internally uses a **dynamic array** to store the elements.	LinkedList internally uses a **doubly linked list** to store the elements.
2) Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the other elements are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.

3) An ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
4) ArrayList is **better for storing and accessing** data.	LinkedList is **better for manipulating** data.
5) The memory location for the elements of an ArrayList is contiguous.	The location for the elements of a linked list is not contagious.
6) Generally, when an ArrayList is initialized, a default capacity of 10 is assigned to the ArrayList.	There is no case of default capacity in a LinkedList. In LinkedList, an empty list is created when a LinkedList is initialized.
7) To be precise, an ArrayList is a resizable array.	LinkedList implements the doubly linked list of the list interface.

Example of ArrayList and LinkedList in Java

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

  List<String> al=new ArrayList<String>();//creating arraylist
  al.add("Ravi");//adding object in arraylist
  al.add("Vijay");
  al.add("Ravi");
  al.add("Ajay");

  List<String> al2=new LinkedList<String>();//creating linkedlist
  al2.add("James");//adding object in linkedlist
  al2.add("Serena");
  al2.add("Swati");
  al2.add("Junaid");
```

```
  System.out.println("arraylist: "+al);
  System.out.println("linkedlist: "+al2);
 }
}
```

Note:

The following are some important points to remember regarding an ArrayList and LinkedList.

- When the rate of addition or removal rate is more than the read scenarios, then go for the LinkedList. On the other hand, when the frequency of the read scenarios is more than the addition or removal rate, then ArrayList takes precedence over LinkedList.
- Since the elements of an ArrayList are stored more compact as compared to a LinkedList; therefore, the ArrayList is more cache-friendly as compared to the LinkedList. Thus, chances for the cache miss are less in an ArrayList as compared to a LinkedList. Generally, it is considered that a LinkedList is poor in cache-locality.
- Memory overhead in the LinkedList is more as compared to the ArrayList. It is because, in a LinkedList, we have two extra links (next and previous) as it is required to store the address of the previous and the next nodes, and these links consume extra space. Such links are not present in an ArrayList.

Java List

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

The List interface is found in the **java.util** package and inherits the Collection interface.

List Interface declaration

public interface List<E> **extends** Collection<E>

Java List Methods

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

List Interface declaration

public interface List<E> **extends** Collection<E>

Java List Methods

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

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

Java List Example1

```
import java.util.*;
public class ListExample1{
public static void main(String args[]){
 //Creating a List
 List<String> list=new ArrayList<String>();
 //Adding elements in the List
 list.add("Mango");
 list.add("Apple");
 list.add("Banana");
 list.add("Grapes");
 //Iterating the List element using for-each loop
 for(String fruit:list)
  System.out.println(fruit);

}
}
```

How to convert Array to List?

We can convert the Array to List by traversing the array and adding the element in list one by one using list.add() method.

```
import java.util.*;
public class ArrayToListExample{
public static void main(String args[]){
//Creating Array
```

```
String[] array={"Java","Python","PHP","C++"};
System.out.println("Printing Array: "+Arrays.toString(array));
//Converting Array to List
List<String> list=new ArrayList<String>();
for(String lang:array){
list.add(lang);
}
System.out.println("Printing List: "+list);

}
}
```

Output:

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

How to convert List to Array?

```
import java.util.*;
public class ListToArrayExample{
public static void main(String args[]){
 List<String> fruitList = new ArrayList<>();
 fruitList.add("Mango");
 fruitList.add("Banana");
 fruitList.add("Apple");
 fruitList.add("Strawberry");
 //Converting ArrayList to Array
 String[] array = fruitList.toArray(new String[fruitList.size()]);
 System.out.println("Printing Array: "+Arrays.toString(array));
 System.out.println("Printing List: "+fruitList);
}
}
```

Output:

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

How to Sort List

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

```
    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
```

```
        Collections.sort(list2);
         //Traversing list through the for-each loop
        for(Integer number:list2)
          System.out.println(number);
       }

      }
```

Output:

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

Java ListIterator Interface

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

ListIterator Interface declaration

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

Methods of Java ListIterator Interface:

Method	Description
void add(E e)	This method inserts the specified element into the list.

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

Example of ListIterator Interface

```
import java.util.*;
public class ListIteratorExample1{
public static void main(String args[]){
List<String> al=new ArrayList<String>();
        al.add("Amit");
        al.add("Vijay");
        al.add("Kumar");
        al.add(1,"Sachin");
        ListIterator<String> itr=al.listIterator();
        System.out.println("Traversing elements in forward direction");
```

```
    while(itr.hasNext()){

    System.out.println("index:"+itr.nextIndex()+" value:"+itr.next());
    }
    System.out.println("Traversing elements in backward direction");
    while(itr.hasPrevious()){

    System.out.println("index:"+itr.previousIndex()+" value:"+itr.previous());
    }
}
}
```

Output:

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

Example of List: Book

```
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 ListExample5 {
public static void main(String[] args) {
    //Creating list of Books
    List<Book> list=new ArrayList<Book>();
    //Creating Books
    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
    Book b2=new Book(102,"Data Communications and Networking","Forouzan","Mc Graw Hill",4)
;
    Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
    //Adding Books to list
    list.add(b1);
    list.add(b2);
    list.add(b3);
    //Traversing list
    for(Book b:list){
    System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
    }
}
}
```

Output:

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

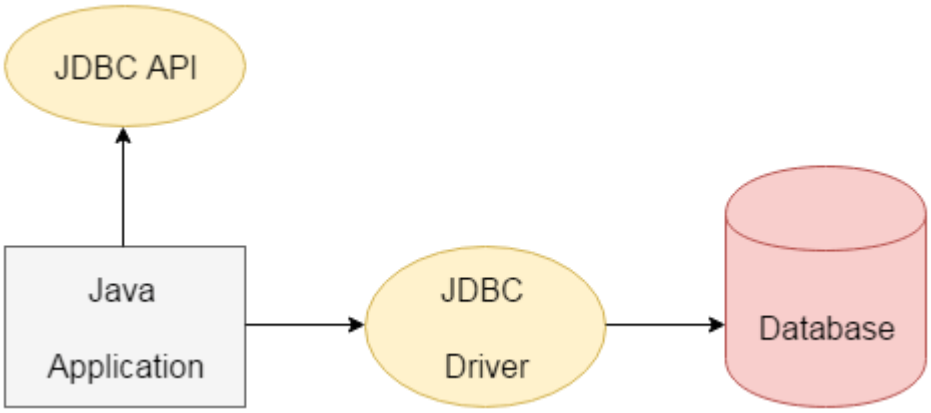
JDBC

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

We can 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) provided by Microsoft.



The **java.sql** package contains classes and interfaces for JDBC API. A list of popular *interfaces* of JDBC API are given below:

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

A list of popular *classes* of JDBC API are given below:

- DriverManager class
- Blob class
- Clob class
- Types class

Why Should We Use JDBC?

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.

What is 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.

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)

1) 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.

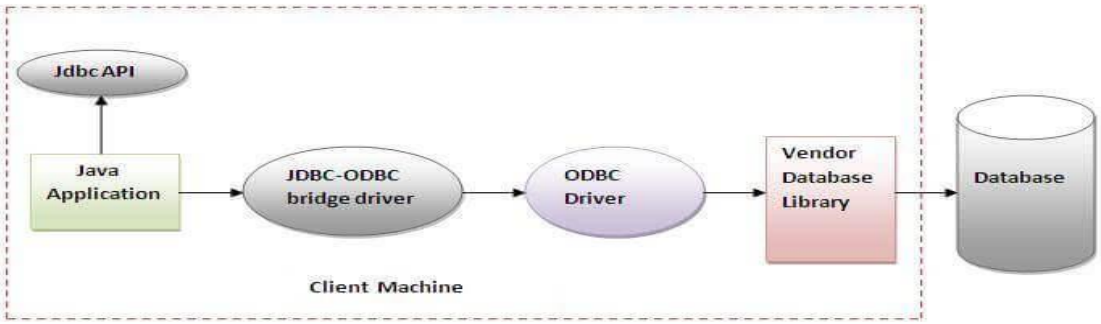


Figure- JDBC-ODBC Bridge Driver

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.

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 method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

2) 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.

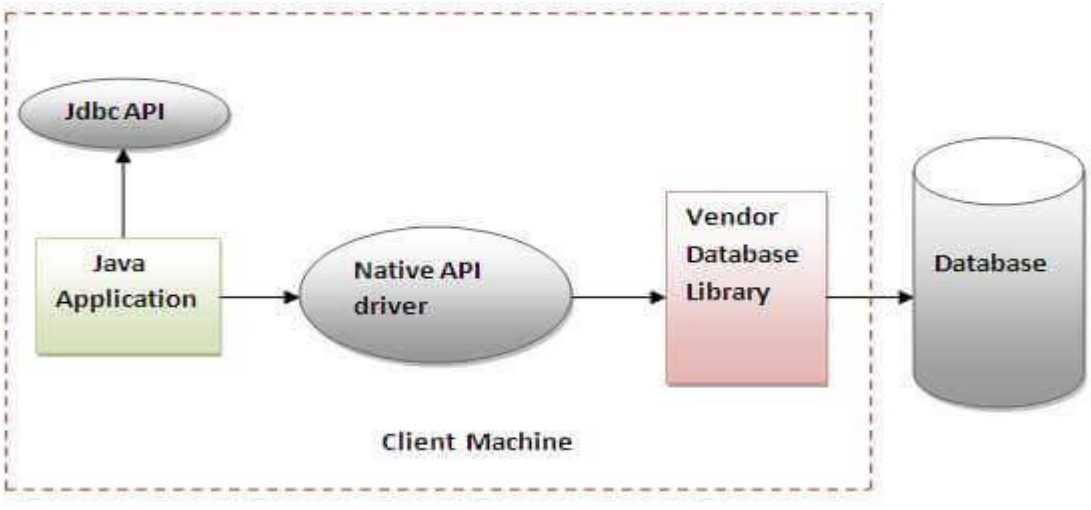


Figure- Native API Driver

Advantage:

- performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

3) 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.

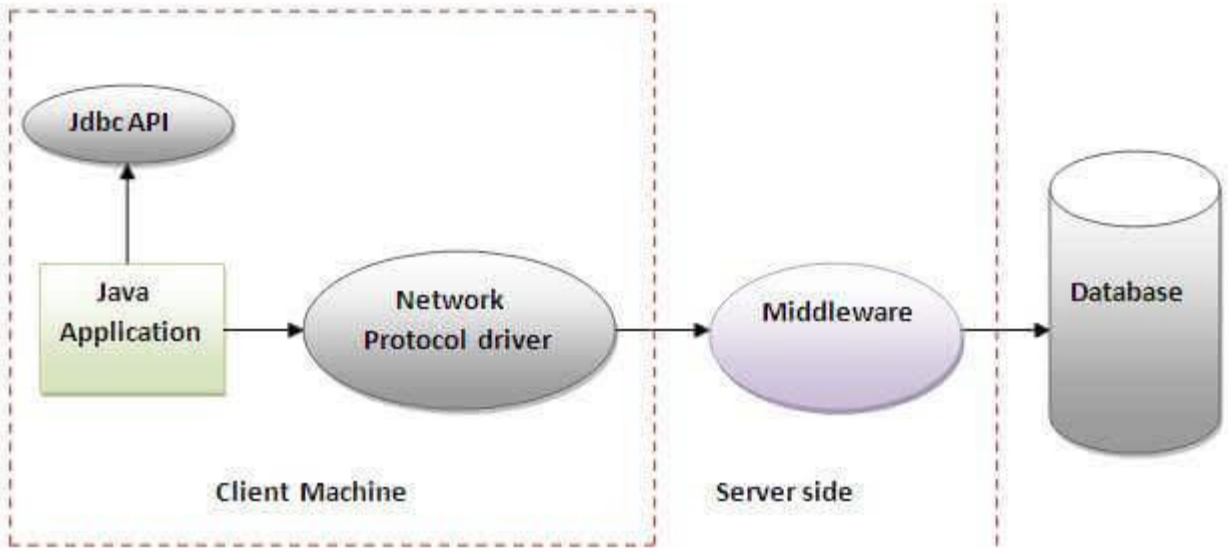


Figure- Network Protocol Driver

Advantage:

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

4) 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.

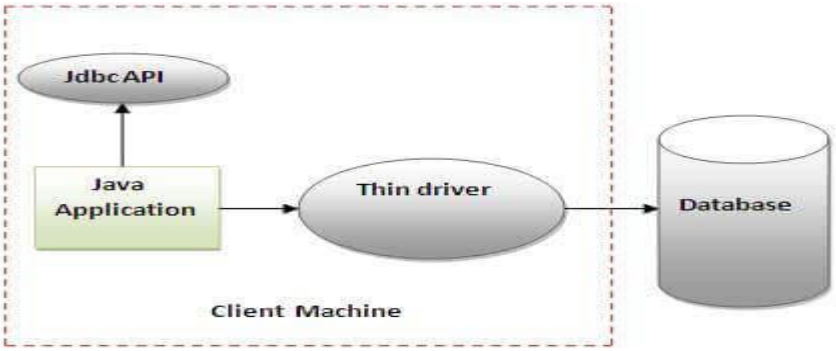


Figure- Thin Driver

Advantage:

- Better performance than all other drivers.
- No software is required at client side or server side.

Disadvantage:

- Drivers depend on the Database.

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

- Register the Driver class
- Create connection
- Create statement
- Execute queries
- Close connection

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

Example to register the OracleDriver class

.Class.forName("oracle.jdbc.driver.OracleDriver");

2) Create the connection object

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

Syntax of getConnection() method

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

public static Connection getConnection(String url,String name,String password) **throws** SQLE xception

Example to establish connection with the Oracle database

Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe", "system","password");

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));
}
```

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();
```

Java Database Connectivity with Oracle

To connect java application with the oracle database, we need to follow 5 following steps. In this example, we are using Oracle 10g as the database.

1. **Driver class:** The driver class for the oracle database is **oracle.jdbc.driver.OracleDriver**.

2. **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.
3. **Username:** The default username for the oracle database is **system**.
4. **Password:** It is the password given by the user at the time of installing the oracle database.

Create a Table

Before establishing connection, create a table in oracle database. Following is the SQL query to create a table.

```
create table emp(id number(10),name varchar2(40),age number(3));
```

Example to Connect Java Application with Oracle database

In this example, we are connecting to an Oracle database and getting data from **emp** table. Here, **system** and **oracle** are the username and password of the Oracle database.

```
import java.sql.*;
class OracleCon{
public static void main(String args[]){
try{
//step1 load the driver class
Class.forName("oracle.jdbc.driver.OracleDriver");

//step2 create  the connection object
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

//step3 create the statement object
Statement stmt=con.createStatement();
```

```
//step4 execute query
ResultSet rs=stmt.executeQuery("select * from emp");
while(rs.next())
System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));

//step5 close the connection object
con.close();

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

}
}
```

Java Database Connectivity with MySQL

To connect Java application with the MySQL database, we need to follow 5 following steps.

1. **Driver class:** The driver class for the mysql database is **com.mysql.jdbc.Driver**.
2. **Connection URL:** The connection URL for the mysql database is **jdbc:mysql://localhost:3306/sonoo** 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 sonoo is the database name. We may use any database, in such case, we need to replace the sonoo with our database name.
3. **Username:** The default username for the mysql database is **root**.
4. **Password:** It is the password given by the user at the time of installing the mysql database. In this example, we are going to use root as the password.

create a table in the mysql database, but before creating table, we need to create database first.

```
create database sonoo;
use sonoo;
create table emp(id int(10),name varchar(40),age int(3));
```

```
import java.sql.*;
class MysqlCon{
public static void main(String args[]){
try{
Class.forName("com.mysql.jdbc.Driver");
Connection con=DriverManager.getConnection(
"jdbc:mysql://localhost:3306/sonoo","root","root");
//here sonoo is database name, root is username and password
Statement stmt=con.createStatement();
ResultSet rs=stmt.executeQuery("select * from emp");
while(rs.next())
System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
con.close();
}catch(Exception e){ System.out.println(e);}
}
}
```

DriverManager class

The DriverManager class is the component of JDBC API and also a member of the *java.sql* package. The DriverManager class acts as an interface between users and drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver. It contains all the appropriate methods to register and deregister the database driver class and to create a connection between a Java application and the database.

Methods of the DriverManager Class

Method	Description

1) public static synchronized void registerDriver(Driver driver):	is used to register the given driver with DriverManager. No action is performed by the method when the given driver is already registered.
2) public static synchronized void deregisterDriver(Driver driver):	is used to deregister the given driver (drop the driver from the list) with DriverManager. If the given driver has been removed from the list, then no action is performed by the method.
3) public static Connection getConnection(String url) throws SQLException:	is used to establish the connection with the specified url. The SQLException is thrown when the corresponding Driver class of the given database is not registered with the DriverManager.
4) public static Connection getConnection(String url,String userName,String password) throws SQLException:	is used to establish the connection with the specified url, username, and password. The SQLException is thrown when the corresponding Driver class of the given database is not registered with the DriverManager.
5) public static Driver getDriver(String url)	Those drivers that understand the mentioned URL (present in the parameter of the method) are returned by this method provided those drivers are mentioned in the list of registered drivers.
6) pubic static int getLoginTimeout()	The duration of time a driver is allowed to wait in order to establish a connection with the database is returned by this method.
7) pubic static void setLoginTimeout(int sec)	The method provides the time in seconds. sec mentioned in the parameter is the maximum time that a driver is allowed to wait in order to establish a

	connection with the database. If 0 is passed in the parameter of this method, the driver will have to wait infinitely while trying to establish the connection with the database.
8) public static Connection getConnection(String URL, Properties prop) throws SQLException	A connection object is returned by this method after creating a connection to the database present at the mentioned URL, which is the first parameter of this method. The second parameter, which is "prop", fetches the authentication details of the database (username and password.). Similar to the other variation of the getConnection() method, this method also throws the SQLException, when the corresponding Driver class of the given database is not registered with the DriverManager.

Connection interface

A Connection is a session between a Java application and a database. It helps to establish a connection with the database.

The Connection interface is a factory of Statement, PreparedStatement, and DatabaseMetaData, i.e., an object of Connection can be used to get the object of Statement and DatabaseMetaData.

Commonly used methods of Connection interface:

1) public Statement createStatement(): creates a statement object that can be used to execute SQL queries.

2) public Statement createStatement(int resultSetType,int resultSetConcurrency): Creates a Statement object that will generate ResultSet objects with the given type and concurrency.

3) public void setAutoCommit(boolean status): is used to set the commit status. By default, it is true.

4) public void commit(): saves the changes made since the previous commit/rollback is permanent.

5) public void rollback(): Drops all changes made since the previous commit/rollback.

6) public void close(): closes the connection and Releases a JDBC resources immediately.

Statement interface

The **Statement interface** provides methods to execute queries with the database. The statement interface is a factory of ResultSet i.e. it provides factory method to get the object of ResultSet.

Commonly used methods of Statement interface:

The important methods of Statement interface are as follows:

1) public ResultSet executeQuery(String sql): is used to execute SELECT query. It returns the object of ResultSet.

2) public int executeUpdate(String sql): is used to execute specified query, it may be create, drop, insert, update, delete etc.

3) public boolean execute(String sql): is used to execute queries that may return multiple results.

4) public int[] executeBatch(): is used to execute batch of commands.

Example of Statement interface

```
import java.sql.*;
class FetchRecord{
```

```
public static void main(String args[])throws Exception{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","or
acle");
Statement stmt=con.createStatement();

//stmt.executeUpdate("insert into emp765 values(33,'Irfan',50000)");
//int result=stmt.executeUpdate("update emp765 set name='Vimal',salary=10000 where id=33");
int result=stmt.executeUpdate("delete from emp765 where id=33");
System.out.println(result+" records affected");
con.close();
}}
```

ResultSet interface

The object of ResultSet maintains a cursor pointing to a row of a table. Initially, cursor points to before the first row.

Commonly used methods of ResultSet interface

1) public boolean next():	is used to move the cursor to the one row next from the current position.
2) public boolean previous():	is used to move the cursor to the one row previous from the current position.
3) public boolean first():	is used to move the cursor to the first row in result set object.
4) public boolean last():	is used to move the cursor to the last row in result set object.
5) public boolean absolute(int row):	is used to move the cursor to the specified row number in the ResultSet object.

6) public boolean relative(int row):	is used to move the cursor to the relative row number in the ResultSet object, it may be positive or negative.
7) public int getInt(int columnIndex):	is used to return the data of specified column index of the current row as int.
8) public int getInt(String columnName):	is used to return the data of specified column name of the current row as int.
9) public String getString(int columnIndex):	is used to return the data of specified column index of the current row as String.
10) public String getString(String columnName):	is used to return the data of specified column name of the current row as String.

PreparedStatement interface

The PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized query.

Methods of PreparedStatement interface

The important methods of PreparedStatement interface are given below:

Method	Description
public void setInt(int paramIndex, int value)	sets the integer value to the given parameter index.
public void setString(int paramIndex, String value)	sets the String value to the given parameter index.

public void setFloat(int paramIndex, float value)	sets the float value to the given parameter index.
public void setDouble(int paramIndex, double value)	sets the double value to the given parameter index.
public int executeUpdate()	executes the query. It is used for create, drop, insert, update, delete etc.
public ResultSet executeQuery()	executes the select query. It returns an instance of ResultSet.

Java ResultSetMetaData Interface

The metadata means data about data i.e. we can get further information from the data.

If you have to get metadata of a table like total number of column, column name, column type etc. , ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.

Commonly used methods of ResultSetMetaData interface

Method	Description
public int getColumnCount()throws SQLException	it returns the total number of columns in the ResultSet object.
public String getColumnName(int index)throws SQLException	it returns the column name of the specified column index.
public String getColumnTypeName(int index)throws SQLException	it returns the column type name for the specified index.

public String getTableName(int index)throws SQLException	it returns the table name for the specified column index.

Syntax

public ResultSetMetaData getMetaData()**throws** SQLException

Transaction Management in JDBC

Transaction represents **a single unit of work**.

The ACID properties describes the transaction management well. ACID stands for Atomicity, Consistency, isolation and durability.

Atomicity means either all successful or none.

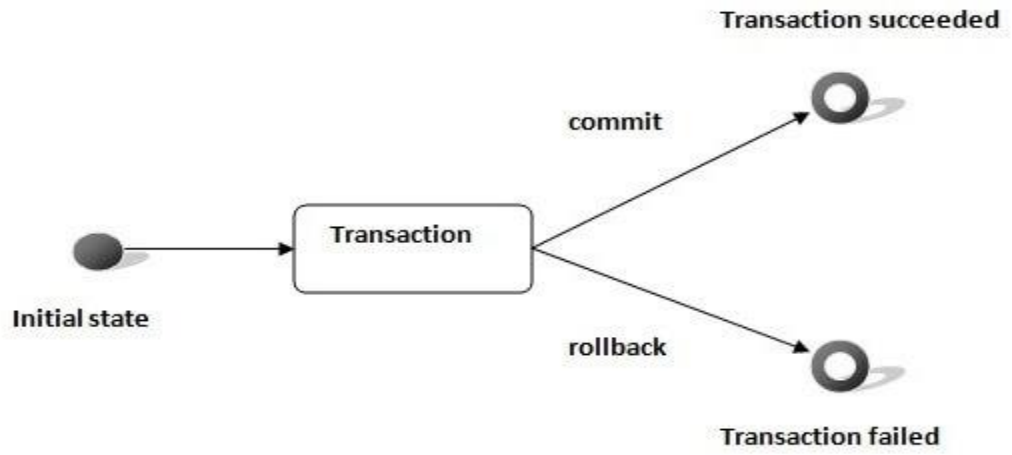
Consistency ensures bringing the database from one consistent state to another consistent state.

Isolation ensures that transaction is isolated from other transaction.

Durability means once a transaction has been committed, it will remain so, even in the event of errors, power loss etc.

Advantage of Transaction Mangaement

fast performance It makes the performance fast because database is hit at the time of commit.



In JDBC, **Connection interface** provides methods to manage transaction.

Method	Description
void setAutoCommit(boolean status)	It is true bydefault means each transaction is committed bydefault.
void commit()	commits the transaction.
void rollback()	cancels the transaction.

Simple example of transaction management in jdbc using Statement

```
import java.sql.*;
class FetchRecords{
public static void main(String args[])throws Exception{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","or
acle");
con.setAutoCommit(false);
```

```
Statement stmt=con.createStatement();
stmt.executeUpdate("insert into user420 values(190,'abhi',40000)");
stmt.executeUpdate("insert into user420 values(191,'umesh',50000)");

con.commit();
con.close();
}}
```