US05CBCA25

Object Oriented Programming – III

UNIT - I

Introduction

1. Database
   - Collection of data
2. DBMS
   - Database Management System
   - Storing and organizing data
3. SQL
   - Relational database
   - Structured Query Language
4. JDBC
   - Java Database Connectivity
   - JDBC driver

1.1 Introduction to JDBC

- JDBC (Java Database Connectivity) is a java API which enables the java programs to execute SQL statements.
- It is an application programming interface that defines how a java programmer can access the database in tabular format from Java code using a set of standard interfaces and classes written in the Java programming language.
- The Java application programming interface provides a mechanism for dynamically loading the correct Java packages and drivers and registering them with the JDBC Driver Manager that is used as a connection factory for creating JDBC connections which supports creating and executing statements such as SQL INSERT, UPDATE and DELETE.

Introduction to JDBC

- In short JDBC helps the programmers to write java applications that manage these three programming activities:

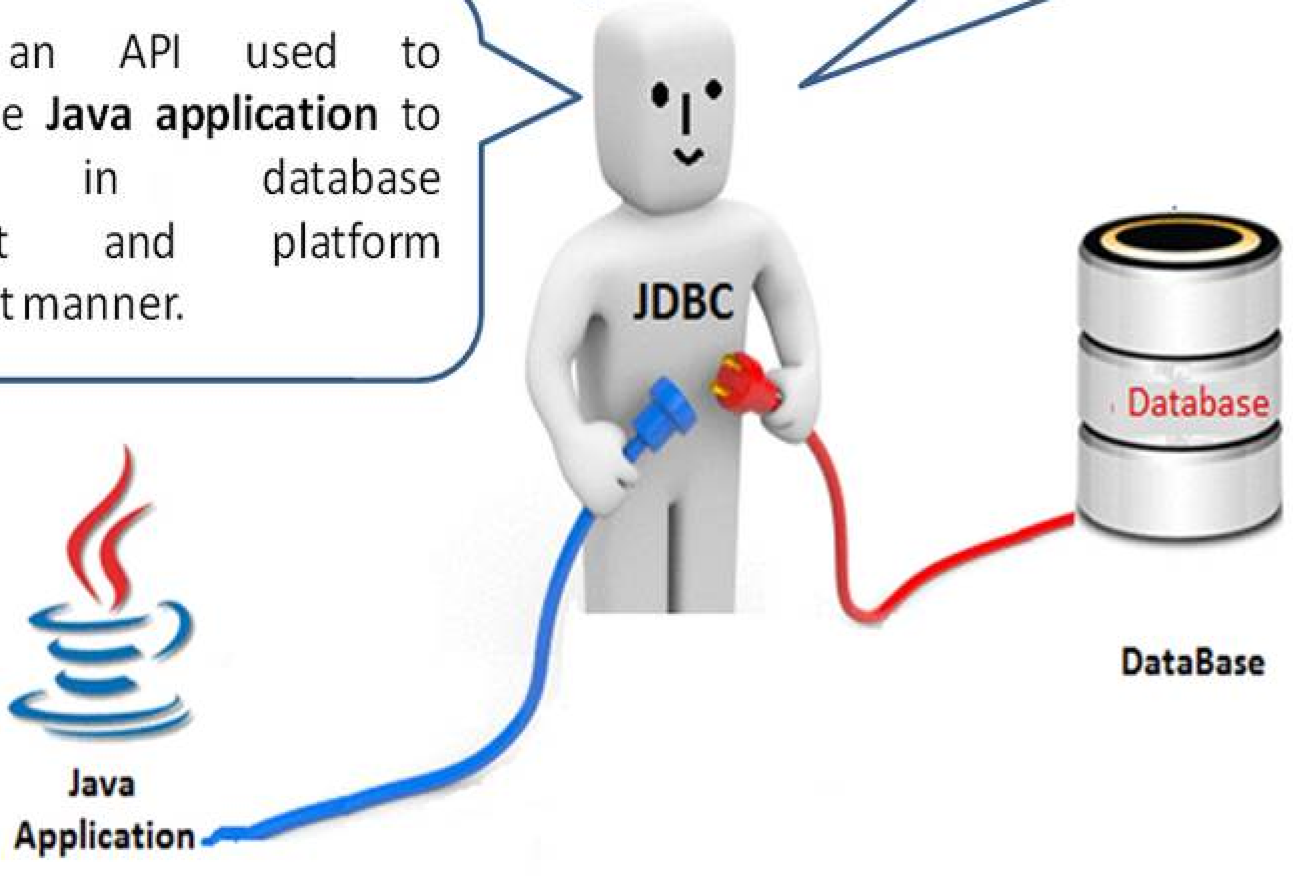
1. It helps us to connect to a data source, like a database.
2. It helps us in sending queries and updating statements to the database
3. Retrieving and processing the results received from the database in terms of answering query.

Introduction: JDBC

JDBC (Java Database Connectivity) is used to connect java application with database.

It provides **classes** and **interfaces** to connect or communicate Java application with database.

JDBC is an API used to communicate **Java application** to **database** in database independent and platform independent manner.



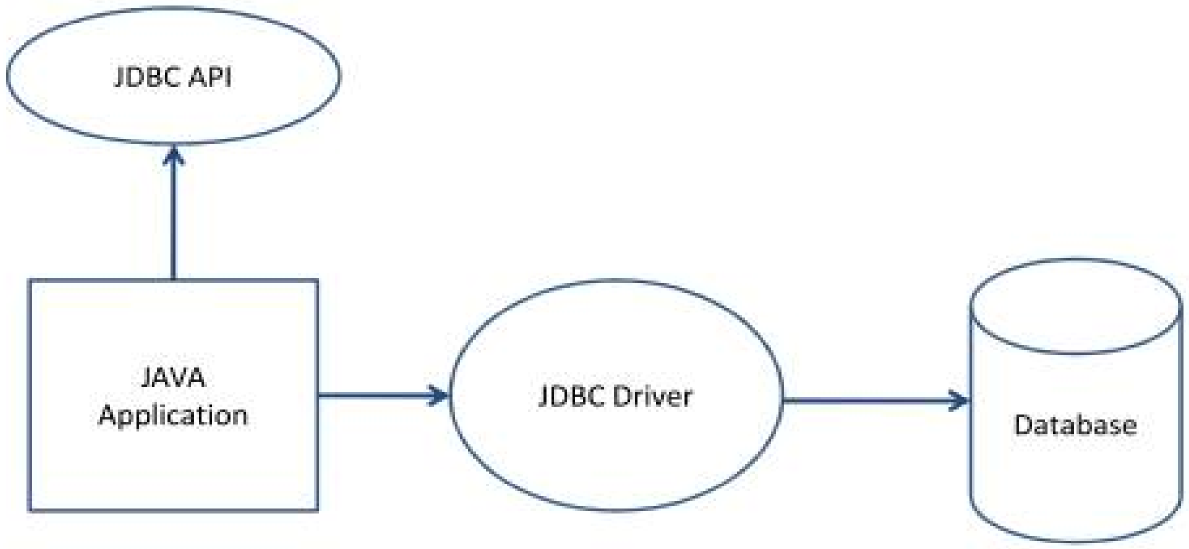
Example
Oracle
MS Access
My SQL
SQL Server
..
.

What is API ?

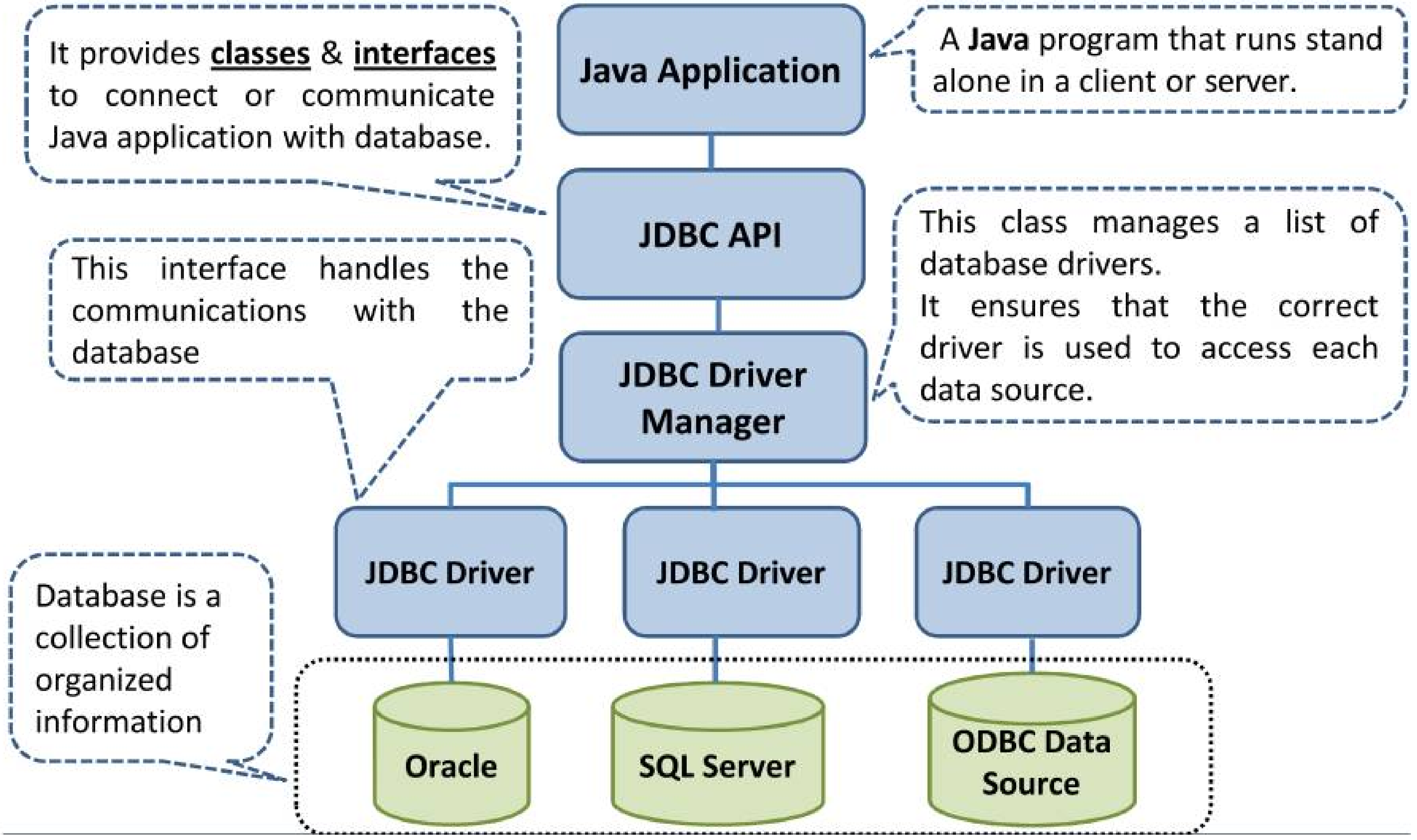
- Application Program Interface
- A set of routines, protocols, and tools for building software applications.
- JDBC is an API, which is used in java programming for interacting with database.

JDBC Components

1. The JDBC API: The JDBC API provides programmatic access to relational data from the Java programming language.
2. JDBC Driver Manager: The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver.
3. JDBC Test Suite — The JDBC driver test suite helps you to determine that JDBC drivers will run your program
4. JDBC-ODBC Bridge — The Java Software bridge provides JDBC access via ODBC drivers.

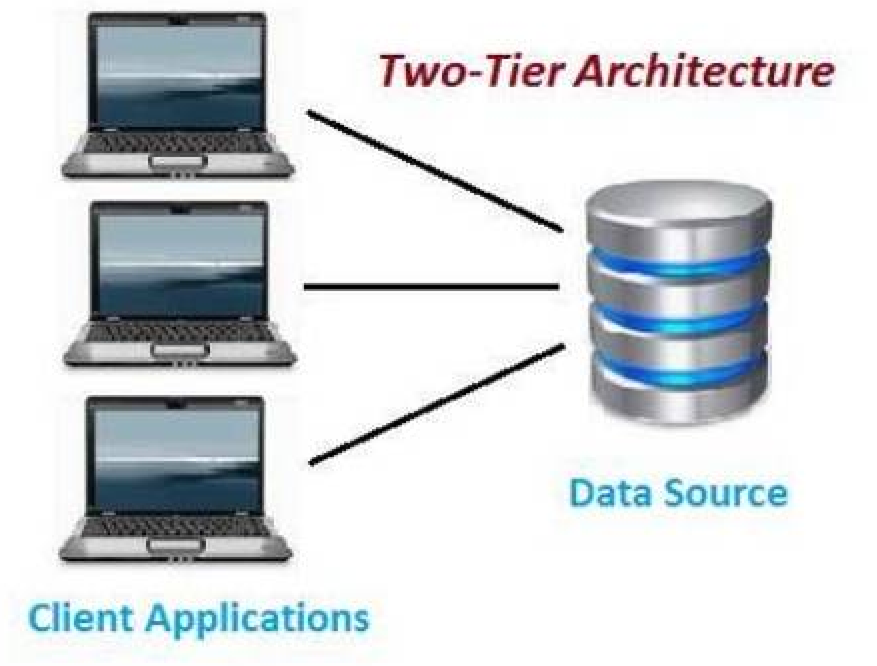


1.2 JDBC Architecture


It provides **classes** & **interfaces** to connect or communicate Java application with database.
Java Application
A **Java** program that runs stand alone in a client or server.
JDBC API
This class manages a list of database drivers.
It ensures that the correct driver is used to access each data source.
This interface handles the communications with the database
JDBC Driver Manager
JDBC Driver
JDBC Driver
JDBC Driver
Database is a collection of organized information
Oracle
SQL Server
ODBC Data Source

JDBC two-tier Architecture

- It is client-server architecture
- Direct communication
- Run faster(tight coupled)



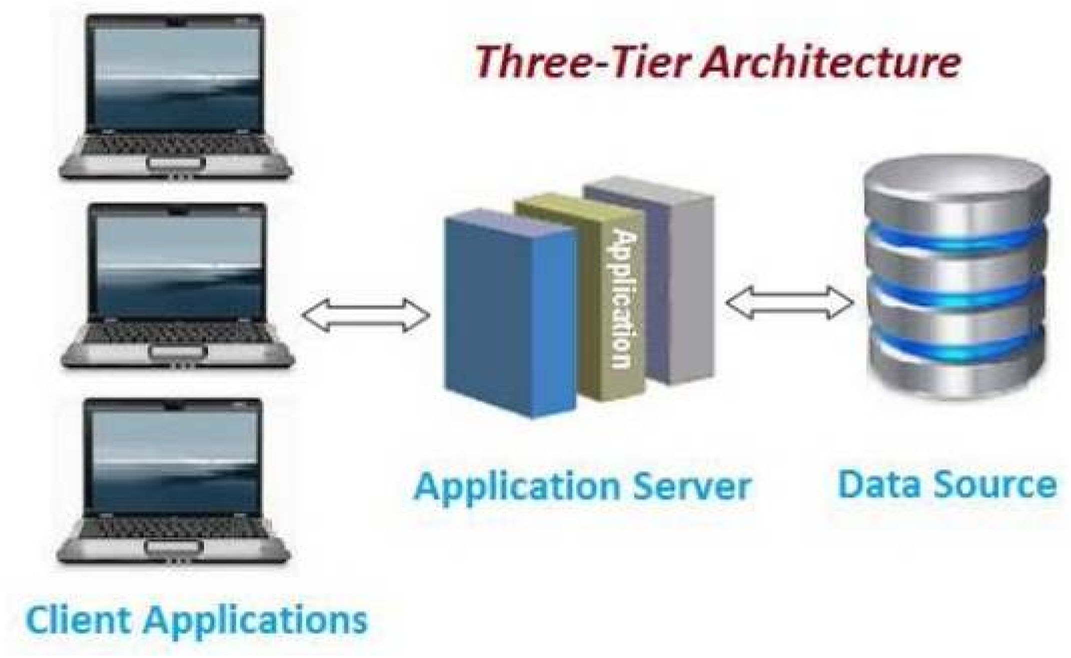
JDBC three-tier Architecture

- **Web based application**
- **Three layers:**

  1) Client layer

  2) Business layer

  3) Data layer



Types of JDBC Driver

- Type1 (JDBC-ODBC Driver)
- Type 2 (Native Code Driver)
- Type 3 (Java Protocol)
- Type 4 (Database Protocol)

JDBC Driver: Type 1 (JDBC-ODBC Driver)

Depends on support for ODBC

- Not portable
- Translate JDBC calls into ODBC calls and use Windows ODBC built in Drivers.
- ODBC must be set up on every client.
  - for server side servlets ODBC must be set up on web server
- driver sun.jdbc.odbc.JdbcOdbc provided by JavaSoft with JDK
- No support from JDK 1.8 (Java 8)
- E.g. MS Access

JDBC Driver: Type 1 (JDBC-ODBC Driver)

Depends on support for ODBC

- Not portable
- Translate JDBC calls into ODBC calls and use Windows ODBC built in Drivers
- ODBC must be set up on every client
  - for server side servlets ODBC must be set up on web server
- driver sun.jdbc.odbc.JdbcOdbc provided by JavaSoft with JDK
- No support from JDK 1.8 (Java 8)
- E.g. MS Access

JDBC Driver: Type 1 (JDBC-ODBC Driver)

Advantages :

- Allow to communicate with all database supported by ODBC driver
- It is vendor independent driver

Disadvantages:

- Due to large number of translations, execution speed is decreased
- Dependent on the ODBC driver
- ODBC binary code or ODBC client library to be installed in every client machine.
- Uses java native interface to make ODBC call

Type1 driver is not used in production environment. It can only be used, when database doesn’t have any other JDBC driver implementation.

JDBC Driver: Type 2 (Native Code Driver)

- JDBC API calls are converted into native API calls, which are unique to the database.
- These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge.
- Native code Driver are usually written in C, C++.
- The vendor-specific driver must be installed on each client machine.
- Type 2 Driver is suitable to use with server side applications.
- E.g. Oracle OCI driver, Weblogic OCI driver, Type2 for Sybase

JDBC Driver: Type 2 (Native Code Driver)

- ***<u>Advantages</u>***
  - As there is no implementation of JDBC-ODBC bridge, it may be considerably faster than a Type 1 driver.
- ***<u>Disadvantages</u>***
  - The vendor client library needs to be installed on the client machine.
  - This driver is platform dependent.
  - This driver supports all java applications except applets.
  - It may increase cost of application, if it needs to run on different platform(since we may require buying the native libraries for all of the platform).

JDBC Driver: Type 3 (Java Protocol)

- Pure Java Driver
- Depends on Middleware server
- Can interface to multiple databases – Not vendor specific.
- Follows a three-tier communication approach.
- The JDBC clients use standard network sockets to communicate with a middleware application server.
- The socket information is then translated by the middleware application server into the call format required by the DBMS.
- This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.

JDBC Driver: Type 3 (Java Protocol)

- ***Advantages***
- Since the communication between client and the middleware server is database independent, there is no need for the database vendor library on the client.
- A single driver can handle any database, provided the middleware supports it.
- We can switch from one database to other without changing the client-side driver class, by just changing configurations of middleware server.
- E.g.: IDS Driver, Weblogic RMI Driver
- ***Disadvantages***
- Compared to Type 2 drivers, Type 3 drivers are slow due to increased number of network calls.
- Requires database-specific coding to be done in the middle tier.
- The middleware layer added may result in additional latency, but is typically overcome by using better middleware services.

JDBC Driver: Type 4 (Database Protocol)

- It is known as the Direct to Database Pure Java Driver
- Need to download a newdriver for each database engine e.g. Oracle, MySQL
- Type 4 driver, a pure Java-based driver communicates directly with the vendor’s database through socket connection.
- This kind of driver is extremely flexible, you don’t need to install special software on the client or server.
- Such drivers are implemented by DBMS vendors.

JDBC Driver: Type 4 (Database Protocol)

Advantages

- Completely implemented in Java to achieve platform independence.
- No native libraries are required to be installed in client machine.
- These drivers don’t translate the requests into an intermediary format (such as ODBC).
- Secure to use since, it uses database server specific protocol.
- The client application connects directly to the database server.
- No translation or middleware layers are used, improving performance.
- The JVM manages all the aspects of the application-to-database connection.

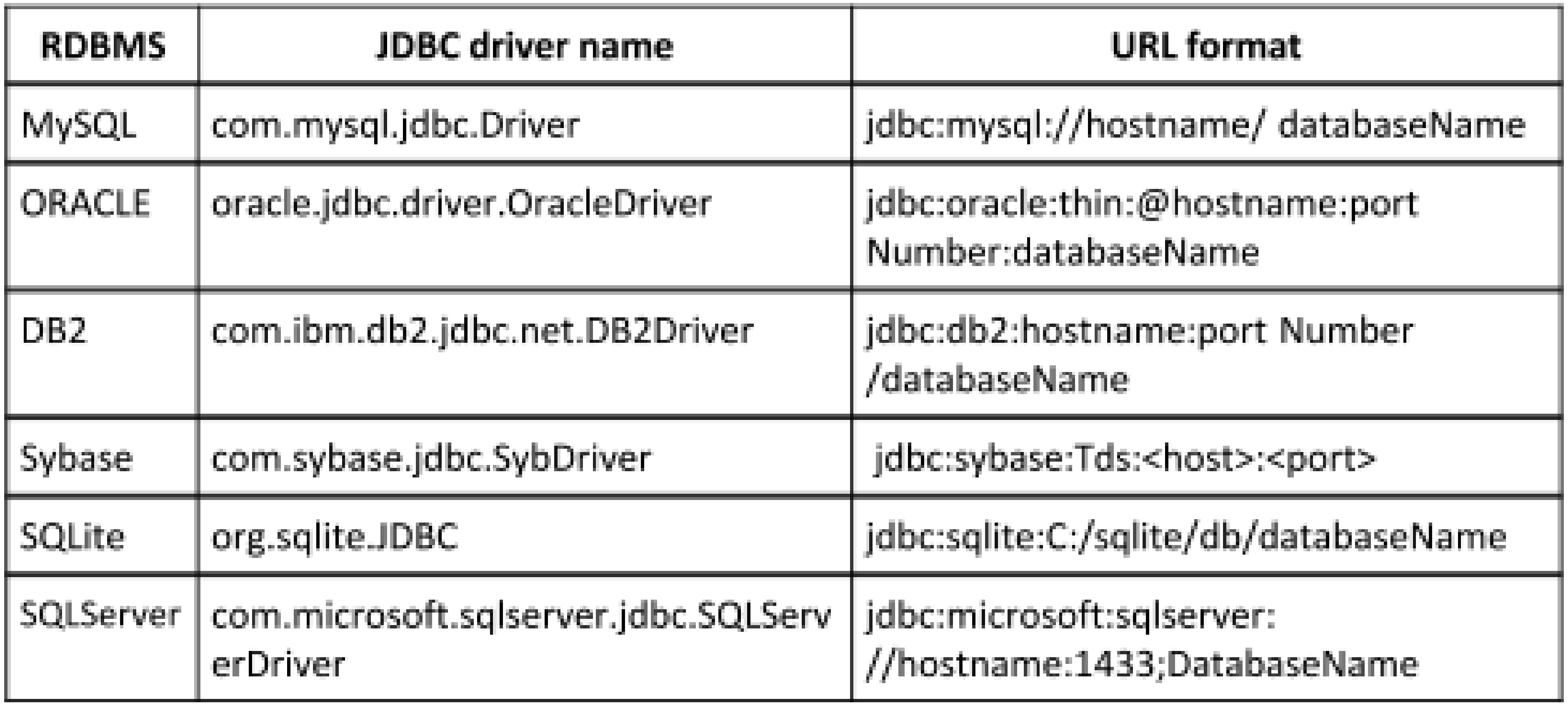
Disadvantage

- This Driver uses database specific protocol and it is DBMS vendor dependent.

Which driver should be used?

- If you are accessing one type of database such as MySql, Oracle, Sybase or IBM etc., the preferred driver type is 4.
- If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.
- Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database.
- The type 1 driver is not considered a deployment-level driver, and is typically used for development and testing purposes only.

JDBC with different RDBMS



RDBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver	jdbc:mysql://hostname/ databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@hostname:port Number:databaseName
DB2	com.ibm.db2.jdbc.net.DB2Driver	jdbc:db2:hostname:port Number /databaseName
Sybase	com.sybase.jdbc.SybDriver	jdbc:sybase:Tds:<host>:<port>
SQLite	org.sqlite.JDBC	jdbc:sqlite:C:/sqlite/db/databaseName
SQLServer	com.microsoft.sqlserver.jdbc.SQLServerDriver	jdbc:microsoft:sqlserver: //hostname:1433;DatabaseName

1.3 Java Database Connectivity Steps

Before you can create a java jdbc connection to the database, you must first import the java.sql package.
import java.sql.*; The star (*) indicates that all of the classes in the package java.sql are to be imported.
There are following six steps involved in building a JDBC application:

1). Import the packages: Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using import java.sql.* will suffice.

2). Register the JDBC driver: Requires that you initialize driver so you can open a communications channel with the database.

3). Open a connection: Requires using the DriverManager .getConnection() method to create a Connection object, which represents a physical connection with the database.

Java Database Connectivity Steps

4). Execute a query: Requires using an object of type Statement for building and submitting an SQL statement to the database.

5). Extract data from result set: Requires that you use the appropriate ResultSet.getXXX() method to retrieve the data from the result set.

6). Clean up the environment: Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

1.3.1 JDBC Connection

- The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver. DriverManager is considered the backbone of JDBC architecture.
- DriverManager class manages the JDBC drivers that are installed on the system.
- Its getConnection() method is used to establish a connection to a database. It uses a username, password, and a jdbc url to establish a connection to the database and returns a connection object. A jdbc Connection represents a session/connection with a specific database.
- With the context of a Connection, SQL, PL/SQL statements are executed and results are returned. An application can have one or more connections with a single database, or it can have many connections with different databases. A Connection object provides metadata i.e. information about the database, tables, and fields. It also contains methods to deal with transactions.

1.3.1 JDBC Connection

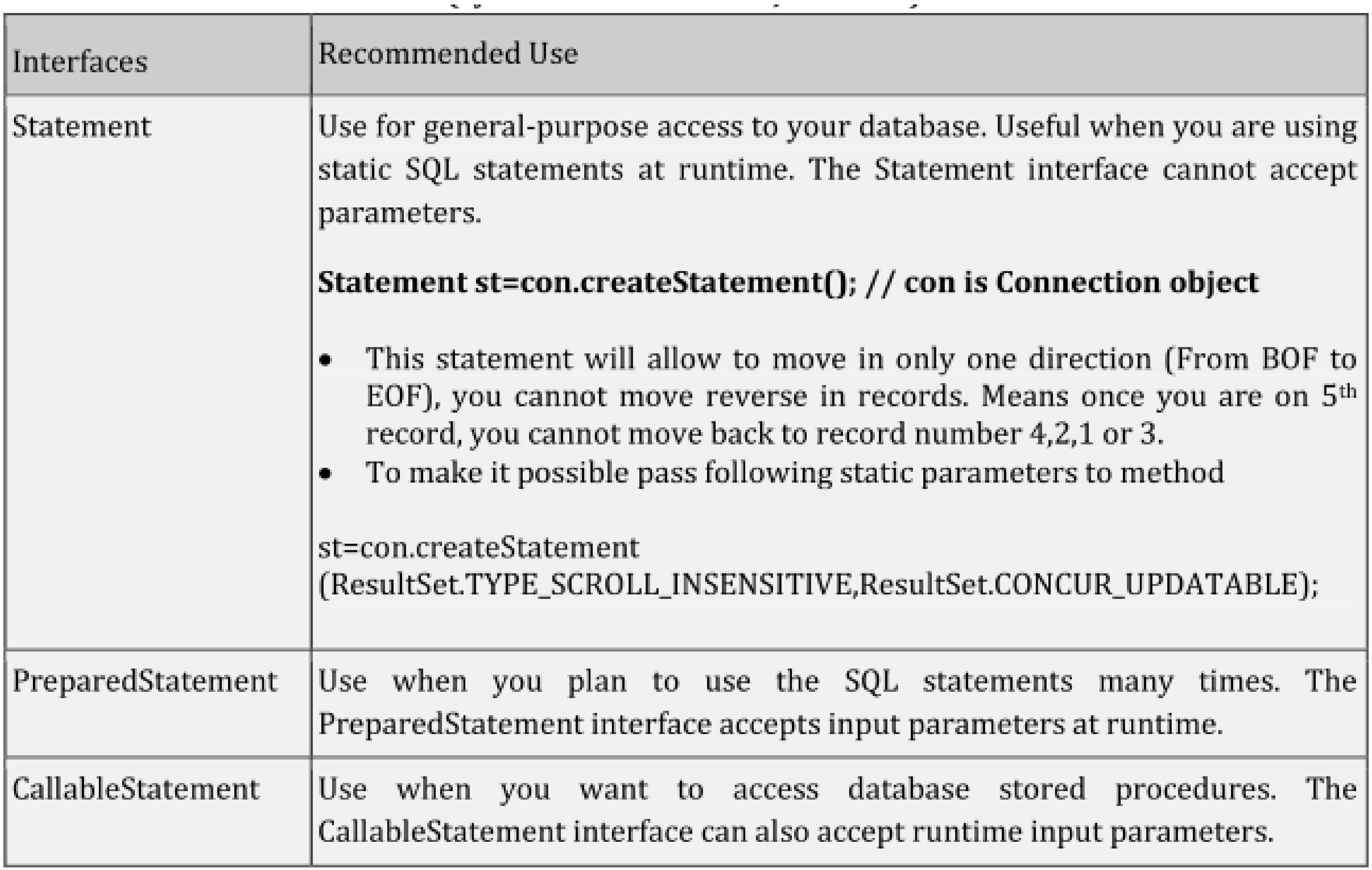
- JDBC URL Example:: jdbc: <subprotocol>: <subname>
  - Each driver has its own subprotocol
  - Each subprotocol has its own syntax for the source. We're using the jdbc odbc subprotocol, so the DriverManager knows to use the sun.jdbc.odbc.JdbcOdbcDriver.

```
Try
{
    Connection con =DriverManager.
            getConnection(url,"loginName","Password")
}
catch( SQLException x ){
    System.out.println( "Couldn't get connection!" );
}
```

1.3.2 Types of Statements

- Once a connection is obtained we can interact with the database. Connection interface defines methods for interacting with the database via the established connection. To execute SQL statements, you need to instantiate a Statement object from your connection object by using the createStatement() method.
    - Statement st = con.createStatement();
- A statement object is used to send and execute SQL statements to a database.
- Three kinds of Statements
- They also define methods that help bridge data type differences between Java and SQL data types used in a database.

1.3.2 Types of Statements



Interfaces	Recommended Use

Statement	Use for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
	
	Statement st=con.createStatement(); // con is Connection object
	
	• This statement will allow to move in only one direction (From BOF to EOF), you cannot move reverse in records. Means once you are on 5th record, you cannot move back to record number 4,2,1 or 3.
	• To make it possible pass following static parameters to method
	
	st=con.createStatement (ResultSet.TYPE_SCROLL_INSENSITIVE,ResultSet.CONCUR_UPDATABLE);

PreparedStatement	Use when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use when you want to access database stored procedures. The CallableStatement interface can also accept runtime input parameters.

1.4 Execution Query

- Statement interface defines methods that are used to interact with database via the execution of SQL statements. The Statement class has three methods for executing statements:
- executeQuery(), executeUpdate(), and execute().
- For a SELECT statement, the method to use is executeQuery().
- For statements that create or modify tables, the method to use is executeUpdate().
- Note: Statements that create a table, alter a table, or drop a table are all examples of DDL statements and are executed with the method executeUpdate.
- execute() executes an SQL statement that is written as String object.
- Example :
  - ResultSet rs=stmt.executeQuery("select * from student");
  - stmt.executeUpdate(“insert into student values(1,’abc’)”);

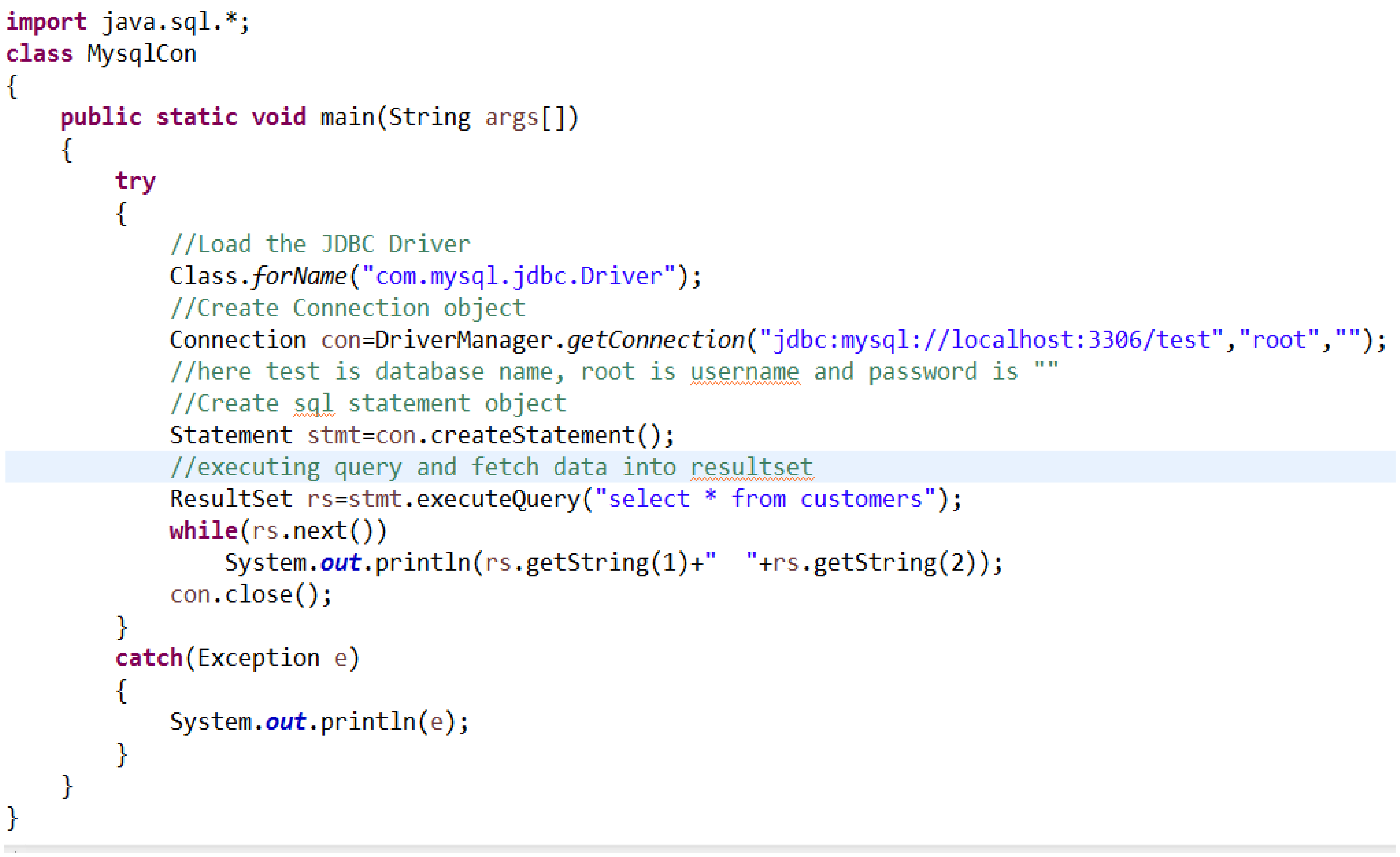
1.4 ResultSet

- The SQL statements that read data from a database query return the data in a result set. The SELECT statement is the standard way to select rows from a database and view them in a result set.
- A ResultSet object maintains a cursor that points to the current row in the result set. The term "result set" refers to the row and column data contained in a ResultSet object.
- The next() method is used to successively step through the rows of the tabular results.

1.4 Types of ResultSet

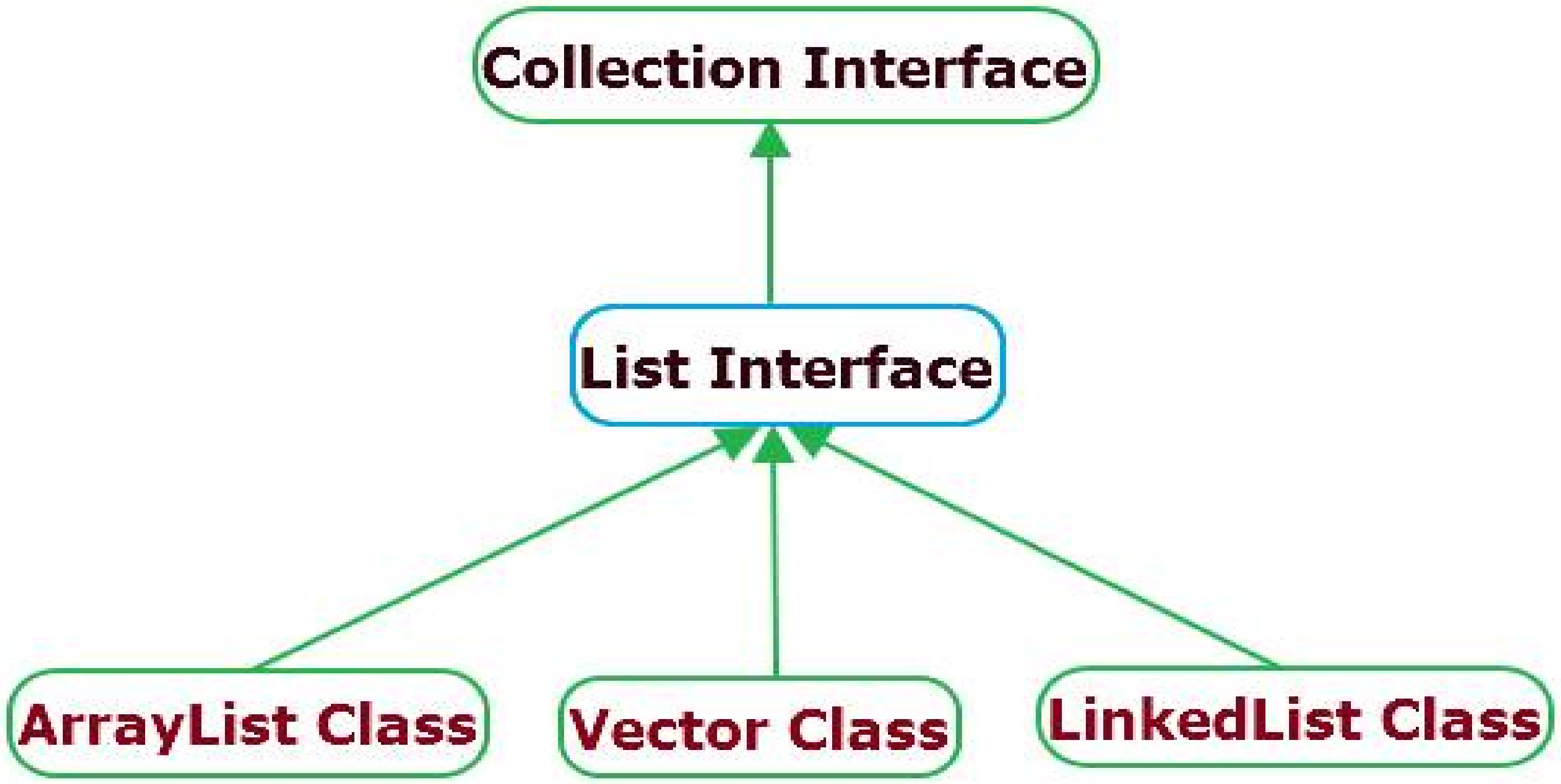
Type	Description
ResultSet.TYPE_FORWARD_ONLY **Default**	The cursor can only move forward in the result set.
ResultSet.TYPE_SCROLL_INSENSITIVE	The cursor can scroll forward and backward, and the result set is not sensitive to changes made by others to the database that occur after the result set was created.
ResultSet.TYPE_SCROLL_SENSITIVE.	The cursor can scroll forward and backward, and the result set is sensitive to changes made by others to the database that occur after the result set was created.

JDBC Example



```java
import java.sql.*;
class MysqlCon
{
    public static void main(String args[])
    {
        try
        {
            //Load the JDBC Driver
            Class.forName("com.mysql.jdbc.Driver");
            //Create Connection object
            Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/test","root","");
            //here test is database name, root is username and password is ""
            //Create sql statement object
            Statement stmt=con.createStatement();
            //executing query and fetch data into resultset
            ResultSet rs=stmt.executeQuery("select * from customers");
            while(rs.next())
                System.out.println(rs.getString(1)+"  "+rs.getString(2));
            con.close();
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}
```

1.5 List Interface



1.5 List Interface

- **List** in Java provides the facility to maintain the *ordered collection*. It contains the index-based methods to insert, update, delete and search the elements. It can have the duplicate elements also. We can also store the null elements in the list.
- The List interface is found in the java.util package and inherits the Collection interface. It is a factory of ListIterator interface. Through the ListIterator, we can iterate the list in forward and backward directions. The implementation classes of List interface are ArrayList, LinkedList, Stack and Vector. The ArrayList and LinkedList are widely used in Java programming.

1.5 Operation on List

- List Interface extends Collection, hence it supports all the operations of Collection Interface, along with following additional operations:

1. Positional Access: List allows add, remove, get and set operations based on numerical positions of elements in List. List provides following methods for these operations:

void add(int index, Object O):	This method adds given element at specified index.
boolean addAll(int index, Collection c):	This method adds all elements from specified collection to list. First element gets inserted at given index. If there is already an element at that position, that element and other subsequent elements(if any) are shifted to the right by increasing their index.
Object remove(int index):	This method removes an element from the specified index. It shifts subsequent elements(if any) to left and decreases their indexes by 1.
Object get(int index):	This method returns element at the specified index
Object set(int index, Object new):	This method replaces element at given index with new element. This function returns the element which was just replaced by new element.

1.5 Operation on List

2. Search: List provides methods to search element and returns its numeric position. Following two methods are supported by List for this operation:

int indexOf(Object o):	This method returns first occurrence of given element or -1 if element is not present in list.
int lastIndexOf(Object o):	This method returns the last occurrence of given element or -1 if element is not present in list

3. Iteration: ListIterator(extends Iterator) is used to iterate over List element. List iterator is bidirectional iterator. For more details about ListIterator refer Iterators in Java.
4. Range-view: List Interface provides a method to get the List view of the portion of given List between two indices.

1.5 ArrayList Class

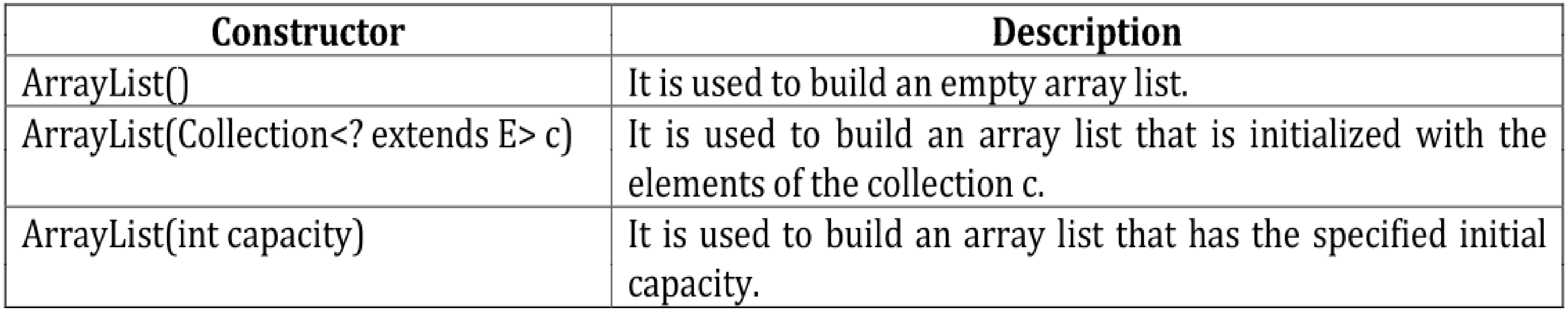
- Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.
- The important points about Java ArrayList class are:
  - Java ArrayList class can contain duplicate elements.
  - Java ArrayList class maintains insertion order.
  - Java ArrayList class is non synchronized.
  - Java ArrayList allows random access because array works at the index basis.
  - In ArrayList class, manipulation is slow because a lot of shifting needs to occur if any element is removed from the array list.

1.5 ArrayList Class

- Creating ArrayList

```
//Creating a List of type String using ArrayList
List<String> list=new ArrayList<String>();

//Creating a List of type Integer using ArrayList
List<Integer> list=new ArrayList<Integer>();
```



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.

1.5 ArrayList Class

```
/* This program is for ArrayList Example*/
import java.util.*;
public class ArrayListExample {
public static void main(String args[])
{
   ArrayList<String>  list=new  ArrayList<String>();                    //Creating
  arraylist
   list.add("Hero");              //Adding object in arraylist
   list.add("Honda");
   list.add("Bajaj");
   System.out.println(list);       //Invoking arraylist object
   //Iterating the List element using for-each loop
   System.out.println("Using for method");
   for(String byke:list)
      System.out.println(byke);
```

1.5 ArrayList Class

```
//adding element in specific position
 System.out.println("Using add() method for specific position");
 list.add(1,"yamaha");
 System.out.println(list);

//Using get method to get ArrayList elements
 System.out.println("Using get() method");
 System.out.println(list.get(1));

 //Using set method to set ArrayList elements
 System.out.println("Using set() method");
 // set() replaces the "Enfield" as 2nd element.
 list.set(1,"Enfield");
 System.out.println(list);
```

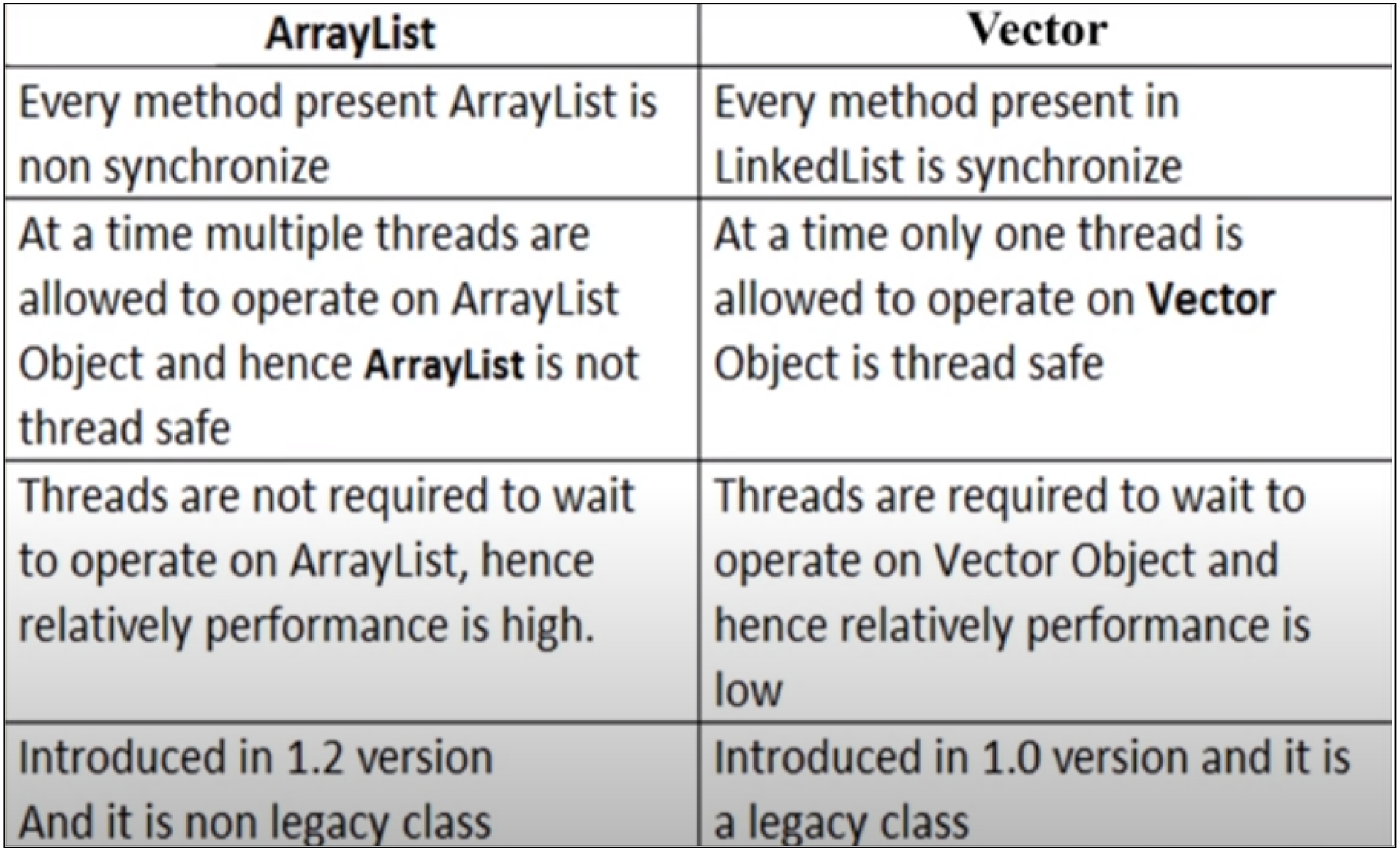
1.5 ArrayList Class

```
   //sorting elements
   System.out.println("Using sort() method");
   Collections.sort(list);
   System.out.println(list);

   //removing last element
   System.out.println("Using remove() method");
   list.remove(2);
   System.out.println(list);

   //removing all element
   System.out.println("Using clear() method");
   list.clear();
   System.out.println(list);
}
}
```

1.5 ArrayList v/s Vector



ArrayList	**Vector**
Every method present ArrayList is non synchronize	Every method present in LinkedList is synchronize
At a time multiple threads are allowed to operate on ArrayList Object and hence **ArrayList** is not thread safe	At a time only one thread is allowed to operate on **Vector** Object is thread safe
Threads are not required to wait to operate on ArrayList, hence relatively performance is high.	Threads are required to wait to operate on Vector Object and hence relatively performance is low
Introduced in 1.2 version And it is non legacy class	Introduced in 1.0 version and it is a legacy class

1.5 Vector

- **Vector** is like the *dynamic array* which can grow or shrink its size. Unlike array, we can store n-number of elements in it as there is no size limit. It is a part of Java Collection framework since Java 1.2. It is found in the java.util package and implements the *List* interface, so we can use all the methods of List interface here.
- It is recommended to use the Vector class in the thread-safe implementation only. If you don't need to use the thread-safe implementation, you should use the ArrayList, the ArrayList will perform better in such case.

1.5 Vector

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

1.5 Vector

add()	**It is used to append the specified element in the given vector.**
addAll()	It is used to append all of the elements in the specified collection to the end of this Vector.
addElement()	It is used to append the specified component to the end of this vector. It increases the vector size by one.
capacity()	It is used to get the current capacity of this vector.
clear()	It is used to delete all of the elements from this vector.
contains()	It returns true if the vector contains the specified element.
elementAt()	It is used to get the component at the specified index.
indexOf()	It is used to get the index of the first occurrence of the specified element in the vector. It returns -1 if the vector does not contain the element.
get()	It is used to get an element at the specified position in the vector.
set()	It is used to replace the element at the specified position in the vector with the specified element.
remove()	It is used to remove the specified element from the vector. If the vector does not contain the element, it is unchanged.

1.5 Vector

```
import java.util.*;
public class VectorClassExample {
    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());
```

1.5 Vector

```
  //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());
System.out.println("Capacity after addition is: "+vec.capacity());
//Display Vector elements again
System.out.println("Elements are: "+vec);
//Checking if Tiger is present or not in this vector
 if(vec.contains("Rat"))
 { System.out.println("Rat is present at the index "
  +vec.indexOf("Rat"));          }
 else          {
  System.out.println("Tiger is not present in the list.");          }
```

1.5 Vector

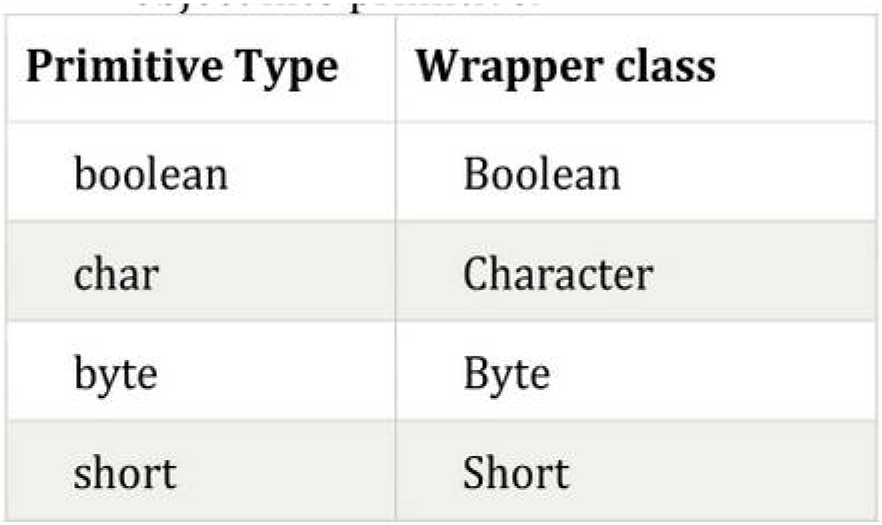
```
              //Get the first element
          System.out.println("The first animal of the vector is = "+vec.firstElement());
          //Get the last element
          System.out.println("The last animal of the vector is = "+vec.lastElement());
System.out.println("get method = "+vec.get(4));
           //set the 5th element
          vec.set(4,"Tiger");
          System.out.println("After using set method = "+vec);
          //remove first occurance of element
          vec.remove("Tiger");
          System.out.println("After using remove method"+vec);
          //remove specified index of element
          vec.remove(2);
          System.out.println("After using remove with specified index method"+vec);
        }
}
```

1.5 Iterator

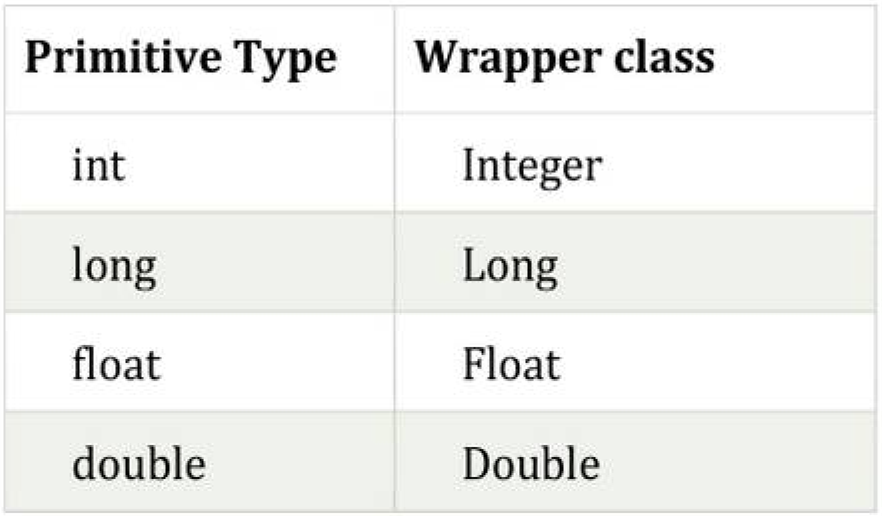
```
import java.util.*;
public class IteratorExample
{      public static void main(String args[])
              {
       ArrayList<String> list=new ArrayList<String>();//Creating arraylist
       list.add("Ravi");//Adding object in arraylist
       list.add("Vijay");
       list.add("Ravi");
       list.add("Ajay");
 //Traversing list through for-each loop
       System.out.println("Using for loop");
       for(String name:list)
          System.out.println(name);
       //Traversing list through Iterator
       System.out.println("Using Iterator");
       Iterator itr=list.iterator();
       while(itr.hasNext())
       {          System.out.println(itr.next());            }
   }
}
```

1.6 Wrapper Class

- Each of Java's eight primitive data types has a class dedicated to it.
- These are known as wrapper classes, because they "wrap" the primitive data type into an object of that class.
- The wrapper classes are part of the java.lang package, which is imported by default into all Java programs.
- The wrapper class in Java provides the mechanism to convert primitive into object and object into primitive.



Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short



Primitive Type	Wrapper class
int	Integer
long	Long
float	Float
double	Double

1.6 Wrapper Class

Autoboxing

- The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing.
- For example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Unboxing

- The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing.
- It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

1.6 Wrapper Class

```
public class WrapperClassExample {
   public static void main(String arg[])
   {
      System.out.println("Autoboxing");
      int a=20;      //Premetive Data type
      //converting premitivve Datatype to object
      Integer i=Integer.valueOf(a);//converting int into Integer explicitly

      //Converting premitivve Datatype to object automatically
      Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
      System.out.println(a+" "+i+" "+j);

      System.out.println("Unboxing");
      Integer x=new Integer(3);      //Object
      //converting object to premitivve Datatype
      int y=x.intValue();//converting Integer to int explicitly
      //converting object to premitivve Datatype automatically
      int z=x;//unboxing, now compiler will write a.intValue() internally
      System.out.println(x+" "+y+" "+z);
   }}
```