UNIT – V

Overview of Networking in Java: URL class and its usage through connection, Sockets based connectivity, TCP/IP Sockets and server sockets, Datagram Sockets. Collections in Java-Array List, stack, queue, Hash table, Collection class hierarchy, JDBC and Jar files.

UNIT – V

15. Networking in Java 241-249

15.1 Networking in Java 241
15.2 InetAddress Class 241
15.3 URL Class 242
15.4 URLConnection Class. 242
15.5 Socket 244
15.6 TCP/IP Socket (Client Socket) 244
15.7 ServerSocket Class 246
15.8 Java UDP Networking 247
15.9 UDP Verses TCP 247
15.10 Datagram Socket 248

16. Collection Framework 250-263

16.1 Collection Framework 250
16.2 Collection Hierarchy 250
16.3 Benefits of the Java Collections Framework 251
16.4 Collection Interface Bulk Operations 252
16.5 Iterators 252
16.6 The Collection Classes 253

17. JDBC Architecture 264-276

17.1 JDBC 264
17.2 JDBC Driver and JDBC Drivers Types 264
17.3 JDBC Architecture 267
17.4 Common JDBC Components 268
17.5 Creating JDBC Application 268

CHAPTER » 15

Networking in Java

15.1 Networking in Java

The network is the soul of Java. Java is the first mainstream programming language to provide built in support for high level Network/Internet programming. Where using other programming languages like C, to develop internet applications the programmer has to use operating system dependent APIs for which he has to refer volumes of operating system manuals and machine manuals. The Java environment is designed in such a way that application program that access computers on any part of the globe can be written very easily. Java derives this power from a set of classes defined in the java.net package.

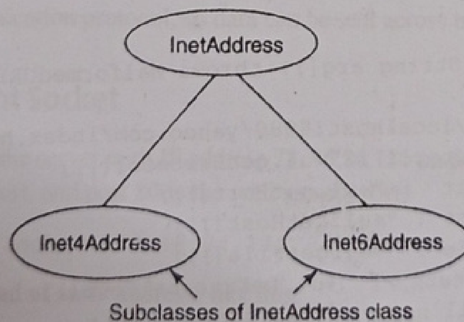
Most of what is new and exciting about Java centers on the potential for new kinds of dynamic networked applications. Java is a premier language for network programming. *java.net* package encapsulate large number of classes and interface that provides an easy-to use means to access network resources.

15.2 InetAddress Class

InetAddress encapsulates both numerical IP address and the domain name for that address. Inet address can handle both IPv4 and Ipv6 addresses. InetAddress class has no visible constructor. To create an inet Address object you have to use *Factory methods.*

Three commonly used InetAddress factory methods are.

1. static InetAddress *getLocalHost()* throws *UnknownHostException*
2. static InetAddress *getByName* (String hostname) throws *UnknownHostException*
3. static *InetAddress[ ] getAllByName* (String hostname) throws *UnknownHostException*



Subclasses of InetAddress class

Example 1: Program using InetAddress class.

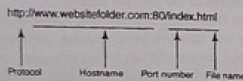
```
import java.net.*;
import java.io.*;
class Net_4
{
  public static void main(String arg[]) throws UnknownHostException
```

```
    {
     InetAddress inet=InetAddress.getByName("www.yahoo.com");
     System.out.println(" "+inet.getAddress());
     byte b[]=inet.getAddress();
     for(int i=0;i<4;i++)
     {
       System.out.println(b[i]&255);
     }
     InetAddress inet2=InetAddress.getByAddress(b);
     System.out.println(" "+inet.getHostName());

    }
  }
```

15.3 URL Class

URL stands for Uniform Resource Locator and represents a resource on the World Wide Web such as a Web page or FTP directory. Java URL Class present in java.net package deals with URL (Uniform Resource Locator) which uniquely identify or locate resources on internet.



- **getProtocol():** Returns protocol of URL.
- **getHost():** Returns hostname(domain name) of URL.
- **getPort():** Returns port number of URL.
- **getFile():** Returns filename of URL.

Example 2: Program using URL class

```
import java.net.*;
class Net_1

{
 public static void main(String arg[]) throws MalformedURLException
 {
  URL u1=new URL("http://localhost:8080/yahoo.com/index.html");
  System.out.println("Protocol : "+u1.getProtocol());
  System.out.println("Port   : "+u1.getPort());
  System.out.println("Host : "+u1.getHost());
  System.out.println("File : "+u1.getFile());
 //System.out.println("Extern  : "+u1.toExternalForm());
  System.out.println("Full name : "+u1.toString());
 }
}
```

15.4 URLConnection Class

The URL class can also be used to access files in the local file system. Thus the URL class can be a use to open a file. it does not matter whether the file came from the network or local file system.

Here is code to open a file in the local file system using the URL class:

```
URL url = new URL("file:/c:/data/test.txt");
URLConnection urlConnection = url.openConnection();
InputStream input = urlConnection.getInputStream();
int data = input.read();
while(data != -1)
{
 System.out.print((char) data);
 data = input.read();
}
input.close();
```

Example 3: Program to access a file.

```
import java.net.MalformedURLException;
import java.net.URL:
import java.io.*;
class Net_2

{
 public static void main(String arg[]) throws IOException
 {
URL u1=new URL("http://localhost:9080/yahoo.com/index.html");
   InputStream is=u1.openStream();
   FileOutputStream fout=new FileOutputStream("d:/index.jsp");
   int c=0;
   while((c=is.read())!=1)
   {
     fout.write(c);
   }

  fout.close();
  is.close();
  }
}
```

Example 4: Program to access file from internet.

```
import java.net.MalformedURLException;
import java.net.URL;
import java.net.*;
import java.io.*;
class Net_1

{
 public static void main(String arg[]) throws IOException
 {
   URL u1=new URL("http://localhost:8080/yahoo.com/index.html");
   URLConnection uc=u1.openConnection();
   System.out.println("Date : "+uc.getDate());
```

```
        System.out.println("Content Type  : "+uc.getContentType());
        //System.out.println("Expire : "+uc.getExpiration());
        System.out.println("Late Modify : "+uc.getLastModified());

        int l=uc.getContentLength();
        System.out.println("Length  : "+l);

        InputStream is=uc.getInputStream();
        int c;
        while((c=is.read())!=1)
        {
          System.out.println((char)c);
        }

        is.close();
      }
   }
```

15.5 Socket

A *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent. An end point is a combination of an IP address and a port number.

15.5.1 Socket Based Connectivity

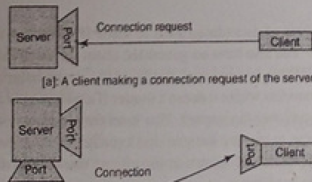
Sockets provide the communication mechanism between two computers using TCP. A client program creates a socket on its end of the communication and attempts to connect that socket to the server.

When the connection is made the server creates a socket object on its end of the communication. The client and server can now communicate by writing to and reading from the socket. The java.net.Socket class represents a socket and the java.net.ServerSocket class provides a mechanism for the server program to listen for clients and establish connections with them.

15.6 TCP/IP Socket (Client Socket)

A TCP/IP client socket is used to create a reliable, bi-directional, stream-based connection between two computers on a network. The client socket is implemented by creating an instance of the Socket class. It is designed to connect to the server and initialize protocol exchanges.

Typically a client opens a TCP/IP connection to a server. The client then starts to communicate with the server. When the client is finished it closes the connection again.



(a): A client making a connection request of the server

A client may send more than one request through an open connection. In fact, a client can send as much data as the server is ready to receive. The server can also close the connection if it wants to.

15.6.1 Methods to Create TCP Client Sockets

An object of the Socket class can be created by these methods:

- By specifying the hostname and port number as follows:
  Socket (String hostname1, int port1)

Where hostname1 is a string type variable that refers to the destination address and port1 refers to the port number of the destination address. This method can throw the exceptions UnknownHostException or IOException, in case of errors.

- By specifying an object of InetAddress and the port number as follows:
  Socket(InetAddress ipaddr1, int port1)

Where ipaddr1 is object of the InetAddress class and port1 refers to the port number of the destination. This method can throw the exception IOException in case of errors.

The following steps occur when establishing a TCP connection between two computers using sockets:

- The server instantiates a ServerSocket object, denoting which port number communication is to occur on.
- The server invokes the accept method of the ServerSocket class. This method waits until a client connects to the server on the given port. After the server is waiting a client instantiates a Socket object specifying the server name and port number to connect to.
- The constructor of the Socket class attempts to connect the client to the specified server and port number. If communication is established the client now has a Socket object capable of communicating with the server.
- On the server side, the accept method returns a reference to a new socket on the server that is connected to the client's socket. After the connections are established, communication can occur using I/O streams.
- Each socket has both an OutputStream and an InputStream. The client's OutputStream is connected to the server's InputStream and the client's InputStream is connected to the server's OutputStream.

TCP is a two way communication protocol, so data can be sent across both streams at the same time.

15.6.2 Creating a Client Socket

This code example connects to the server with IP address 78. 46.84.171 on port 80. That server happens to be my web server (www.yahoo.com), and port 80 is the web servers port.

```
Socket socket = new Socket("78.46.84.171", 80);
```

Domain name can be instead of an IP address, like this:

```
Socket socket = new Socket("yahoo.com", 80);
```

Socket Program code:

```
Socket socket = new Socket("yahoo.com", 80);
OutputStream out = socket.getOutputStream();
out.write("some data".getBytes());
out.flush();
```

```
out.close();
socket.close();
```

Don't forget to call flush() to the data sent across the internet to the server. The underlying TCP/IP implementation in OS may buffer the data and send it in larger chunks to fit with the size of TCP/IP packets.

15.6.3 Reading from a Socket

To read from a Java Socket you will need to obtains its *InputStream*. Here is how that is done:

```
Socket socket = new Socket("yahoo.com", 80);
InputStream in = socket.getInputStream();
int data = in.read();
//... read more data...
in.close();
socket.close();
```

Keep in mind that you cannot always just read from the Socket's InputStream until it returns -1, as you can when reading a file. The reason is that -1 is only returned when the server closes the connection. But a server may not always close the connection. Perhaps you want to send multiple requests over the same connection. In that case it would be pretty stupid to close the connection.

Instead you must know how many bytes to read from the Socket's InputStream. This can be done by either the server telling how many bytes it is sending, or by looking for a special end-of-data character.

15.6.4 Closing a Socket

When you are done using a Java Socket you must close it to close the connection to the server. This is done by calling the.close() method, like this:

```
Socket socket = new Socket("yahoo.com", 80);
socket.close();
```

15.7 ServerSocket Class

The *java.net.ServerSocket* class is used by server applications to obtain a port and listen for client requests The ServerSocket class has four constructors:

15.7.1 Constructor with Description

1 **public ServerSocket (*int port*) throws IOException**
Attempts to create a server socket bound to the specified port. An exception occurs if the port is already bound by another application.

2 **public ServerSocket (*int port, int backlog*) throws IOException**
Similar to the previous constructor the backlog parameter specifies how many incoming clients to store in a wait queue.

3 **public ServerSocket (*int port, int backlog, InetAddress address*) throws IOException**
Similar to the previous constructor the InetAddress parameter specifies the local IP address to bind to. The InetAddress is used for servers that may have multiple IP addresses, allowing the server to specify which of its IP addresses to accept client requests.

UNIT • V

4 **public ServerSocket() throws IOException**
Creates an unbound server socket. When using this constructor use the bind method when you are ready to bind the server socket. If the ServerSocket constructor does not throw an exception it means that your application has successfully bound to the specified port and is ready for client requests.

Here are some of the common methods of the ServerSocket class:

15.7.2 Methods with Description

1 **public int getLocalPort()**
Returns the port that the server socket is listening on. This method is useful if you passed in 0 as the port number in a constructor and let the server find a port for you.

2 **public Socket accept() throws IOException**
Waits for an incoming client. This method blocks until either a client connects to the server on the specified port or the socket times out assuming that the time-out value has been set using the setSoTimeout method. Otherwise this method blocks indefinitely.

3 **public void setSoTimeout (*int timeout*)**
Sets the time-out value for how long the server socket waits for a client during the accept.

4 **public void bind (*SocketAddresshost, int backlog*)**
Binds the socket to the specified server and port in the SocketAddress object Use this method if you instantiated the ServerSocket using the no-argument constructor. When the ServerSocket invokes accept the method does not return until a client connects. After a client does connect the ServerSocket creates a new Socket on an unspecified port and returns a reference to this new Socket. A TCP connection now exists between the client and server, and communication can begin.

15.8 Java UDP Networking

UDP works a bit differently from TCP. Using UDP there is no connection between the client and server. A client may send data to the server and the server may (or may not) receive this data. The client will never know if the data was received at the other end. The same is true for the data sent the other way from the server to the client.

Because there is no guarantee of data delivery the UDP protocol has less protocol overhead.

There are several situations in which the connectionless UDP model is preferable over TCP. These are covered in more detail in the text on Java's UDP.

15.9 UDP Verses TCP

UDP works a bit differently from TCP. When you send data via TCP you first create a connection. Once the TCP connection is established TCP guarantess that your data arrives at the other end, or it will tell you that an error occurred.

With UDP you just send packets of data (datagrams) to some IP address on the network. You have no guarantee that the data will arrive. You also have no guarantee about the order which UDP packets arrive in at the receiver. This means that UDP has less protocol overhead (no stream integrity checking) than TCP.

UDP is appropriate for data transfers where it doesn't matter if a packet is lost in transition. For instance, imagine a transfer of a live TV-signal over the internet. You want the signal to arrive at the clients as close to live as possible. Therefore if a frame or two are lost you don't really care. You don't want the live broadcast to be delayed just to make sure all frames are shown at the client. You'd rather skip the missed frames and move directly to the newest frames at all times.

15.10 Datagram Socket

DatagramSocket's are Java's mechanism for network communication via UDP instead of TCP. UDP is still layered on top of IP. You can use Java's DatagramSocket both for sending and receiving UPD datagrams.

15.10.1 Sending Data via a DatagramSocket

To send data via Java's DatagramSocket you must first create DatagramPacket. Here is how that is done:

```
byte[] buffer = new byte[65508];
InetAddress address = InetAddress.getByName("yahoo.com");
DatagramPacket packet = new DatagramPacket(
  buffer, buffer.length, address, 9000);
```

The byte buffer (the byte array) is the data that is to be sent in the UDP datagram. The length of the above buffer, 65508 bytes, is the maximum amount of data you can send in a single UDP packet.

The length given to the DatagramPacket constructor is the length of the data in the buffer to send. All data in the buffer after that amount of data is ignored.

The InetAddress instance contains the address of the node (e.g. server) to send the UDP packet to. TheInetAddress class represents an IP address (Internet Address). The getByName() method returns anInetAddress instance with the IP address matching the given host name.

The port parameter is the UDP port the server to receiver the data is listeing on. UDP and TCP ports are not the same. A computer can have different processes listening on e.g. port 80 in UDP and in TCP at the same time.

To send the DatagramPacket you must create a DatagramSocket targeted at sending data. Here is code.

```
DatagramSocket datagramSocket = new DatagramSocket();
```

To send data you call the send() method, like this:

```
datagramSocket.send(packet);
```

Here is full code:

```
DatagramSocket datagramSocket = new DatagramSocket();
byte[] buffer = "0123456789".getBytes();
InetAddress receiverAddress = InetAddress.getLocalHost();
DatagramPacket   packet   =   new   DatagramPacket(  buffer,     buffer.length,
receiverAddress, 80);
datagramSocket.send(packet);
```

15.10.2 Receiving Data via a DatagramSocket

Receiving data via a DatagramSocket is done by first creating a DatagramPacket and then receiving data into it via the DatagramSocket's receive() method.

Here is code:

```
DatagramSocket datagramSocket = new DatagramSocket(80);
byte[] buffer = new byte[10];
DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
datagramSocket.receive(packet);
```

Notice how the DatagramSocket is instantiated with the parameter value 80 passed to its constructor. This parameter is the UDP port the DatagramSocket is to receive UDP packets on. As mentioned earlier, TCP and UDP ports are not the same, and thus do not overlap. You can have two different processes listening on both TCP and UDP port 80, without any conflict.

Second a byte buffer and a DatagramPacket is created. We are going to use the DatagramPacket for receiving data not sending it. Thus no destination address is needed.

Finally the DatagramSocket's receive() method is called. This method blocks until a DatagramPacket is received.

The data received is located in the DatagramPacket's byte buffer. This buffer can be obtained by calling:

```
byte[] buffer = packet.getData();
```

How much data was received in the buffer is up to you to find out. The protocol you are using should specify either how much data is sent per UDP packet or specify an end-of-data marker you can look for instead.

A real server program would probably call the receive() method in a loop and pass all received DatagramPacket's to a pool of worker threads just like a TCP server does with incoming connections.

■ ■ ■

Very Short Questions

1. What is socket?
2. What is URL?
3. What is main difference between String Class and URL class?
4. What do you mean by port number?

Short Questions

1. Explain InetAddress class with example.
2. Explain DatagramSocket with methods and constructors.
3. Write short note on TCP/IP Socket.
4. Describe URLConnection class.

Long Questions

1. Explain networking in java. Explain DatagramSocket with methods and constructors. Write short note on TCP/IP Socket.

CHAPTER » 16

Collection Framework

16.1 Collection Framework

Collection framework was not part of original Java release. Collection was added to J2SE 1.2. Prior to Java 2, Java provided adhoc classes such as Dictionary, Vector, Stack and properties to store and manipulates groups of objects. Collection framework provides many important classes and interfaces to collect and organize group of alike(dissimilar) objects. As collections doesn't store primitive data types (stores only refrences) hence Autoboxing facilitates the storing of primitive data types in collection by boxing it into its wrapper type. Java provides a set of Collection classes that implements Collection interface. Some of these classes provide full implementations that can be used as it is and other abstract classes provides skeletal implementations that can be used as starting points for creating concrete collections.

A *collections framework* is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

- **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
- **Implementations:** These are the concrete implementations of the collection interfaces. In essence they are reusable data structures.
- **Algorithms:** These are the methods that perform useful computations such as searching and sorting on objects that implement collection interfaces. The algorithms are said to be *polymorphic*: that is the same method can be used on many different implementations of the appropriate collection interface. In essence algorithms are reusable functionality.

16.2 Collection Hierarchy

The following list describes the core collection interfaces:

- **Collection:** It is root of the collection hierarchy. A collection represents a group of objects known as its *elements*. The Collection interface is the least common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired. Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific sub interfaces such as Set and List.

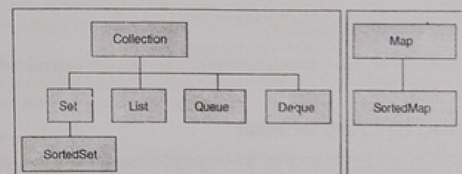
Set: A Set is a Collection that cannot contain duplicate elements. There are three main implementations of Set interface: HashSet, TreeSet, and LinkedHashSet. HashSet which stores its elements in a hash table is the best-performing implementation; however it makes no guarantees concerning the order of iteration. TreeSet which stores its elements in a red-black tree, orders its elements based on their values; it

is substantially slower than HashSet. LinkedHashSet, which is implemented as a hash table with a linked list running through it orders its elements based on the order in which they were inserted into the set (insertion-order). This interface models the mathematical set abstraction and is used to represent sets such as the cards comprising a poker hand the courses making up a student's schedule, or the processes running on a machine.

- **List:** It is an ordered collection (sometimes called a *sequence*). Lists can contain duplicate elements. The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position).
- **Queue:** It is a collection used to hold multiple elements prior to processing. Besides basic Collection operations a Queue provides additional insertion, extraction and inspection operations. Queues typically but do not necessarily order elements in a FIFO (first-in, first-out) manner. In a FIFO queue all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every Queue implementation must specify its ordering properties.
- **Deque:** It is a collection used to hold multiple elements prior to processing. Besides basic Collection operations a Deque provides additional insertion, extraction and inspection operations. Deques can be used both as FIFO (first-in, first-out) and LIFO (last-in, first-out). In a deque all new elements can be inserted retrieved and removed at both ends. Also see *The Deque Interface* section.
- **Map:** It is an object that maps keys to values. A Map cannot contain duplicate keys each key can map to at most one value.

The last two core collection interfaces are merely sorted versions of Set and Map:

- **SortedSet:** It is a Set that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets such as word lists and membership rolls.
- **SortedMap:** It is a Map that maintains its mappings in ascending key order. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key and value pairs such as dictionaries and telephone directories.



16.3 Benefits of the Java Collections Framework

The Java Collections Framework provides the following benefits:

- **Reduces programming effort:** By providing useful data structures and algorithms the Collections Framework frees you to concentrate on the important perts of your program rather than on the low-level "plumbing" required to make it work. By facilitating interoperability among unrelated APIs the Java Collections Framework frees you from writing adapter objects or conversion code to connect APIs.

- **Increases program speed and quality:** This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms. The various implementations of each interface are interchangeable so programs can be easily tuned by switching collection implementations.
- **Allows interoperability among unrelated APIs:** The collection interfaces are the vernacular by which APIs pass collections back and forth. If my network administration API furnishes a collection of node names and if your GUI toolkit expects a collection of column headings our APIs will interoperate seamlessly even though they were written independently.
- **Reduces effort to learn and to use new APIs:** Many APIs naturally take collections on input and furnish them as output. In the past each such API had a small sub-API devoted to manipulating its collections. There was little consistency among these adhoc collections sub-APIs so you had to learn each one from scratch and it was easy to make mistakes when using them. With the advent of standard collection interfaces the problem went away.
- **Reduces effort to design new APIs:** This is the flip side of the previous advantage. Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead they can use standard collection interfaces.
- **Fosters software reuse:** New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

16.4 Collection Interface Bulk Operations

Bulk operations perform an operation on an entire Collection. You could implement these shorthand operations using the basic operations, though in most cases such implementations would be less efficient. The following are the bulk operations:

- **containsAll():** returns true if the target Collection contains all of the elements in the specified Collection.
- **addAll():** adds all of the elements in the specified Collection to the target Collection.
- **removeAll():** removes from the target Collection all of its elements that are also contained in the specified Collection.
- **retainAll():** removes from the target Collection all its elements that are not also contained in the specified Collection. That is it retains only those elements in the targetCollection that are also contained in the specified Collection.
- **Clear():** removes all elements from the Collection.

16.5 Iterators

An Iterator is an object that enables you to traverse through a collection and to remove elements from the collection selectively if desired. You get an Iterator for a collection by calling its iterator method. The following is the Iterator interface.

```
public interface Iterator<E>
{
 boolean hasNext();
 E next();
 void remove(); //optional
}
```

The hasNext() method returns true if the iteration has more elements and the next method returns the next element in the iteration. The remove() method removes the last element that was returned by next() from

the underlying Collection. The remove() method may be called only once per call to next() and throws an exception if this rule is violated.

Note that Iterator.remove() is the *only* safe way to modify a collection during iteration; the behavior is unspecified if the underlying collection is modified in any other way while the iteration is in progress.

Use Iterator instead of the *for-each* construct when you need to:

- Remove the current element. The for-each construct hides the iterator so you cannot call remove. Therefore the for-each construct is not usable for filtering.
- Iterate over multiple collections in parallel.

The following method shows you how to use an Iterator to filter an arbitrary Collection — that is, traverse the collection removing specific elements.

```
static void filter(Collection<?> c)
{
 for (Iterator<?> it = c.iterator(); it.hasNext(); )
 if (!cond(it.next()))
 it.remove();
}
```

This simple piece of code is polymorphic, which means that it works for any Collection regardless of implementation.

16.6 The Collection Classes

16.6.1 ArrayList Class

It is a resizable-array implementation of the List interface. Implements all optional list operations and permits all elements including null. In addition to implementing the List interface this class provides methods to manipulate the size of the array that is used internally to store the list. This class is roughly equivalent to Vector except that it is unsynchronized. The size(), isEmpty(), get(), set(), iterator(), and listIterator() methods are used to manipulate the objects. The add() operation is, adding n elements requires. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation.

1. ArrayList class extends *AbstractList* class and implements the *List* interface.
2. ArrayList supports dynamic array that can grow as needed. ArrayList has three constructors.
3. ArrayLists are created with an initial size when this size is exceeded it gets enlarged automatically.
4. It can contain Duplicate elements and maintains the insertion order.
5. ArrayLists are not synchronized.
6. toArray() method is used to get an array containing all the contents of the list.

Following are the reasons why you must obtain array from your ArrayList whenever required.

- To obtain faster processing.
- To pass array to methods who do not accept Collection as arguments.
- To integrate and use collections with legacy code.

Example 1: Program using ArratyList collection class.

```
import java.util.*;
class ArrayList_3
{
 public static void main(String[]args)
```

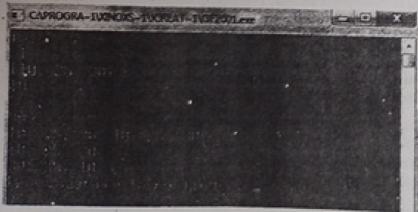
```
{
 ArrayList<Integer> al = new ArrayList<Integer>();
 al.add(10);
 al.add(20);
 al.add(30);
 al.set(1,25);
 al.add(40);
 al.remove(3);
 al.trimToSize();
 System.out.println(al.contains(5));
 System.out.println(al.indexOf(5));
 //al.clear();
 System.out.println(al);
 for(int i=0;i<al.size();i++)
  System.out.println(al.get(i));
 for(int i:al)//for each
  System.out.print(i+"  ");
 Iterator itr=al.iterator();
 while(itr.hasNext())
 {
  System.out.print(itr.next()+"  ");
 }
 System.out.println();
 ListIterator litr=al.listIterator();
 while(litr.hasNext())
 {
  System.out.print(litr.next()+"  ");
 }

 System.out.println();
 while(litr.hasPrevious())
 {
  System.out.print(litr.previous()+"  ");
 }

 System.out.println();

 }
}
```

Output:



Example 2: Program using ArayList Collrction class.

```
import java.util.*;
class ArrayListDemo
{
  public static void main(String[]args)
  {
   ArrayList<String> al = new ArrayList<String>();
   System.out.print("initial size of al:"+al.size());
   al.add("C");
   al.add("A");
   al.add("E");
   al.add("B");
   al.add("D");
   al.add("F");
   al.add(1,"A2");
   //al.add(new Integver(5)); will not compile
   System.out.println("Size of al after additions: "+al.size());
   System.out.println("Contents of al: "+al);
   al.remove("F");
   al.remove("2");
   System.out.println("Size of al after deletions: "+al.size());
   System.out.println("Contents of al: "+al);
  }
}
```

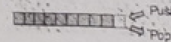
Output:

16.6.2 Stack Class and Queue

Each is defined by two basic operations: *insert* a new item and *remove* an item. When we insert an item our intent is clear. But when we remove an item the rule used for a queue is to always remove the item that has been in the collection the *most* amount of time. This policy is known as *first-in-first-out* or *FIFO*. The rule used for a stack is to always remove the item that has been in the collection the *least* amount of time. This policy is known as *last-in first-out* or LIFO.

16.6.2.1 Stack class

A *stack* (or just a *stack*) is a collection that is based on the last-in-first-out (LIFO) policy. The last-in-first-out policy offered by a stack provides just the behavior that you expect.



By tradition, we name the stack *insert* method push() and the stack *remove* operation pop(). There is method to test whether the stack is empty.

The java.util.Stack class is use for stack opration. The List interface is as an implementation on the Stack class. The typical use of a Stack is not as a List though.

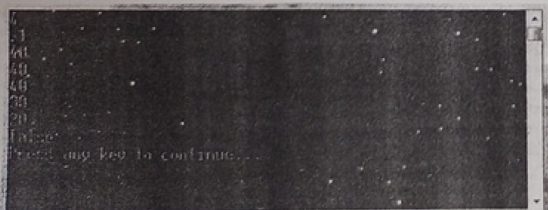
A Stack is a data structure where you add elements to the "top" of the stack and also remove elements from the top again. This is also referred to as the "Last In First Out (LIFO)" principle.

Example 3: Program using stack class.

```
import java.util.*;
class Stack_2

{
  public static void main(String arg[])
  {
    Stack<Integer> sl=new Stack<Integer>();
    sl.push(10);
    sl.push(20);
    sl.push(30);
    sl.push(40);
    System.out.println(sl.search(10));//lifo 4
    System.out.println(sl.search(11));
    try
    {
      System.out.println(sl.peek());
      System.out.println(sl.peek());
      System.out.println(sl.pop());
      System.out.println(sl.pop());
      System.out.println(sl.peek());
    }
    catch(EmptyStackException e)
    {
      System.out.println(e);
    }
    System.out.println(sl.empty());
  }
}
```

Output:



The push() method pushes an object onto the top of the Stack.

The peek() method returns the object at the top of the Stack but leaves the object on of the Stack.

The pop() method returns the object at the top of the stack and removes the object from the Stack. The search() method is used for searching an object. The object's equals() method is called on every object on the Stack to determine if the searched-for object is present on the Stack. The index you get is the index from the *top* of the Stack meaning the top element on the Stack has index 1.

16.6.2.2 Queue

In contrast a Queue uses a "First In First Out (FIFO)" principle. The *java.util.Queue* interface is subtype of the *java.util.Collection* interface. It represents an ordered list of objects just like a List but its intended use is slightly different. A queue is designed to have elements inserted at the end of the queue and elements removed from the beginning of the queue. Just like a queue in a supermarket.

Being a Collection subtype all methods in the Collection interface are also available in the Queue interface.

Since Queue is an interface you need to instantiate a concrete implementation of the interface in order to use it. The following are Queue implementations in the Java Collections API:

- java.util.LinkedList
- java.util.PriorityQueue

16.6.2.2.1 LinkedList class

LinkedList is a pretty standard queue implementation.

1. LinkedList class extends *AbstractSequentialList* and implements *List,Deque* and *Queue* inteface.
2. It can be used as List, stack or Queue as it implements all the related interfaces.
3. It can contain duplicate elements and is not synchronized.
4. LinkedList class extends *AbstractSequentialList* and implements *List,Deque* and *Queue* inteface.
5. It can be used as List, stack or Queue as it implements all the related interfaces.
6. It can contain duplicate elements and is not synchronized.

Example 4: Program using LinkedList.

```
import java.util.*;
class LinkList_4

{
  public static void main(String[]args)
  {
    LinkedList<Integer> al = new LinkedList<Integer>();
    al.add(10);
    al.addFirst(10);
    al.addLast(30);
    al.set(0,11);//replace
    System.out.println(al.contains(30)+" Contains");
    System.out.println(al.indexOf(30)+"indexOf");
    System.out.println(al.getFirst()+"First");
    System.out.println(al.getLast()+"Last");
    System.out.println(al.contains(30)+"contains");
    System.out.println(al);
    //al.clear();
    System.out.println(al);
```

```
for(int i=0;i<al.size();i++)
 System.out.println(al.get(i));
 System.out.println();
for(int i:al)//for each
 System.out.print(i+"  ");
 System.out.println();
Iterator itr=al.descendingIterator();
while(itr.hasNext())
{
 System.out.print(itr.next()+"  ");
}
System.out.println();
Iterator itr2=al.iterator();
while(itr2.hasNext())
{
 System.out.print(itr2.next()+"  ");
}
System.out.println();
ListIterator litr=al.listIterator();
while(litr.hasNext())
{
 System.out.print(litr.next()+"  ");
}

System.out.println();
while(litr.hasPrevious())
{
 System.out.print(litr.previous()+"  ");
}

System.out.println();
al.removeFirst();
al.removeLast();
System.out.println(al);


  }
 }
```

Output:

To add elements to a Queue you call its add() method. This method is inherited from the Collectioninterface. To take the first element out of the queue, use the remove() method.

16.6.2.2.2 PriorityQueue

PriorityQueue stores its elements internally according to their natural order (If they implement Comparable) or according to a Comparator passed to the PriorityQueue.

```
Queue queueB = new PriorityQueue();
```

16.6.2.2.3 Generic Queue

By default you can put any Object into a Queue but from Java 5, Java Generics makes it possible to limit the types of object you can insert into a Queue. Here is an example:

```
Queue<MyObject> queue = new LinkedList<MyObject>();
```

This Queue can now only have MyObject instances inserted into it. You can then access and iterate its elements without casting them. Here is how it looks:

```
MyObject myObject = queue.remove();
for(MyObject anObject : queue)
{
 //do someting to anObject...
}
```

16.6.3 HashSet class

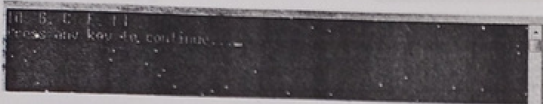
1. HashSet extends *AbstractSet* class and implements the *Set* interface.
2. It creates a collection that uses hash table for storage.
3. HashSet does not maintain any order of elements.

Example 5: Program using HashSet.

```
import java.util.*;
class HashSetDemo

{
 public static void main(String[]args)
 {
  //create a hash set
  HashSet<String> hs = new HashSet<String>();
  //add elements to the hash set
  hs.add("B");
  hs.add("A");
  hs.add("E");
  hs.add("C");
  hs.add("F");
  System.out.println(hs);
 }
}
```

Output:



16.6.4 Map interface

A Map is an object that maps keys to values. A map cannot contain duplicate keys. There are three main implementations of Map interfaces: HashMap, TreeMap and LinkedHashMap.

- **HashMap:** it makes no guarantees concerning the order of iteration.
- **TreeMap:** It stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than HashMap.
- **LinkedHashMap:** It orders its elements based on the order in which they were inserted into the set (insertion-order).

16.6.4.1 HashMap Class

java.util.*Package

Hash table based implementation of the Map interface. This class makes no guarantees for the order of the map. HashMap is a Map based collection class that is used for storing Key and value pairs. This class makes no guarantees as to the order of the map. It is similar to the Hashtable class except that it is unsynchronized and permits nulls(null values and null key). This implementation provides all of the optional map operations and permits null values and the null key. HashMap class is roughly equivalent to Hashtable except that it is unsynchronized and permits null.

This implementation provides constant-time performance for the basic operations (get and put). It provides constructors to set initial capacity and load factor for the collection.

Example 6: Program using HashMap.

```
import java.util.*;
class Hash_map

{
  public static void main(String[]args)
  {
   HashMap<String,Integer> al = new HashMap<String,Integer>();

   al.put("C",1000);
   al.put("C++",2000);
   al.put("java",3000);
   System.out.println(al);
   Set sl=al.keySet();
   Iterator itr=sl.iterator();
   while(itr.hasNext())
   {
    System.out.print(itr.next()+"  ");
   }
   System.out.println();
```



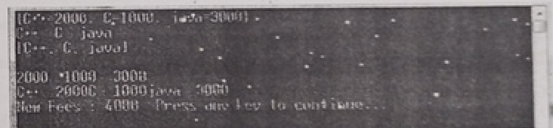
```
   System.out.println(s1);
   System.out.println();
   Collection c=al.values();
   Iterator itr2=c.iterator();
   while(itr2.hasNext())
   {
    System.out.print(itr2.next()+"  ");
   }
   System.out.println();

   Set s2=al.entrySet();
   Iterator itr3=s2.iterator();
   while(itr3.hasNext())
   {
    Map.Entry me=(Map.Entry)itr3.next();
    System.out.print(me.getKey()+"  ");
    System.out.print(me.getValue());
   }
   System.out.println();

   int fee=al.get("java");
   al.put("java",fee+1000);
   System.out.print("New Fees : "+al.get("java")+"  ");


  }
}
```

Output:



Differences

HashSet	HashMap
1. HashSet class implements the Set interface.	1. HashMap class implements the Map interface.
2. In HashSet we store objects (elements or values) e.g. If we have a HashSet of string elements then it could depict a set of HashSet elements: {"Hello", "Hi", "Bye", "Run"}	2. HashMap is used for storing key and value pairs. In short it maintains the mapping of key & value (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This is how you could represent HashMap elements if it has integer key and value of String type: e.g. {1->"Hello", 2->"Hi", 3->"Bye", 4->"Run"}

....table continued

HashSet	HashMap
3. HashSet does not allow duplicate elements that means you can not store duplicate values in HashSet.	3. HashMap does not allow duplicate keys however it allows to have duplicate values.
4. HashSet permits to have a single null value.	4. HashMap permits single null key and any number of null values.

Similarities

1. Both HashMap and HashSet are not synchronized which means they are not suitable for thread-safe operations unitl unless synchronized explicitly. This is how you can synchronize them explicitly:

HashSet

```
Set s = Collections.synchronizedSet(new HashSet(...));
```

HashMap

1. Map m = Collections.synchronizedMap(new HashMap(...));
2. Both of these classes do not guarantee that the order of their elements will remain constant over time.
3. If you look at the source code of HashSet then you may find that it is backed up by a HashMap. So basically it internally uses a HashMap for all of its operations.
4. They both provide constant time performance for basic operations such as adding, removing element etc.

16.6.4.2 Hashtable Class

HashMap and *Hashtable* both classes implements *java.util.Map* interface however there are differences in the way they work and their usage. Here we will discuss the differences between these classes.

HashMap vs Hashtable

1. *HashMap* is non-synchronized. This means if it's used in multithread environment then more than one thread can access and process the HashMap simultaneously.
   Hashtable is synchronized. It ensures that no more than one thread can access the Hashtable at a given moment of time. The thread which works on Hashtable acquires a lock on it to make the other threads wait till its work gets completed.
2. HashMap allows one null key and any number of null values. Hashtable doesn't allow null keys and null values.
3. HashMap implementation *LinkedHashMap* maintains the insertion order and *TreeMap* sorts the mappings based on the ascending order of keys. Hashtable doesn't guarantee any kind of order. It doesn't maintain the mappings in any particular order.
4. Initially Hashtable was not the part of *collection framework* it has been made a collection framework member later after being retrofitted to implement the Map interface.
   HashMap implements Map interface and is a part of collection framework since the beginning.
5. Another difference between these classes is that the Iterator of the HashMap is a fail-fast and it throws *ConcurrentModificationException* if any other Thread modifies the map structurally by adding or removing any element except iterator's own remove() method. In Simple words fail-fast means: When calling iterator.next() if any modification has been made between the moment the iterator was created and the moment next() is called a ConcurrentModificationException is immediately thrown.
   Enumerator for the Hashtable is not fail-fast.

Very Short Questions

1. What is collection framework?
2. What is adhoc classes in java.
3. Write elements of collection frame work in java.
4. What are uses Iterators?
5. What is use of for each loop?
6. What is use of HashSet class?

Short Questions

1. What is collection framework? Write down advantage collection frame over java adhoc classes.
2. Write note on collection hierarchy.
3. Explain for each loop with example.
4. What are differences between HashMap and HashTable ?classes.
5. What is difference between Stack and Queue class?

Long Questions

1. Explain java collection framework in details.

CHAPTER » 17

JDBC Architecture

17.1 JDBC

Java Database Connectivity (JDBC) is a Java API that allows programmers to access and work with relational database management systems from the Java programming language. JDBC stands for **Java Database Connectivity** which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

- Making a connection to a database.
- Creating SQL or MySQL statements.
- Executing SQL or MySQL queries in the database.
- Viewing & Modifying the resulting records.

17.2 JDBC Driver and JDBC Drivers Types

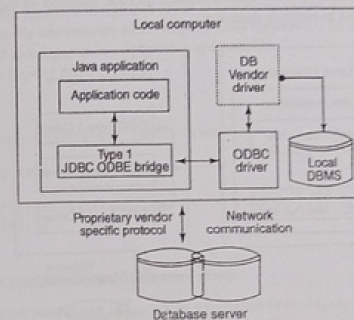
JDBC drivers implement the defined interfaces in the JDBC API, for interacting with your database server. Using JDBC drivers enable to open database connections and to interact with it by sending SQL or database commands then receiving results with Java.

JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates. Sun has divided the implementation types into four categories, Types 1, 2, 3, and 4, which is explained below:

17.2.1 Type 1: JDBC-ODBC Bridge Driver

In a Type 1 driver a JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC requires configuring on system a Data Source Name (DSN) that represents the target database.

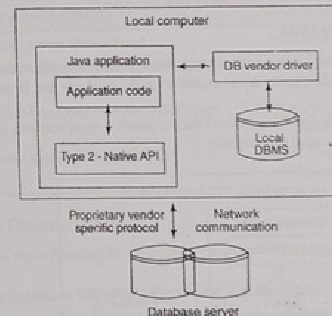
When Java first came out this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available.



17.2.2 Type 2: JDBC-Native API

In a Type 2 driver JDBC API calls are converted into native C/C++ 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. The vendor-specific driver must be installed on each client machine.

If we change the Database we have to change the native API as it is specific to a database and they are mostly obsolete but speed is increase with a Type 2 driver because it eliminates ODBC's overhead.

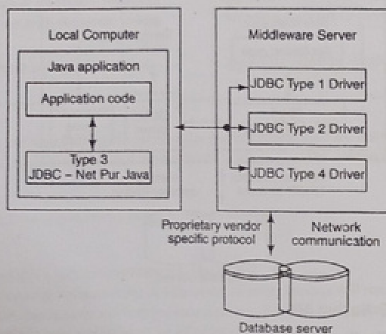


The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

17.2.3 Type 3: JDBC-Net Pure Java

In a Type 3 driver a three-tier approach is used to access databases. 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 and forwarded to the database server.

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.



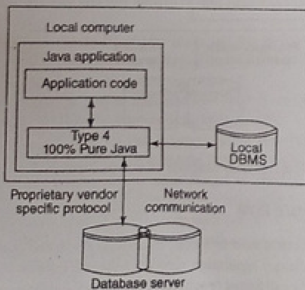
You can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result you need some knowledge of the application server's configuration in order to effectively use this driver type.

Your application server might use a Type 1, 2 or 4 drivers to communicate with the database.

17.2.4 Type 4: 100% Pure Java

In a Type 4 driver a pure Java-based driver communicates directly with the vendor's database through socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself.

This kind of driver is extremely flexible you don't need to install special software on the client or server. Further these drivers can be downloaded dynamically.



After installed the appropriate driver establish a database connection using JDBC is required. Here are these simple four steps:

Import JDBC Packages: Add **import** statements to Java program to import required classes in your Java code.
Register JDBC Driver: This step causes the JVM to load the desired driver implementation into memory so it can fulfill your JDBC requests.
Database URL Formulation: This is to create a properly formatted address that points to the database to which you wish to connect.
Create Connection Object: Finally, code a call to the *Driver Manager object's get Connection ()* method to establish actual database connection.

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as:

- Java Applications
- Java Applets
- Java Servlets
- Java ServerPages (JSPs)
- Enterprise JavaBeans (EJBs).

All of these different executables are able to use a JDBC driver to access a database, and take advantage of the stored data.

JDBC provides the same capabilities as ODBC allowing Java programs to contain database-independent code.

17.3 JDBC Architecture

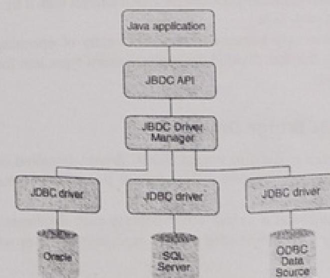
The JDBC API supports both two-tier and three-tier processing models for database access but in general JDBC Architecture consists of two layers:

- **JDBC API:** This provides the application-to-JDBC Manager connection.
- **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.

The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.

The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

Following is the architectural diagram which shows the location of the driver manager with respect to the JDBC drivers and the Java application:



17.4 Common JDBC Components

The JDBC API provides the following interfaces and classes:

1. **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.
2. **Driver:** This interface handles the communications with the database server and interact directly with Driver objects.
3. **Connection:** This interface with all methods for contacting a database. The connection object represents communication context i.e. all communication with database is through connection object only.
4. **Statement:** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
5. **ResultSet:** These objects hold data retrieved from a database after execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
6. **SQLException:** This class handles any errors that occur in a database application.

17.5 Creating JDBC Application

There are following six steps involved in building a JDBC application:

- **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.
- **Register the JDBC driver:** Requires that you initialize a driver so you can open a communication channel with the database.
- **Open a connection:** Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with the database.
- **Execute a query:** Requires using an object of type Statement for building and submitting an SQL statement to the database.
- **Extract data from result set:** Requires that you use the appropriate *ResultSet.getXXX()* method to retrieve the data from the result set.
- **Clean up the environment:** Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

17.5.1 Import JDBC Packages

The *Import* statements tell the Java compiler where to find the classes you reference in your code and are placed at the very beginning of your source code.

To use the standard JDBC package which allows you to select, insert, update and delete data in SQL tables add the following *imports* to your source code:

```
import java.sql.* ; // for standard JDBC programs
```

17.5.2 Register JDBC Driver

It is to must register the driver in your program before you use it. Registering the driver is the process by which the MySql driver's class file is loaded into the memory so it can be utilized as an implementation of the JDBC interfaces.

The most common approach to register a driver is to use Java's *Class.forName()* method to dynamically load the driver's class file into memory which automatically registers it. This method is preferable because it allows to make the driver registration configurable and portable.

Here is code for register driver:

```
try
{
  Class.forName("com.mysql.jdbc.Driver");
}
catch(ClassNotFoundException ex)
{
  System.out.println("Error: unable to load driver class!");
  System.exit(1);
}
```

17.5.3 Database URL Formulation

After loaded the driver connection can establish a using the *DriverManager.getConnection()* method. For easy reference there are three overloaded DriverManager.getConnection() methods:

```
getConnection(String url)
getConnection(String url, Properties prop)
getConnection(String url, String user, String password)
```

Here each form requires a database *URL*. A database URL is an address that points to database. Formulating a database URL is where most of the problems associated with establishing a connection occurs.

Following table lists down the popular JDBC driver names and database URL.

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/database Name
Sybase	com.sybase.jdbc.SybDriver	jdbc:sybase:Tds:hostname: port Number/ databaseName

17.5.4 Create Connection Object

We have listed down three forms of *DriverManager.getConnection()* method to create a connection object.

17.5.4.1 Using a Database URL with a Username and Password

The most commonly used form of getConnection() requires to pass a database URL, a *username* and a *password*:

Using MySql's *thin* driver specify a jdbc:mysql://hostname/ databaseName value for the database portion of the URL.

Call getConnection() method with appropriate username and password to get a *Connection* object as follows:

```
String URL = " jdbc:mysql://hostname/ databaseName ";
String USER = "username";
String PASS = "password"
Connection conn = DriverManager.getConnection(URL, USER, PASS);
```

17.5.4.2 Using Only a Database URL

A second form of the DriverManager.getConnection( ) method requires only a database URL:

```
DriverManager.getConnection(String url);
```

However in this case the database URL includes the username and password and has the following general form:

So the above connection can be created as follows:

```
String URL = " jdbc:mysql://hostname/ databaseName:username/password@amrood:1521:EMP";

Connection conn = DriverManager.getConnection(URL);
```

17.5.4.3 Using a Database URL and a Properties Object

A third form of the DriverManager.getConnection( ) method requires a database URL and a Properties object:

```
DriverManager.getConnection(String url, Properties info);
```

A Properties object holds a set of keyword-value pairs. It is used to pass driver properties to the driver during a call to the getConnection() method.

To make the same connection made by the previous examples use the following code:

```
import java.util.*;
String URL = " jdbc:mysql://hostname/ databaseName ";
Properties info = new Properties( );
info.put( "user", "username" );
info.put( "password", "password" );
Connection conn = DriverManager.getConnection(URL, info);
```

17.5.5 Closing JDBC Connections

At the end of JDBC program it is required explicitly to close all the connections to the database to end each database session. However if we forget Java's garbage collector will close the connection when it cleans up state objects.

Relying on the garbage collection especially in database programming, is a very poor programming practice. It should make a habit of always closing the connection with the close() method associated with connection object.

To ensure that a connection is closed, it is better to provide a 'finally' block in code. A *finally* block always executes regardless of an exception occurs or not.

To close the above opened connection, you should call close() method as follows:

```
conn.close();
```

Explicitly closing a connection conserves DBMS resources.

Once a connection is obtained we can interact with the database. The JDBC *Statement*, *CallableStatement*, and *PreparedStatement* interfaces define the methods and properties that enable to send SQL or PL/SQL commands and receive data from your database. They also define methods that help bridge data type differences between Java and SQL data types used in a database.

The following table provides a summary of each interface's purpose to decide on the interface to use:

Interfaces	Recommended Use
Statement	Use this for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use this when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use this when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

17.5.6 Creating Statement Object

Before use a Statement object to execute a SQL statement it need to create the Connection object's createStatement( ) method as in the following example:

```
Statement stmt = null;
try
{
  stmt = conn.createStatement( );
. . .
}
catch (SQLException e)
{
. ..
}
finally
{
  . . .
}
```

Once Statement object is created then use it to execute an SQL statement with one of its three execute methods.

- **boolean execute (String SQL):** Returns a boolean value of true if a ResultSet object can be retrieved; otherwise it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.
- **int executeUpdate (String SQL):** Returns the number of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example an INSERT, UPDATE or DELETE statement.
- **ResultSet executeQuery (String SQL):** Returns a ResultSet object. Use this method when you expect to get a result set as you would with a SELECT statement.

17.5.7 Closing Statement Object

As closing a Connection object to save database resources for the same reason it should also close the Statement object.

The close() method is used to close the Statement. If we are close the Connection object first it will close the Statement object as well. However it should always explicitly close the Statement object to ensure proper cleanup.

```
Statement stmt = null;
try
```

```
{
  stmt = conn.createStatement( );
  . . .
}
catch (SQLException e)
{
  . . .
}
finally
{
  Stmt.close();
}
```

xample 1:

llowing is the example which makes use of the following three queries along with the opening and closing atment:

```
//STEP 1. Import required packages
import java.sql.*;
public class JDBCExample
{
  // JDBC driver name and database URL
  static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
  static final String DB_URL = "jdbc:mysql://localhost/EMP";

  // Database credentials
  static final String USER = "username";
  static final String PASS = "password";
  public static void main(String[] args)
  {
    Connection conn = null;
    Statement stmt = null;
    try
    {
    //STEP 2: Register JDBC driver
    Class.forName("com.mysql.jdbc.Driver");

    //STEP 3: Open a connection
    System.out.println("Connecting to database...");
    conn = DriverManager.getConnection(DB_URL,USER,PASS);

    //STEP 4: Execute a query
    System.out.println("Creating statement...");
    stmt = conn.createStatement();
    String sql = "UPDATE Employees set age=30 WHERE id=103";

    // Let us check if it returns a true Result Set or not.
    Boolean ret = stmt.execute(sql);
    System.out.println("Return value is : " + ret.toString() );

    // Let us update age of the record with ID = 103;
    int rows = stmt.executeUpdate(sql);
```



```
    System.out.println("Rows impacted : " + rows );
    // Let us select all the records and display them.
    sql = "SELECT id, first, last, age FROM Employees";
    ResultSet rs = stmt.executeQuery(sql);

    //STEP 5: Extract data from result set
    while(rs.next())
    {
      //Retrieve by column name
      int id = rs.getInt("id");
      int age = rs.getInt("age");
      String first = rs.getString("first");
      String last = rs.getString("last");

      //Display values
      System.out.print("ID: " + id);
      System.out.print(", Age: " + age);
      System.out.print(", First: " + first);
      System.out.println(", Last: " + last);
    }

    //STEP 6: Clean-up environment
    rs.close();
    stmt.close();
    conn.close();
   }
   catch(SQLException se)
   {

     //Handle errors for JDBC
     se.printStackTrace();
   }
   catch(Exception e)
   {
     //Handle errors for Class.forName
     e.printStackTrace();
   }
   finally
   {
     //finally block used to close resources
     try
     {
       if(stmt!=null)
       stmt.close();
     }catch(SQLException se2)
     {
     }
     // nothing we can do
     try
     {
       if(conn!=null)
       conn.close();
```

```
    }
    catch(SQLException se)
    {
      se.printStackTrace();
    }
    //end finally try
    }//end try
    System.out.println("End");
}//end main
```

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. The *java.sql.ResultSet* interface represents the result set of a database query.

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 methods of the ResultSet interface can be broken down into three categories:

- **Navigational methods:** Used to move the cursor around.
- **Get methods:** Used to view the data in the columns of the current row being pointed by the cursor.
- **Update methods:** Used to update the data in the columns of the current row. The updates can then be updated in the underlying database as well.

The cursor is movable based on the properties of the ResultSet. These properties are designated when the corresponding Statement that generates the ResultSet is created.

JDBC provides the following connection methods to create statements with desired ResultSet:

- createStatement(int RSType, int RSConcurrency);
- prepareStatement(String SQL, int RSType, int RSConcurrency);
- prepareCall(String sql, int RSType, int RSConcurrency);

The first argument indicates the type of a ResultSet object and the second argument is one of two ResultSet constants for specifying whether a result set is read-only or updatable.

17.5.8 Type of ResultSet

The possible ResultSet type are given below. If ResultSet type is not specify then TYPE_FORWARD_ONLY ResultSet type will automatically apply.

Type	Description
ResultSet.TYPE_FORWARD_ONLY	The cursor can only move forward in the ResultSet
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.

There are several methods in the ResultSet interface that involve moving the cursor including:

S.N.	Methods & Description
1	**public void beforeFirst() throws SQLException** Moves the cursor just before the first row.
2	**public void afterLast() throws SQLException** Moves the cursor just after the last row.
3	**public boolean first() throws SQLException** Moves the cursor to the first row.
4	**public void last() throws SQLException** Moves the cursor to the last row.
5	**public boolean absolute(int row) throws SQLException** Moves the cursor to the specified row.
6	**public boolean relative(int row) throws SQLException** Moves the cursor the given number of rows forward or backward from where it is currently pointing.
7	**public boolean previous() throws SQLException** Moves the cursor to the previous row. This method returns false if the previous row is off the result set.
8	**public boolean next() throws SQLException** Moves the cursor to the next row. This method returns false if there are no more rows in the result set.
9	**public int getRow() throws SQLException** Returns the row number that the cursor is pointing to.
10	**public void moveToInsertRow() throws SQLException** Moves the cursor to a special row in the result set that can be used to insert a new row into the database. The current cursor location is remembered.
11	**public void moveToCurrentRow() throws SQLException** Moves the cursor back to the current row if the cursor is currently at the insert row; otherwise this method does nothing.

17.5.9 Viewing a ResultSet

The ResultSet interface contains many methods for getting the data of the current row. There is a *get* method for each of the possible data types and each *get* method has two versions:

- One that takes in a column name.
- One that takes in a column index.

For example if the column you are interested in viewing contains an int you need to use one of the getInt() methods of ResultSet:

S.N.	Methods & Description
1	**public int getInt(String columnName) throws SQLException** Returns the int in the current row in the column named *columnName*.
2	**public int getInt(int columnIndex) throws SQLException** Returns the int in the current row in the specified column index. The column index starts at 1, meaning the first column of a row is 1, the second column of a row is 2 and so on.

17.5.10 Updating a Result Set

The ResultSet interface contains a collection of update methods for updating the data of a result set.

As with the get methods there are two update methods for each data type:

- One that takes in a column name.
- One that takes in a column index.

For example to update a String column of the current row of a result set you would use one of the following updateString() methods:

S.N.	Methods & Description
1	**public void updateString(int columnIndex, String s) throws SQLException** Changes the String in the specified column to the value of String s.
2	**public void updateString(String columnName, String s) throws SQLException** Similar to the previous method except that the column is specified by its name instead of its index.

■ ■ ■

Very Short Questions

1. Define JDBC.
2. Write the package name of JDBC classes.
3. What is JDBC driver?
4. What is JDBC API?
5. What is JDBC driver API?

Short Questions

1. What is JDBC? Write uses of JDBC in java.
2. Define JDBC drivers. List the type of JDBC drivers.
3. Explain common components of JDBC.
4. Write down steps to create JDBC Application.
5. Write functions of forName(" ") method.
6. Write java code to create connection.

Long Questions

1. What do you mean by JDBC Drivers? Explain all types of drivers in details.
2. Explain steps to create JDBC Application in detail with java code.