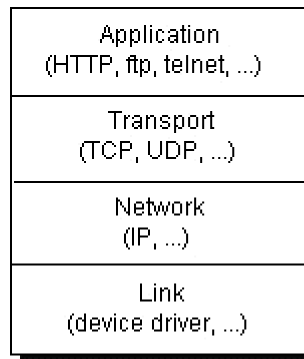
Unit-III

NETWORKING

Data Networking continues to evolve. The demand for a High-Speed Network Infrastructure has been growing at an alarming rate. Just a few short years ago, 4 Mbps (Million bits per second) **Token Ring** and 10 Mbps **Ethernet** shared networks were the norm. Now they can't keep up with the growing demands from end-users. End-user applications and files are growing in size and number.

TCP AND UDP

Computers running on the Internet communicate to each other using either the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP), as this diagram illustrates:



When you write Java programs that communicate over the network, you are programming at the application layer. Typically, you don't need to concern yourself with the TCP and UDP layers. Instead, you can use the classes in the `java.net` package. These classes provide system-independent network communication. However, to decide which Java classes your programs should use, you do need to understand how TCP and UDP differ.

TCP

When two applications want to communicate to each other reliably, they establish a connection and send data back and forth over that connection. This is analogous to making a telephone call. If you want to speak to Aunt Beatrice in Kentucky, a connection is established when you dial her phone number and she answers. You send data back and forth over the connection by speaking to one another over the phone lines. Like the phone company, TCP guarantees that data sent from one end of the connection actually gets to the other end and in the same order it was sent. Otherwise, an error is reported.

TCP provides a point-to-point channel for applications that require reliable communications. The Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), and Telnet are all examples of

applications that require a reliable communication channel. The order in which the data is sent and received over the network is critical to the success of these applications. When HTTP is used to read from a URL, the data must be received in the order in which it was sent. Otherwise, you end up with a jumbled HTML file, a corrupt zip file, or some other invalid information.

UDP

The UDP protocol provides for communication that is not guaranteed between two applications on the network. UDP is not connection-based like TCP. Rather, it sends independent packets of data, called *datagrams*, from one application to another. Sending datagrams is much like sending a letter through the postal service: The order of delivery is not important and is not guaranteed, and each message is independent of any other.

For many applications, the guarantee of reliability is critical to the success of the transfer of information from one end of the connection to the other. However, other forms of communication don't require such strict standards. In fact, they may be slowed down by the extra overhead or the reliable connection may invalidate the service altogether.

Consider, for example, a clock server that sends the current time to its client when requested to do so. If the client misses a packet, it doesn't really make sense to resend it because the time will be incorrect when the client receives it on the second try. If the client makes two requests and receives packets from the server out of order, it doesn't really matter because the client can figure out that the packets are out of order and make another request. The reliability of TCP is unnecessary in this instance because it causes performance degradation and may hinder the usefulness of the service.

Another example of a service that doesn't need the guarantee of a reliable channel is the ping command. The purpose of the ping command is to test the communication between two programs over the network. In fact, ping needs to know about dropped or out-of-order packets to determine how good or bad the connection is. A reliable channel would invalidate this service altogether.

The UDP protocol provides for communication that is not guaranteed between two applications on the network. UDP is not connection-based like TCP. Rather, it sends independent packets of data from one application to another. Sending datagrams is much like sending a letter through the mail service: The order of delivery is not important and is not guaranteed, and each message is independent of any others.

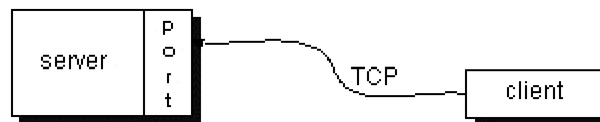
Understanding Ports

Generally speaking, a computer has a single physical connection to the network. All data destined for a particular computer arrives through that connection. However, the data may be intended for different

applications running on the computer. So how does the computer know to which application to forward the data? Through the use of *ports*.

Data transmitted over the Internet is accompanied by addressing information that identifies the computer and the port for which it is destined. The computer is identified by its 32-bit IP address, which IP uses to deliver data to the right computer on the network. Ports are identified by a 16-bit number, which TCP and UDP use to deliver the data to the right application.

In connection-based communication such as TCP, a server application binds a socket to a specific port number. This has the effect of registering the server with the system to receive all data destined for that port. A client can then rendezvous with the server at the server's port, as illustrated here:

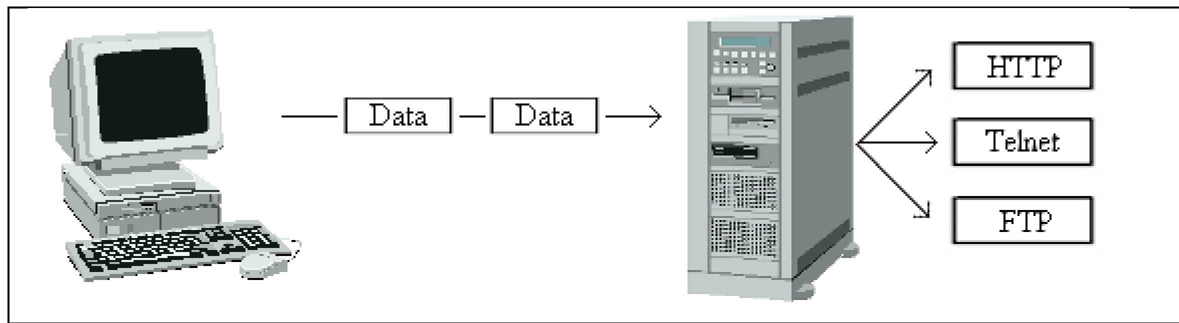


NETWORKING WITH JAVA

Network access is crucial to computer operations in the twenty first century. Java provides many built-in networking class objects through its *.net, .nio* and *.rmi* packages. *java.net* provides http connections and streams as well as protocol sockets. *java.nio*, *java.nio.charset* and *java.nio.channels* provide buffers, character sets and channels for multiplexed non-blocking applications that are more tolerant of dropped connections and time delays. *java.rmi* provides methods for remote method invocation

The Internet is composed of millions of computers, located all across the globe, communicating and transmitting information over a variety of computing systems, platforms, and networking equipment. Each of these computers (unless they are connecting via an intranet) will have a unique IP address.

Often, computers connected to the Internet provide services. This page is provided by a web server, for example. Because computers are capable of providing more than one type of service, we need a way to uniquely identify each service. Like an IP address, we use a number. We call this number a port. Common services (such as HTTP, FTP, Telnet, SMTP) have well known port numbers. For example, most web servers use port 80. Of course, you can use any port you like - there's no rule that says you *must* use 80.



Ports help computers identify which service data is for.

Handling internet addresses (domain names, and IP addresses) is made easy with Java. Internet addresses are represented in Java by the InetAddress class. InetAddress provides simple methods to convert between domain names, and numbered addresses.

CLIENT / SERVER

You often hear the term client/server mentioned in the context of networking. It seems complicated when you read about it in corporate marketing statements, but it is actually quite simple. A server is anything that has some resource that can be shared. There are compute servers, which provide computing power; print servers, which manage a collection of printers; disk servers, which provide networked disk space; and web servers, which store web pages. A client is simply any other entity that wants to gain access to a particular server. The interaction between client and server is just like the interaction between a lamp and an electrical socket. The power grid of the house is the server, and the lamp is a power client. The server is a permanently available resource, while the client is free to “unplug” after it is has been served. in Berkeley sockets, the notion of a socket allows a single computer to serve many different clients at once, as well as serving many different types of information. This feat is managed by the introduction of a port, which is a numbered socket on a particular machine. A server process is said to “listen” to a port until a client connects to it. A server is allowed to accept multiple clients connected to the same port number, although each session is unique. To manage multiple client connections, a server process must be multithreaded or have some other means of multiplexing the simultaneous I/O .

EXAMPLE:

```
import java.net.InetAddress;

import java.net.UnknownHostException;


/*
```

```
 * Main.java
 *
 * @author R.G.Gupta
 */
public class Main {

  /*
   * This method performs a NS Lookup
   */
  public void performNSLookup() {

     try {

        InetAddress inetHost = InetAddress.getByName("cnn.com");
        String hostName = inetHost.getHostName();
        System.out.println("The host name was: " + hostName);
        System.out.println("The hosts IP address is: " + inetHost.getHostAddress());

     } catch(UnknownHostException ex) {

        System.out.println("Unrecognized host");
     }
  }
  /**
   * @param args the command line arguments
```

```
  */
    public static void main(String[] args) {
      new Main().performNSLookup();
    }
  }
```

SOCKET CLASS

This class implements client sockets (also called just "sockets"). A socket is an endpoint for communication between two machines.

The actual work of the socket is performed by an instance of the `SocketImpl` class. An application, by changing the socket factory that creates the socket implementation, can configure itself to create sockets appropriate to the local firewall.

A network socket is a lot like an electrical socket. Various plugs around the network have a standard way of delivering their payload. Anything that understands the standard protocol can "plug in" to the socket and communicate. With electrical sockets, it doesn't matter if you plug in a lamp or a toaster; as long as they are expecting 60Hz, 115-volt electricity, the devices will work. Think how your electric bill is created. There is a metersomewhere between your house and the rest of the network. For each kilowatt of power that goes through that meter, you are billed. The bill comes to your "address." So even though the electricity flows freely around the power grid, all of the sockets in your house have a particular address. The same idea applies to network sockets, except we talk about TCP/IP packets and IP addresses rather than electrons and street addresses. Internet Protocol (IP) is a low-level routing protocol that breaks data into small packets and sends them to an address across a network, which does not guarantee to deliver said packets to the destination. Transmission Control Protocol (TCP) is a higher-level protocol that manages to robustly string together these packets, sorting and retransmitting them as necessary to reliably transmit your data. A third protocol, User Datagram Protocol (UDP), sits next to TCP and can be used directly to support fast, connectionless, unreliable transport of packets.

CONSTRUCTORS:

1) `protected` **`Socket`**`()`

Creates an unconnected socket, with the system-default type of SocketImpl.

2) `protected` **`Socket`**`(SocketImpl impl) throws SocketException`

Creates an unconnected Socket with a user-specified SocketImpl.

Parameters:

`impl` - an instance of a **SocketImpl** the subclass wishes to use on the Socket.

Throws:

`SocketException` - if there is an error in the underlying protocol, such as a TCP error

```
3) public Socket(String host,
             int port)
       throws UnknownHostException,
             IOException
```

Creates a stream socket and connects it to the specified port number on the named host.

If the application has specified a server socket factory, that factory's `createSocketImpl` method is called to create the actual socket implementation. Otherwise a "plain" socket is created.

If there is a security manager, its `checkConnect` method is called with the host address and `port` as its arguments. This could result in a SecurityException.

Parameters:

`host` - the host name.

`port` - the port number.

Throws:

`UnknownHostException` - if the IP address of the host could not be determined.

`IOException` - if an I/O error occurs when creating the socket.

`SecurityException` - if a security manager exists and its `checkConnect` method doesn't allow the operation

```
4) public Socket(InetAddress address,
             int port)
       throws IOException
```

Creates a stream socket and connects it to the specified port number at the specified IP address.

If the application has specified a socket factory, that factory's `createSocketImpl` method is called to create the actual socket implementation. Otherwise a "plain" socket is created.

If there is a security manager, its `checkConnect` method is called with the host address and `port` as its arguments. This could result in a SecurityException.

Parameters:

`address` - the IP address.

`port` - the port number.

Throws:

`IOException` - if an I/O error occurs when creating the socket.

`SecurityException` - if a security manager exists and its `checkConnect` method doesn't allow the operation.

```
5) public Socket(InetAddress host,
              int port,
              boolean stream)
       throws IOException
```

Deprecated. *Use DatagramSocket instead for UDP transport.*

Creates a socket and connects it to the specified port number at the specified IP address.

If the stream argument is `true`, this creates a stream socket. If the stream argument is `false`, it creates a datagram socket.

If the application has specified a server socket factory, that factory's `createSocketImpl` method is called to create the actual socket implementation. Otherwise a "plain" socket is created.

If there is a security manager, its `checkConnect` method is called with `host.getHostAddress()` and `port` as its arguments. This could result in a SecurityException.

Parameters:

`host` - the IP address.

`port` - the port number.

`stream` - if `true`, create a stream socket; otherwise, create a datagram socket.

Throws:

`IOException` - if an I/O error occurs when creating the socket.

`SecurityException` - if a security manager exists and its `checkConnect` method doesn't allow the operation.

EXAMPLE:

This example introduces you to Java socket programming. The server listens for a connection. When a connection is established by a client. The client can send data. In the current example the client sends the message "Hi my server". To terminate the connection, the client sends the message "bye". Then the server sends the message "bye" too. Finally the connection is ended and the server waits for an other connection. The two programs should be runned in the same machine. however if you want to run them in two different machines, you may simply change the adress "localhost" by the IP adress of the machine where you will run the server.

The server:-

```
import java.io.*;
import java.net.*;
public class Provider{
        ServerSocket providerSocket;
        Socket connection = null;
        ObjectOutputStream out;
        ObjectInputStream in;
        String message;
        Provider(){}
        void run()
        {
                try{
                        //1. creating a server socket
                        providerSocket = new ServerSocket(2004, 10);
                        //2. Wait for connection
                        System.out.println("Waiting for connection");
                        connection = providerSocket.accept();
                        System.out.println("Connection received from " +
connection.getInetAddress().getHostName());
                        //3. get Input and Output streams
                        out = new
ObjectOutputStream(connection.getOutputStream());
                        out.flush();
                        in = new
ObjectInputStream(connection.getInputStream());
                        sendMessage("Connection successful");
                        //4. The two parts communicate via the input and output
streams
                        do{
                                try{
                                        message = (String)in.readObject();
                                        System.out.println("client>" +
message);
                                        if (message.equals("bye"))
                                                sendMessage("bye");
                                }
```

```
                                catch(ClassNotFoundException classnot){
                                        System.err.println("Data received in
unknown format");
                                }
                        }while(!message.equals("bye"));
                }
                catch(IOException ioException){
                        ioException.printStackTrace();
                }
                finally{
                        //4: Closing connection
                        try{
                                in.close();
                                out.close();
                                providerSocket.close();
                        }
                        catch(IOException ioException){
                                ioException.printStackTrace();
                        }
                }
        }
        void sendMessage(String msg)
        {
                try{
                        out.writeObject(msg);
                        out.flush();
                        System.out.println("server>" + msg);
                }
                catch(IOException ioException){
                        ioException.printStackTrace();
                }
        }
        public static void main(String args[])
        {
                Provider server = new Provider();
                while(true){
                        server.run();
                }
        }
}
```

The client:-

```
import java.io.*;
import java.net.*;
public class Requester{
        Socket requestSocket;
        ObjectOutputStream out;
        ObjectInputStream in;
        String message;
        Requester(){}
        void run()
        {
                try{
                        //1. creating a socket to connect to the server
                        requestSocket = new Socket("localhost", 2004);
```

```
                        System.out.println("Connected to localhost in port
2004");
                        //2. get Input and Output streams
                        out = new
ObjectOutputStream(requestSocket.getOutputStream());
                        out.flush();
                        in = new
ObjectInputStream(requestSocket.getInputStream());
                        //3: Communicating with the server
                        do{
                                try{
                                        message = (String)in.readObject();
                                        System.out.println("server>" +
message);
                                        sendMessage("Hi my server");
                                        message = "bye";
                                        sendMessage(message);
                                }
                                catch(ClassNotFoundException classNot){
                                        System.err.println("data received in
unknown format");
                                }
                        }while(!message.equals("bye"));
                }
                catch(UnknownHostException unknownHost){
                        System.err.println("You are trying to connect to an
unknown host!");
                }
                catch(IOException ioException){
                        ioException.printStackTrace();
                }
                finally{
                        //4: Closing connection
                        try{
                                in.close();
                                out.close();
                                requestSocket.close();
                        }
                        catch(IOException ioException){
                                ioException.printStackTrace();
                        }
                }
        }
        void sendMessage(String msg)
        {
                try{
                        out.writeObject(msg);
                        out.flush();
                        System.out.println("client>" + msg);
                }
                catch(IOException ioException){
                        ioException.printStackTrace();
                }
        }
        public static void main(String args[])
```

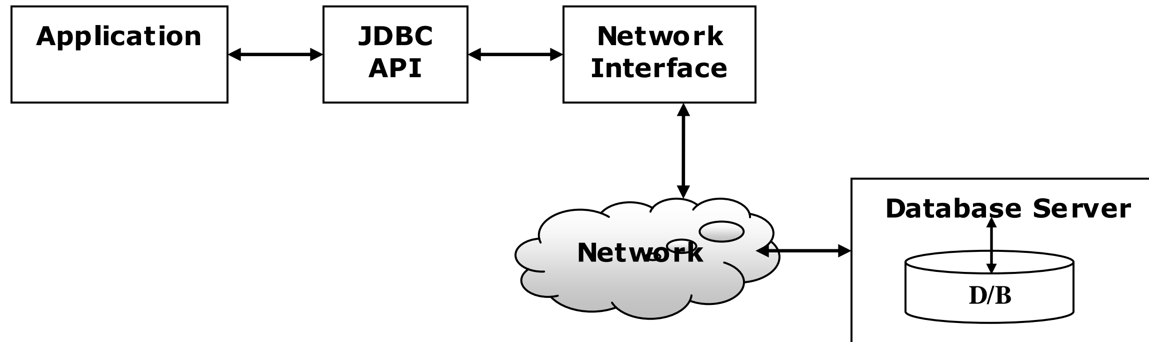
```
        {
                Requester client = new Requester();
                client.run();
        }
}
```

JDBC

Java Database Connectivity or in short JDBC is a technology that enables the java program to manipulate data stored into the database. It was developed by JavaSoft, a subsidiary of Sun Microsystems

With the "write once, compile once, run anywhere" power that JDBC offers us, Java's database connectivity allows us to concentrate on the translation of relational data into objects instead of how we can get that data from the database.

JDBC enables Java programs to execute SQL statements. This allows Java programs to interact with any SQL-compliant database. Since nearly all relational database management systems (DBMSs) support SQL, and because Java itself runs on most platforms, JDBC makes it possible to write a single database application that can run on different platforms and interact with different DBMSs.



JDBC is a Java Database Connectivity API that lets you access virtually any tabular data source from a Java application. In addition to providing connectivity to a wide range of SQL databases, JDBC allows you to access other tabular data sources such as spreadsheets or flat files. Although JDBC is often thought of as an acronym for Java Database Connectivity, the trademarked API name is actually JDBC.

JDBC Drivers

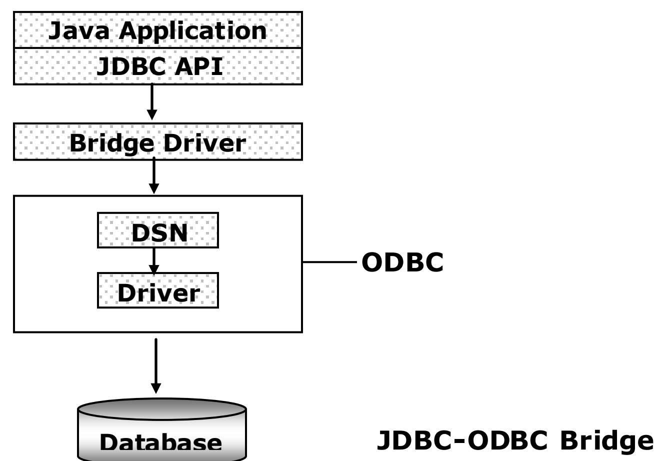
The JDBC API contains two major sets of interfaces: the first is the JDBC API for application writers, and the second is the lower-level JDBC driver API for driver writers. JDBC technology drivers fit into one of four categories.

Proprietary Database drivers:

Bridge drivers: The JDBC-ODBC Bridge driver is recommended only for experimental use or when no other alternative is available.

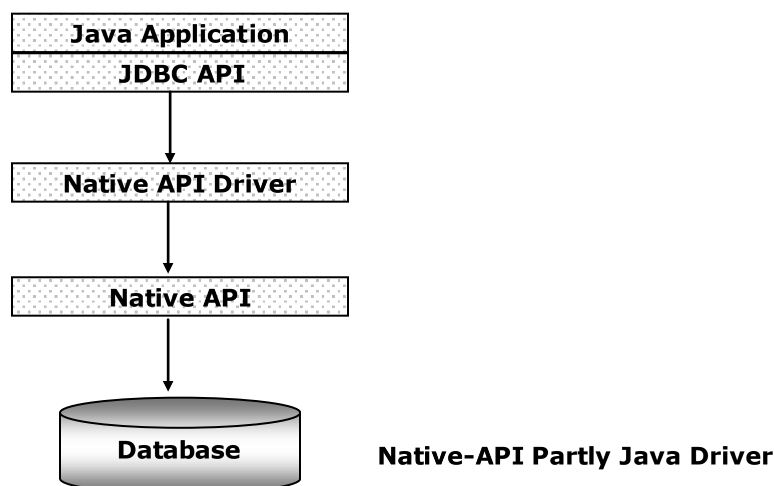
DBMS-independent all-Java net-drivers: These are written in 100% Java and use vendor independent Net-protocol to access a vendor independent remote listener.

1. A ***JDBC-ODBC BRIDGE*** provides JDBC API access via one or more ODBC drivers. Note that some ODBC native code and in many cases native database client code must be loaded on each client machine that uses this type of driver. Hence, this kind of driver is generally most appropriate when automatic installation and downloading of a Java technology application is not important.



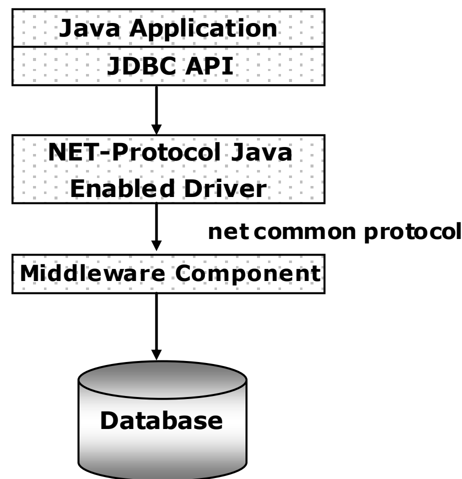
JDBC-ODBC Bridge

2. A ***NATIVE-API PARTLY JAVA TECHNOLOGY-ENABLED DRIVER*** converts JDBC calls into calls on the client API for Oracle, Sybase, Informix, DB2, or other DBMS. Note that, like the bridge driver, this style of driver requires that some binary code be loaded on each client machine.



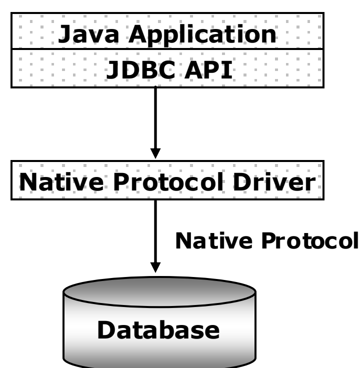
Native-API Partly Java Driver

3. A ***NET-PROTOCOL FULLY JAVA TECHNOLOGY-ENABLED DRIVER*** translates JDBC API calls into a DBMS-independent net protocol, which is then translated, to a DBMS protocol by a server. This net server middleware is able to connect all of its Java technology-based clients to many different databases. The specific protocol used depends on the vendor. In general, this is the most flexible JDBC API alternative. It is likely that all vendors of this solution will provide products suitable for Intranet use. In order for these products to also support Internet access they must handle the additional requirements for security, access through firewalls, etc., that the Web imposes. Several vendors are adding JDBC technology-based drivers to their existing database middleware products.



NET-Protocol Fully Java Enabled Driver

4. A ***NATIVE-PROTOCOL FULLY JAVA TECHNOLOGY-ENABLED DRIVER*** converts JDBC technology calls into the network protocol used by DBMSs directly. This allows a direct call from the client machine to the DBMS server and is a practical solution for Intranet access. Since many of these protocols are proprietary the database vendors themselves will be the primary source for this style of driver. Several database vendors have these in progress.



Native-Protocol Fully Java Technology-Enabled Driver

Creating Connection

A JDBC application connects to a target data source using one of two mechanisms:

1. **DriverManager:** This fully implemented class requires an application to load a specific driver, using a hardcoded URL. As part of its initialization, the DriverManager class attempts to load the driver classes referenced in the jdbc.drivers system property. This allows you to customize the JDBC Drivers used by your applications.
2. **DataSource:** This interface is preferred over DriverManager because it allows details about the underlying data source to be transparent to your application. A DataSource object's properties are set so that it represents a particular data source.

Establishing a connection involves two steps:

- Loading the driver
- Making the connection.

1. In order to connect to a database, JDBC driver has to be loaded by the Java Virtual Machine classloader, and your application needs to check to see that the driver was successfully loaded. Here we'll be using the ODBC bridge driver, but if your database vendor supplies a JDBC driver, feel free to use it instead.

```
// load the JDBC driver
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

2. Once our driver is loaded, we can connect to the database. We'll connect via the DriverManager class, which selects the appropriate driver for the database we specify. In this case, we'll only be using an ODBC database, but in more complex applications, we might wish to use different drivers to connect to multiple databases. We identify our database through a URL.
   i. A JDBC URL starts with "jdbc:" This indicates the protocol (JDBC). We also specify our database in the URL. As an example, here's the URL for an ODBC datasource called 'demo'.

```
// Create a URL that identifies database
String url = " jdbc:odbc:demo“;
```

   ii. To connect to the database,

```
// Now attempt to create a database connection
Connection db_conn =
        DriverManager.getConnection (url, "user", "password");
```

Building Statements

In JDBC, a Statement object is used to execute queries. A Statement object is responsible for sending the SQL statement, and returning a set of results, if needed, from the query. Statement objects support two main types of statements - an update statement that is normally used for operations, which don't generate a response, and a query statement that returns data.

```
// Create a statement to send SQL
Statement db_stmt = db_conn.createStatement();
```

Once an instance of a statement object has been created, you can call its executeUpdate and executeQuery methods. To illustrate the executeUpdate command, we'll create a table that stores information about student. To keep things simple and limit it to name and student rollno.

```
// Create a simple table, which stores an employee ID and name
db_stmt.executeUpdate ("create table student (rollno number(4), name varchar(50))");
// Insert an employee, so the table contains data
db_stmt.executeUpdate ("insert into student values (1, 'ABC')");
// Commit changes
db_conn.commit();
```

Now that there's data in the table, we can execute queries. The response to a query will be returned by the executeQuery method as a ResultSet object.

Handling Results

ResultSet objects store the last response to a query for a given statement object. Instances of ResultSet have methods following the pattern of getXX where XX is the name of a data type. Such data types include numbers (bytes, ints, shorts, longs, doubles, big-decimals), as well as strings, booleans, timestamps and binary data.

```
// Execute query
ResultSet result = db_stmt.executeQuery
                                        ("select * from student");

// While more rows exist, print them
while (result.next() )
{
          // Use the getInt method to obtain emp. id
          System.out.println ("ID : " + result.getInt("rollno"));

          // Use the getString method to obtain emp. name
```

How to Create Data Source Name (DSN)

Creating a new ODBC Data Source Name (DSN) is a straight-forward operation. Using the OBDC Data Source Administrator, you can create User, System, and File DSNs as you need them.

Data Source Name Categories

ODBC has three different categories or types of DSNs:

- User DSN
- System DSN
- File DSN

Each DSN category serves a specific purpose and has a specific scope.

User DSN

A user DSN is just that, a DSN for a specific user. If you create a user DSN under your user account, no other user can see it or use it. The DSN is for me and me alone. If you need a connection to a data source that only you should use, choose a user DSN.

System DSN

A system DSN is a DSN that is seen by the entire system. Any user can see it, as well as any process or service. If you need a data source connection that should be seen more than just your user account, choose to use a system DSN. This is especially true if you are trying to establish a connection through IIS or some other service.

File DSN

A file DSN is simply where the connection settings are written to a file. The reason for having a file DSN is if you want to distribute a data source connection to multiple users on different systems without having to configure a DSN for each system. For instance, you can create a file DSN to a reporting database on your desktop. you can then send the file to your users. your users can save the file DSN to their hard drives and then point their reporting applications at the file DSN.

Steps to Create a DSN:

- Click **Start**, point to **Control Panel**, double-click **Administrative Tools**, and then double-click **Data Sources(ODBC)**.
- Click the **System DSN** tab, and then click **Add**.
- Click the database driver that corresponds with the database type to which you are connecting, and then click **Finish**.
- Type the data source name (e.g demo). Make sure that you choose a name that you can remember. You will need to use this name later.
- Click **Select**.
- Click the correct database, and then click **OK**.

- Click **OK**, and then click **OK**.

Putting All Concepts together:

```
import java.sql.*;
class JdbcExample
{
public static void main(String args[]) throws Exception
    {
        // load the JDBC driver
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        // Create a URL that identifies database
        String url = " jdbc:odbc:demo";
        // Now attempt to create a database connection
        Connection db_conn =
         DriverManager.getConnection (url, "user", "password");
        // Create a statement to send SQL
        Statement db_stmt = db_conn.createStatement();
        // Create a simple table, which stores an employee ID and name
        db_stmt.executeUpdate ("create table student (rollno number(4), name
        varchar(50))");
        // Insert an employee, so the table contains data
        db_stmt.executeUpdate ("insert into student values (1, 'ABC')");
        // Commit changes
        db_conn.commit();
        // Execute query
        ResultSet result = db_stmt.executeQuery
                                   ("select * from student");

        // While more rows exist, print them
        while (result.next() )
        {
            // Use the getInt method to obtain emp. id
            System.out.println ("ID : " + result.getInt("rollno"));

            // Use the getString method to obtain emp. name
            System.out.println ("Name : " + result.getString("name"));
            System.out.println ();

         }
            result.close();
            db_stmt.close();
            db_conn.close();
     }
  }
```