T. Y. B. C. A.

Semester VI

Subject Name -: Advance Java

Course Code -: 602

Prepared By Prof.Bhujbal V.N.

Unit No.	Topic
1.	**JDBC**
1.1	The design of JDBC
1.2	Basic JDBS program Concept
1.3	Drivers
1.4	Making the Connection, Statement , ResultSet
1.5	Executing SQL commands
1.6	Executing queries
2	**Networking**
2.1	The java.net package
2.2	Connection oriented transmission – Stream Socket Class
2.3	Creating a Socket to a remote host on a port (creating TCP client and server)
3	**Servlet**
3.1	Introduction
3.2	How It differ from CGI
3.3	Types of servlet
3.4	Life cycle of servlet
3.5	Execution process of Servlet Application
3.6	Session Tracking
3.7	Cookie class
3.8	Servlet- Jdbc
	Introduction to JSP
3.9	Components of JSP – Directives , Tags, Scripting Elements
3.10	Building a simple application using JSP
4	**Multithreading**
4.1	Introduction to Thread
4.2	Life cycle of thread
4.3	Thread Creation - By using Thread Class - By Using Runnable interface
4.4	Priorities and Synchronization
4.5	Inter thread communication
4.6	Implementation of Thread with Applet

1. JDBC

What is JDBC?

JDBC is Java application programming interface that allows the Java programmers to access database management system from Java code. It was developed by **JavaSoft**, a subsidiary of **Sun Microsystems**.

Definition

Java Database Connectivity in short called as JDBC. It is a java API which enables the java programs to execute SQL statements. It is an application programming interface that defines how a java programmer can access the database in tabular format from Java code using a set of standard interfaces and classes written in the Java programming language.

Introduction

JDBC has been developed under the Java Community Process that allows multiple implementations to exist and be used by the same application. JDBC provides methods for querying and updating the data in Relational Database Management system such as SQL, Oracle etc.

The Java application programming interface provides a mechanism for dynamically loading the correct Java packages and drivers and registering them with the JDBC **Driver Manager** that is used as a connection factory for creating JDBC connections which supports creating and executing statements such as SQL INSERT, UPDATE and DELETE. Driver Manager is the backbone of the jdbc architecture.

Generally all Relational Database Management System supports SQL and we all know that Java is platform independent, so JDBC makes it possible to write a single database application that can run on different platforms and interact with different Database Management Systems.

Java Database Connectivity is similar to Open Database Connectivity (ODBC) which is used for accessing and managing database, but the difference is that JDBC is designed specifically for Java programs, whereas ODBC is not depended upon any language.

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

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

Product Components of JDBC

JDBC has four Components:

1. The JDBC API.
2. The JDBC Driver Manager.
3. The JDBC Test Suite.
4. The JDBC-ODBC Bridge.

1. The JDBC API.

The JDBC application programming interface provides the facility for accessing the relational database from the Java programming language. The API technology provides the industrial standard for independently connecting Java programming language and a wide range of databases. The user not only execute the SQL statements, retrieve results, and update the data but can also access it anywhere within a network because of it's "Write Once, Run Anywhere" (WORA) capabilities.

Due to JDBC API technology, user can also access other tabular data sources like spreadsheets or flat files even in the a heterogeneous environment. JDBC application programming interface is a part of the Java platform that have included Java Standard Edition (Java SE) and the Java Enterprise Edition (Java EE) in itself.

The JDBC API has four main interface:

The latest version of JDBC 4.0 application programming interface is divided into two packages
i-) java.sql
ii-) javax.sql.

Java SE and Java EE platforms are included in both the packages.

2. The JDBC Driver Manager.

The JDBC Driver Manager is a very important class that defines objects which connect Java applications to a JDBC driver. Usually Driver Manager is the backbone of the JDBC architecture. It's very simple and small that is used to provide a means of managing the different types of JDBC database driver running on an application. The main responsibility of JDBC database driver is to load all the drivers found in the system properly as well as to select the most appropriate driver from opening a connection to a database. The Driver Manager also helps to select the most appropriate driver from the previously loaded drivers when a new open database is connected.

3. The JDBC Test Suite.
The function of JDBC driver test suite is to make ensure that the JDBC drivers will run user's program or not . The test suite of JDBC application program interface is very useful for testing a driver based on JDBC technology during testing period. It ensures the requirement of Java Platform Enterprise Edition (J2EE).

4. The JDBC-ODBC Bridge.
The JDBC-ODBC bridge, also known as JDBC type 1 driver is a database driver that utilize the ODBC driver to connect the database. This driver translates JDBC method calls into ODBC function calls. The Bridge implements Jdbc for any database for which an Odbc driver is available. The Bridge is always implemented as the sun.jdbc.odbc Java package and it contains a native library used to access ODBC.

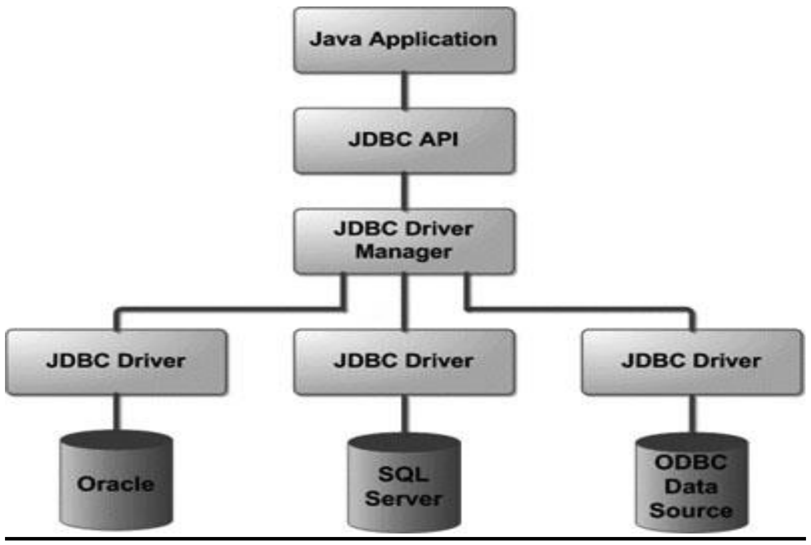
Now we can conclude this topic: This first two component of JDBC, the JDBC API and the JDBC Driver Manager manages to connect to the database and then build a java program that utilizes SQL commands to communicate with any RDBMS. On the other hand, the last two components are used to communicate with ODBC or to test web application in the specialized environment.

1.1 The design of JDBC

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:



1.2 Basic JDBC program Concept

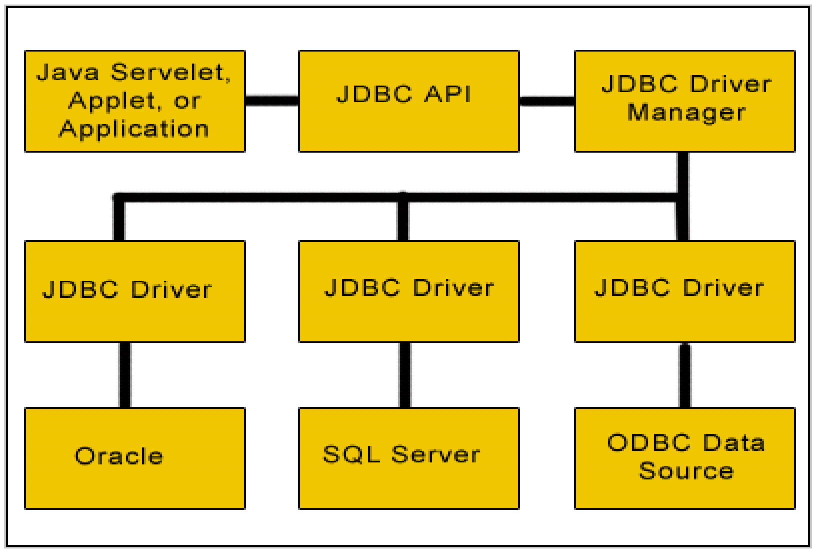
JDBC is an API specification developed by ***Sun Microsystems*** that defines a uniform interface for accessing various relational databases. JDBC is a core part of the Java platform and is included in the standard JDK distribution.

The primary function of the JDBC API is to provide a means for the developer to issue SQL statements and process the results in a consistent, database-independent manner. JDBC provides rich, object-oriented access to databases by defining classes and interfaces that represent objects such as:

1. Database connections
2. SQL statements
3. Result Set
4. Database metadata
5. Prepared statements
6. Binary Large Objects (BLOBs)
7. Character Large Objects (CLOBs)
8. Callable statements
9. Database drivers
10. Driver manager

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. The location of the driver manager with respect to the JDBC drivers and the servlet is shown in Figure 1.

Layers of the JDBC Architecture



A **JDBC** *driver* translates standard *JDBC* calls into a network or database protocol or into a database library API call that facilitates communication with the database. This translation layer provides JDBC applications with database independence. If the back-end database changes, only the JDBC driver need be replaced with few code modifications required

JDBC Driver Manager

The **JDBC DriverManager** class defines objects which can connect Java applications to a JDBC driver. DriverManager has traditionally been the backbone of the JDBC architecture. It is quite small and simple.

This is a very important class. Its main purpose is to provide a means of managing the different types of JDBC database driver. On running an application, it is the DriverManager's responsibility to load all the drivers found in the system property jdbc. drivers. For example, this is where the driver for the Oracle database may be defined. This is not to say that a new driver cannot be explicitly stated in a program at runtime which is not included in jdbc.drivers. When opening a connection to a database it is the DriverManager' s role to choose the most appropriate driver from the previously loaded drivers.

The JDBC API defines the Java interfaces and classes that programmers use to connect to databases and send queries. A JDBC driver implements these interfaces and classes for a particular DBMS vendor.

A Java program that uses the JDBC API loads the specified driver for a particular DBMS before it actually connects to a database. The JDBC DriverManager class then sends all JDBC API calls to the loaded driver.

1.3 Drivers

JDBC Driver

This topic defines the Java(TM) Database Connectivity (JDBC) driver types. Driver types are used to categorize the technology used to connect to the database. A JDBC driver vendor uses these types to describe how their product operates. Some JDBC driver types are better suited for some applications than others.

Types of JDBC drivers

This topic defines the Java(TM) Database Connectivity (JDBC) driver types. Driver types are used to categorize the technology used to connect to the database. A JDBC driver vendor uses these types to describe how their product operates. Some JDBC driver types are better suited for some applications than others.

There are four types of JDBC drivers known as:

- JDBC-ODBC bridge plus ODBC driver, also called Type 1.
- Native-API, partly Java driver, also called Type 2.
- JDBC-Net, pure Java driver, also called Type 3.
- Native-protocol, pure Java driver, also called Type 4.

Type 1 Driver- the JDBC-ODBC bridge

The JDBC type 1 driver, also known as the JDBC-ODBC bridge is a database driver implementation that employs the ODBC driver to connect to the database. The driver converts JDBC method calls into ODBC function calls. The bridge is usually used when there is no pure-Java driver available for a particular database.

The driver is implemented in the sun.jdbc.odbc.JdbcOdbcDriver class and comes with the Java 2 SDK, Standard Edition. The driver is platform-dependent as it makes use of ODBC which in turn depends on native libraries of the operating system. Also, using this driver has got other dependencies such as ODBC must be installed on the computer having the driver and the database which is being connected to must support an ODBC driver. Hence the use of this driver is discouraged if the alternative of a pure-Java driver is available.

Type 1 is the simplest of all but platform specific i.e only to Microsoft platform.

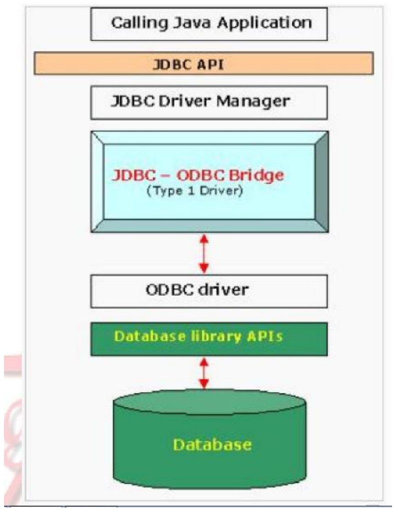
A JDBC-ODBC bridge provides JDBC API access via one or more ODBC drivers. Note that some ODBC native code and in many cases native database client code must be loaded on each client machine that uses this type of driver. Hence, this kind of driver is generally most appropriate when automatic installation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

Type 1 drivers are "bridge" drivers. They use another technology such as Open Database Connectivity (ODBC) to communicate with a database. This is an advantage because ODBC drivers exist for many Relational Database Management System (RDBMS) platforms. The Java Native Interface (JNI) is used to call ODBC functions from the JDBC driver.

A Type 1 driver needs to have the bridge driver installed and configured before JDBC can be used with it. This can be a serious drawback for a production application. Type 1 drivers cannot be used in an applet since applets cannot load native code.

Type 1 JDBC-ODBC Bridge. Type 1 drivers act as a *"bridge"* between **JDBC** and another database connectivity mechanism such as **ODBC.** The **JDBC- ODBC** bridge provides JDBC access using most standard ODBC drivers. This driver is included in the Java 2 SDK within the **sun.jdbc.odbc** package. In this driver the java statements are converted to a jdbc statements. JDBC statements calls the ODBC by using the **JDBC-ODBC Bridge.** And finally the query is executed by the database. This driver has serious limitation for many applications. (See Figure 2.)

Type 1 JDBC Architecture



Functions:

1. Translates query obtained by JDBC into corresponding ODBC query, which is then handled by the ODBC driver.
2. Sun provides a JDBC-ODBC Bridge driver. sun.jdbc.odbc.JdbcOdbcDriver. This driver is native code and not Java, and is closed
   source.
3. Client -> JDBC Driver -> ODBC Driver -> Database
4. There is some overhead associated with the translation work to go from JDBC to ODBC.

Advantages:

Almost any database for which ODBC driver is installed, can be accessed.

Disadvantages:

1. Performance overhead since the calls have to go through the JDBC overhead bridge to the ODBC driver, then to the native database connectivity interface.
2. The ODBC driver needs to be installed on the client machine.
3. Considering the client-side software needed, this might not be suitable for applets.

Type 2 Driver - the Native-API Driver

The JDBC type 2 driver, also known as the Native-API driver is a database driver implementation that uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API.

The type 2 driver is not written entirely in Java as it interfaces with non-Java code that makes the

final database calls.
The driver is compiled for use with the particular operating system. For platform interoperability, the Type 4 driver, being
a full-Java implementation, is preferred over this driver.

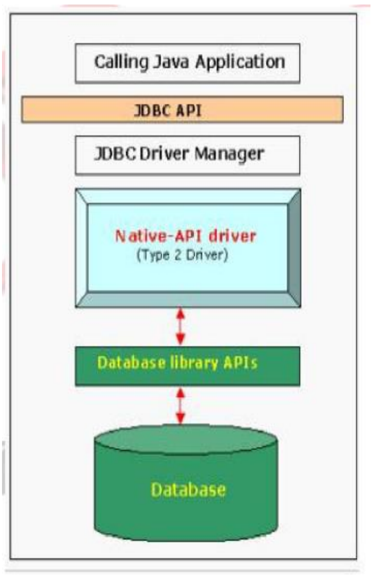
A native-API partly Java technology-enabled driver converts JDBC calls into calls on the client API for Oracle, Sybase, Informix, DB2, or other DBMS. Note that, like the bridge driver, this style of driver requires that some binary code be loaded on each client machine.

However the type 2 driver provides more functionality and performance than the type 1 driver as it does not have the overhead of the additional ODBC function calls.

Type 2 drivers use a native API to communicate with a database system. Java native methods are used to invoke the API functions that perform database operations. Type 2 drivers are generally faster than Type 1 drivers.

Type 2 drivers need native binary code installed and configured to work. A Type 2 driver also uses the JNI. You cannot use a Type 2 driver in an applet since applets cannot load native code. A Type 2 JDBC driver may require some Database Management System (DBMS) networking software to be installed.
The Developer Kit for Java JDBC driver is a Type 2 JDBC driver.

Type 2 Java to Native API. Type 2 drivers use the **Java Native Interface (JNI)** to make calls to a local database library API. This driver converts the JDBC calls into a database specific call for databases such as SQL, ORACLE etc. This driver communicates directly with the database server. It requires some native code to connect to the database. Type 2 drivers are usually faster than Type 1 drivers. Like Type 1 drivers, Type 2 drivers require native database client libraries to be installed and configured on the client machine. (See Figure 3.)
Type 2 JDBC Architecture



Functions:

1. This type of driver converts JDBC calls into calls to the client API for that database.
2. Client -> JDBC Driver -> Vendor Client DB Library -> Database

Advantage

Better performance than Type 1 since no jdbc to odbc translation is needed.

Disadvantages

1. The vendor client library needs to be installed on the client machine.
2. Cannot be used in internet due the client side software needed.
3. Not all databases give the client side library.

Type 3 driver - the Network-Protocol Driver

The JDBC type 3 driver, also known as the network-protocol driver is a database driver implementation which makes use of a middle-tier between the calling program and the database. The middle-tier (application server) converts JDBC calls directly or indirectly into the vendor-specific database protocol.

This differs from the type 4 driver in that the protocol conversion logic resides not at the client, but in the middle-tier. However, like type 4 drivers, the type 3 driver is written entirely in Java.

The same driver can be used for multiple databases. It depends on the number of databases the middleware has been configured to support. The type 3 driver is platform-independent as the platform-related differences are taken care by the middleware. Also, making use of the middleware provides additional advantages of security and firewall access.

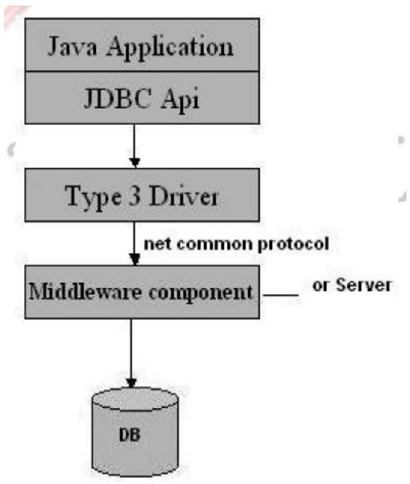
A net-protocol fully Java technology-enabled driver translates JDBC API calls into a DBMS-independent net protocol which is then translated to a DBMS protocol by a server. This net server middleware is able to connect all of its Java technology-based clients to many different databases. The specific 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.

These drivers use a networking protocol and middleware to communicate with a server. The server then translates the protocol to DBMS function calls specific to DBMS.

Type 3 JDBC drivers are the most flexible JDBC solution because they do not require any native binary code on the client. A Type 3 driver does not need any client installation.

Type 3 Java to Network Protocol Or All- Java Driver. Type 3 drivers are pure Java drivers that use a proprietary network protocol to communicate with JDBC middleware on the server. The middleware then translates the network protocol to database-specific function calls. Type 3 drivers are the most flexible JDBC solution because they do not require native database libraries on the client and can connect to many different databases on the back end. Type 3 drivers can be deployed over the Internet without client installation. (See Figure 4.)
Java-------> JDBC statements------> SQL statements ------> databases.

Type 3 JDBC Architecture



Functions:

1. Follows a three tier communication approach.
2. Can interface to multiple databases - Not vendor specific.
3. The JDBC Client driver written in java, communicates with a middleware-net-server using a database independent protocol, and then this net server translates this request into database commands for that database.
4. Thus the client driver to middleware communication is database independent.
5. Client -> JDBC Driver -> Middleware-Net Server -> Any Database

Advantages

1. Since the communication between client and the middleware server is database independent, there is no need for the vendor db library on the client machine. Also the client to middleware need'nt be changed for a new database.

2. The Middleware Server (Can be a full fledged J2EE Application server) can provide typical middleware services like caching (connections, query results, and so on), load balancing, logging, auditing etc..
3. eg. for the above include jdbc driver features in Weblogic.
4. Can be used in internet since there is no client side software needed.
5. At client side a single driver can handle any database.(It works provided the middlware supports that database!!)

Disadvantages

1. Requires database-specific coding to be done in the middle tier.
2. An extra layer added may result in a time-bottleneck. But typically this is overcome by providing efficient middleware
   services described above.

Type 4 - the Native-Protocol Driver

The JDBC type 4 driver, also known as the native-protocol driver is a database driver implementation that converts JDBC calls directly into the vendor-specific database protocol.

The type 4 driver is written completely in Java and is hence platform independent. It is installed inside the Java Virtual Machine of the client. It provides better performance over the type 1 and 2 drivers as it does not have the overhead of conversion of calls into ODBC or database API calls. Unlike the type 1 and 2 drivers, it does not need associated software to work.

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.

As the database protocol is vendor-specific, separate drivers, usually vendor-supplied, need to be used to connect to the database.

A Type 4 driver uses Java to implement a DBMS vendor networking protocol. Since the protocols are usually proprietary, DBMS vendors are generally the only companies providing a Type 4 JDBC driver.

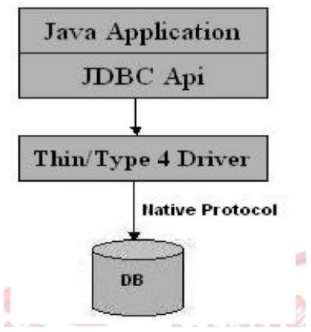
Type 4 drivers are all Java drivers. This means that there is no client installation or configuration. However, a Type 4 driver may not be suitable for some applications if the underlying protocol does not handle issues such as security and network connectivity well.

The IBM Toolbox for Java JDBC driver is a Type 4 JDBC driver, indicating that the API is a pure Java networking protocol driver.

Type 4 Java to Database Protocol. Type 4 drivers are pure Java drivers that implement a proprietary database protocol (like Oracle's SQL*Net) to communicate directly with the database. Like Type 3 drivers, they do not require native database libraries and can be deployed over the Internet without client installation. One drawback to Type 4 drivers is that they are database specific. Unlike Type 3 drivers, if your back-end database changes, you may save to purchase and deploy a new Type 4 driver (some Type 4 drivers are available free of charge from the database manufacturer). However, because Type drivers communicate directly with the database engine rather than through middleware or a native library, they are usually the fastest JDBC drivers available. This driver directly converts the java statements to SQL statements.

(See Figure 5.)

Type 4 JDBC Architecture



Functions

1. Type 4 drivers are entirely written in Java that communicate directly with a vendor's database through socket connections. No translation or middleware layers, are required, improving performance.
2. The driver converts JDBC calls into the vendor-specific database protocol so that client applications can communicate directly with the database server.
3. Completely implemented in Java to achieve platform independence.
4. e.g include the widely used Oracle thin driver - oracle.jdbc.driver. OracleDriver which connect to jdbc:oracle:thin URL format.
5. Client Machine -> Native protocol JDBC Driver -> Database server

Advantages

These drivers don't translate the requests into db request to ODBC or pass it to client api for the db, nor do they need a middleware layer for request indirection. Thus the performance is considerably improved.

Disadvantage

At client side, a separate driver is needed for each database.

1.4 Making the Connection, Statement , ResultSet

The JDBC API is comprised of two Java packages: java.sql and javax.sql.

The following are core JDBC classes, interfaces, and exceptions in the **java.sql** package:

• **DriverManager:**

This class loads JDBC drivers in memory. You can also use it to create java.sql.Connection objects to data sources (such as Oracle, MySQL, and so on).

• **Connection**:

This interface represents a connection with a data source. You can use the Connectionobject for creating Statement, PreparedStatement, and CallableStatement objects.

• **Statement**:

A statement object is used to send and execute SQL statements to a database. It Execute simple sql queries without parameters.Statement createStatement() Creates an SQL Statement object.

This interface represents a static SQL statement. You can use it to retrieve ResultSet objects.Statement interface defines methods that are used to interact with database via the execution of SQL statements. The Statement class has three methods for executing statements:executeQuery(), executeUpdate(), and execute(). For a SELECT statement, the method to use is executeQuery . For statements that create or modify tables, the method to use is executeUpdate. Note: Statements that create a table, alter a table, or drop a table are all examples of DDLstatements and are executed with the method executeUpdate. execute() executes an SQLstatement that is written as String object.

• **PreparedStatement:**

This interface extends Statement and represents a precompiled SQL statement. You can use it to retrieve ResultSet objects.It Execute precompiled sql queries with or without parameters. PreparedStatement prepareStatement(String sql)returns a new PreparedStatement object. PreparedStatement objects are precompiled SQL statements.

• **CallableStatement**:

This interface represents a database stored procedure. You can use it to execute stored procedures in a database server.It Execute a call to a database stored procedure.

CallableStatement prepareCall(String sql)returns a new CallableStatement object. CallableStatement objects are SQL stored procedure call statements.

• **ResultSet**:

This interface represents a database result set generated by using SQL's SELECT statement. It provides access to a table of data generated by executing a Statement. The table rows are retrieved in sequence. A ResultSet maintains a cursor pointing to its current row of data. The next() method is used to successively step through the rows of the tabular results.

ResultSetMetaData

Interface holds information on the types and properties of the columns in a ResultSet. It is constructed from the Connection object.

• **SQLException**:

This class is an exception class that provides information on a database access error or other errors.

1.5 Executing SQL commands

1.6 Executing queries

2.Networking

2.1 The java.net package

- The ***java.net*** package contains the ***Socket*** class. This class speaks TCP (connection-oriented protocol).
- The ***DatagramSocket*** class uses UDP (connectionless protocol).
- The java.net.Socket class represents a single side of a socket connection on either the client or server. In addition, the server uses the ***java.net.ServerSocket*** class to wait for connections from clients.
- The server creates a *ServerSocket* object and waits, blocked in a call to its ***accept()*** method, until a connection arrives. When a connection request arrives, the ***accept()*** creates a *Socket* object. The server uses this *Socket* object to communicate with the client.

What are sockets?

The socket is the software abstraction used to represent the "terminals" of a connection between two machines. For a given connection, there's a socket on each machine, and you can imagine a hypothetical "cable" running between the two machines with each end of the "cable" plugged into a socket. Of course, the physical hardware and cabling between machines is completely unknown. The whole point of the abstraction is that we don't have to know more than is necessary.
a socket on one computer that talks to a socket on another computer creates a communication channel. A programmer can use that channel to send data between the two machines. When you send data, each layer of the TCP/IP stack adds appropriate header information to wrap your data. These headers help the stack get your data to its destination. The good news is that the Java language hides all of this from you by providing the data to your code on streams, which is why they are sometimes called *streaming sockets*.
Think of sockets as handsets on either side of a telephone call -- you and I talk and listen on our handsets on a dedicated channel. The conversation doesn't end until we decide to hang up (unless we're using cell phones). And until we hang up, our respective phone lines are busy.
If you need to communicate between two computers without the overhead of higher-level mechanisms like ORBs (and CORBA, RMI, IIOP, and so on), sockets are for you. The low-level details of sockets get rather involved. Fortunately, the Java platform gives you some simple yet powerful higher-level abstractions that make creating and using sockets easy.

Types of sockets

Generally speaking, sockets come in two flavors in the Java language:
* TCP sockets (implemented by the Socket class, which we'll discuss later)
* UDP sockets (implemented by the DatagramSocket class)
TCP and UDP play the same role, but they do it differently. Both receive transport protocol packets and pass along their contents to the Presentation Layer. TCP divides messages into packets (*datagrams*) and reassembles them in the correct sequence at the receiving end. It also handles requesting retransmission of missing packets. With TCP, the upper-level layers

have much less to worry about. UDP doesn't provide these assembly and retransmission requesting features. It simply passes packets along. The upper layers have to make sure that the message is complete and assembled in correct sequence.

In general, UDP imposes lower performance overhead on your application, but only if your application doesn't exchange lots of data all at once and doesn't have to reassemble lots of datagrams to complete a message. Otherwise, TCP is the simplest and probably most efficient choice.

Because most readers are more likely to use TCP than UDP, we'll limit our discussion to the TCP-oriented classes in the Java language.

4. Servlet

4.1 Introduction

What is Java Servlets?

Servlets are server side components that provide a powerful mechanism for developing server side programs. Servlets provide component-based, platform-independent methods for building Web-based applications, without the performance limitations of CGI programs. Unlike proprietary server extension mechanisms (such as the Netscape Server API or Apache modules), servlets are server as well as platform-independent. This leaves you free to select a "best of breed" strategy for your servers, platforms, and tools. Using servlets web developers can create fast and efficient server side application which can run on any servlet enabled web server. Servlets run entirely inside the Java Virtual Machine. Since the Servlet runs at server side so it does not checks the browser for compatibility. Servlets can access the entire family of Java APIs, including the JDBC API to access enterprise databases. Servlets can also access a library of HTTP-specific calls, receive all the benefits of the mature java language including portability, performance, reusability, and crash protection. Today servlets are the popular choice for building interactive web applications. Third-party servlet containers are available for Apache Web Server, Microsoft IIS, and others. Servlet containers are usually the components of web and application servers, such as BEA WebLogic Application Server, IBM WebSphere, Sun Java System Web Server, Sun Java System Application Server and others.

Servlets are not designed for a specific protocols. It is different thing that they are most commonly used with the HTTP protocols Servlets uses the classes in the java packages javax.servlet and javax.servlet.http. Servlets provides a way of creating the sophisticated server side extensions in a server as they follow the standard framework and use the highly portable java language.

HTTP Servlet typically used to:

- Priovide dynamic content like getting the results of a database query and returning to the client.
- Process and/or store the data submitted by the HTML.
- Manage information about the state of a stateless HTTP. e.g. an online shopping car manages request for multiple concurrent customers.

Introduction to Server Side Programming

All of us (or most of us) would have started programming in Java with the ever famous “Hello World!” program. If you can recollect, we saved this file with a .java extension and later compiled the program using javac and then executed the class file with java. Apart from introducing you to the language basics, the point to be noted about this program is that – "It is a client side program". This means that you write, compile and also execute the program on a client machine (e.g. Your PC). No doubt, this is the easiest and fastest way to write, compile and execute programs. But, it has little practical significance when it comes to real world programming.

1. **Why Server Side Programming?**

   Though it is technically feasible to implement almost any business logic using client side programs, logically or functionally it carries no ground when it comes to enterprise applications (e.g. banking, air ticketing, e-shopping etc.). To further explain, going by the client side programming logic; a bank having 10,000 customers would mean that each customer should have a copy of the program(s) in his or her PC which translates to 10,000 programs! In addition, there are issues like security, resource pooling, concurrent access and manipulations to the database which simply cannot be handled by client side programs. The answer to most of the issues cited above is – "Server Side Programming". Figure-1 illustrates Server side architecture in the simplest terms.

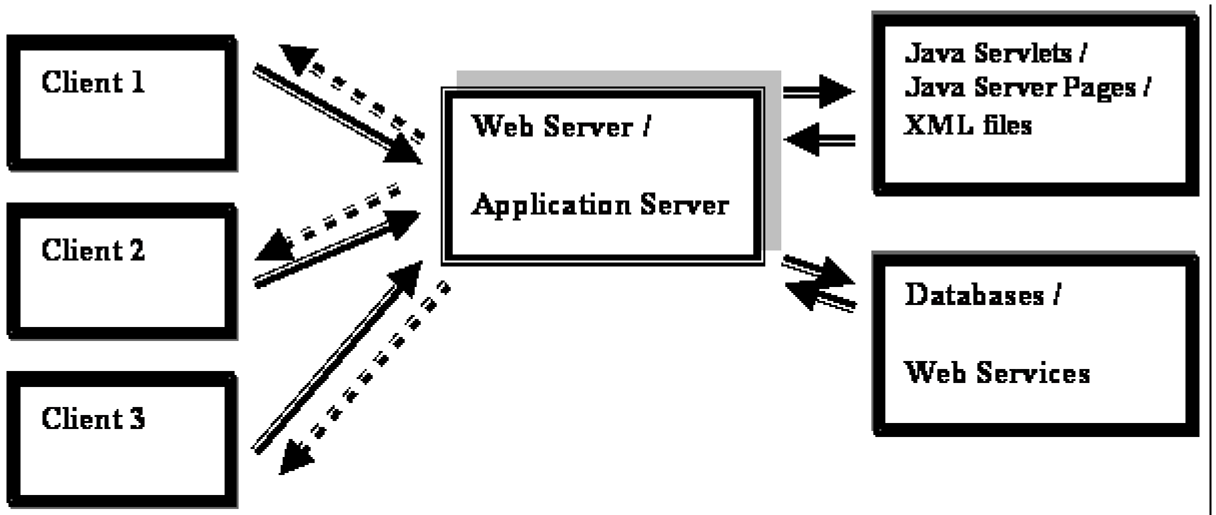


Figure – Server Side Programming (architecture)

2. **Advantages of Server Side Programs**

   The list below highlights some of the important advantages of Server Side programs.

   i. All programs reside in one machine called the Server. Any number of remote machines (called clients) can access the server programs.
   ii. New functionalities to existing programs can be added at the server side which the clients' can advantage without having to change anything from their side.
   iii. Migrating to newer versions, architectures, design patterns, adding patches, switching to new databases can be done at the server side without having to bother about clients' hardware or software capabilities.

iv. Issues relating to enterprise applications like resource management, concurrency, session management, security and performance are managed by service side applications.
v. They are portable and possess the capability to generate dynamic and user-based content (e.g. displaying transaction information of credit card or debit card depending on user's choice).

3. **Types of Server Side Programs**
   i. Active Server Pages (ASP)
   ii. Java Servlets
   iii. Java Server Pages (JSPs)
   iv. Enterprise Java Beans (EJBs)
   v. PHP

   To summarize, the objective of server side programs is to centrally manage all programs relating to a particular application (e.g. Banking, Insurance, e-shopping, etc). Clients with bare minimum requirement (e.g. Pentium II, Windows XP Professional, MS Internet Explorer and an internet connection) can experience the power and performance of a Server (e.g. IBM Mainframe, Unix Server, etc) from a remote location without having to compromise on security or speed. More importantly, server programs are not only portable but also possess the capability to generate dynamic responses based on user's request.

Introduction to Java Servlet

Java Servlets are server side Java programs that require either a Web Server or an Application Server for execution. Examples for Web Servers include Apache's Tomcat Server and Macromedia's JRun. Web Servers include IBM's Weblogic and BEA's Websphere server. Examples for other Server programs include Java Server Pages (JSPs) and Enterprise Java Beans (EJBs). In the forthcoming sections, we will get acquainted with Servlet fundamentals and other associated information required for creating and executing Java Servlets.

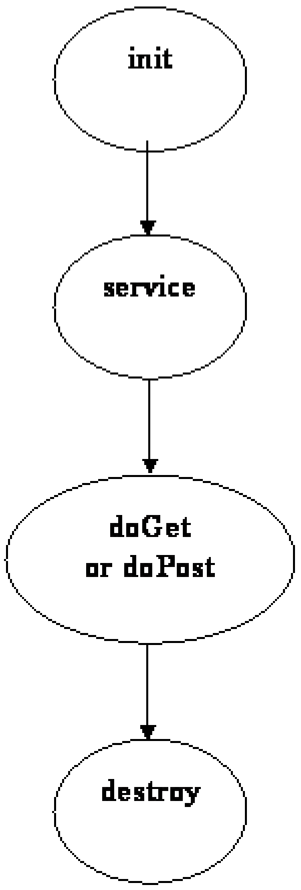
1. **Basic Servlet Structure**

   As seen earlier, Java servlets are server side programs or to be more specific; web applications that run on servers that comply HTTP protocol. The javax.servlet and javax.servlet.http packages provide the necessary interfaces and classes to work with servlets. Servlets generally extend the HttpServlet class and override the doGet or the doPost methods. In addition, other methods such as init, service and destroy also called as life cycle methods might be used which will be discussed in the following section. The skeleton of a servlet is given in Figure

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class <<servlet name>> extends HttpServlet {

public void doGet (HttpServlet request, HttpResponse
response) throws ServletException, IOException {

// code for business logic here

// use request object to read client's requests

// user response object to throw output back to the client

} // close doGet
} // end program
```

Figure-Basic Servlet Structure

2. **A Servlet's Life Cycle**
   The first time a servlet is invoked, it is the init method which is called. And remember that this is called only once during the lifetime of a servlet. So, you can put all your initialization code here. This method next calls the service method. The service method in turn calls the doGet or doPost methods (whichever the user has overridden). Finally, the servlet calls the destroy method. It is in a sense equivalent to the finally method. You can reset or close references / connections done earlier in the servlet's methods (e.g. init, service or doGet /doPost). After this method is called, the servlet ceases to exist for all practical purposes. However, please note that it is not mandatory to override all these methods. More often than not, it is the doGet or doPost method used with one or more of the other life cycle methods.



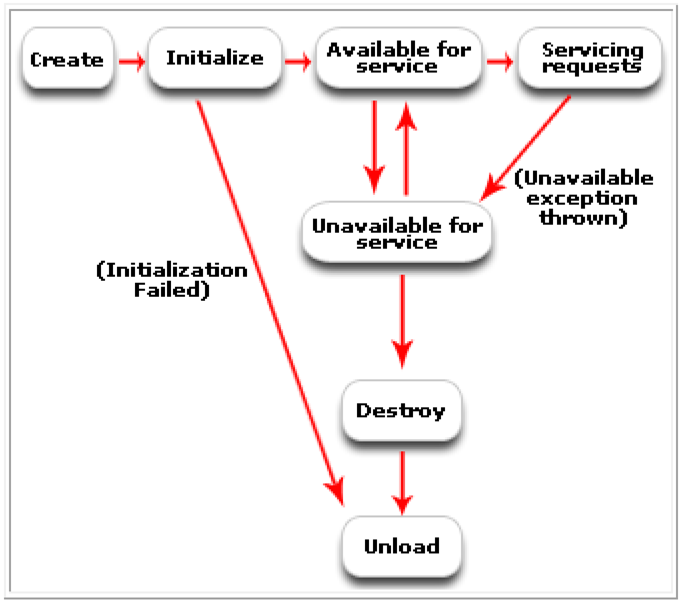
4.2 Life cycle of servlet

The life cycle of a servlet can be categorized into four parts:

1. **Loading and Inatantiation:** The servlet container loads the servlet during startup or when the first request is made. The loading of the servlet depends on the attribute <load-on-startup> of web.xml file. If the attribute <load-on-startup> has a positive value then the servlet is load with loading of the container otherwise it load when the first request comes for service. After loading of the servlet, the container creates the instances of the servlet.
2. **Initialization:** After creating the instances, the servlet container calls the init() method and passes the servlet initialization parameters to the init() method. The init() must be called by the servlet container before the servlet can service any request. The initialization parameters persist untill the servlet is destroyed. The init() method is called only once throughout the life cycle of the servlet.

   The servlet will be available for service if it is loaded successfully otherwise the servlet container unloads the servlet.

3. **Servicing the Request:** After successfully completing the initialization process, the servlet will be available for service. Servlet creates seperate threads for each request. The sevlet container calls the service() method for servicing any request. The service() method determines the kind of request and calls the appropriate method (doGet() or doPost()) for handling the request and sends response to the client using the methods of the response object.
4. **Destroying the Servlet:** If the servlet is no longer needed for servicing any request, the servlet container calls the destroy() method . Like the init() method this method is also called only once throughout the life cycle of the servlet. Calling the destroy() method indicates to the servlet container not to sent the any request for service and the servlet releases all the resources associated with it. Java Virtual Machine claims for the memory associated with the resources for garbage collection.



Life Cycle of a Servlet

Methods of Servlets

A Generic servlet contains the following five methods:

init()

public void init(ServletConfig config) throws ServletException

The **init() method** is called only once by the servlet container throughout the life of a servlet. By this init() method the servlet get to know that it has been placed into service.

The servlet cannot be put into the service if

- The init() method does not return within a fix time set by the web server.
- It throws a ServletException

Parameters - The init() method takes a **ServletConfig** object that contains the initialization parameters and servlet's configuration and throws a **ServletException** if an exception has occurred.

service()

public void service(ServletRequest req, ServletResponse res) throws ServletException, IOException

Once the servlet starts getting the requests, the service() method is called by the servlet container to respond. The servlet services the client's request with the help of two objects. These two objects **javax.servlet.ServletRequest** and **javax.servlet.ServletResponse** are passed by the servlet container.

The status code of the response always should be set for a servlet that throws or sends an error.

Parameters - The service() method takes **the ServletRequest** object that contains the client's request and the object **ServletResponse** contains the servlet's response. The service() method throws **ServletException and IOExceptions** exception.

getServletConfig()

public ServletConfig getServletConfig()

This method contains parameters for initialization and startup of the servlet and returns a **ServletConfig object.** This object is then passed to the init method. When this interface is implemented then it stores **the ServletConfig object** in order to return it. It is done by the generic class which implements this inetrface.

Returns - the ServletConfig object

getServletInfo()

public String getServletInfo()

The information about the servlet is returned by this method like version, author etc. This method returns a string which should be in the form of plain text and not any kind of markup.

Returns - a string that contains the information about the servlet

destroy()

public void destroy()

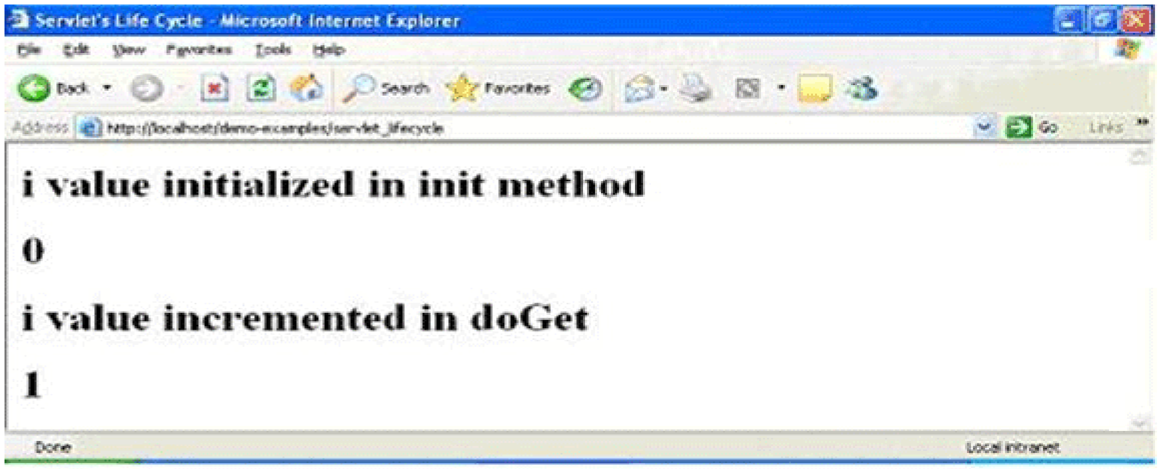
This method is called when we need to close the servlet. That is before removing a servlet instance from service, the servlet container calls the destroy() method. Once the servlet container calls the destroy() method, no service methods will be then called . That is after the exit of all the threads running in the servlet, the destroy() method is called. Hence, the servlet gets a chance to clean up all the resources like memory, threads etc which are being held.

3. **A Servlet Program Demostrating it's life cycle**

```
// Servlet program demonstrating it's life cycle
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class servlet_lifecycle extends HttpServlet {
    int i;

public void init() throws ServletException
{
i=0;  // initializing i value
}

    // incrementing i value in doGet method
    public void doGet(HttpServletRequest request,
HttpServletResponse response)
    throws IOException, ServletException
    {
     response.setContentType("text/html");
        PrintWriter out = response.getWriter();
     if (i==0)
     {
     out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet's Life Cycle</title>");
        out.println("</head>");
        out.println("<body>");
        out.print("<h1>i value initialized in init
method</h1>"+ "<h1>" + i+ "</h1>");
        out.println("</body>");
        out.println("</html>");

     }
    i =i+1;
if (i ==10)
     {
     out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet's Life Cycle</title>");
        out.println("</head>");
        out.println("<body>");
        out.print("<h1>i value reaches 10, hence calling
destroy method to reset it</h1>"+ "<h1>" + i+ "</h1>");
        out.println("</body>");
        out.println("</html>");

     destroy(); // call destroy method if i=10
     }

     if ( i < 10 ) // display incremented value of i
     {
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet's Life Cycle</title>");
        out.println("</head>");
        out.println("<body>");
        out.print("<h1>i value incremented in doGet</h1>"+
"<h1>" + i+ "</h1>");
        out.println("</body>");
        out.println("</html>");
     }

    }
    public void destroy() // reset i value here
    {
    i=0;
    }
    }
```

Output Screens

To appreciate the execution of the servlet life cycle methods, keep refreshing the browser (F5 in Windows). In the background, what actually happens is – with each refresh, the doGet method is called which increments i's value and displays the current value. Find below the screen shots (Figures 5 through 7) captured at random intervals. The procedure to run the servlets using a Web Server will be demonstrated in the next section (1.3.).


Servlet's Life Cycle - Microsoft Internet Explorer
http://localhost/demo-examples/servlet_lifecycle
Done
Local intranet

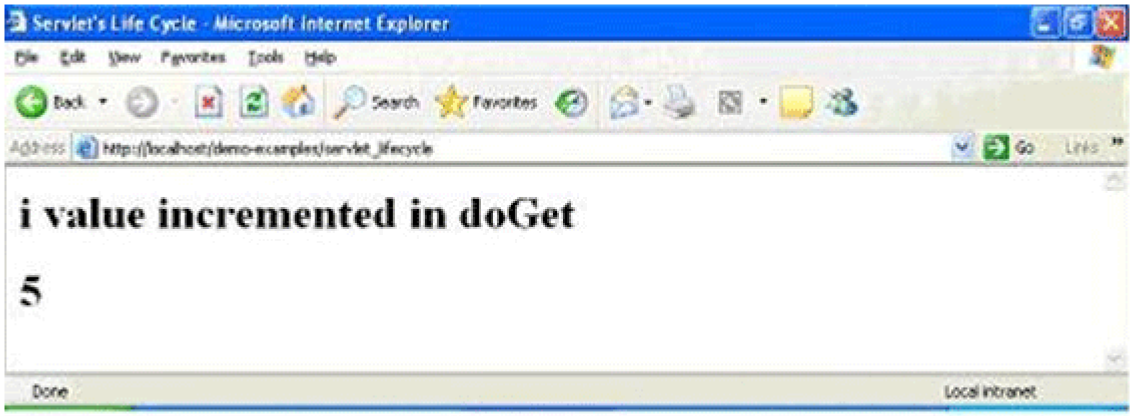
i value initialized in init method

0

i value incremented in doGet

1

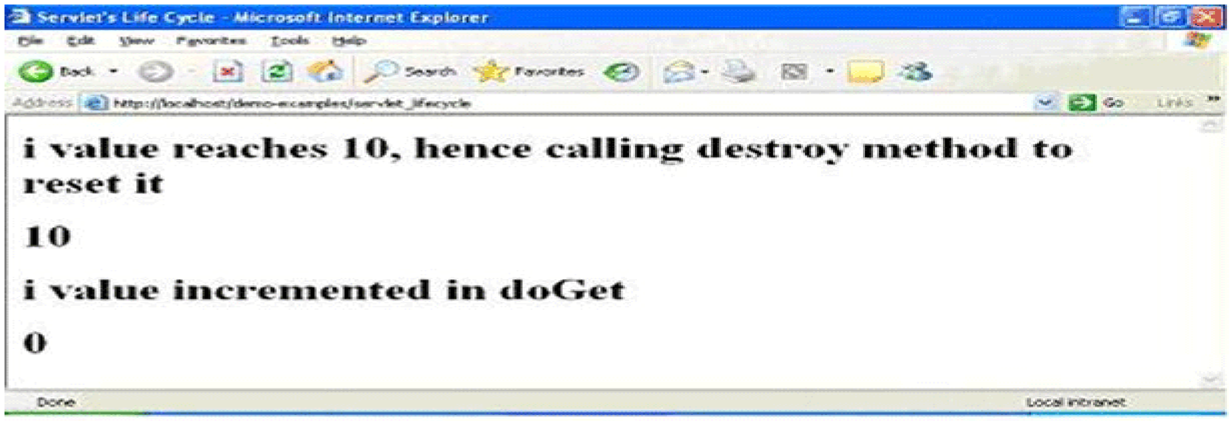
Figure– initial i value and incremented value


Servlet's Life Cycle - Microsoft Internet Explorer
http://localhost/demo-examples/servlet_lifecycle
Done
Local intranet

i value incremented in doGet

5

Figure- i value after repeated refreshing of browser


Servlet's Life Cycle - Microsoft Internet Explorer
http://localhost/demo-examples/servlet_lifecycle
Done
Local intranet

i value reaches 10, hence calling destroy method to reset it

10

i value incremented in doGet

0

Figure- i value gets reset when its value equals 10

How to run a servlet

In this section, we will see as how to install a WebServer, configure it and finally run servlets using this server. Throughout this tutorial, we will be using Apache's Tomcat server as the WebServer. Tomcat is not only an open and free server, but also the most preferred WebServer across the world. A few reasons we can attribute for its popularity is – Easy to install and configure, very less memory footprint, fast, powerful and portable. It is the ideal server for learning purpose.

1. **Installation of Tomcat Server and JDK**

   As mentioned earlier, Apache's Tomcat Server is free software available for download @ www.apache.org. The current version of Tomcat Server is 6.0 (as of November 2007). This Server supports Java Servlets 2.5 and Java Server Pages (JSPs) 2.1 specifications. In case of doubt or confusion, you can refer to the abundant documentation repository available on this site.

   Important software required for running this server is Sun's JDK (Java Development Kit) and JRE (Java Runtime Environment). The current version of JDK is 6.0. Like Tomcat, JDK is also free and is available for download at www.java.sun.com.
2. **Configuring Tomcat Server**

   - Set JAVA_HOME variable - You have to set this variable which points to the base installation directory of JDK installation. (e.g. c:\program file\java\jdk1.6.0). You can either set this from the command prompt or from My Computer -> Properties -> Advanced -> Environment Variables.
   - Specify the Server Port – You can change the server port from 8080 to 80 (if you wish to) by editing the server.xml file in the conf folder. The path would be something like this – c:\program files\apache software foundation\tomcat6\conf\server.xml
3. **Run Tomcat Server**

   Once the above pre-requisites are taken care, you can test as whether the server is successfully installed as follows:

   Step 1

   • Go to C:\Program Files\Apache Software Foundation\Tomcat 6.0\bin and double click on tomcat6

   OR

   • Go to Start->Programs->Apache Tomcat 6.0 -> Monitor Tomcat. You will notice an icon appear on the right side of your Status Bar. Right click on this icon and click on Start service.

   Step 2

• Open your Browser (e.g. MS Internet Explorer) and type the following URL :

http://localhost/ (If you have changed to port # to 80)

OR

• Open your Browser (e.g. MS Internet Explorer) and type the following URL :

http://localhost:8080/ (If you have NOT changed the default port #)

In either case, you should get a page similar to the one in Figure-8 which signifies that the Tomcat Server is successfully running on your machine.

4. **Compile and Execute your Servlet**

This section through a step by step (and illustration) approach explains as how to compile and then run a servlet using Tomcat Server. Though this explanation is specific to Tomcat, the procedure explained holds true for other Web servers too (e.g. JRun, Caucho's Resin).

Step 1 – Compile your servlet program

The first step is to compile your servlet program. The procedure is no different from that of writing and compiling a java program. But, the point to be noted is that neither the javax.servlet.* nor the javax.servlet.http.* is part of the standard JDK. It has to be exclusively added in the CLASSPATH. The set of classes required for writing servlets is available in a jar file called servlet-api.jar. This jar file can be downloaded from several sources. However, the easiest one is to use this jar file available with the Tomcat server (C:\Program Files\Apache Software Foundation\Tomcat 6.0\lib\servlet-api.jar). You need to include this path in CLASSPATH. Once you have done this, you will be able to successfully compile your servlet program. Ensure that the class file is created successfully.

Step 2 – Create your Web application folder

The next step is to create your web application folder. The name of the folder can be any valid and logical name that represents your application (e.g. bank_apps, airline_tickets_booking, shopping_cart,etc). But the most important criterion is that this folder should be created under webapps folder. The path would be similar or close to this - C:\Program Files\Apache Software Foundation\Tomcat 6.0\webapps. For demo purpose, let us create a folder called demo-examples under the webapps folder.

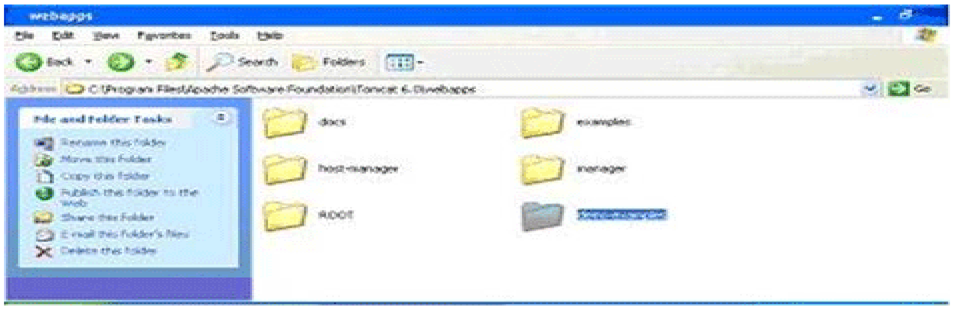


Figure- depicts the same.

Step 3 – Create the WEB-INF folder

The third step is to create the WEB-INF folder. This folder should be created under your web application folder that you created in the previous step. Figure-10 shows the WEB-INF folder being placed under the demo-examples folder.

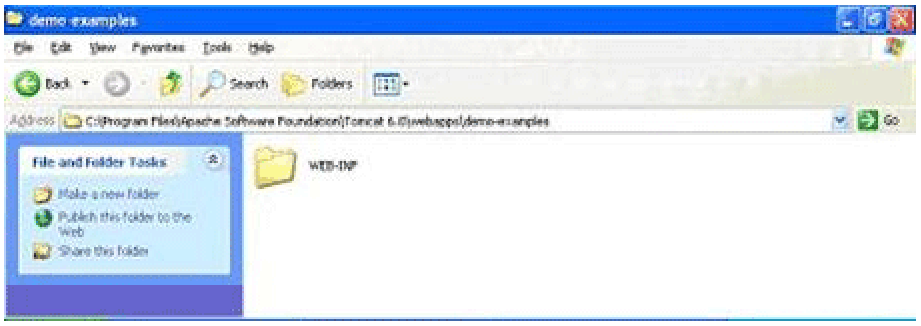

demo-examples
C:\Program Files\Apache Software Foundation\Tomcat 6.0\webapps\demo-examples
File and Folder Tasks
Make a new folder
Publish this folder to the Web
Share this folder
WEB-INF

Figure – WEB-INF folder inside web application folder

Step 4 – Create the web.xml file and the classes folder

The fourth step is to create the web.xml file and the classes folder. Ensure that the web.xml and classes folder are created under the WEB-INF folder. Figure-11 shows this file and folder being placed under the WEB-INF folder.

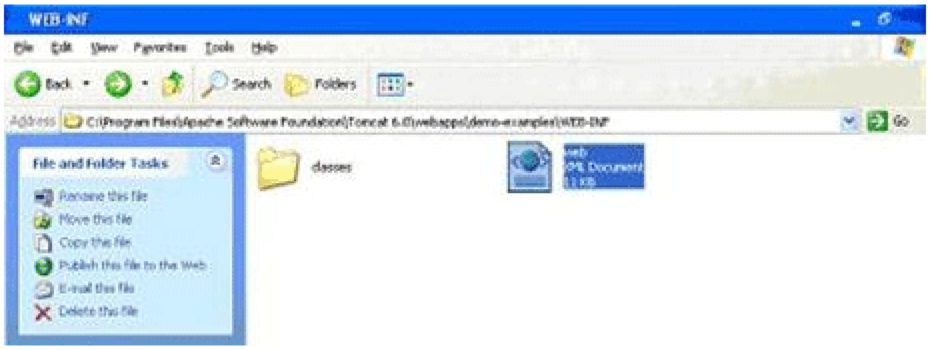


Figure – web.xml file and the classes folder

Note – Instead of creating the web.xml file an easy way would be to copy an existing web.xml file (e.g. C:\Program Files\Apache Software Foundation\Tomcat 6.0\webapps\examples\WEB-INF) and paste it into this folder. You can later edit this file and add relevant information to your web application.

Step 5 – Copy the servlet class to the classes folder

We need to copy the servlet class file to the classes folder in order to run the servlet that we created. All you need to do is copy the servlet class file (the file we obtained from Step 1) to this folder. Figure-12 shows the servlet_lifecycle (refer section 1.2.3.) class being placed in the classes folder.

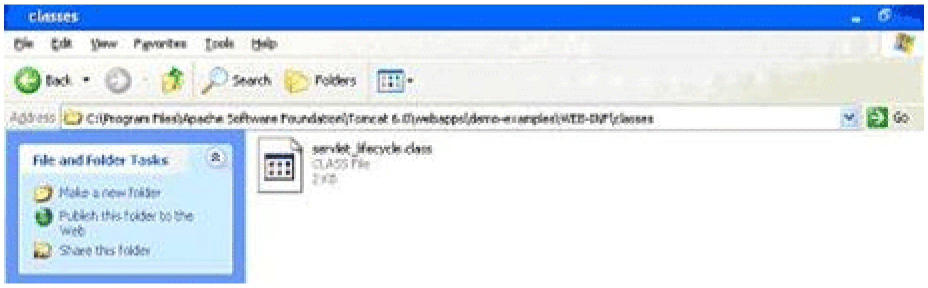
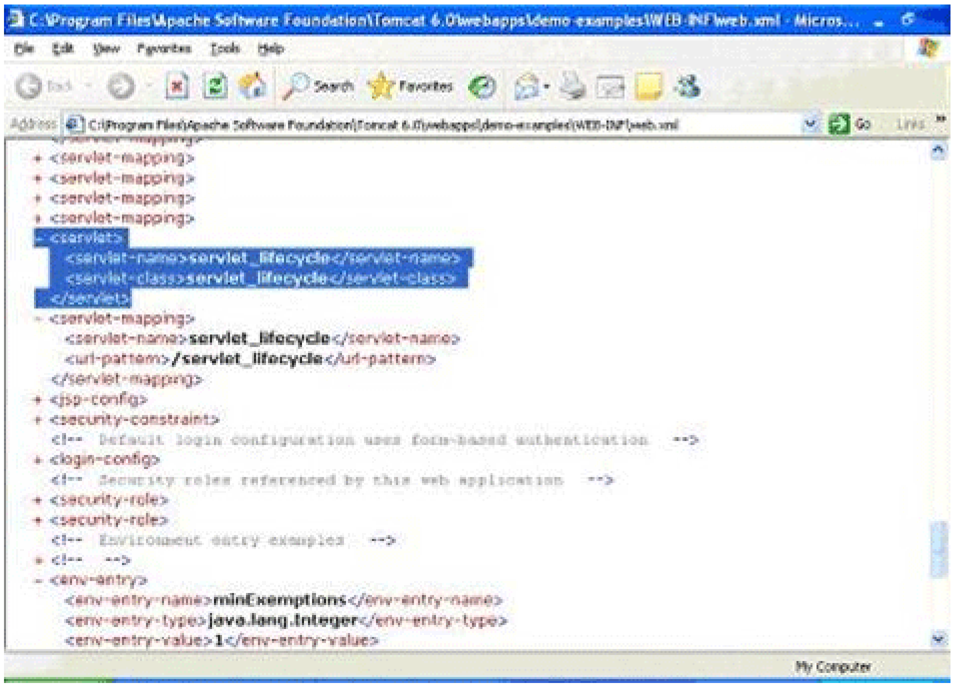

classes
File Edit View Favorites Tools Help
Back Search Folders
Address C:\Program Files\Apache Software Foundation\Tomcat 6.0\webapps\demo-examples\WEB-INF\classes
File and Folder Tasks
Make a new folder
Publish this folder to the Web
Share this folder
servlet_lifecycle.class
CLASS File
2 KB

Figure – servlet class file placed under classes folder

Step 6 – Edit web.xml to include servlet's name and url pattern

This step involves two actions viz. including the servlet's name and then mentioning the url pattern. Let us first see as how to include the servlet's name in the web.xml file. Open the web.xml file and include the servlet's name as shown in Figure-13.

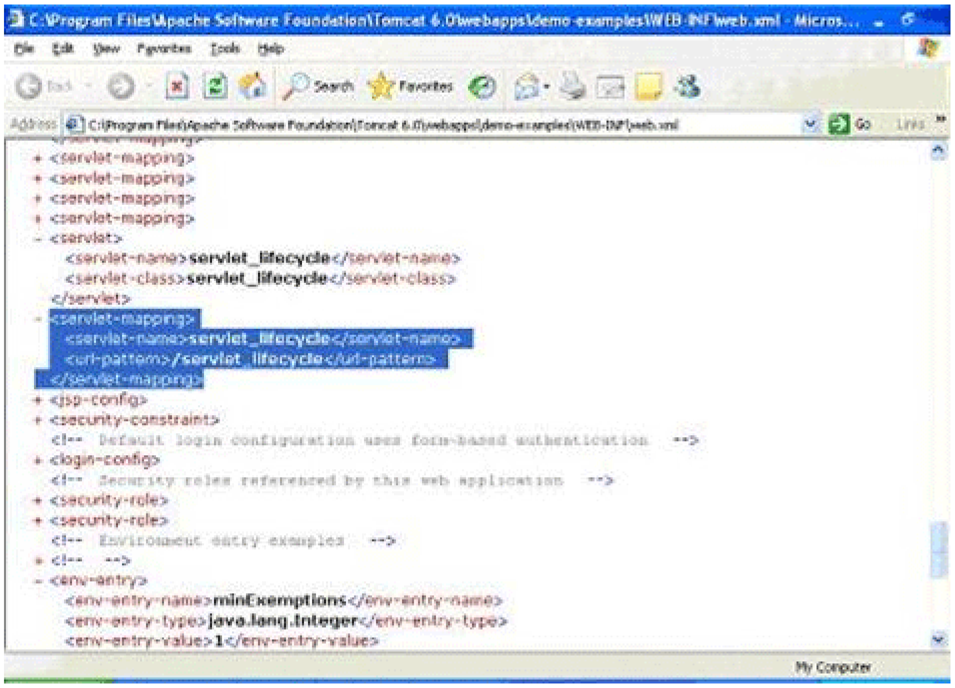

C:\Program Files\Apache Software Foundation\Tomcat 6.0\webapps\demo-examples\WEB-INF\web.xml - Micros...
File Edit View Favorites Tools Help
Back Search Favorites
Address C:\Program Files\Apache Software Foundation\Tomcat 6.0\webapps\demo-examples\WEB-INF\web.xml
Go Links
My Computer

```
+ <servlet-mapping>
+ <servlet-mapping>
+ <servlet-mapping>
+ <servlet-mapping>
- <servlet>
    <servlet-name>servlet_lifecycle</servlet-name>
    <servlet-class>servlet_lifecycle</servlet-class>
  </servlet>
- <servlet-mapping>
    <servlet-name>servlet_lifecycle</servlet-name>
    <url-pattern>/servlet_lifecycle</url-pattern>
  </servlet-mapping>
+ <jsp-config>
+ <security-constraint>
  <!-- Default login configuration uses form-based authentication -->
+ <login-config>
  <!-- Security roles referenced by this web application -->
+ <security-role>
+ <security-role>
  <!-- Environment entry examples -->
+ <!-- -->
- <env-entry>
    <env-entry-name>minExemptions</env-entry-name>
    <env-entry-type>java.lang.Integer</env-entry-type>
    <env-entry-value>1</env-entry-value>
```

Figure– Include servlet's name using the <servlet> </servlet> tag

Note – The servlet-name need not be the same as that of the class name. You can give a different name (or alias) to the actual servlet. This is one of the main reasons as why this tag is used for.

Next, include the url pattern using the <servlet-mapping> </servlet-mapping> tag. The url pattern defines as how a user can access the servlet from the browser. Figure-14 shows the url pattern entry for our current servlet.



```
+ <servlet-mapping>
+ <servlet-mapping>
+ <servlet-mapping>
+ <servlet-mapping>
- <servlet>
    <servlet-name>servlet_lifecycle</servlet-name>
    <servlet-class>servlet_lifecycle</servlet-class>
  </servlet>
- <servlet-mapping>
    <servlet-name>servlet_lifecycle</servlet-name>
    <url-pattern>/servlet_lifecycle</url-pattern>
  </servlet-mapping>
+ <jsp-config>
+ <security-constraint>
  <!--  Default login configuration uses form-based authentication  -->
+ <login-config>
  <!--  Security roles referenced by this web application  -->
+ <security-role>
+ <security-role>
  <!--  Environment entry examples  -->
+ <!--  -->
- <env-entry>
    <env-entry-name>minExemptions</env-entry-name>
    <env-entry-type>java.lang.Integer</env-entry-type>
    <env-entry-value>1</env-entry-value>
```

Figure – Include url-pattern using the <servlet-mapping> </servlet-mapping> tag

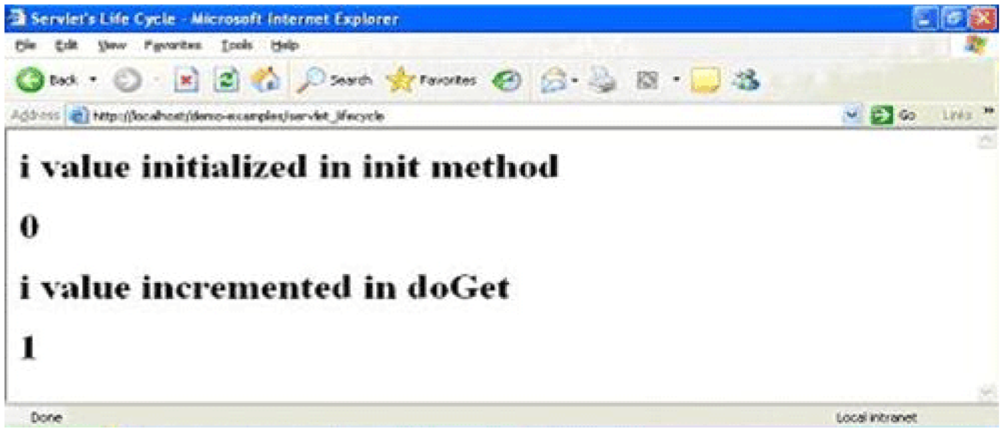
Note – Please remember that the path given in the url-pattern is a relative path. This means that this path is w.r.t. your web applications folder (demo-examples in this case).

Step 7 – Run Tomcat server and then execute your Servlet

This step again involves two actions viz. running the Web Server and then executing the servlet. To run the server, follow the steps explained in Section 1.3.3.

After ensuring that the web server is running successfully, you can run your servlet. To do this, open your web browser and enter the url as specified in the web.xml file. The complete url that needs to be entered in the browser is:

http://localhost/demo-examples/servlet_lifecycle


Servlet's Life Cycle - Microsoft Internet Explorer
http://localhost/demo-examples/servlet_lifecycle
Done
Local intranet

```
i value initialized in init method
0
i value incremented in doGet
1
```

Figure – Our servlet's output!

4.3 Types of servlet

There are two types of servlets, **GenericServlet** and **HttpServlet**. **GenericServlet** defines the generic or protocol independent servlet. HttpServlet is subclass of GenericServlet and provides some http specific functionality like doGet and doPost methods.

HttpServlet Provides an abstract class to be subclassed to create an HTTP servlet suitable for a Web site. A subclass of `HttpServlet` must override at least one method, usually one of these:

- `doGet`, if the servlet supports HTTP GET requests
- `doPost`, for HTTP POST requests
- `doPut`, for HTTP PUT requests
- `doDelete`, for HTTP DELETE requests
- `init` and `destroy`, to manage resources that are held for the life of the servlet
- `getServletInfo`, which the servlet uses to provide information about itself

There's almost no reason to override the `service` method. `service` handles standard HTTP requests by dispatching them to the handler methods for each HTTP request type (the `do`*XXX* methods listed above). Likewise, there's almost no reason to override the `doOptions` and `doTrace` methods.

GenericServlet defines a generic, protocol-independent servlet. To write an HTTP servlet for use on the Web, extend `HttpServlet` instead.

`GenericServlet` implements the `Servlet` and `ServletConfig` interfaces. `GenericServlet` may be directly extended by a servlet, although it's more common to extend a protocol-specific subclass such as `HttpServlet`.

`GenericServlet` makes writing servlets easier. It provides simple versions of the lifecycle methods `init` and `destroy` and of the methods in the `ServletConfig` interface. `GenericServlet` also implements the `log` method, declared in the `ServletContext` interface.

To write a generic servlet, you need only override the abstract `service` method.

4.4 Session Tracking

As we know that the Http is a *stateless* protocol, means that it can't persist the information. It always treats each request as a new request. In Http client makes a connection to the server, sends the request. gets the response, and closes the connection.

In session management client first make a request for any servlet or any page, the container receives the request and generate a unique session ID and gives it back to the client along with the response. This ID gets stores on the client machine. Thereafter when the client request again sends a request to the server then it also sends the session Id with the request. There the container sees the Id and sends back the request.

Session Tracking can be done in three ways:

1. **Hidden Form Fields:** This is one of the way to support the session tracking. As we know by the name, that in this fields are added to an HTML form which are not displayed in the client's request. The hidden form field are sent back to the server when the form is submitted. In hidden form fields the html entry will be like this : <input type ="hidden" name = "name" value="">. This means that when you submit the form, the specified name and value will be get included in get or post method. In this session ID information would be embedded within the form as a hidden field and submitted with the Http POST command.
2. **URL Rewriting:** This is another way to support the session tracking. **URLRewriting** can be used in place where we don't want to use cookies. It is used to maintain the session. Whenever the browser sends a request then it is always interpreted as a new request because http protocol is a stateless protocol as it is not persistent. Whenever we want that out request object to stay alive till we decide to end the request object then, there we use the concept of session tracking. In session tracking firstly a session object is created when the first request goes to the server. Then server creates a token which will be used to maintain the session. The token is transmitted to the client by the response object and gets stored on the client machine. By default the server creates a cookie and the cookie get stored on the client machine.
3. **Cookies:** When cookie based session management is used, a token is generated which contains user's information, is sent to the browser by the server. The cookie is sent back to the server when the user sends a new request. By this cookie, the server is able to identify the user. In this way the session is maintained. Cookie is nothing but a name-value pair, which is stored on the client machine. By default the cookie is implemented in most of the browsers. If we want then we can also disable the cookie. For security reasons, cookie based session management uses two types of cookies.

Program To Determine whether the Session is New or Old

In this program we are going to make one servlet on session in which we will check whether the session is new or old.

To make this program firstly we need to make one class named **CheckingTheSession**. Inside the **doGet()** method, which takes two objects one of request and second of response. Inside this method call the method **getWriter()** of the response object. Use **getSession()** of the request object, which returns the **HttpSession** object. Now by using the **HttpSession** we can find out whether the session is new or old.

The code of the program is given below:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class CheckingTheSession extends HttpServlet{
  protected void doGet(HttpServletRequest request, HttpServletResponse respon
se)
                                    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.println("Checking whether the session is new or old<br>");
    HttpSession session = request.getSession();
    if(session.isNew()){
      pw.println("You have created a new session");
    }
    else{
      pw.println("Session already exists");
    }
  }
}
```

4.5 Cookie class

Cookies are state objects stored by a Web browser (or other HTTP client) and can be used by server-side applications to store and retrieve information. Cookies can be created by servlets (or CGI scripts) and sent to the browser. For every subsequent request made by the browser, the cookie is sent as part of the HTTP request. This allows server-side applications to access state information, without the effort of encoding it in a hyperlink or HTML form. Figure 1 shows an example transaction between a server-side application and a browser where a cookie is stored and then returned.

Storing cookies from a Java servlet

Support for cookies has been included in the Servlet API and provides an extremely easy interface for storing and retrieving cookies. Cookies are represented by the

`javax.servlet.http.Cookie`

class. The constructor takes two strings as parameters — the name of the cookie (which is fixed) and the value (which can be changed at a later date).

```
// Create a new cookie
Cookie myCookie = new Cookie ( "accountID" , "212994234");
```

Servlets that wish to set cookies must add their cookie to the response sent back to the browser.

```
HttpServletResponse
```

offers an

```
addCookie(Cookie)
```

method, which can be invoked once or multiple times to add additional cookies.

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
{
 // Store state information in browser cookies
 res.addCookie (new Cookie ("thecounter", "1");

 // Additional servlet code would go here.....
}
```

Reading cookies from a Java servlet

Accessing stored cookies from a servlet is also easy. Cookies are sent each time a request is made, so that if a cookie is already stored in the browser, it can be accessed by invoking the

```
Cookies[] getCookies()
```

method of

```
javax.servlet.http.HttpServletRequest
```

. This returns an array of Cookie objects, or

```
Null
```

if no cookies are present.

```
// Get cookie array from HttpServletRequest
Cookie[] cookieArray = request.getCookies();

// Guard statement to check for missing cookies
if (cookieArray != null)
{
 // Print a list of all cookies sent by browser
 for (int i =0; i< cookieArray.length; i++)
 {
  Cookie c = cookieArray[i];
  pout.println ("Name : " + c.getName());
  pout.println ("Value: " + c.getValue());
 }
}
else
pout.println ("No cookies present, or browser does not support cookies");
```

4.6 Servlet- Jdbc

Accessing Access Database From Servlet

```
import java.io.*;
import java.util.Enumeration;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;
import java.net.*;

public class emaildb extends HttpServlet{
     Connection theConnection;
     private ServletConfig config;

public void init(ServletConfig config)
  throws ServletException{
     this.config=config;
   }

public void service (HttpServletRequest req,
HttpServletResponse res)
throws ServletException, IOException {

   HttpSession session = req.getSession(true);

   res.setContentType("text/html");

   PrintWriter out = res.getWriter();

   out.println("<HTML><HEAD><TITLE>Emai
List.</TITLE>");
```

```
   out.println("</HEAD>");

   out.println("<BODY bgColor=blanchedalmond
text=#008000 topMargin=0>");

   out.println("<P align=center><FONT
face=Helvetica><FONT color=fuchsia
style=\"BACKGROUND-COLOR: white\"><BIG><BIG>List of
E-mail addresses.</BIG></BIG></FONT></P>");

    out.println("<P align=center>");

out.println("<TABLE align=center border=1
cellPadding=1 cellSpacing=1 width=\"75%\">");



    out.println("<TR>");

    out.println("<TD>Name</TD>");

    out.println("<TD>E-mail</TD>");

    out.println("<TD>Website</TD></TR>");

try{


     //Loading Sun's JDBC ODBC Driver
     Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");


   //Connect to emaildb Data source
   theConnection =
DriverManager.getConnection("jdbc:odbc:emaildb",
"admin", "");


   Statement
theStatement=theConnection.createStatement();

   ResultSet
theResult=theStatement.executeQuery("select * from
emaillists"); //Select all records from emaillists
table.

  //Fetch all the records and print in table
    while(theResult.next()){

   out.println();
   out.println("<TR>");

   out.println("<TD>" + theResult.getString(1) +
"</TD>");

   out.println("<TD>" + theResult.getString(2) +
"</TD>");

   String s=theResult.getString(3);

   out.println("<TD><a href=" + s + ">" + s +
"</a></TD>");

   out.println("</TR>");

    }

  theResult.close();//Close the result set

  theStatement.close();//Close statement

  theConnection.close(); //Close database Connection

  }catch(Exception e){

   out.println(e.getMessage());//Print trapped
error.

  }

  out.println("</TABLE></P>");

  out.println("<P>&nbsp;</P></FONT></BODY></HTML>");

 }

  public void destroy(){

  }

}
```

Compile emaildb.java file, move emaildb.class file to Java Web Servers servlets directory and register the servlet. Now open your browser and run the servlet.

4.Introduction to JSP

Java Server Pages or JSP for short is Sun's solution for developing dynamic web sites. JSP provide excellent server side scripting support for creating database driven web applications. JSP enable the developers to directly insert java code into jsp file, this makes the development process very simple and its maintenance also becomes very easy. JSP pages are efficient, it loads into the web servers memory on receiving the request very first time and the subsequent calls are served within a very short period of time.

In today's environment most web sites servers dynamic pages based on user request. Database is very convenient way to store the data of users and other things. JDBC provide excellent database connectivity in heterogeneous database environment. Using JSP and JDBC its very easy to develop database driven web application.

Java is known for its characteristic of "write once, run anywhere." JSP pages are platform independent. Your port your .jsp pages to any platform.

To process a JSP file, we need a JSP engine that can be connected with a web server or can be accommodated inside a web server. Firstly when a web browser seeks a JSP file through an URL from the web server, the web server recognizes the .jsp file extension in the URL requested by the browser and understands that the requested resource is a JavaServer Page. Then the web server passes the request to the JSP engine. The JSP page is then translated into a Java class, which is then compiled into a servlet.

This translation and compilation phase occurs only when the JSP file is requested for the first time, or if it undergoes any changes to the extent of getting retranslated and recompiled. For each additional request of the JSP page thereafter, the request directly goes to the servlet byte code, which is already in memory. Thus when a request comes for a servlet, an init() method is called when the Servlet is first loaded into the virtual machine, to perform any global initialization that every request of the servlet will need. Then the individual requests are sent to a service() method, where the response is put together. The servlet creates a new thread to run service() method for each request. The request from the browser is converted into a Java object of type HttpServletRequest, which is passed to the Servlet along with an HttpServletResponse object that is used to send the response back to the browser. The servlet code performs the operations specified by the JSP elements in the .jsp file.

How JSP and JSP Container function

A JSP page is executed in a JSP container or a JSP engine, which is installed in a web server or in a application server. When a client asks for a JSP page the engine wraps up the request and delivers it to the JSP page along with a response object. The JSP page processes the request and modifies the response object to incorporate the communication with the client. The container or the engine, on getting the response, wraps up the responses from the JSP page and delivers it to the client. The underlying layer for a JSP is actually a servlet implementation. The abstractions of the request and response are the same as the ServletRequest and ServletResponse respectively.

If the protocol used is HTTP, then the corresponding objects are HttpServletRequest and HttpServletResponse.

The first time the engine intercepts a request for a JSP, it compiles this translation unit (the JSP page and other dependent files) into a class file that implements the servlet protocol. If the dependent files are other JSPs they are compiled into their own classes. The servlet class generated at the end of the translation process must extend a superclass that is either

1. specified by the JSP author through the use of the extends attribute in the page directive or

2. is a JSP container specific implementation class that implements javax.servlet.jsp.JspPage interface and provides some basic page specific behavior.

Since most JSP pages use HTTP, their implementation classes must actually implement the javax.servlet.jsp.HttpJspPage interface, which is a sub interface of javax.servlet.jsp.JspPage.

The javax.servlet.jsp.JspPage interface contains two methods:

1. public void jspInit() - This method is invoked when the JSP is initialized and the page authors are free to provide initialization of the JSP by implementing this method in their JSPs.

2. public void jspDestroy() - This method is invoked when the JSP is about to be destroyed by the container. Similar to above, page authors can provide their own implementation.

The javax.servlet.jsp.HttpJspPage interface contains one method:

public void _jspService(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException

This method generated by the JSP container is invoked, every time a request comes to the JSP. The request is processed and the JSP generates appropriate response. This response is taken by the container and passed back to the client.

JSP Architecture

There are two basic ways of using the JSP technology. They are the client/server (page-centric) 2-tier approach and the N-tier approach (dispatcher).

The Page-Centric Approach

Applications built using a client-server (2-tier) approach consist of one or more application programs running on client machines and connecting to a server-based application to work. With the arrival of Servlets technology, 2-tier applications could also be developed using Java programming language. This model allows JSPs or Servlets direct access to some resource such as database or legacy application to service a client's request. The JSP page is where the incoming request is intercepted, processed and the response sent back to the client. JSPs differ from Servlets in this scenario by providing clean code, separating code from the content by placing data access in EJBs. Even though this model makes application development easier, it does not scale up well for a large number of simultaneous clients as it entails a significant amount of request processing to be performed and each request must establish or share a potentially scarce/expensive connection to the resource in question.

Page-view - This basic architecture involves direct request invocations to a server page with embedded Java code, and markup tags which dynamically generate output for substitution within the HTML. This approach has been blessed a number of benefits. It is very straightforward and is a low-overhead approach from a developerment perspective. All the Java code may be embedded within the HTML, so changes are confined to a very limited area, reducing complexity drastically.

The big trade-off here is in the level of sophistication. As the scale of the system grows, some limitations begin to surface, such as bloating of business logic code in the page instead of factoring forward to a mediating Servlet or factoring back to a worker bean. It is a fact that utilizing a Servlet and helper beans helps to separate developer roles more cleanly and improves the potential for code reuse.

Page-view with bean - This pattern is used when the above architecture becomes too cluttered with business-related code and data access code. The Java code representing the business logic and simple data storage implementation in the previous model moves from the JSP to the JavaBean worker. This refactoring leaves a much cleaner JSP with limited Java code, which can be comfortably owned by an individual in a web-production role, since it encapsulates mostly markup tags.

The Dispatcher Approach

In this approach, a Servlet or JSP acts as a mediator or controller, delegating requests to JSP pages and JavaBeans. There are three different architectures. They are mediator-view, mediator-composite view and service to workers.

In an N-tier application, the server side of the architecture is broken up into multiple tiers. In this case, the application is composed of multiple tiers, where the middle tier, the JSP, interacts with the back end resources via another object or EJBs component. The Enterprise JavaBeans server and the EJB provide managed access to resources, support transactions and access to underlying security mechanisms, thus addressing the resource sharing and performance issues of the 2-tier approach.

The first step in N-tiered application design should be identifying the correct objects and their interaction and the second step is identifying the JSPs or Servlets. These are divided into two categories.

Front end JSPs or Servlets manage application flow and business logic evaluation. They act as a point to intercept the HTTP requests coming from the users. They provide a single entry point to an application, simplifying security management and making application state easier to maintain.

Presentation JSPs or Servlets generate HTML or XML with their main purpose in life being presentation of dynamic content. They contain only presentation and rendering logic.

These categories resemble to the Modal-View design pattern, where the front-end components is the model and the presentation component the view. In this approach, JSPs are used to generate the presentation layer and either JSPs or Servlets to perform process-intensive tasks. The front-end component acts as the controller and is in charge of the request processing and the creation of any beans or objects used by the presentation JSP, as well as deciding, depending on the user's actions, which JSP to forward this request to. There is no processing logic within the presentation JSP itself and it simply responsible for retrieving any objects or beans that may have been previously created by the Servlet and extracting the dynamic content for insertion within static templates.

Benefits of JSP

One of the main reasons why the JavaServer Pages technology has evolved into what it is today and it is still evolving is the overwhelming technical need to simplify application design by separating dynamic content from static template display data. Another benefit of utilizing JSP is that it allows to more cleanly separate the roles of web application/HTML designer from a software developer. The JSP technology is blessed with a number of exciting benefits, which are chronicled as follows:

1. The JSP technology is platform independent, in its dynamic web pages, its web servers, and its underlying server components. That is, JSP pages perform perfectly without any hassle on any platform, run on any web server, and web-enabled application server. The JSP pages can be accessed from any web server.

2. The JSP technology emphasizes the use of reusable components. These components can be combined or manipulated towards developing more purposeful components and page design. This definitely reduces development time apart from the At development time, JSPs are very different from Servlets, however, they are precompiled into Servlets at run time and executed by a JSP engine which is installed on a Web-enabled application server such as BEA WebLogic and IBM WebSphere.

7.1 Components of JSP – Directives , Tags, Scripting Elements

JSP page is built using components such as :

5. Directives
6. Declarations
7. Scriplets
8. Expressions
9. Standard Actions
10. Custom Tags

Directives :

JSP directives serve as messages to the JSP container from the JSP. They are used to set global values such as class declaration, methods to be implemented, output content type, etc. They do not produce any output to the client. All directives have scope of the entire JSP file. That is, a directive affects the whole JSP file, and only that JSP file. Directives are characterized by the @ character within the tag .

Listing some of them:

1) **Page**

Syntax : < %@ page Language="Java" extends="<Class name>" import="<class> or <package>" %>

Attributes:

a. Language = "Java"
b. Import = "Class"
c. Buffersize = ""
d. Scope = "REQUEST/PAGE/SESSION/APPLICATION"
e. And etc….

Page directive is aimed to define certain attribute of a JSP page for e.g. Language of the page in which the page content should be written , which class to be imported so that it can be used within the JSP page.

2) **Include**

Syntax: <%@ include file="<filename>" %>

Attributes:

a. file = "<filename>"

This directive is to include the a HTML, JSP or Sevlet file into a JSP file. This is a static inclusion of file i.e. it will be included into the JSP file at the time of compilation and once the JSP file is compiled any changes in the included the file will not the reflected.

Declarations:

A declaration is a block of Java code in a JSP that is used to define class-wide variables and methods in the generated class file. Declarations are initialized when the JSP page is initialized and have class scope. Anything defined in a declaration is available throughout the JSP, to other declarations, expressions or code

JSP Declaratives begins with <%! and ends %> with .We can embed any amount of java code in the JSP Declaratives. Variables and functions defined in the declaratives are class level and can be used anywhere in the JSP page.

Syntax: <%! Declare all the variables here %>

Scriplets:

A scriptlet consists of one or more valid Java statements. A scriptlet is a block of Java code that is executed at request-processing time. A scriptlet is enclosed between "<%" and "%>". What the scriptlet actually does depends on the code, and it can produce output into the output stream to the client. Multiple scriptlets are combined in the compiled class in the order in which they appear in the JSP. Scriptlets like any other Java code block or method, can modify objects inside them as a result of method invocations.

JSP Scriptlets begins with <% and ends %> .We can embed any amount of java code in the JSP Scriptlets. JSP Engine places these code in the *_jspService()* method.

Syntax: <% All your scripts will come here %>

Expressions:

An expression is a shorthand notation for a scriptlet that outputs a value in the response stream back to the client. When the expression is evaluated, the result is converted to a string and displayed, An expression is enclosed within <%= and %> "<%=" and "%>". If any part of expression is an object, the conversion is done using the toString() method of the object.

Syntax: <%= expression evaluation and display the variable %>

Standard Action:

What is JSP Actions?

Standard actions are specific tags that affect the runtime behavior of the JSP and affect the response sent back to the client. The JSP specification lists some standard action types to be provided by all containers, irrespective of the implementation. Standard actions provide page authors with some basic functionality to exploit; the vendor is free to provide other actions to enhance behavior.

Servlet container provides many built in functionality to ease the development of the applications. Programmers can use these functions in JSP applications. The JSP Actions tags enables the programmer to use these functions. The JSP Actions are XML tags that can be used in the JSP page.

Here is the list of JSP Actions:

- **jsp:include**
  The **jsp:include** action work as a subroutine, the Java servlet temporarily passes the request and response to the specified JSP/Servlet. Control is then returned back to the current JSP page.

- **jsp:param**
  The **jsp:param** action is used to add the specific parameter to current request. The jsp:param tag can be used inside a jsp:include, jsp:forward or jsp:params block.

- **jsp:forward**
  The **jsp:forward** tag is used to hand off the request and response to another JSP or servlet. In this case the request never return to the calling JSP page.

- **jsp:plugin**
  In older versions of Netscape Navigator and Internet Explorer; different tags is used to embed applet. The **jsp:plugin** tag actually generates the appropriate HTML code the embed the Applets correctly.

- **jsp:fallback**
  The **jsp:fallback** tag is used to specify the message to be shown on the browser if applets is not supported by browser.
  Example:
  ```
   <jsp:fallback>
       <p>Unable to load applet</p>
     </jsp:fallback>
  ```

- **jsp:getProperty**
  The **jsp:getPropertyB** is used to get specified property from the JavaBean object.

- **jsp:setProperty**
  The **jsp:setProperty** tag is used to set a property in the JavaBean object.

- **jsp:useBean**
  The **jsp:useBean** tag is used to instantiate an object of Java Bean or it can re-use existing java bean object.

Custom Tags:

taglib

Syntax: <%@ taglib uri="<tag library uri>" prefix="<tagprefix>" %>

Attributes:

a. uri = "<relative path of the tag library uri>"
b. prefix = "<tagprefix>"

prefix is alias name for the tag library name.

JSP provides certain Implicit Objects listed below.

Object	**Of Kind**
Out	JSP writer
Request	HttpServletRequest
Response	HttpServletRespose
Session	HttpSession
Application	ServletContext
Config	Sevlet Config
Page	Object
PageContext	Page Context => is responsible for generating all other implicit objects.

Out:

This object is instantiated implicitly from JSP Writer class and can be used for displaying anything within delimiters.

For e.g. out.println("Hi Buddy");

Request:

It is also an implicit object of class HttpServletRequest class and using this object request parameters can be accessed.

For e.g. in case of retrieval of parameters from last form is as follows:

```
request.getParameters(“Name”);
```

Where “Name” is the form element.

Response:

It is also an implicit object of class HttpServletResponse class and using this object response(sent back to browser) parameters can be modified or set.

For e.g. in case of modifying the HTTP headers you can use this object.

```
Response.setBufferSize(“50”);
```

Session:

Session object is of class HttpSession and used to maintain the session information. It stores the information in Name-Value pair.

For e.g.

```
session.setValue(“Name”,”Jakes”);
session.setValue(“Age”,”22”);
```

Application:

This object belongs to class SevletContext and used to maintain certain information throughout the scope of Application.

For e.g.

```
Application.setValue(“servername”,”www.myserver.com”);
```

PageContext:

This object is of class pageContext and is utilized to access the other implicit objects.

7.2 Building a simple application using JSP

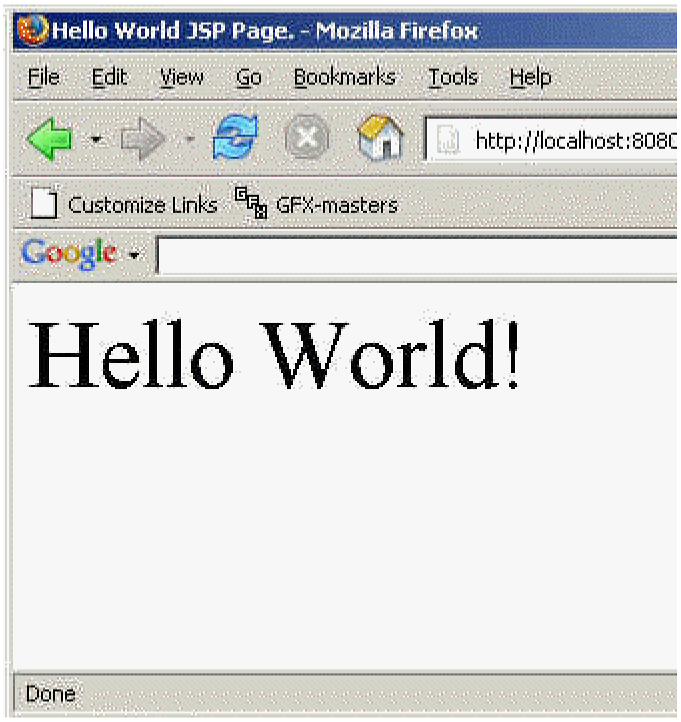
This simple page contains a JSP program and it's output on browser in running state. This given program of JSP illustrates you how to print simple "Hello World!" string on the browser through the server side code (provided by JSP).

This program also contains HTML (Hypertext Markup Language) code for designing the page and the contents. Following code of the program prints the string "Hello World!" by using **<%="Hello World!" %>** while you can also print the string by using **out.println("Hello World!")** in the **<%** and **%>** JSP tags.

Here is the code of the program:

```
<html>
        <head><title>Hello World JSP Page.</title></head>
        <body>
                <font size="10"><%="Hello World!" %></font>
        </body>
</html>
```

Output of the program:



Hello World!

2. MultiThreading

2.1 Threading basics

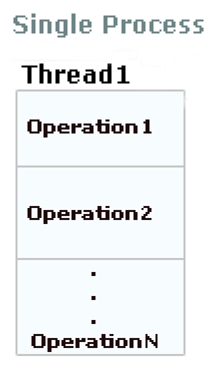
Process

A **process** is an instance of a computer program that is executed sequentially. It is a collection of instructions which are executed simultaneously at the rum time. Thus several processes may be associated with the same program. For example, to check the **spelling** is a single process in the **Word Processor** program and you can also use other processes like **printing, formatting, drawing, etc**. associated with this program.

Thread

A thread is a **lightweight** process which exist within a program and executed to perform a special task. Several threads of execution may be associated with a single process. Thus a process that has only one thread is referred to as a **single-threaded** process, while a process with multiple threads is referred to as a **multi-threaded** process.

In Java Programming language, thread is a sequential path of code execution within a program. Each thread has its own local variables, program counter and lifetime. In single threaded runtime environment, operations are executes sequentially i.e. next operation can execute only when the previous one is complete. It exists in a common memory space and can share both data and code of a program. Threading concept is very important in Java through which we can increase the speed of any application. You can see diagram shown below in which a thread is executed along with its several operations with in a single process.



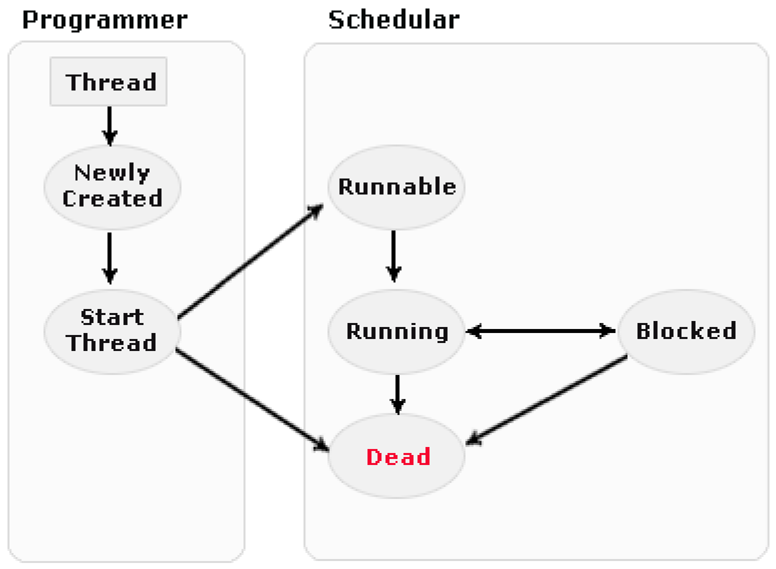
Main Thread

When any standalone application is running, it firstly execute the **main()** method runs in a one thread, called the main thread. If no other threads are created by the main thread, then program terminates when the main( ) method complete its execution. The main thread creates some other threads called child threads. The main() method execution can finish, but the program will keep running until the all threads have complete its execution.

2.2 Life cycle of thread

When you are programming with threads, understanding the life cycle of thread is very valuable. While a thread is alive, it is in one of several states. By invoking start() method, it doesn't mean that the thread has access to CPU and start executing straight away. Several factors determine how it will proceed.

Different states of a thread are:



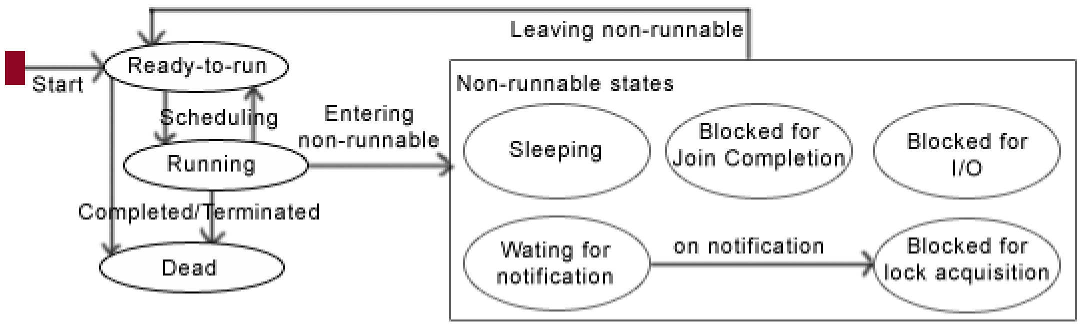
1. **New state –** After the creations of Thread instance the thread is in this state but before the start() method invocation. At this point, the thread is considered not alive.

2. **Runnable (Ready-to-run) state –** A thread start its life from Runnable state. A thread first enters runnable state after the invoking of start() method but a thread can return to this state after either running, waiting, sleeping or coming back from blocked state also. On this state a thread is waiting for a turn on the processor.

3. **Running state –** A thread is in running state that means the thread is currently executing. There are several ways to enter in Runnable state but there is only one way to enter in Running state: the scheduler select a thread from runnable pool.

4. **Dead state –** A thread can be considered dead when its run() method completes. If any thread comes on this state that means it cannot ever run again.
5. **Blocked -** A thread can enter in this state because of waiting the resources that are hold by another thread.

Different states implementing Multiple-Threads are:



As we have seen different states that may be occur with the single thread. A running thread can enter to any non-runnable state, depending on the circumstances. A thread cannot enters directly to the running state from non-runnable state, firstly it goes to runnable state. Now lets understand the some non-runnable states which may be occur handling the multithreads.

- **Sleeping –** On this state, the thread is still alive but it is not runnable, it might be return to runnable state later, if a particular event occurs. On this state a thread sleeps for a specified amount of time. You can use the method **sleep( )** to stop the running state of a thread.

  static void sleep(long millisecond) throws InterruptedException
- **Waiting for Notification –** A thread waits for notification from another thread. The thread sends back to runnable state after sending notification from another thread.

  final void wait(long timeout) throws InterruptedException
  final void wait(long timeout, int nanos) throws InterruptedException

final void wait() throws InterruptedException

- **Blocked on I/O –** The thread waits for completion of blocking operation. A thread can enter on this state because of waiting I/O resource. In that case the thread sends back to runnable state after availability of resources.
- **Blocked for joint completion –** The thread can come on this state because of waiting the completion of another thread.
- **Blocked for lock acquisition –** The thread can come on this state because of waiting to acquire the lock of an object.

Methods that can be applied apply on a Thread:

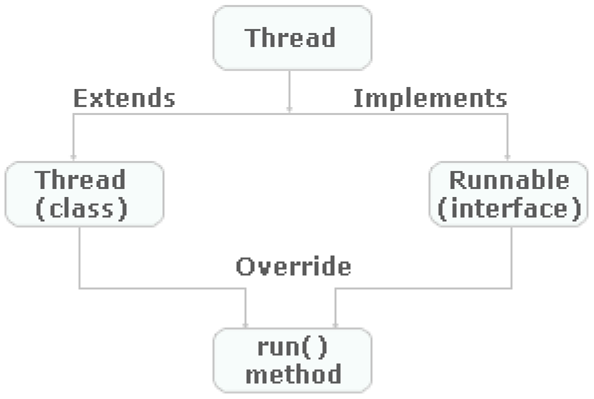
Some Important Methods defined in **java.lang.Thread** are shown in the table:

Method	**Return Type**	**Description**
currentThread( )	Thread	Returns an object reference to the thread in which it is invoked.
getName( )	String	Retrieve the name of the thread object or instance.
start( )	void	Start the thread by calling its run method.
run( )	void	This method is the entry point to execute thread, like the main method for applications.
sleep( )	void	Suspends a thread for a specified amount of time (in milliseconds).
isAlive( )	boolean	This method is used to determine the thread is running or not.
activeCount( )	int	This method returns the number of active threads in a particular thread group and all its subgroups.
interrupt( )	void	The method interrupt the threads on which it is invoked.
yield( )	void	By invoking this method the current thread pause its execution temporarily and allow other threads to execute.
join( )	void	This method and **join(long millisec)** Throws InterruptedException. These two methods are invoked on a thread. These are not returned until either the thread has completed or it is timed out respectively.

2.3 Creating Threads

In Java, an object of the Thread class can represent a thread. Thread can be implemented through any one of two ways:

- **Extending the java.lang.Thread Class**
- **Implementing the java.lang.Runnable Interface**



I. Extending the java.lang.Thread Class

For creating a thread a class have to extend the Thread Class. For creating a thread by this procedure you have to follow these steps:

1. Extend the **java.lang.Thread** Class.
2. Override the **run( )** method in the subclass from the Thread class to define the code executed by the thread.
3. Create an **instance** of this subclass. This subclass may call a Thread class constructor by subclass constructor.
4. Invoke the **start( )** method on the instance of the class to make the thread eligible for running.

The following program demonstrates a single thread creation extending the "Thread" Class:

```
class MyThread extends Thread{

  String s=null;

  MyThread(String s1){
    s=s1;
    start();
  }
  public void run(){
      System.out.println(s);
    }
}
public class RunThread{
  public static void main(String args[]){

    MyThread m1=new MyThread("Thread started....");
   }
}
```

Output of the Program is :

```
C:\j2se6\thread>javac
RunThread.java

C:\j2se6\thread>java RunThread
Thread started....
```

II. Implementing the java.lang.Runnable Interface

The procedure for creating threads by implementing the Runnable Interface is as follows:

1. A Class implements the **Runnable** Interface, override the run() method to define the code executed by thread. An object of this class is Runnable Object.
2. Create an object of **Thread** Class by passing a Runnable object as argument.
3. Invoke the **start( )** method on the instance of the Thread class.

The following program demonstrates the thread creation implenting the Runnable interface:

```
class MyThread1 implements Runnable{
  Thread t;
  String s=null;

  MyThread1(String s1){
     s=s1;
    t=new Thread(this);
    t.start();
  }
  public void run(){
      System.out.println(s);
     }
}
public class RunableThread{
  public static void main(String args[]){
    MyThread1 m1=new MyThread1("Thread started....");
    }
}
```

However, this program returns the output same as of the output generated through the previous program.

Output of the Program is:

```
C:\j2se6\thread>javac
RunableThread.java

C:\j2se6\thread>java RunableThread
Thread started....
```

There are two reasons for implementing a Runnable interface preferable to extending the Thread Class:

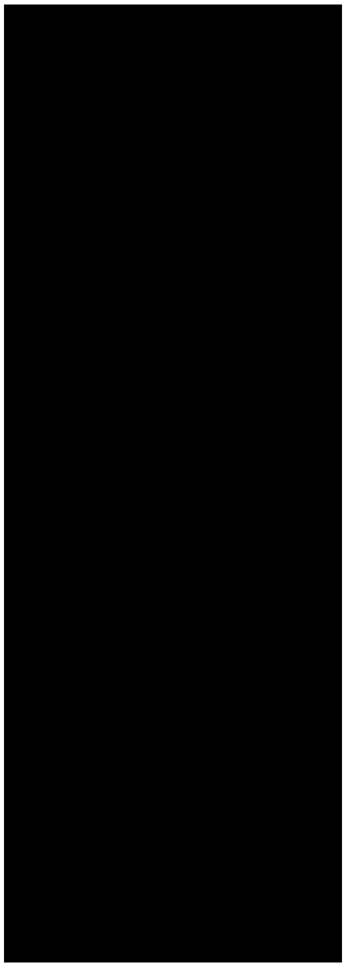
1. If you extend the **Thread** Class, that means that subclass cannot extend any other Class, but if you implement **Runnable** interface then you can do this.
2. The class implementing the **Runnable** interface can avoid the full overhead of **Thread** class which can be excessive.

join() & isAlive() methods:

The following program demonstrates the **join()** & **isAlive()** methods:

```
class DemoAlive extends Thread {
    int value;

    public DemoAlive(String str){
        super(str);
        value=0;
        start();
    }

   public void run(){
        try{
            while (value < 5){
    System.out.println(getName() + ": " + (value++));
             Thread.sleep(250);
            }
        } catch (Exception e) {}
        System.out.println("Exit from thread: " + getName());
    }
}

public class DemoJoin{

  public static void main(String[] args){
        DemoAlive da = new DemoAlive("Thread a");
        DemoAlive db = new DemoAlive("Thread b");
        try{
            System.out.println("Wait for the child threads to finish.");
            da.join();
            if (!da.isAlive())
                System.out.println("Thread A not alive.");

            db.join();

            if (!db.isAlive())
                System.out.println("Thread B not alive.");
        } catch (Exception e) { }
        System.out.println("Exit from Main Thread.");
    }
}
```

Output of this program is:



Creation of Multiple Threads

Like creation of a single thread, You can also create more than one thread (multithreads) in a program using class **Thread** or implementing interface **Runnable**.

Let’s see an example having the implementation of the multithreads by extending **Thread** Class:

```
 class MyThread extends Thread{
  MyThread(String s){
    super(s);
    start();
  }
  public void run(){
    for(int i=0;i<5;i++){
      System.out.println("Thread Name  :"
              +Thread.currentThread().getName());
      try{
        Thread.sleep(1000);
      }catch(Exception e){}
    }
  }
}
  public class MultiThread1{
  public static void main(String args[]){
    System.out.println("Thread Name :"
           +Thread.currentThread().getName());
    MyThread m1=new MyThread("My Thread 1");
    MyThread m2=new MyThread("My Thread 2");
  }
}
```

Output of the Program

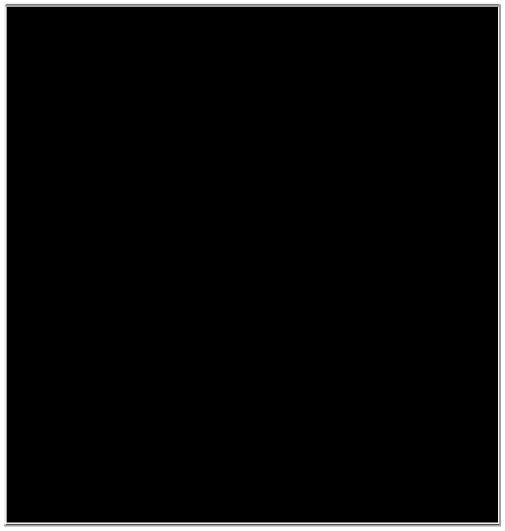
```
C:\nisha>javac
MultiThread1.java

C:\nisha>java MultiThread1
Thread Name :main
Thread Name :My Thread 1
Thread Name :My Thread 2
Thread Name :My Thread 1
Thread Name :My Thread 2
Thread Name :My Thread 1
Thread Name :My Thread 2
Thread Name :My Thread 1
Thread Name :My Thread 2
Thread Name :My Thread 1
Thread Name :My Thread 2
```

In this program, two threads are created along with the "**main**" thread. The **currentThread()** method of the **Thread** class returns a reference to the currently executing thread and the **getName( )** method returns the name of the thread. The sleep( ) method pauses execution of the current thread for 1000 milliseconds(1 second) and switches to the another threads to execute it. At the time of execution of the program, both threads are registered with the **thread scheduler** and the CPU scheduler executes them one by one.

Now, lets create the same program implenting the **Runnable** interface:

```
class MyThread1 implements Runnable{
  Thread t;
  MyThread1(String s)  {
    t=new Thread(this,s);
    t.start();
  }

  public void run()  {
    for(int i=0;i<5;i++) {
      System.out.println("Thread Name  :"+Thread.currentThread().getName());
      try {
      Thread.sleep(1000);
      }catch(Exception e){}
    }
  }
}

public class RunnableThread1{
  public static void main(String args[])  {
    System.out.println("Thread Name :"+Thread.currentThread().getName());
    MyThread1 m1=new MyThread1("My Thread 1");
    MyThread1 m2=new MyThread1("My Thread 2");
  }
}
```

Output of the program:



Note that, this program gives the same output as the output of the previous example. It means, you can use either class **Thread** or interface **Runnable** to implement thread in your program.

2.4 Priorities and Synchronization

Priorities

In Java, thread scheduler can use the thread **priorities** in the form of **integer value** to each of its thread to determine the execution schedule of threads . Thread gets the **ready-to-run** state according to their priorities. The **thread scheduler** provides the CPU time to thread of highest priority during ready-to-run state.
Priorities are integer values from 1 (lowest priority given by the constant **Thread.MIN_PRIORITY**) to 10 (highest priority given by the constant **Thread.MAX_PRIORITY**). The default priority is 5(**Thread.NORM_PRIORITY**).

Constant	**Description**
Thread.MIN_PRIORITY	The maximum priority of any thread (an int value of 10)
Thread.MAX_PRIORITY	The minimum priority of any thread (an int value of 1)
Thread.NORM_PRIORITY	The normal priority of any thread (an int value of 5)

The methods that are used to set the priority of thread shown as:

Method	**Description**
setPriority()	This is method is used to set the priority of thread.
getPriority()	This method is used to get the priority of thread.

When a Java thread is created, it inherits its priority from the thread that created it. At any given time, when multiple threads are ready to be executed, the runtime system chooses the runnable thread with the highest priority for execution. In Java runtime system, **preemptive scheduling** algorithm is applied. If at the execution time a thread with a higher priority and all other threads are runnable then the runtime system chooses the new **higher priority** thread for execution. On the other hand, if two threads of the same priority are waiting to be executed by the CPU then the **round-robin** algorithm is applied in which the scheduler chooses one of them to run according to their round of **time-slice**.
Thread Scheduler

In the implementation of threading scheduler usually applies one of the two following strategies:

- **Preemptive scheduling –** If the new thread has a higher priority then current running thread leaves the runnable state and higher priority thread enter to the runnable state.

- **Time-Sliced (Round-Robin) Scheduling –** A running thread is allowed to be execute for the fixed time, after completion the time, current thread indicates to the another thread to enter it in the runnable state.

You can also set a thread's priority at any time after its creation using the `setPriority` method. Lets see, how to set and get the priority of a thread.

```
class MyThread1 extends Thread{
  MyThread1(String s){
    super(s);
    start();
  }
  public void run(){
    for(int i=0;i<3;i++){
      Thread cur=Thread.currentThread();
      cur.setPriority(Thread.MIN_PRIORITY);
      int p=cur.getPriority();
      System.out.println("Thread Name  :"+Thread.currentThread().getName());
      System.out.println("Thread Priority  :"+cur);
      }
  }
}
  class MyThread2 extends Thread{
  MyThread2(String s){
    super(s);
    start();
  }

public void run(){
    for(int i=0;i<3;i++){
      Thread cur=Thread.currentThread();
      cur.setPriority(Thread.MAX_PRIORITY);
      int p=cur.getPriority();
      System.out.println("Thread Name  :"+Thread.currentThread().getName());
      System.out.println("Thread Priority  :"+cur);
      }
  }
}
public class ThreadPriority{
  public static void main(String args[]){
    MyThread1 m1=new MyThread1("My Thread 1");
    MyThread2 m2=new MyThread2("My Thread 2");
  }
}
```

Output of the Program:

In this program two threads are created. We have set up maximum priority for the first thread "**MyThread2**" and minimum priority for the first thread "**MyThread1**" i.e. the after executing the program, the first thread is executed only once and the second thread "**MyThread2**" started to run until either it gets end or another thread of the equal priority gets ready to run state.

<u>Synchronization</u>

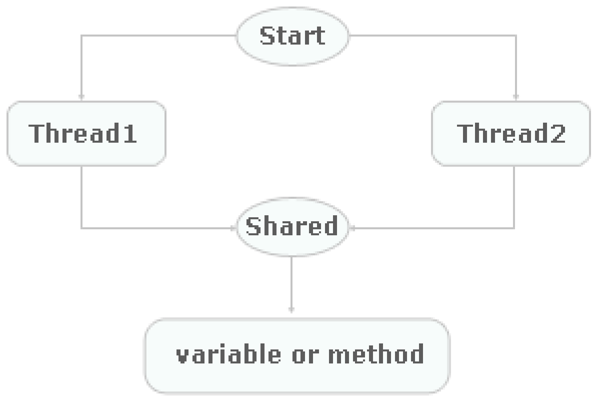
In Java, the threads are executed independently to each other. These types of threads are called as **asynchronous threads**. But there are two problems may be occur with asynchronous threads.

- Two or more threads share the same resource (variable or method) while only one of them can access the resource at one time.
- If the producer and the consumer are sharing the same kind of data in a program then either producer may produce the data faster or consumer may retrieve an order of data and process it without its existing.

Suppose, we have created two methods as **increment( )** and **decrement( )**. which increases or decreases value of the variable **"count"** by 1 respectively shown as:

```
public void increment() {
    count++;
  }

  public void decrement( )
{
    count--;
 }
public int value() {
        return count;
    }
```

When the two threads are executed to access these methods (one for **increment( )**,another for **decrement( )**) then both will share the variable **"count"**. in that case, we can't be sure that what value will be returned of variable "count".
We can see this problem in the diagram shown below:



To avoid this problem, Java uses **monitor** also known as **"semaphore"** to prevent data from being corrupted by multiple threads by a keyword **synchronized** to synchronize them and intercommunicate to each other. It is basically a mechanism which allows two or more threads to share all the available resources in a sequential manner. Java's synchronized is used to ensure that only one thread is in a **critical region**. critical region is a lock area where only one thread is run (or lock) at a time. Once the thread is in its critical section, no other thread can enter to that critical region. In that case, another thread will has to wait until the current thread leaves its critical section.

General form of the synchronized statement is as:

```
synchronized(object)
{
```

```
// statements to be
synchronized

}
```

Lock:

Lock term refers to the access granted to a particular thread that can access the shared resources. At any given time, only one thread can hold the lock and thereby have access to the shared resource. Every object in Java has build-in lock that only comes in action when the object has synchronized method code. By associating a shared resource with a Java object and its lock, the object can act as a guard, ensuring synchronized access to the resource. Only one thread at a time can access the shared resource guarded by the **object lock**.

Since there is one lock per object, if one thread has acquired the lock, no other thread can acquire the lock until the lock is not released by first thread. Acquire the lock means the thread currently in synchronized method and released the lock means exits the synchronized method.
Remember the following points related to **lock** and **synchronization**:

- Only methods (or blocks) can be synchronized, Classes and variable cannot be synchronized.

- Each object has just one lock.

- All methods in a class need not to be synchronized. A class can have both synchronized and non-synchronized methods.

- If two threads wants to execute a synchronized method in a class, and both threads are using the same instance of the class to invoke the method then only one thread can execute the method at a time.

- If a class has both synchronized and non-synchronized methods, multiple threads can still access the class's non-synchronized methods. If you have methods that don't access the data you're trying to protect, then you don't need to synchronize them. Synchronization can cause a hit in some cases (or even deadlock if used incorrectly), so you should be careful not to overuse it.

- If a thread goes to sleep, it holds any locks it has—it doesn't release them.

- A thread can acquire more than one lock. For example, a thread can enter a synchronized method, thus acquiring a lock, and then immediately invoke a synchronized method on a different object, thus acquiring that lock as well. As the stack unwinds, locks are released again.

- You can synchronize a block of code rather than a method.

- Constructors cannot be synchronized

There are two ways to synchronized the execution of code:

1. Synchronized Methods

2. Synchronized Blocks (Statements)
3.

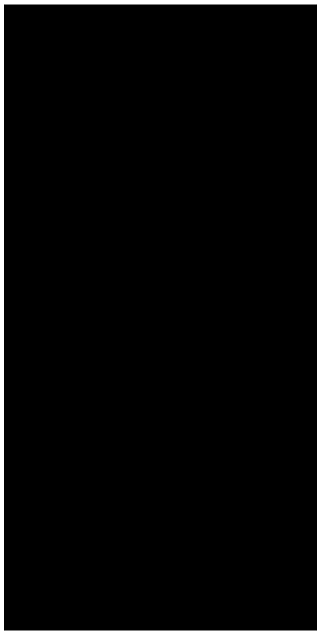
Synchronized Methods:

Any method is specified with the keyword **synchronized** is only executed by one thread at a time. If any thread want to execute the synchronized method, firstly it has to obtain the objects lock. If the lock is already held by another thread, then calling thread has to wait. Synchronized methods are useful in those situations where methods are executed concurrently, so that these can be intercommunicate manipulate the state of an object in ways that can corrupt the state if . Stack implementations usually define the two operations push and pop of elements as synchronized, that's why pushing and popping are mutually exclusive operations. For Example if several threads were sharing a stack, if one thread is popping the element on the stack then another thread would not be able to pushing the element on the stack.

The following program demonstrates the synchronized method:

```java
class Share extends Thread{
  static String msg[]={"This", "is", "a", "synchronized", "variable"};
  Share(String threadname){
    super(threadname);
  }
  public void run(){
    display(getName());
  }
  public synchronized void display(String threadN){
    for(int i=0;i<=4;i++)
      System.out.println(threadN+msg[i]);
      try{
      this.sleep(1000);
  }catch(Exception e){}
    }
}
public class SynThread1 {
  public static void main(String[] args)    {
    Share t1=new Share("Thread One: ");
    t1.start();
    Share t2=new Share("Thread Two: ");
    t2.start();
}
}
```

Output of the program is:



In this program, the method **"display( )"** is synchronized that will be shared by both thread's objects at the time of program execution. Thus only one thread can access that method and process it until all statements of the method are executed.

Synchronized Blocks (Statements)

Another way of handling synchronization is Synchronized Blocks (Statements). Synchronized statements must specify the object that provides the native lock. The synchronized block allows execution of arbitrary code to be synchronized on the lock of an arbitrary object.

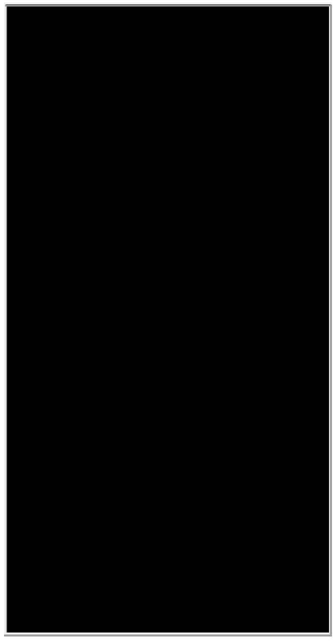
General form of synchronized block is:

```
synchronized (object reference
expression)
    {
// statements to be synchronized
}
```

The following program demonstrates the **synchronized block** that shows the same output as the output of the previous example:

```
class Share extends Thread{
  static String msg[]={"This", "is", "a", "synchronized", "variable"};
  Share(String threadname){
    super(threadname);
  }
  public void run(){
    display(getName());
  }
  public void display(String threadN){
     synchronized(this){
    for(int i=0;i<=4;i++)
      System.out.println(threadN+msg[i]);
      try{
      this.sleep(1000);
  }catch(Exception e){}
    }
}
public class SynStatement {
  public static void main(String[] args)   {
    Share t1=new Share("Thread One: ");
    t1.start();
    Share t2=new Share("Thread Two: ");
    t2.start();
}
}
```

Output of the Program



2.5 Inter Thread Communication

Java provides a very efficient way through which multiple-threads can communicate with each-other. This way reduces the CPU’s idle time i.e. A process where, a thread is paused running in its critical region and another thread is allowed to enter (or lock) in the same critical section to be executed. This technique is known as **Interthread communication** which is implemented by some methods. These methods are defined in "**java.lang**" package and can only be called within synchronized code shown as:

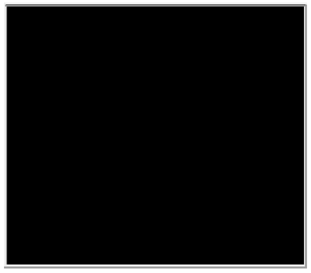
Method	**Description**
wait( )	It indicates the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls method **notify()** or **notifyAll()**.
notify( )	It wakes up the first thread that called **wait()** on the same object.
notifyAll( )	Wakes up (Unloack) all the threads that called **wait( )** on the same object. The highest priority thread will run first.

All these methods must be called within a try-catch block.

Lets see an example implementing these methods :

```
class Shared {

int num=0;
boolean value = false;

synchronized int get() {
  if (value==false)
  try {
    wait();
    }
  catch (InterruptedException e) {
  System.out.println("InterruptedException caught");
    }
System.out.println("consume: " + num);
value=false;
notify();
return num;
}

synchronized void put(int num) {
  if (value==true)
  try {
    wait();
    }
  catch (InterruptedException e) {
  System.out.println("InterruptedException caught");
    }
    this.num=num;
    System.out.println("Produce: " + num);
    value=false;
    notify();
    }
    }

    class Producer extends Thread {
  Shared s;

  Producer(Shared s) {
    this.s=s;
    this.start();
  }

  public void run() {
    int i=0;

    s.put(++i);
    }
}

class Consumer extends Thread{
  Shared s;

  Consumer(Shared s) {
    this.s=s;
    this.start();
  }

  public void run() {
    s.get();
    }
}

public class InterThread{
  public static void main(String[] args)
  {
    Shared s=new Shared();
    new Producer(s);
    new Consumer(s);
  }
}
```

Output of the Program:



In this program, two threads "**Producer**" and "**Consumer**" share the `synchronized` methods of the class **"Shared"**. At time of program execution, the **"put( )"** method is invoked through the

"Producer" class which increments the variable **"num"** by **1**. After producing 1 by the producer, the method **"get( )"** is invoked by through the **"Consumer"** class which retrieves the produced number and returns it to the output. Thus the Consumer can't retrieve the number without producing of it.

Another program demonstrates the uses of wait() & notify() methods:

```java
public class DemoWait extends Thread{
  int val=20;
  public static void main(String args[])  {
    DemoWait d=new DemoWait();
    d.start();
    new Demo1(d);
  }
  public void run(){
    try    {
    synchronized(this){
    wait();
    System.out.println("value is  :"+val);
    }
    }catch(Exception e){}
 }

  public void valchange(int val){
    this.val=val;
    try    {
    synchronized(this)    {
    notifyAll();
    }
    }catch(Exception e){}

  }
}
class Demo1 extends Thread{
  DemoWait d;
  Demo1(DemoWait d)  {
  this.d=d;
  start();
  }
  public void run(){
   try{
    System.out.println("Demo1 value is"+d.val);
    d.valchange(40);
  }catch(Exception e){}
  }
}
```

Output of the program is:

[redacted]

2.6 Runnable Interface

The `Runnable` interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called `run`.

This interface is designed to provide a common protocol for objects that wish to execute code while they are active. For example, `Runnable` is implemented by class `Thread`. Being active simply means that a thread has been started and has not yet been stopped.

In addition, `Runnable` provides the means for a class to be active while not subclassing `Thread`. A class that implements `Runnable` can run without subclassing `Thread` by instantiating a `Thread` instance and passing itself in as the target. In most cases, the `Runnable` interface should be used if you are only planning to override the `run()` method and no other `Thread` methods. This is important because classes should not be subclassed unless the programmer intends on modifying or enhancing the fundamental behavior of the class.