SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT- I Advanced Java – SBS1301

J2EE Platform

J2EE Components J2EE applications are made up of components. A J2EE component is a self-contained functional software unit that is assembled into a J2EE application with its related classes and files and that communicates with other components. The J2EE specification defines the following J2EE components:

Application clients and applets are components that run on the client.

• Java Servlet and JavaServer Pages™ (JSP™) technology components are Web components that run on the server.

• Enterprise JavaBeans™ (EJB™) components (enterprise beans) are business components that run on the server.

J2EE components are written in the Java programming language and are compiled in the same way as any program in the language.

The difference between J2EE components and “standard” Java classes is that J2EE components are assembled into a J2EE application, verified to be well formed and in compliance with the J2EE specification.

J2EE Platform Roles

The J2EE platform uses a set of defined roles to conceptualize the tasks related to the various workflows in the development and deployment life cycle of an enterprise application. These role definitions provide a logical separation of responsibilities for team members involved in the development, deployment, and management of a J2EE application

The J2EE roles are as follows:

J2EE product provider

Application component provider

Application assembler

Application deplorer

System administrator

Tool provider.

Client Side Programming

- It is the program that runs on the client machine (browser)
- Deals with the user interface/display and any other processing that can happen on client machine like reading/writing cookies.

Using Client Side Programming we can

1) Interact with temporary storage
2) Make interactive web pages

3) Interact with local storage
4) Sending request for data to server
5) Send request to server
6) work as an interface between server and user

Programming languages for client-side:

1) Javascript
2) VBScript
3) HTML
4) CSS
5) AJAX

Server-side Programming

It is the program that runs on server dealing with the generation of content of web page.

Need for Server side programming

1) Querying the database
2) Operations over databases
3) Access/Write a file on server.
4) Interact with other servers.
5) Structure web applications.
6) Process user input. For example if user input is a text in search box, run a search algorithm on data stored on server and send the results.

Example Server side programming Languages

The Programming languages for server-side programming are :
1) PHP
2) Java Servlets and Java Server Pages(JSP)

3)Enterprise Java Beans(EJB)
4) Python
5) Ruby on Rails

Server Side programming Overview

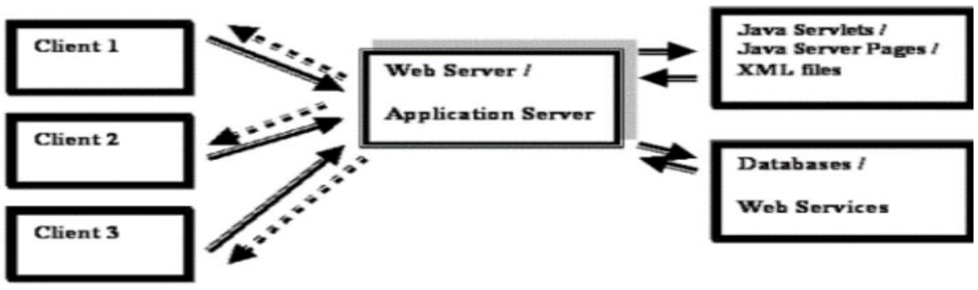


Figure 1.1 Server Side programming Overview

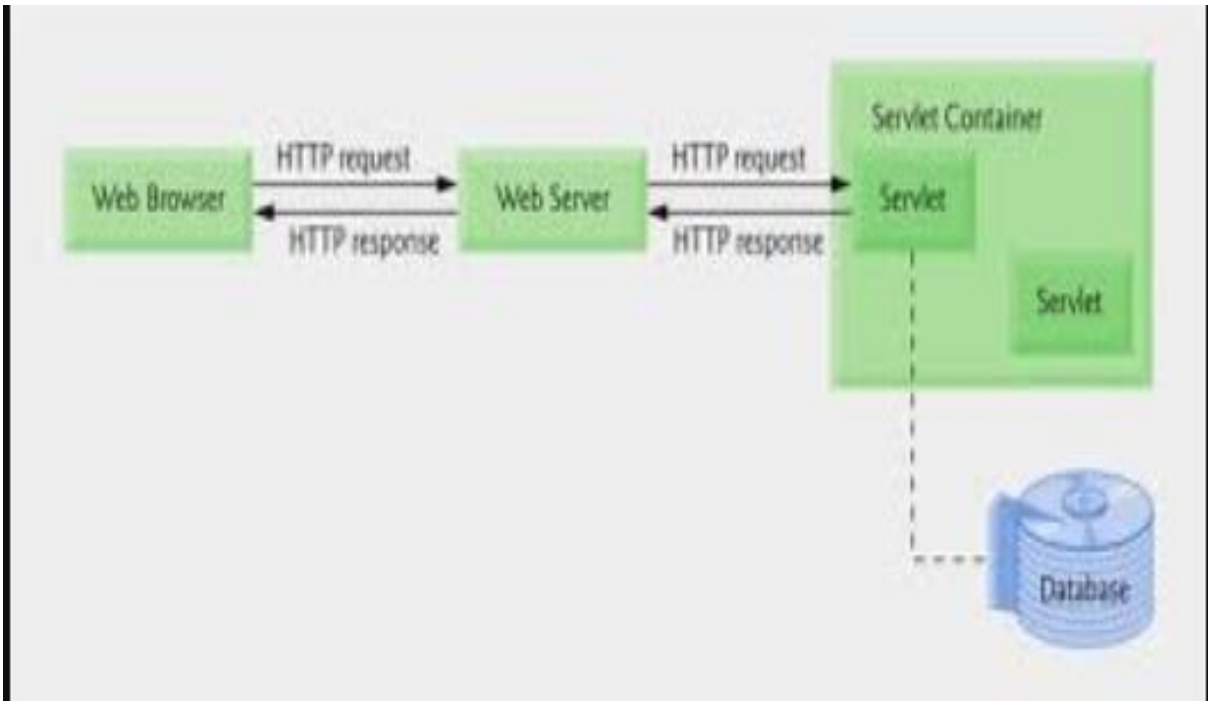
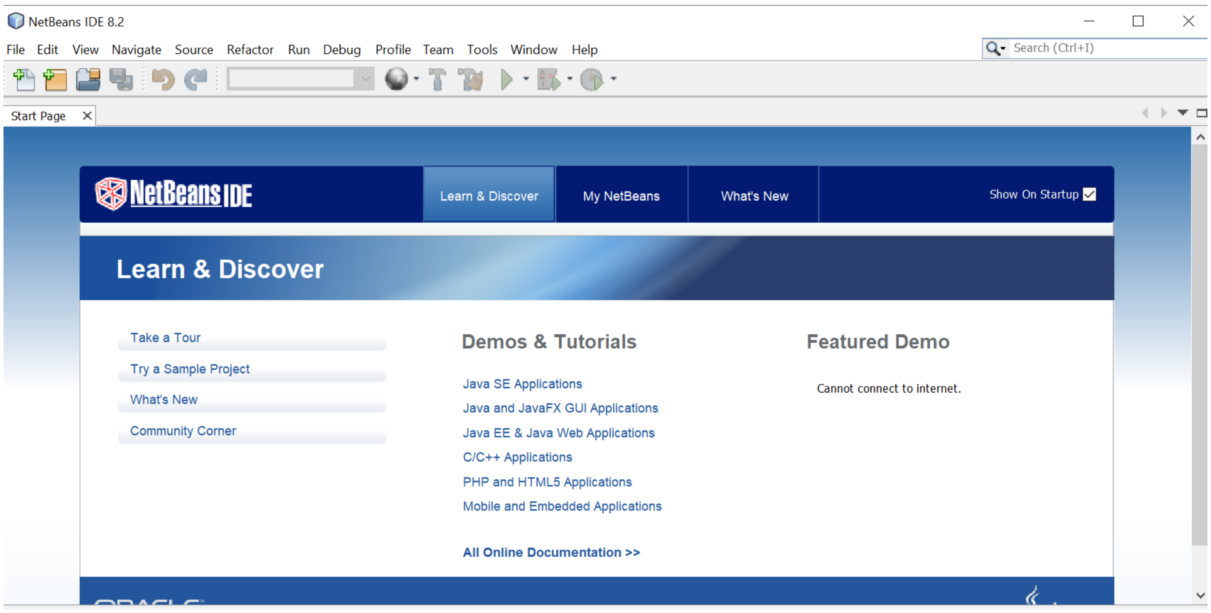


Figure 1.2 Server Side programming Overview

Implementation in Java

1. **Java 2 Standard Edition (J2SE)[12]:** This is the most commonly used form of Java technology. Also referred to as *Java Development Kit* (JDK), J2SE is a software development kit that includes a set of APIs, tools, and a runtime for developing general-purpose applications.
2. **Java 2 Enterprise Edition (J2EE)[13]:** J2EE is the basis for Java-based enterprise applications. J2EE specifies a standard for developing multi-tier distributed component-based applications. J2EE provides an integrated technical framework for developing Internet-enabled enterprise, and e-commerce applications.
3. **Java 2 Micro Edition (J2ME)[14]:** J2ME provides a lightweight and optimized Java environment for applications and consumer and mobile devices.

J2EE Platform-Netbeans IDE



Learn & Discover

Take a Tour
Try a Sample Project
What's New
Community Corner

Demos & Tutorials

Java SE Applications
Java and JavaFX GUI Applications
Java EE & Java Web Applications
C/C++ Applications
PHP and HTML5 Applications
Mobile and Embedded Applications

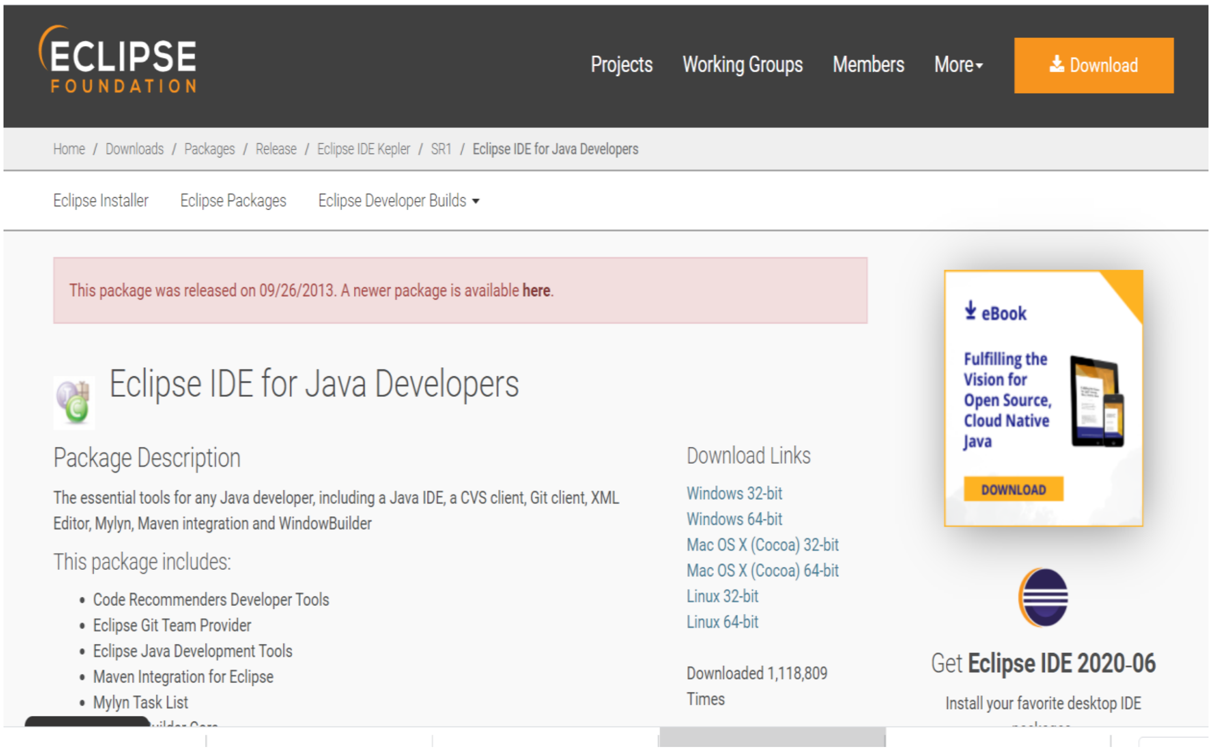
All Online Documentation >>

Featured Demo

Cannot connect to internet.

Figure 1.3 Platform-Netbeans IDE

J2EE Platform-Eclipse IDE



This package was released on 09/26/2013. A newer package is available **here**.

Eclipse IDE for Java Developers

Package Description

The essential tools for any Java developer, including a Java IDE, a CVS client, Git client, XML Editor, Mylyn, Maven integration and WindowBuilder

This package includes:

- Code Recommenders Developer Tools
- Eclipse Git Team Provider
- Eclipse Java Development Tools
- Maven Integration for Eclipse
- Mylyn Task List

Download Links

Windows 32-bit
Windows 64-bit
Mac OS X (Cocoa) 32-bit
Mac OS X (Cocoa) 64-bit
Linux 32-bit
Linux 64-bit

Downloaded 1,118,809 Times

eBook

Fulfilling the Vision for Open Source, Cloud Native Java

DOWNLOAD

Get **Eclipse IDE 2020-06**

Install your favorite desktop IDE

Figure 1.4 J2EE Platform-Eclipse IDE

What is IDE?

- An integrated development environment (IDE) is a software application that provides comprehensive facilities to computer programmers for software development.
- An IDE normally consists of at least a source code editor, build automation tools and a debugger.
- Some IDEs, such as NetBeans and Eclipse, contain the necessary compiler, interpreter, or both

J2EE Platform

- A distributed application sever environment
- Provides
    - ➢ APIs– Programming model
    - ➢ Runtime Infrastructure-For hosting and managing applications

Enterprise Architecture Styles:

- Two-Tier Architecture
- Three-Tier Architecture
- Multitier Architecture
- J2EE Architecture

Two Tier Architecture

- The two-tier architecture is also known as the client/server architecture.
- It consists mainly of two tiers:

  data & client (GUI).
- The application logic can be located in:

➢ client tier which results in a fat client or

➢ in the data tier which results in a fat server

Two Tier Architecture

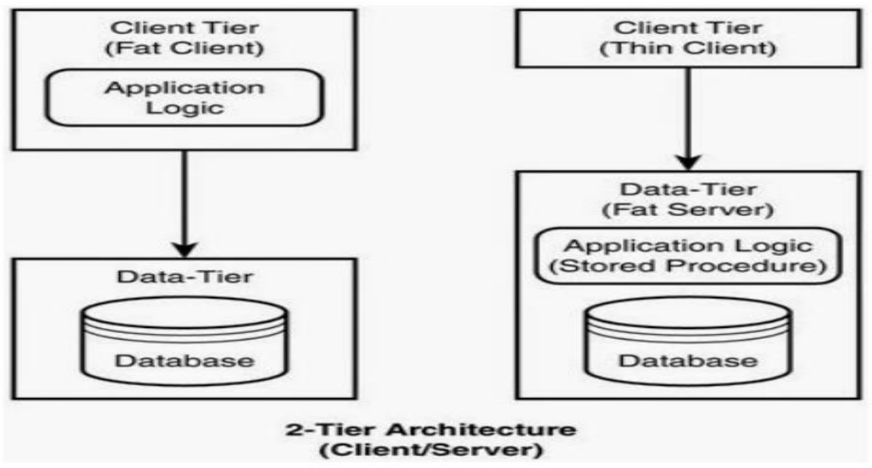


Figure 1.5 Two Tier Architecture

Two Tier Architecture Drawbacks

- This type of architecture suffers from a lack of scalability.
- Because both the client and the server have limited resources.
- Client sends request to server to clear the traffic
- Most time is wasted in request response processing
- Another issue is maintainability.
- Need to update every single client installation

Three Tier Architecture

3-tier schema is an extension of the 2-tier architecture. 3-tier architecture has following layers

1. Presentation layer (your PC, Tablet, Mobile, etc.)
2. Application layer (server)
3. Database Server

The first tier is referred to as the presentation layer, and consists of the application GUI.The middle tier, or the business layer, consists of the business logic to retrieve data for the user requests.The back-end tier, or data layer, consists of the data needed by the application.

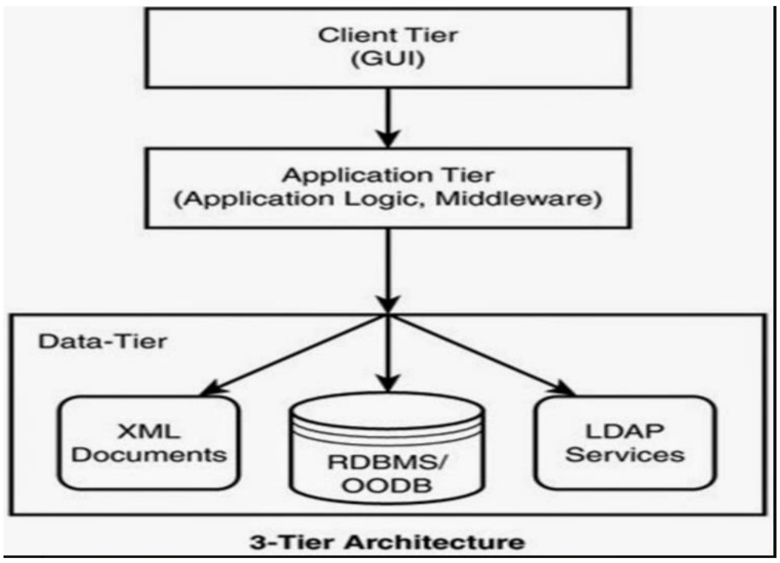


Figure 1.6 Three Tier Architecture

This DBMS architecture contains an Application layer between the user and the DBMS, which is responsible for communicating the user's request to the DBMS system and send the response from the DBMS to the user.The application layer(business logic layer) also processes functional logic, constraint, and rules before passing data to the user or down to the DBMS.Three tier architecture is the most popular DBMS architecture.

The goal of Three-tier architecture is:

- To separate the user applications and physical database
- Proposed to support DBMS characteristics
- Program-data independence
- Support of multiple views of the data

Multitier (n-tier) Architecture:

- Application logic is logically divided rather than physically
- Parts:
  - ➢ User interface: GUI
  - ➢ Presentation Logic: User requests are handled
  - ➢ Business Logic: Business rules are applied
  - ➢ Infrastructure services: eg. Messaging, transaction support
  - ➢ Data Layer: Database
- Applies 3 functional components of MVC
- Model View Controller(MVC):
  - ➢ Model: Data

- View: Presentation Logic
- Controller: Business Logic/Application Logic

Multitier (n-tier) Architecture

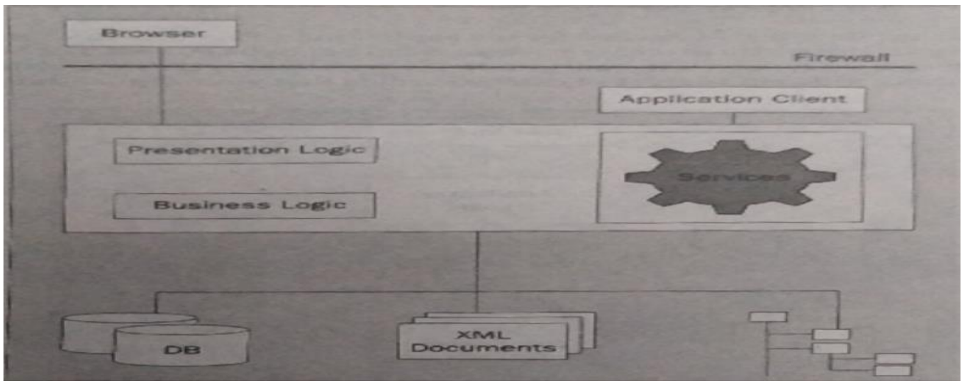


Figure 1.7 Multi-Tier Architecture

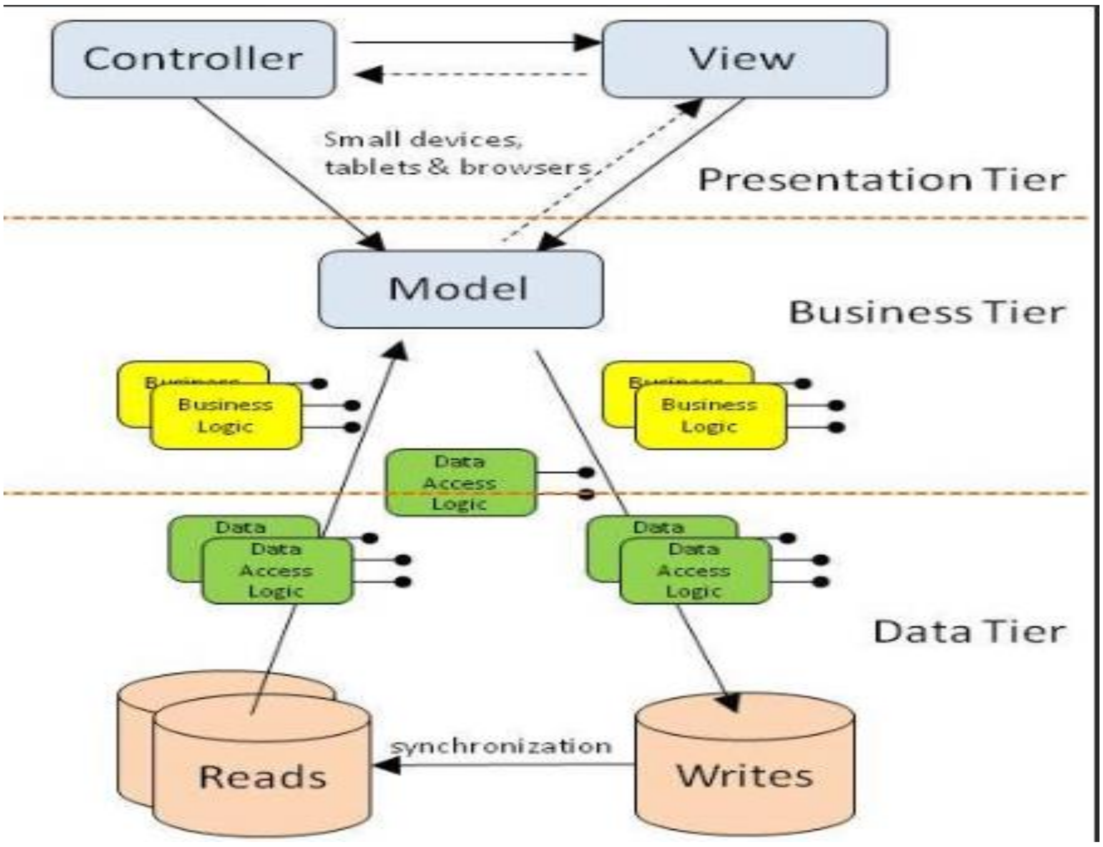


Figure 1.8 Multi-Tier(n-tier) Architecture

N-tier architecture is also called multi-tier architecture because the software is engineered to have the processing, data management, and presentation functions physically and logically separated. That means that these different functions are hosted on several machines or clusters, ensuring that services are provided without resources being shared and, as such, these services are delivered at top capacity. The “N” in the name n-tier architecture refers to any number from 1.

N-tier architecture would involve dividing an application into three different tiers. These would be the

1. logic tier,
2. the presentation tier, and
3. The data tier.

Enterprise Architecture:

- Generic interfaces are used
- Only objects are modified not the interface

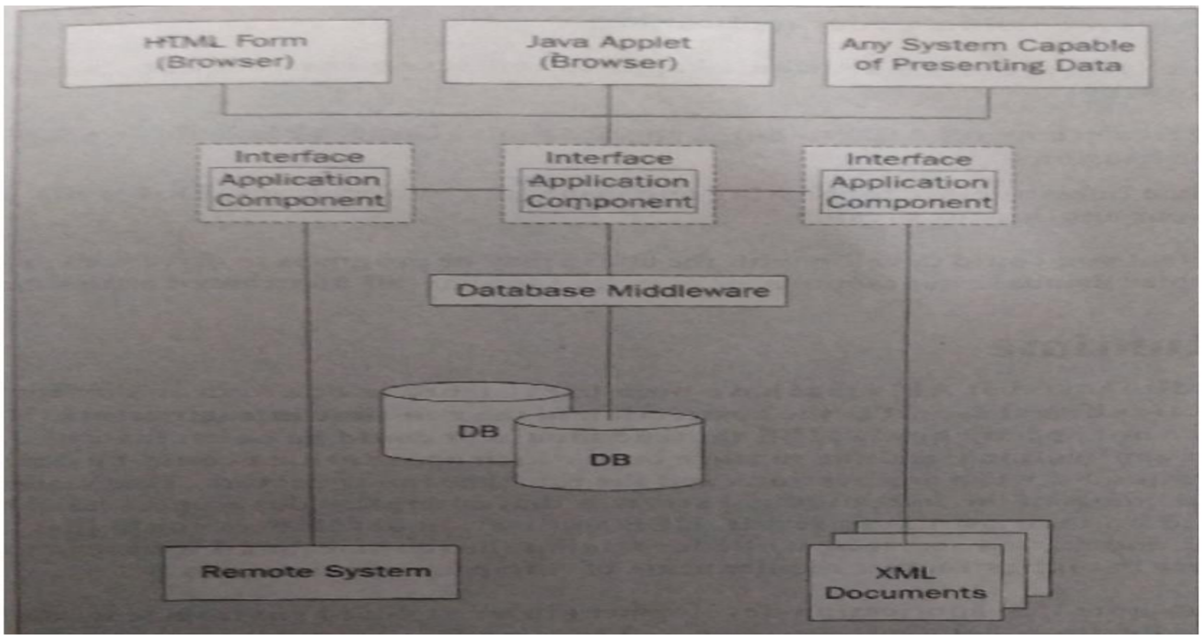

HTML Form (Browser)
Java Applet (Browser)
Any System Capable of Presenting Data
Interface Application Component
Interface Application Component
Interface Application Component
Database Middleware
DB
DB
Remote System
XML Documents

Figure 1.9 Enterprise Architecture

This architecture shows two existing relational databases being used by a J2EE application server. One database is being accessed using JDO and the other by JDBC. A third database is being used in the middle tier as an EJB accelerator. The database used as an EJB accelerator holds data that is a copy of data from one or more of the existing relational databases. All updates from the EJB components are made directly to this database.

J2EE Architecture

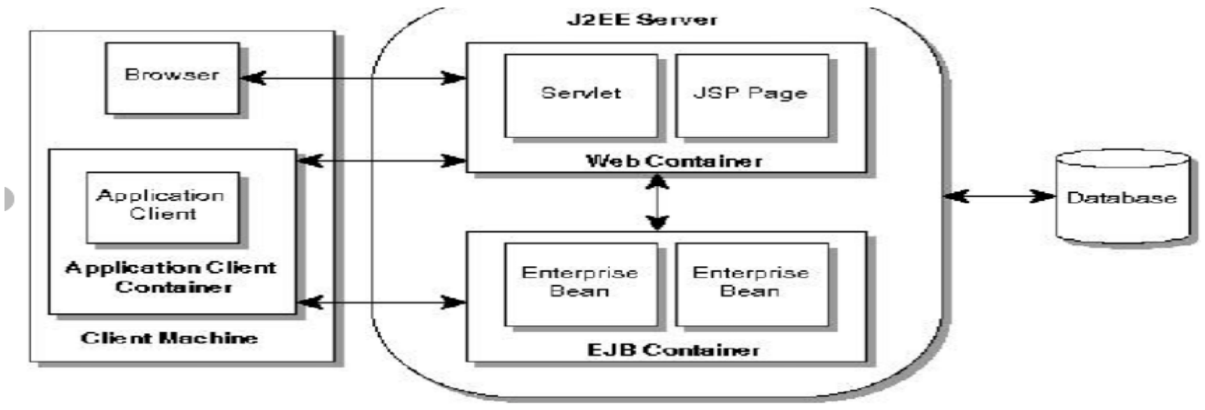


Figure 1.10 J2EE Architecture

- A distributed application sever environment
- Provides
  - ➢ APIs– Programming model
  - ➢ Runtime Infrastructure-For hosting and managing applications

Middleware

- Middleware is software which lies between an operating system and the applications running on it.
- Essentially functioning as hidden translation layer, middleware enables communication and data management for distributed applications.
- It is sometimes called plumbing, as it connects two applications together so data and databases can be easily passed between the “pipe.”
- Using middleware allows users to perform such requests as submitting forms on a web browser or allowing the web server to return dynamic web pages based on a user’s profile.

Static and dynamic web page:

- Web pages can be either static or dynamic. "Static" means unchanged or constant, while "dynamic" means changing or lively. Therefore, static Web pages contain the same prebuilt content each time the page is loaded, while the content of dynamic Web pages can be generated on-the-fly.
- Standard HTML pages are static Web pages. They contain HTML code, which defines the structure and content of the Web page. Each time an HTML page is loaded, it looks the same. The only way the content of an HTML page will change is if the Web developer updates and publishes the file.
- Other types of Web pages, such as PHP, ASP, and JSP pages are dynamic Web pages. These pages contain "server-side" code, which allows the server to generate unique content each time the page is loaded.

Cookie

- An HTTP cookie (also called web cookie, Internet cookie, browser cookie, or simply cookie) is a small piece of data stored on the user's computer by the web browser while browsing a website.
- Cookies were designed to be a reliable mechanism for websites to remember stateful information (such as items added in the shopping cart in an online store) or to record the user's browsing activity

J2EE Runtime

- Applications use interfaces to interact with the runtime. There is a clear demarcation between applications and runtime infrastructure
- Long Term and Short term demands:
  - Short term demands ->Short lived internet designs
  - Long term demands->Maintainable and reusable
- J2EE is more flexible to build applications including long term and short term demands
- Layers in J2EE architecture are loosely coupled
- Highly extendable implementation
- J2EE architecture provides uniform means of accessing platform level services via its runtime environment
- Before J2EE distributed computing was client-server based.
  - Server-> implements the interface
  - Client->Connect to server
  - Eg. CORBA is a distributed application (IDL generates stubs on the client side and skeletons on the server side)

J2EE Runtime

Enterprise services

- Transaction processing
- Database access
- Messaging
- Multithreading

Distributed transactions needs these services

- To access the above services we need plumbing code or middleware solutions using APIs

- Server side resources are scarce. We need to concentrate on server side resources: threads, security, transactions, and database connections
- J2EE is the solution that meets all the above requirements
- It is a platform having built-in solution to meet all the needs
- J2EE does not specify the nature and structure of runtime instead it introduces the container

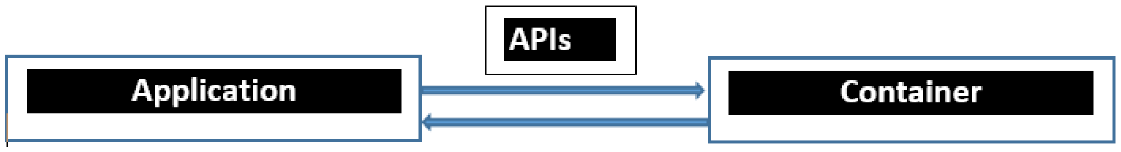


Figure 1.11 Distributed transactions

J2EE Container (Runtime)

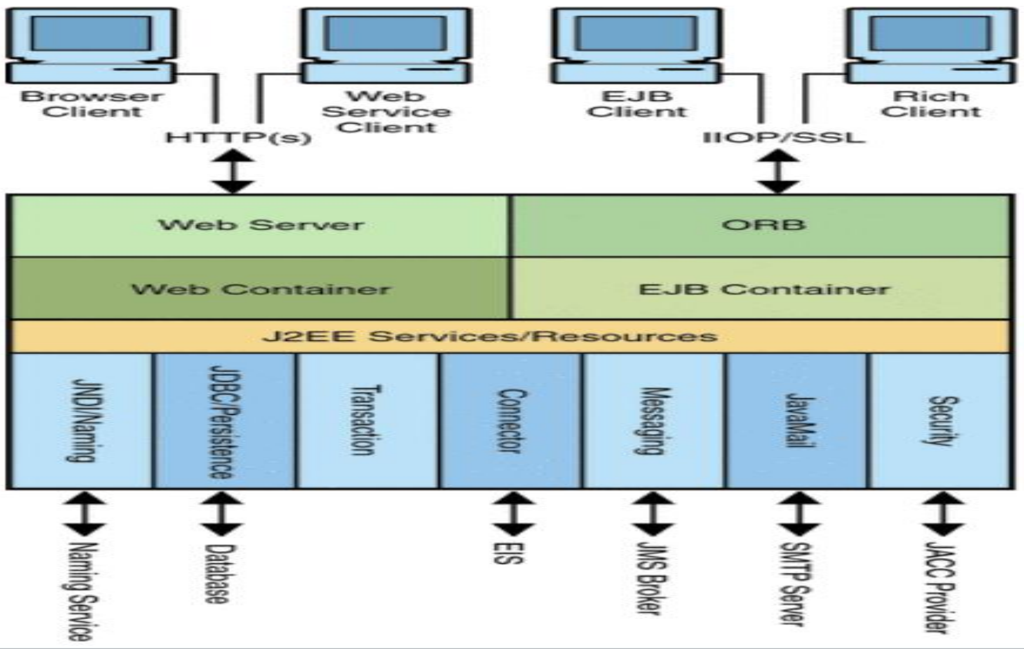


Figure 1.12 J2EE Containers (Runtime)

J2EE APIs

Application Programming Interface: API

- API is a software intermediary that allows two applications to talk to each other. Each time you use an app like Facebook, send an instant message, or check the weather on your phone, you’re using an API

APIs of J2EE platform:

1. Java Database Connectivity(JDBC)
2. Enterprise Java Beans: (EJB)
3. Java Servlets
4. JSP
5. Java Message Services(JMS)
6. Java Transaction API (JTA)
7. Java Mail
8. JavaBeans Activation Framework
9. Java API for XML parsing(JAXP)
10. Java Connector Architecture(JCA)

    Java Authentication and Authorization service(JAAS)

J2SE APIs

J2SE (Standard edition) APIs

1. Java IDL API
2. JDBC Core API
3. RMI-IIOP API
4. JNDI API

J2EE APIs

1. **Java Database Connectivity(JDBC) :** is the Java API that manages connecting to a database, issuing queries and commands, and handling result sets obtained from the database.
2. **Enterprise Java Beans: (EJB) EJB is** a server-side software component that encapsulates business logic of an application. An **EJB** web container provides a runtime environment for web related software components, including computer security, Java servlet lifecycle management, transaction processing, and other web services.
3. **Java Servlets**: **Servlet** technology is used to create a web application (resides at server side and generates a dynamic web page).
4. JSP : JavaServer Pages (***JSP***) is a Java standard technology that enables you to write dynamic, data-driven pages for your Java web applications.
5. **Java Message Services(JMS):** JMS API allows applications to create, send, receive, and read ***messages*** using reliable, asynchronous, loosely coupled communication.

6. **Java Transaction API (JTA):** The Java™ Transaction API (JTA) allows applications to perform distributed transactions, that is, transactions that access and update data on two or more networked computer resources.
7. **Java Mail** : The JavaMail API provides a platform-independent and protocol-independent framework to build mail and messaging applications.
8. **JavaBeans Activation Framework:** The **JavaBeans Activation Framework** (JAF) is used by the JavaMail API. JAF provides standard services to determine the type of an arbitrary piece of data, encapsulate access to it, discover the operations available on it, and create the appropriate **JavaBeans** component to perform those operations.
9. **Java API for XML parsing(JAXP):** Java ***API*** for XML Processing. Java ***API*** for XML Processing (***JAXP***) enables applications to parse, transform, validate and query XML documents using an ***API*** that is independent of a particular XML processor implementation.
10. **Java Connector Architecture(JCA):** The Java EE Connector Architecture (JCA) defines a standard architecture for Java EE systems to external heterogeneous Enterprise Information Systems (EIS). Examples of EISs include Enterprise Resource Planning (ERP) systems, mainframe transaction processing (TP), databases and messaging systems.
11. **Java Authentication and Authorization service(JAAS)**: JAAS provides subject-based authorization on authenticated identities.

J2SE APIs

- **Java IDL API:** Java IDL technology adds CORBA (Common Object Request Broker Architecture) capability to the Java platform, providing standards-based interoperability and connectivity. Java IDL enables distributed Web-enabled Java applications to transparently invoke operations on remote network services using the industry standard IDL (Object Management Group Interface Definition Language) and IIOP (Internet Inter-ORB Protocol) defined by the Object Management Group. Runtime components include a Java ORB for distributed computing using IIOP communication.
- **JDBC Core API:** Provides the API for accessing and processing data stored in a data source (usually a relational database) using the Java programming language.
- **RMI-IIOP API:** Java Remote Method Invocation over Internet Inter-ORB Protocol technology ("RMI-IIOP") is part of the Java Platform, Standard Edition (Java SE). The RMI Programming Model enables the programming of CORBA servers and applications via the rmi API
- **JNDI API:** The Java Naming and Directory Interface (JNDI) provides naming and directory functionality to applications written in the Java programming language. It is designed to be independent of any specific naming or directory service implementation.

J2EE Container Architecture

Definition: A J2EE container is a runtime to manage application components developed according to the API specifications and provide access to J2EE APIs

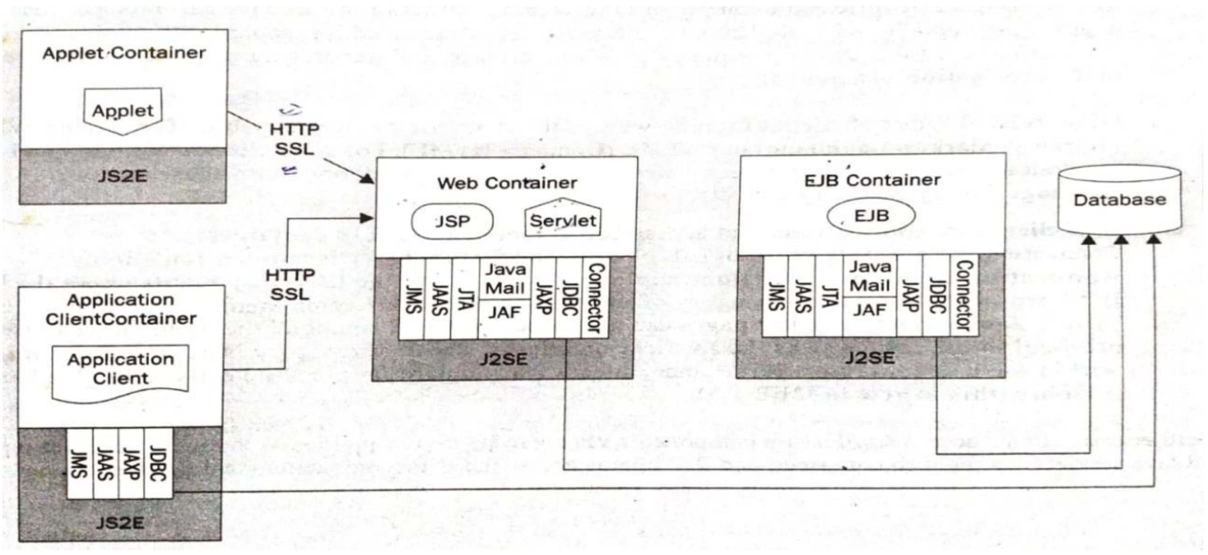


Figure 1.13 J2EE Container Architecture

J2EE Container ArchitectureComponent:

Definition:

- J2EE components are also called **Managed Objects** since they are created and managed within container runtime

Four Containers:

- **Web container:** To host Servlets, JSPs
- **EJB Container:** To host EJB components
- **Applet Container**-To host Java applets
- **Application client-** To host standard Java applications

2 Primary types of Clients:

– Web clients

– EJB clients

Web clients:

- Run in web browsers
- For these clients UI is generated on Server side as HTML/XML
- They use HTTP to communicate with server

- Application components in web containers include Java servlets, JSPs whtich implement functionality required by web clients
- Web containers are responsible for accepting requests from web clients and generating responses with the help of application components

EJB clients:

- Application that access EJB components in EJB containers are called EJB clients

3 types of EJB clients

- Application clients that are standalone applications accessing EJB components using RMI-IIOP protocol
- Components in web container (Java servlets, JSP) also access EJB components using RMI-IIOP protocol
- Other EJBs running within the EJB container. They communicate via standard Java method calls via local interface

Container Architecture

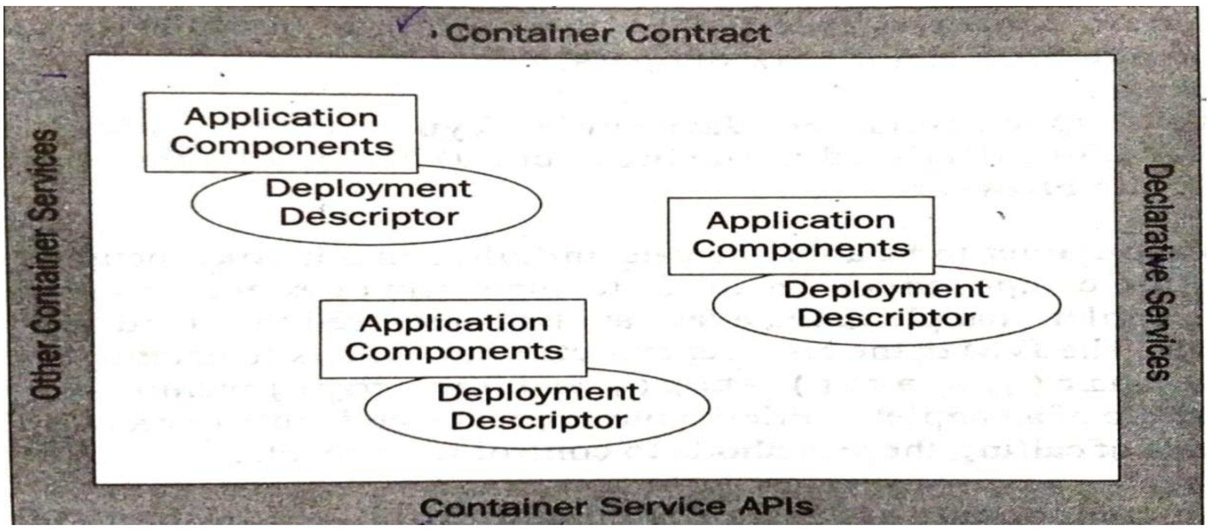


Figure 1.14 Container Architecture

Application Components:

- Servlets
- JSP
- EJB
-

Deployment Descriptors

- It is an XML file that describes application components

Architecture Container: can be divided into 4 parts

- Component contract
- Container service APIs
- Declarative services
- Other container services

Application Components:

- Servlets
- JSP
- EJB

Deployment Descriptors

- It is an XML file that describes application components

Architecture Container: can be divided into 4 parts

- Component contract
- Container service APIs
- Declarative services
- Other container services

Component contract:

- A set of APIs specified by the container which are commonly required to extend or implement
- Basic purpose of container is to provide a runt time for application clients
- Instance of application components are created and invoked within JVM of the container
- This makes container responsible for managing life cycle of application components and they are required to abide by certain contracts specified by the container.
- Eg. Java applet downloaded by browser and instantiated and initialized in browser's JVM
- However the components are required to implement certain interfaces or classes,
- Eg. Java.applet.Applet(Class)->init(),start(), stop(), destroy() methods

J2EE application components are remote to clients. Client can't directly call methods on components

Client makes request to application server and it is container that actually invokes methods. Application components are required to follow contract specified by the container

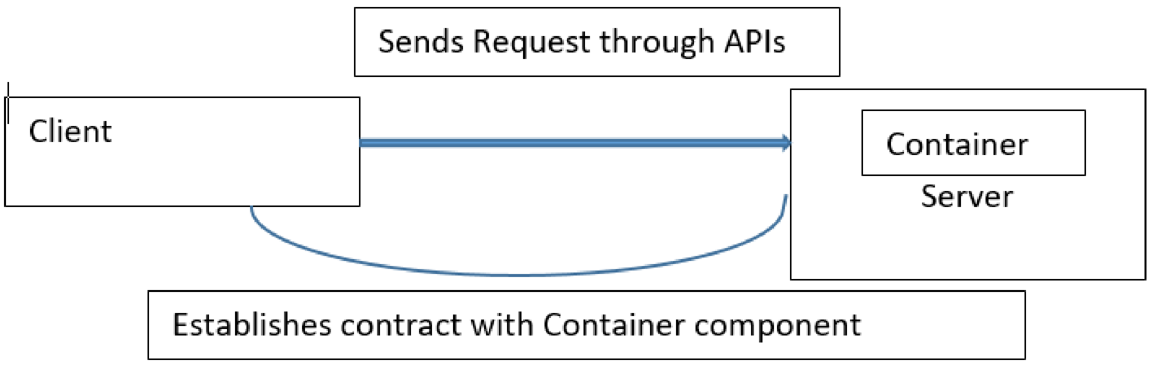

Sends Request through APIs
Client
Container
Server
Establishes contract with Container component

Figure 1.15 Container Architecture

All J2EE application components are managed which includes:

- Location
- Instantiation
- Pooling
- Initialization
- Service invocation
- Removal of components from service

- Eg. Web containers have interfaces:
- Javax.ejb. EJBHome and javax.ejb.EJBObject implementing either javax.ejb.SessionBean or javx.ejb.EntityBean interface
- Message Driven Beans implement both: javx.ejb.MessageDrivenBean and javax.jmx.MessageListener interfaces

Container Service APIs(Additional services required for all applications)

- A container in J2EE architecture provides a consolidated view of various enterprise APIs specified in J2EE platform
- In J2EE, application components can access APIs via appropriate objects created and published in JNDI service or implementation
- Eg. Audio component access by remote client in a distributed application

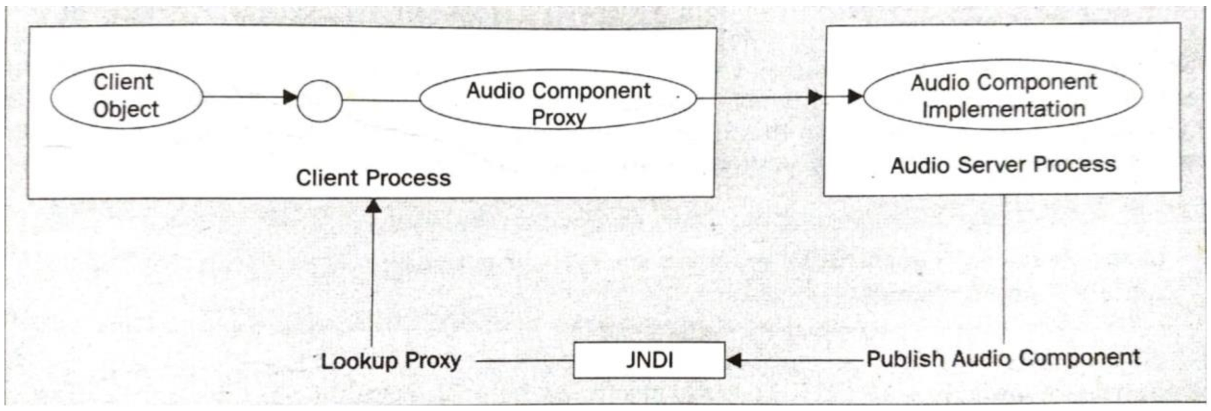


Figure 1.16 Container Service

- Loosely coupling between implementation and client
- Uses delegation (code reusability) instead of inheritance

Declarative services

- Based on deployment description provided for each application component such as security, transaction etc
- A deployment descriptor defines contract between container and component

Two methods for invoking services

- Explicit invocation
- Declarative invocation

Explicit invocation: It is a standard method. Eg. In DBMS we explicitly invoke commit/rollback queries

Declarative invocation: These are not explicitly invoked. Instead, we specify business transaction (start, stop, etc) in deployment descriptor and the container will automatically start a transaction

- Declarative service is a service performed by the container on our behalf
- Web container receives HTTP request and delegates them to servlets and JSP

Advantages:

- Automatic start and end of business transactions
- We can place new services without changing application component
- So decisions can be postponed to run-time instead of design time

Other container Services

- Lifecycle management of application components
- Resource Pooling

- Populating JNDI namespace based on deployment names associated with EJB components
- Populating JNDI namespace with objects necessary for utilizing container service APIs
- Clustering (distributing load of requests across JVMs)
- Enhancing availability and scalability of applications

Java RMI

- RMI stands for **Remote Method Invocation**. It is a mechanism that allows an object residing in one system (JVM) to access/invoke an object running on another JVM.
- RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package **java.rmi**.

Architecture of an RMI Application

- In an RMI application, we write two programs, a **server program** (resides on the server) and a **client program** (resides on the client).
- Inside the server program, a remote object is created and reference of that object is made available for the client (using the registry).
- The client program requests the remote objects on the server and tries to invoke its methods.

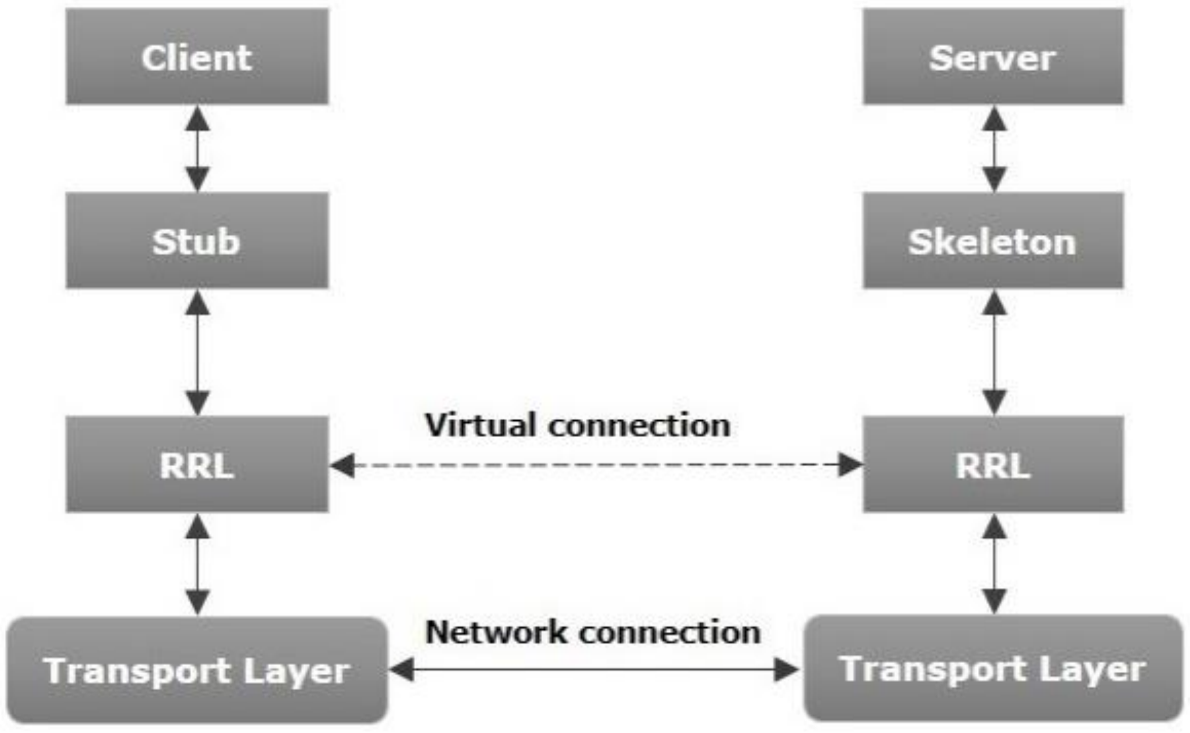


Figure 1.17 Architecture of an RMI

Components of an RMI Architecture

- **Transport Layer** − This layer connects the client and the server. It manages the existing connection and also sets up new connections.
- **Stub** − A stub is a representation (proxy) of the remote object at client. It resides in the client system; it acts as a gateway for the client program.
- **Skeleton** − This is the object which resides on the server side. **stub** communicates with this skeleton to pass request to the remote object.
- **RRL(Remote Reference Layer)** − It is the layer which manages the references made by the client to the remote object.

Working of an RMI Application:

- When the client makes a call to the remote object, it is received by the stub which eventually passes this request to the RRL.
- When the client-side RRL receives the request, it invokes a method called **invoke()** of the object **remoteRef**. It passes the request to the RRL on the server side.
- The RRL on the server side passes the request to the Skeleton (proxy on the server) which finally invokes the required object on the server.
- The result is passed all the way back to the client.

Serialization and Deserialization in Java:

- **Serialization** is a mechanism of converting the state of an object into a byte stream.
- **Deserialization** is the reverse process where the byte stream is used to recreate the actual Java object in memory.
- This mechanism is used to persist the object.

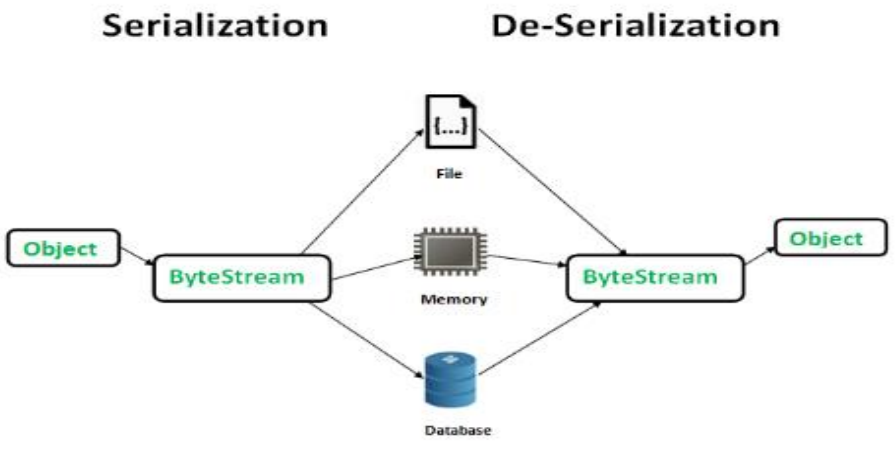


Figure 1.18 Serialization and Deserialization

Marshalling and Unmarshalling

- Whenever a client invokes a method that accepts parameters on a remote object, the parameters are bundled into a message before being sent over the network. These parameters may be of primitive type or objects. In case of primitive type, the parameters are put together and a header is attached to it. In case the parameters are objects, then they are serialized. This process is known as marshaling.
- At the server side, the packed parameters are unbundled and then the required method is invoked. This process is known as **unmarshalling**.

RMI Registry

- RMI registry is a namespace on which all server objects are placed. Each time the server creates an object, it registers this object with the RMIregistry (using **bind()** or **reBind()** methods). These are registered using a unique name known as **bind name**.
- To invoke a remote object, the client needs a reference of that object. At that time, the client fetches the object from the registry using its bind name (using **lookup()** method).

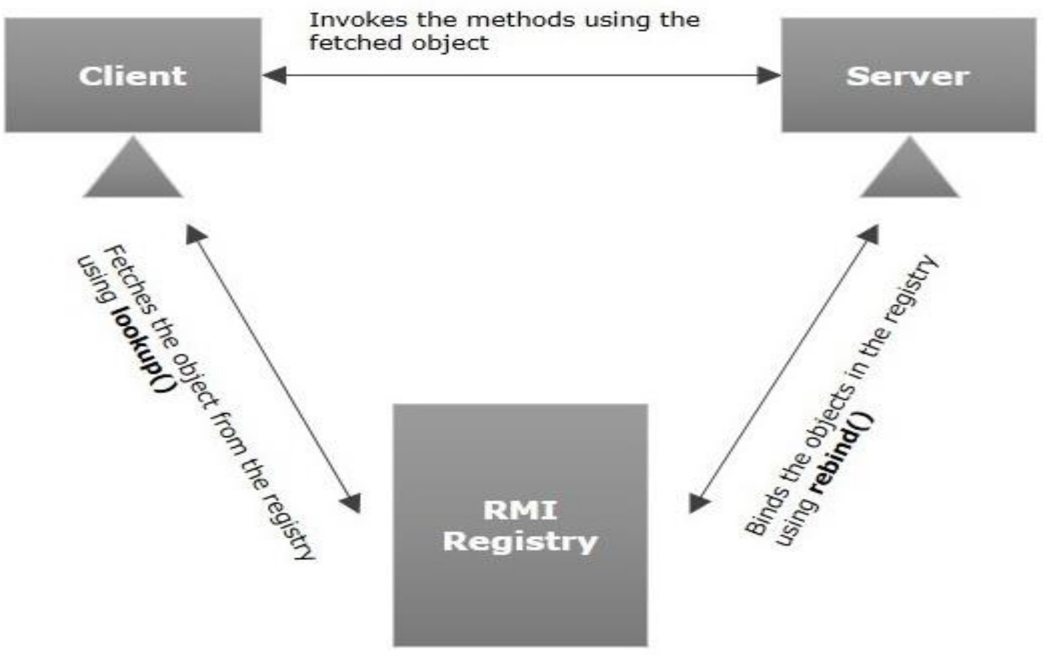

Invokes the methods using the fetched object
Client
Server
Fetches the object from the registry using lookup()
Binds the objects in the registry using rebind()
RMI Registry

Figure 1.19 RMI Registry

Goals of RMI

- To minimize the complexity of the application.
- To preserve type safety.
- Distributed garbage collection.
- Minimize the difference between working with local and remote objects.

Java implementation of RMI

- Define the remote interface
- Develop the implementation class (remote object)
- Develop the server program
- Develop the client program
- Compile the application
- Execute the application

Defining the Remote Interface:

- A remote interface provides the description of all the methods of a particular remote object. The client communicates with this remote interface.
- To create a remote interface –
- Create an interface that extends the predefined interface **Remote** which belongs to the package.
- Declare all the business methods that can be invoked by the client in this interface.
- Since there is a chance of network issues during remote calls, an exception named **RemoteException** may occur; throw it.

Defining the Remote Interface:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

// Creating Remote interface for our application
public interface Hello extends Remote {
   void printMsg() throws RemoteException;
}
```

Developing the Implementation Class (Remote Object)

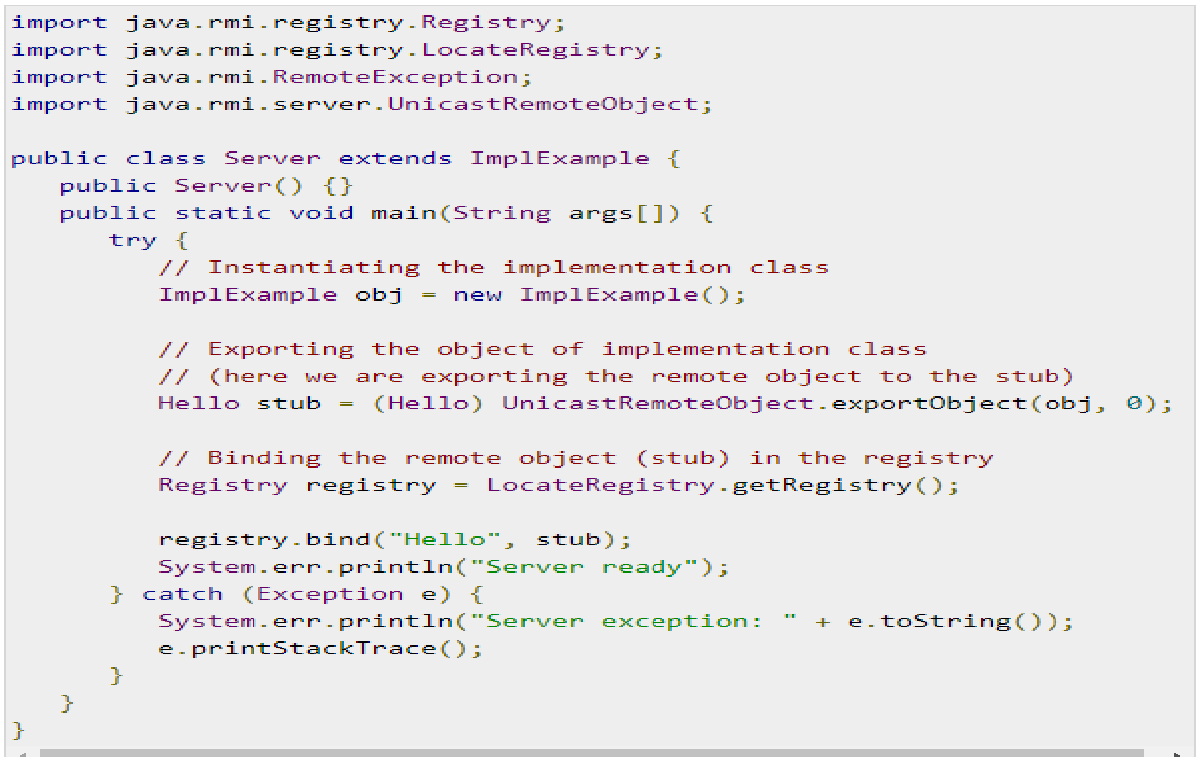
- To develop an implementation class –
- Implement the interface created in the previous step.
- Provide implementation to all the abstract methods of the remote interface.

```
// Implementing the remote interface
public class ImplExample implements Hello {

   // Implementing the interface method
   public void printMsg() {
      System.out.println("This is an example RMI program");
   }
}
```

Developing the Server Program:

- An RMI server program should implement the remote interface or extend the implementation class. Here, we should create a remote object and bind it to the **RMIregistry**.
- To develop a server program –
- Create a client class from where you want invoke the remote object.
- **Create a remote object** by instantiating the implementation class as shown below.
- Export the remote object using the method **exportObject()** of the class named **UnicastRemoteObject** which belongs to the package **java.rmi.server**.
- Get the RMI registry using the **getRegistry()** method of the **LocateRegistry** class which belongs to the package **java.rmi.registry**.
- Bind the remote object created to the registry using the **bind()** method of the class named **Registry**. To this method, pass a string representing the bind name and the object exported, as parameters.

RMI Server



```java
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class Server extends ImplExample {
   public Server() {}
   public static void main(String args[]) {
      try {
         // Instantiating the implementation class
         ImplExample obj = new ImplExample();

         // Exporting the object of implementation class
         // (here we are exporting the remote object to the stub)
         Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);

         // Binding the remote object (stub) in the registry
         Registry registry = LocateRegistry.getRegistry();

         registry.bind("Hello", stub);
         System.err.println("Server ready");
      } catch (Exception e) {
         System.err.println("Server exception: " + e.toString());
         e.printStackTrace();
      }
   }
}
```

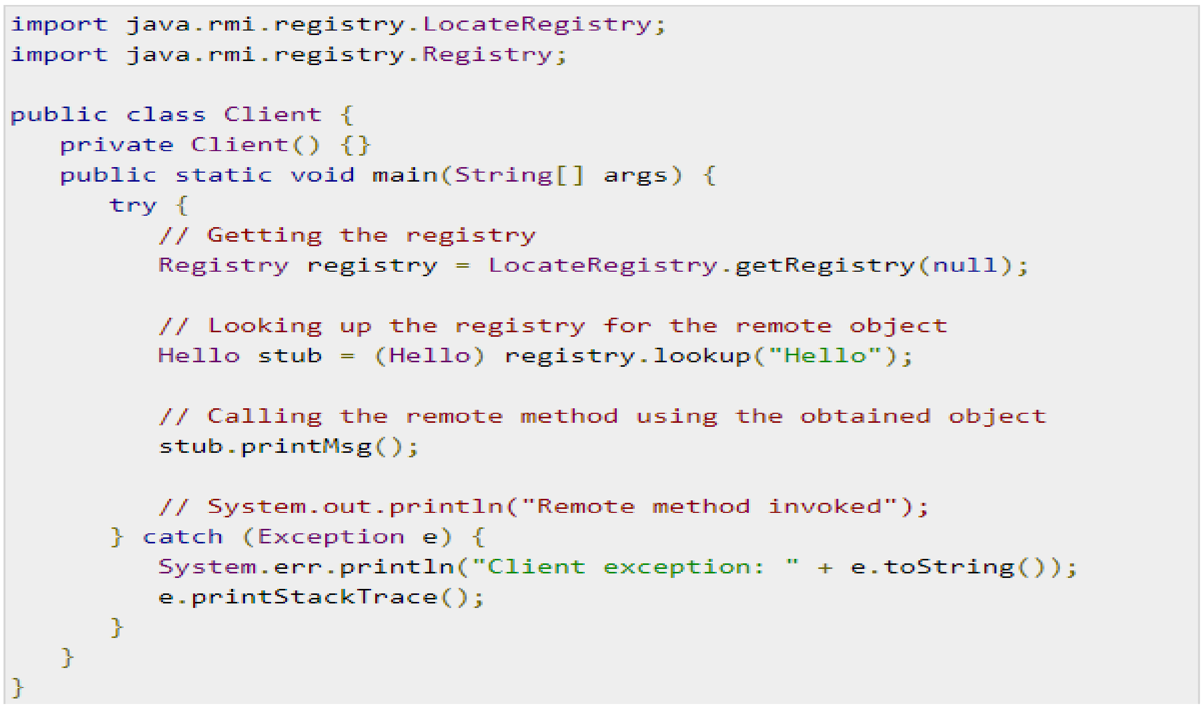
Developing the Client Program

To develop a client program –

- Create a client class from where your intended to invoke the remote object.

- Get the RMI registry using the **getRegistry()** method of the **LocateRegistry** class which belongs to the package **java.rmi.registry**.
- Fetch the object from the registry using the method **lookup()** of the class **Registry** which belongs to the package **java.rmi.registry**.
- To this method, you need to pass a string value representing the bind name as a parameter. This will return you the remote object.
- The lookup() returns an object of type remote, down cast it to the type Hello.
- Finally invoke the required method using the obtained remote object.

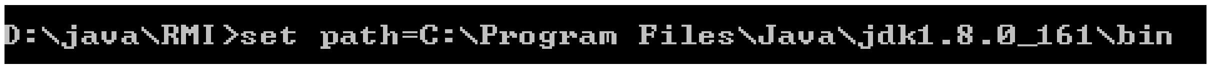
RMI Client code:



```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {
   private Client() {}
   public static void main(String[] args) {
      try {
         // Getting the registry
         Registry registry = LocateRegistry.getRegistry(null);

         // Looking up the registry for the remote object
         Hello stub = (Hello) registry.lookup("Hello");

         // Calling the remote method using the obtained object
         stub.printMsg();

         // System.out.println("Remote method invoked");
      } catch (Exception e) {
         System.err.println("Client exception: " + e.toString());
         e.printStackTrace();
      }
   }
}
```

Execution of RMI codes in Java:

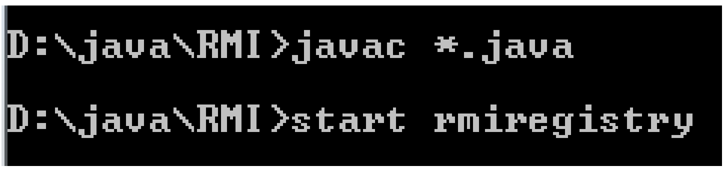
Step1: Set the current path to jdk bin directory(the path where jdk is installed in your computer)



```
D:\java\RMI>set path=C:\Program Files\Java\jdk1.8.0_161\bin
```

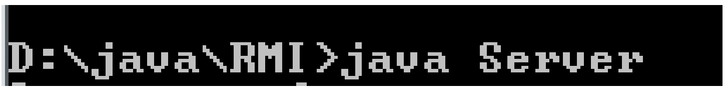
Step 2: Compile all the java files in the RMI directory (*.java)

Step 3: Start the **rmi** registry



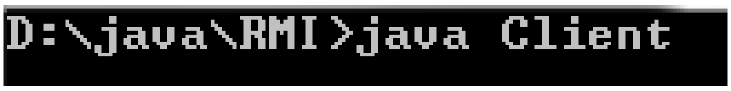
```
D:\java\RMI>javac *.java

D:\java\RMI>start rmiregistry
```

Step 4: Run the Server(Sever will wait for client’s request)



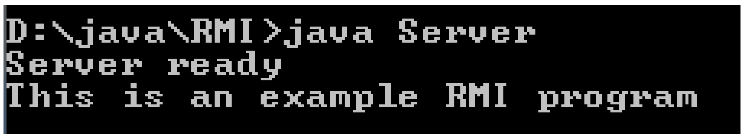
```
D:\java\RMI>java Server
```

Step 5: Run the Client



```
D:\java\RMI>java Client
```

Step 6: Final output on the server side



```
D:\java\RMI>java Server
Server ready
This is an example RMI program
```

JNDI Overview:

- The **Java Naming and Directory Interface** (JNDI) is an application programming interface (API) that provides **naming and directory functionality** to applications written using the Java programming language.
- It is defined to be **independent of any** specific **directory service** implementation.
- Thus a variety of directories--new, emerging, and already deployed--can be accessed in a common way.

What is a Naming Service?

- Naming service is a service that enables the creation of standard name for given set of data
- On internet each host has a Fully Qualified domain Name (FQDN)
- Eg. www. apress.com
  - Host name: www
  - Domain name: apress.com

What is a Directory Service?

- In computing, **directory service** or **name service** maps the names of network resources to their respective network addresses.
- It is a shared information infrastructure for locating, managing, administering and organizing everyday items and network resources, which can include volumes, folders, files, printers, users, groups, devices, telephone numbers and other objects.
- A directory service is a critical component of a network operating system.
- A **directory server** or name server is a server which provides such a service.
- Each resource on the network is considered an object by the directory server. Information about a particular resource is stored as a collection of attributes associated with that resource or object.

Directory Service

- A directory service defines a **namespace** for the network.
- The namespace is used to assign a name (unique identifier) to each of the objects.
- Directories typically have a set of rules determining how network resources are named and identified, which usually includes a requirement that the identifiers be unique and unambiguous.
- When using a directory service, a user does not have to remember the physical address of a network resource; providing a name locates the resource.
- Some directory services include access control provisions, limiting the availability of directory information to authorized users.

General Purpose Directory Services:

- Novell Directory Service(NDS)-file and print server
- Network information Service (NIS)-email in Linux
- Active Directory Service(ADS)-file encryption, remote desktop service, sharepoint service
- Windows NT Domain-MS office

Drawbacks:

- Used only for specific purpose
- Lack of security
- No standard API

JNDI Architecture:

- The JNDI architecture consists of an API and a **service provider interface (SPI).**
- Java applications use the JNDI API to access a variety of naming and directory services.
- The SPI enables a variety of naming and directory services to be plugged in transparently, thereby allowing the Java application using the JNDI API to access their services.

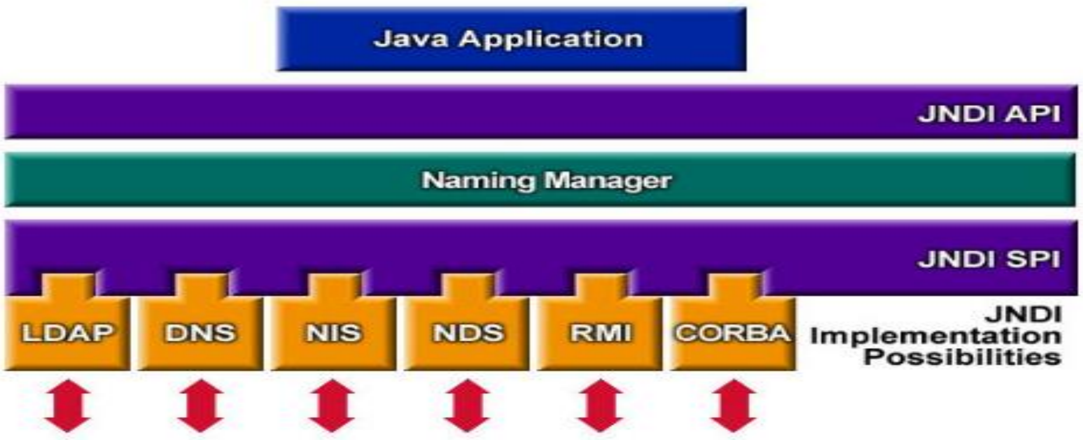


Figure 1:20. JNDI Architecture

Directory Services:

LDAP: Lightweight Directory Access Protocol

DNS: Domain Naming Service

NIS: Network Information System

NDS: Novell Directory Service

RMI : Remote Method invocation

CORBA: Common Object Request Broker Architecture

Lightweight Directory Access Protocol:

LDAP organizes data into a hierarchy, allowing it to be administered based on company, branch, department or any other method you choose. A wide variety of data can be served through LDAP, and it has become a standard with medium sized and larger organizations. As well as the OpenLDAP server that's distributed with Linux, commercial vendors such as Novell and Microsoft are using the LDAP protocols within their Active Directory and directory products. Although LDAP can be set up and operated from command line tools via text files it is found more practically that it's used with graphic tools and editors such as:

http://pegacat.com/jxplorer/
http://www.ldapbrowser.com/
http://sourceforge.net/projects/gqclient
http://www-unix.mcs.anl.gov/~gawor/ldap/
http://www.openldap.org/faq/data/cache/270.html

LDAP:

- LDAP's basic structure is based on a simple information tree metaphor called a directory information tree (DIT).
- Each leaf in the tree is an *entry;* the first or top-level entry is the root entry.

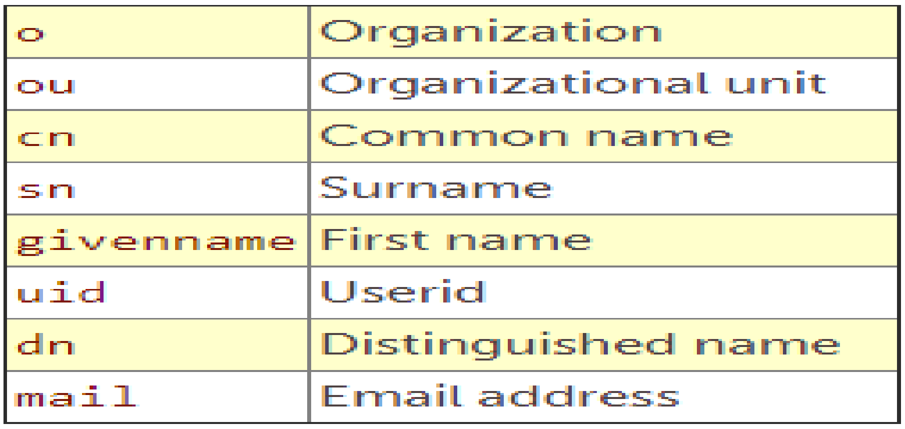
- An entry includes a distinguished name (DN) and any number of attribute/value pairs.

LDAP Data Interchange Format:

uid=styagi,ou=people,o=myserver.com

The leftmost part of the DN, called a relative distinguished name (RDN), is made up of an attribute/value pair. In the above example, this pair would be uid=styagi.

LDAP Attributes



o	Organization
ou	Organizational unit
cn	Common name
sn	Surname
givenname	First name
uid	Userid
dn	Distinguished name
mail	Email address

Figure 1:21 LDAP Attributes

DOMAIN NAME SERVICE

The Domain Name Service - DNS - (or BIND - the Berkeley INternet Domain) is used to resolve a narrow range of services on a wider (usually worldwide) basis, based on your fully qualified domain name.

What is a Fully Qualified Domain Name?

- A fully qualified domain name (FQDN) is the **complete domain name** for a specific computer, or host, on the internet.
- The FQDN consists of two parts: the hostname and the domain name.
- For example, an FQDN for a hypothetical mail server might be mymail.indiana.edu. The hostname is mymail, and the host is located within the domain indiana.edu
- In this example, .edu is **the top-level domain (TLD**).
- This is similar to the **root directory** on a typical workstation, where all other directories (or folders) originate.

Simple website login using DNS:

- When connecting to a host you must specify the FQDN.
- The DNS server then resolves the hostname to its IP address by looking at its DNS table.

- The host is contacted and you receive a login prompt.

NETWORK INFORMATION SERVICE

Originated by Sun Microsystems, the Network Information Service provides for the provision of a master and slave servers to provide information on a wide range of databases such as hosts, users, service, email aliases, etc ... and we can add our own databases too.Known in its very early days as "**yellow pages"** - that's why all the file names start with "yp" NIS provided an excellent tool for **administering** the various **databases centrally**. However, it did **not provide a distributed administration** capability, and this and other issues lead Sun to introduce a new service called NIS+ or nisplus.

NIS+ didn't really catch on though.

Novell Directory Service:

- Novell Directory Services (NDS) is a popular software product for managing access to computer resources and keeping track of the users of a network, such as a company's intranet, from a single point of administration. Using NDS, a network administrator can set up and control a database of users and manage them using a directory with an easy-to-use graphical user interface (GUI).
- Users of computers at remote locations can be added, updated, and managed centrally.
- Applications can be distributed electronically and maintained centrally.
- NDS can be installed to run under Windows NT, Sun Microsystem's Solaris, and IBM's OS/390 as well as under Novell's own NetWare so that it can be used to control a multi-platform network.

CORBA

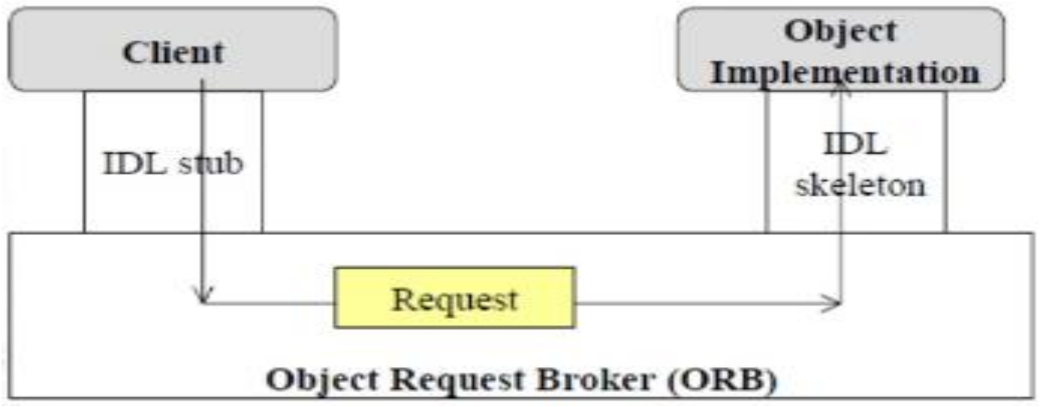


Figure 1:22 CRBA

- Needed by enterprise applications
- These applications need to support existing distributed computing standards
- CORBA uses COSnaming services(CORBA Object Service) to define the location of available objects

Relation between Directory Services, LDAP and JNDI:

- Drawback: Each directory service requires its own API which adds complexity and code bloat(swell up) to our client application

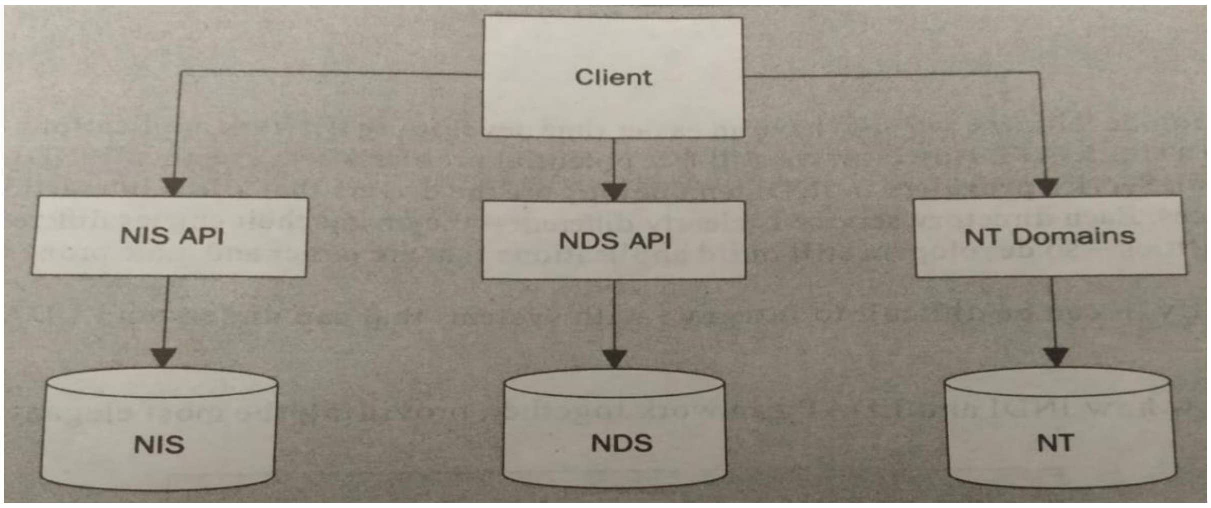


Figure 1:23 Simplified JNDI

Advantage:

Still multiple servers and multiple APIs are there. But for the application developer it appears as a single API

Drawback:

- Service providers in JNDI are drivers that allows interaction with different directory services
- Developers still build larger applications and are more prone to failure
- It is difficult to integrate with systems that can understand LDAP but unable to use Java

Simplified JNDI

Advantage:

Still multiple servers and multiple APIs are there. But for the application developer it appears as a single API

Drawback:

- Service providers in JNDI are drivers that allows interaction with different directory services
- Developers still build larger applications and are more prone to failure
- It is difficult to integrate with systems that can understand LDAP but unable to use Java

JNDI and LDAP

- It is sufficient for the client to know about LDAP protocol and JNDI API alone

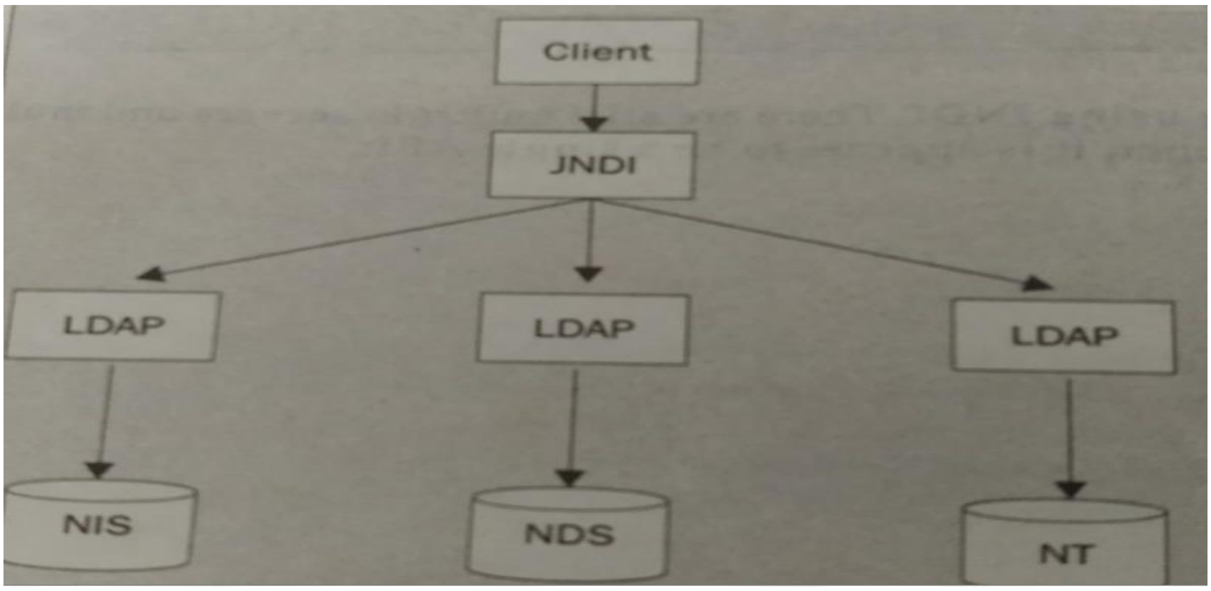


Figure 1:24 JNDI and LDAP

Why we use JNDI when we have LDAP

- LDAP without JNDI

  LDAP is a great way to converge access to directory data through a single protocol

  LDAP is a great way to provide standards based on network address book or easily keep track of devices on a network

- JNDI without LDAP

  JNDI is useful for Java applications

- LDAP and JNDI:

  LDAP is an open standard maintained by IETF(Internet Engineering Task Force)

It is accessible by a variety of clients and vendors.

Role of XML in directory services

- XML is a defacto standard for existing data
- Enables Data transfer between applications written in different languages
- XML specification for LDAP is DSML(Directory Services Markup Language) to distribute directory information via XML-RPC based protocol like SOAP (Simple Object Access Protocol)
- Current Standard is LDIP-(LDAP Data Interchange Format.

Java Database Connectivity-JDBC

- Database Vendor: Provides a set of APIs for accessing data managed by database server. Eg. Oracle, Sybase etc
- Client applications: written in C/C++ can use these vendor specific APIs
- JDBC Driver: A Middleware layer that translates the JDBC calls to vendor Specific APIs
- Packages: java.sql-----J2SE

  javax.sql---J2EE

JDBC Driver Types

- JDBC Driver is a software component that enables java application to interact with the database.
- There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)
4. Thin driver (fully java driver)

Type 1-JDBC-ODBC bridge driver

- The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver.
- In Java 8 and later versions, the JDBC-ODBC Bridge has been removed.
- Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

JDBC-ODBC architecture

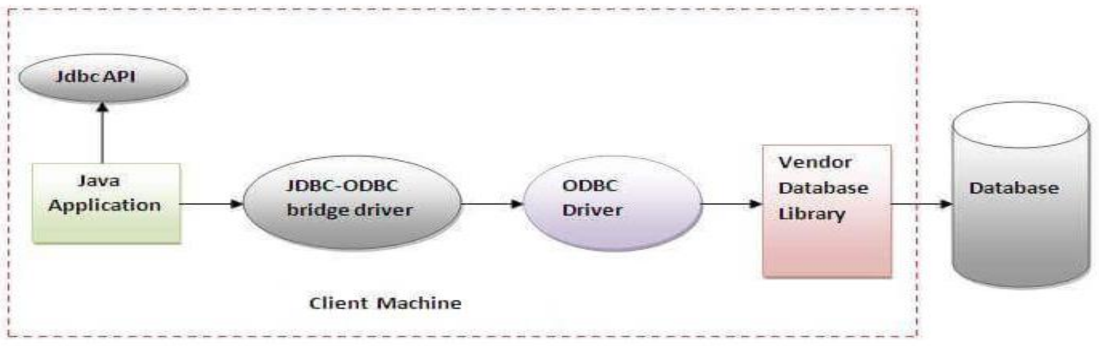


Figure- JDBC-ODBC Bridge Driver

Type 2-Native-API driver(Part Java, Part Native Driver)

- The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.

Native API Driver Architecture

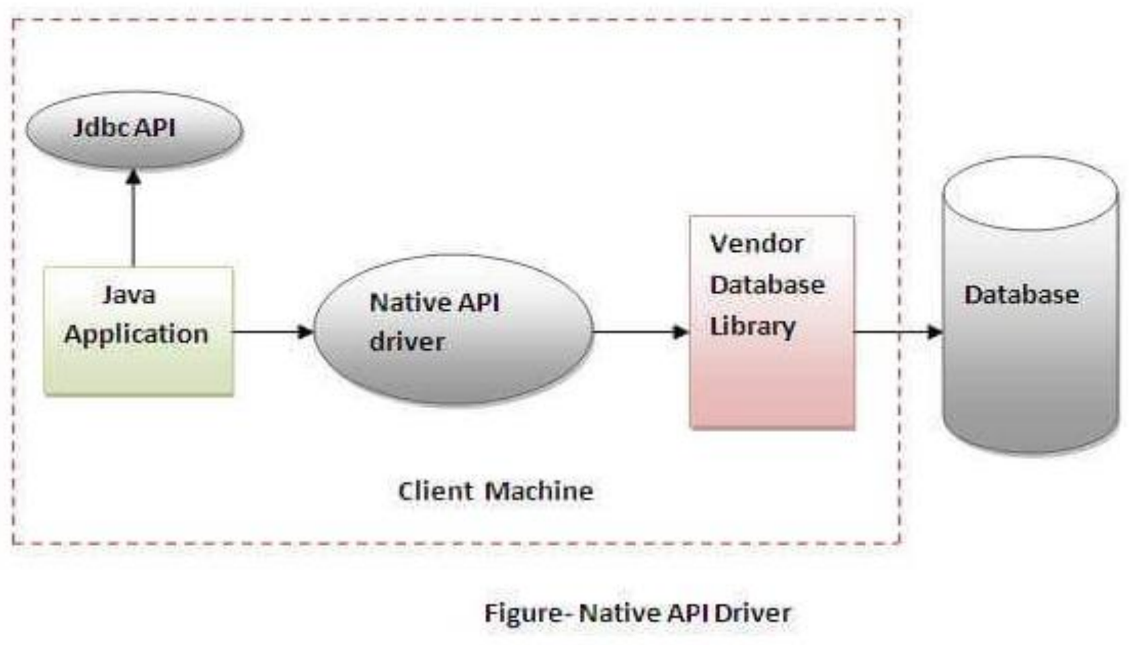


Figure 1:25 Native API Driver Architecture

Advantages and Drawbacks of Type 2 driver

Advantage:

- performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

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

Type 3- Network Protocol driver(Intermediate Database Access Server)

- The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

Type 3 Drive Architecture

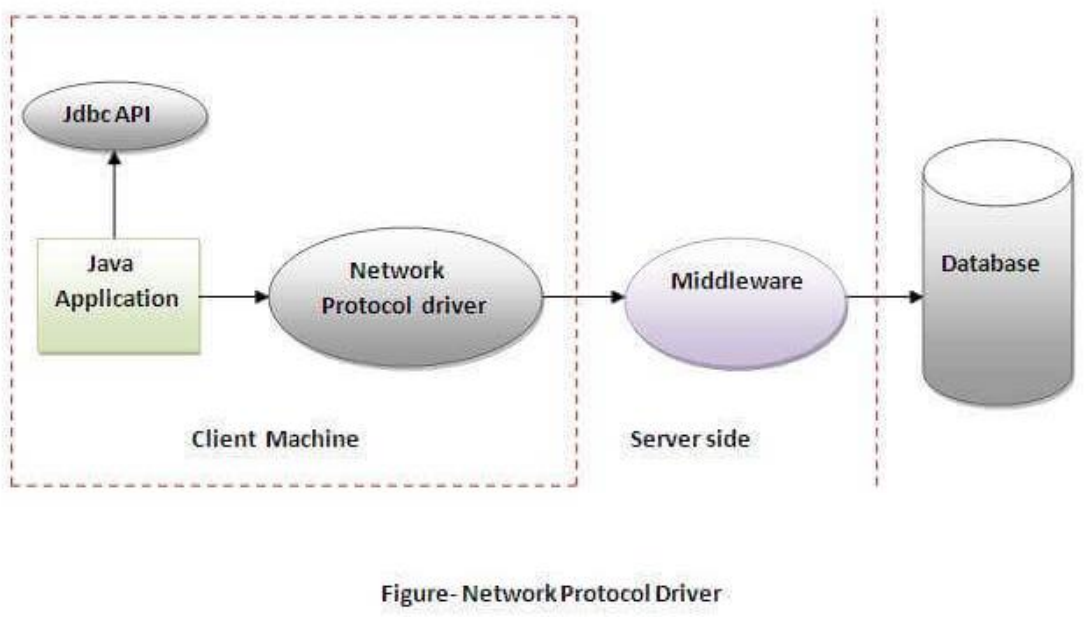


Figure 1:26 Type 3 Drive Architecture

Type 3: Advantages and Drawbacks

Advantage:

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

Disadvantages:

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

Type 4- Thin driver (Pure Java Driver)

- The thin driver converts JDBC calls directly into the vendor-specific database protocol.
- That is why it is known as thin driver.
- It is fully written in Java language.

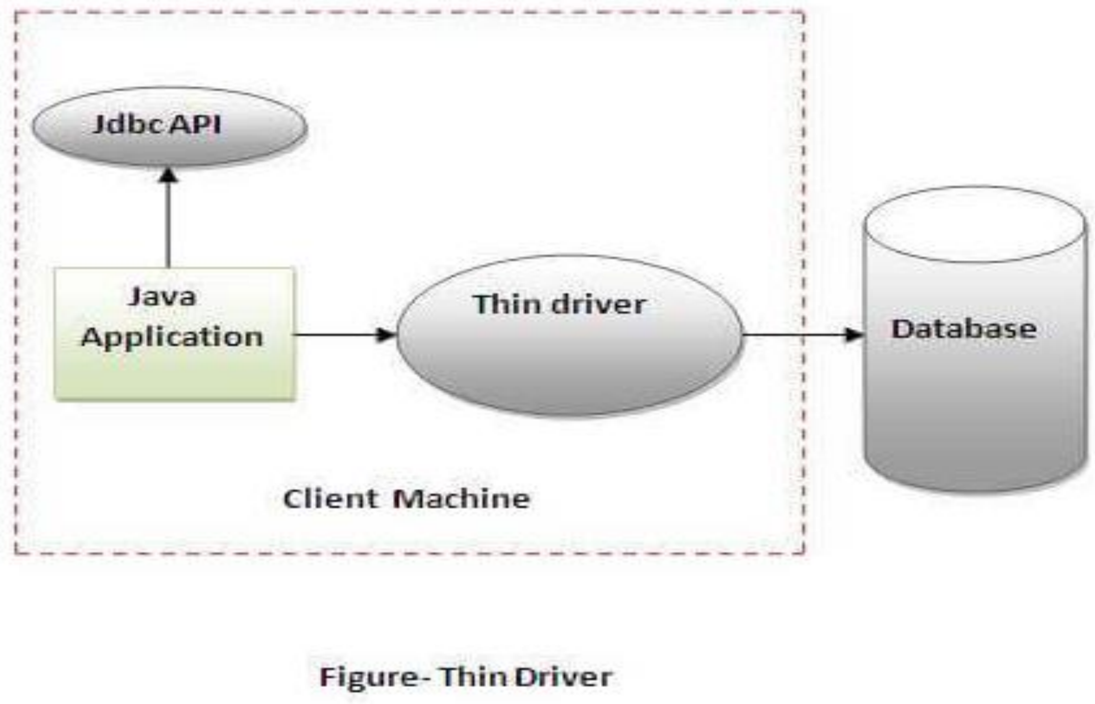


Figure 1:27 Thin Driver

Advantage:

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

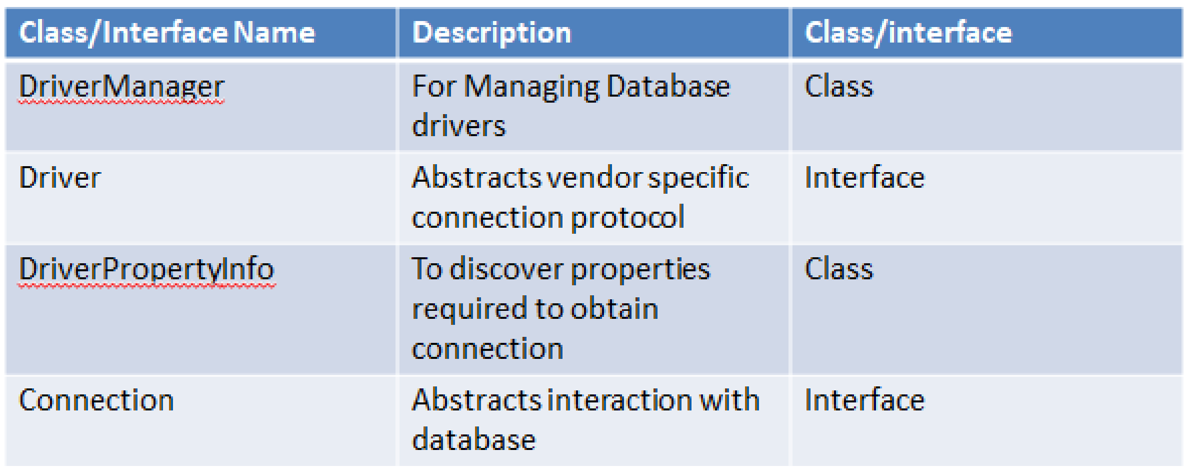
Disadvantage:

- Drivers depend on the Database.

Java.sql package

- Class groups based on functionalities
- Connection Management
- Database Access
- Data Types
- Database MetaData
- Exceptions and Warnings

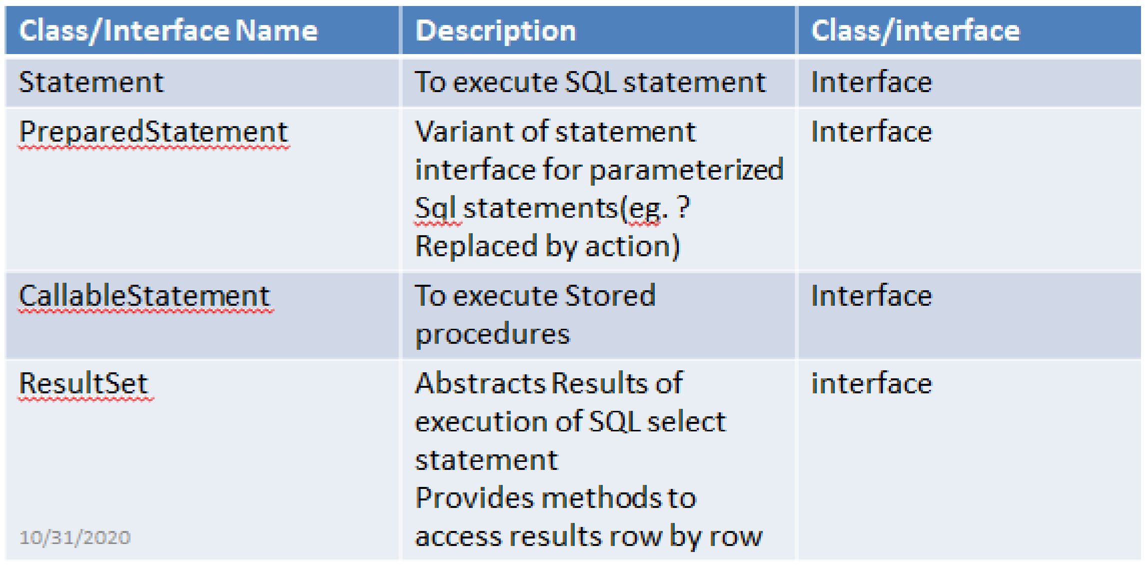
Connection management classes/interfaces of java.sql Package:
To establish connection to database



Class/Interface Name	Description	Class/interface
DriverManager	For Managing Database drivers	Class
Driver	Abstracts vendor specific connection protocol	Interface
DriverPropertyInfo	To discover properties required to obtain connection	Class
Connection	Abstracts interaction with database	Interface

Table: 1.1 Connection management classes/interfaces of java.sql Package

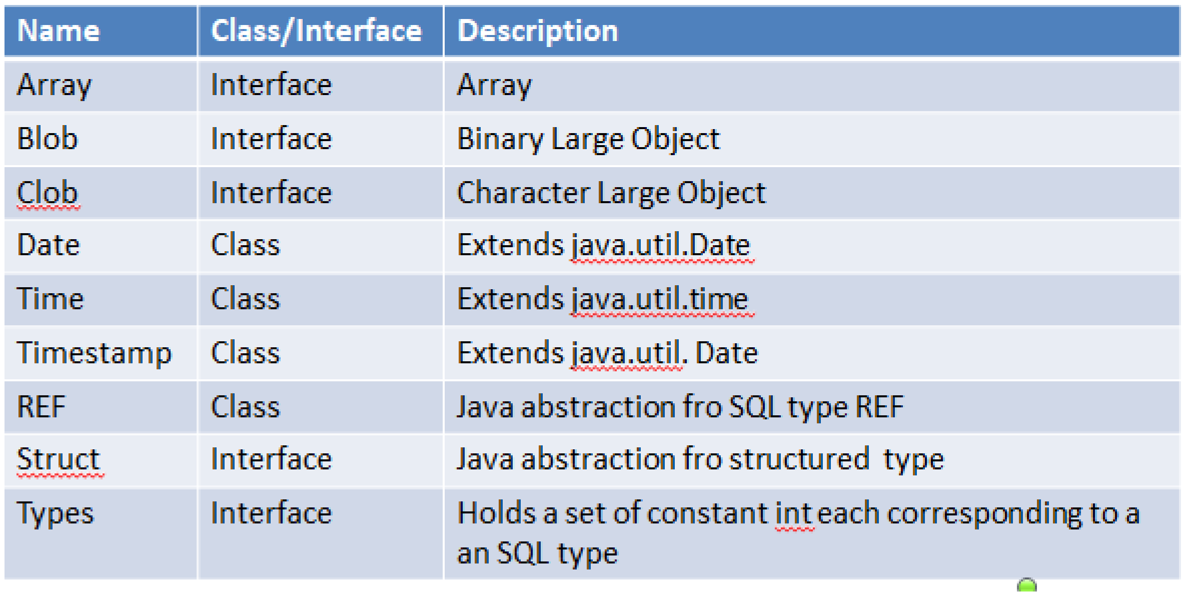
Database Access: classes/interfaces of java.sql Package:
To send SQL statement to database for execution and getting results



Class/Interface Name	Description	Class/interface
Statement	To execute SQL statement	Interface
PreparedStatement	Variant of statement interface for parameterized Sql statements(eg. ? Replaced by action)	Interface
CallableStatement	To execute Stored procedures	Interface
ResultSet 10/31/2020	Abstracts Results of execution of SQL select statement Provides methods to access results row by row	interface

Table: 1.2 Database Access: classes/interfaces of java.sql Package

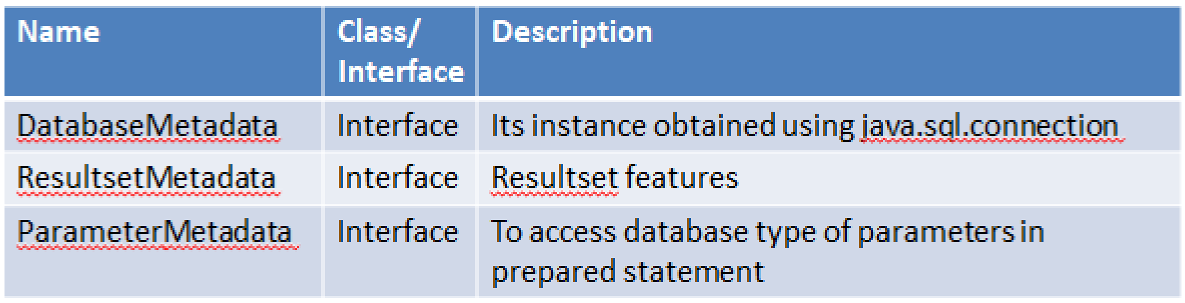
Data Types java.sql Package:
For programming the database



Name	Class/Interface	Description
Array	Interface	Array
Blob	Interface	Binary Large Object
Clob	Interface	Character Large Object
Date	Class	Extends java.util.Date
Time	Class	Extends java.util.time
Timestamp	Class	Extends java.util. Date
REF	Class	Java abstraction fro SQL type REF
Struct	Interface	Java abstraction fro structured type
Types	Interface	Holds a set of constant int each corresponding to a an SQL type

Table: 1.3 Data Types java.sql Package

Metadata: java.sql Package: To obtain metadata about database, statements,resultsets



Name	Class/ Interface	Description
DatabaseMetadata	Interface	Its instance obtained using java.sql.connection
ResultsetMetadata	Interface	Resultset features
ParameterMetadata	Interface	To access database type of parameters in prepared statement

Table: 1.4 Metadata: java.sql Package

JDBC Steps

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

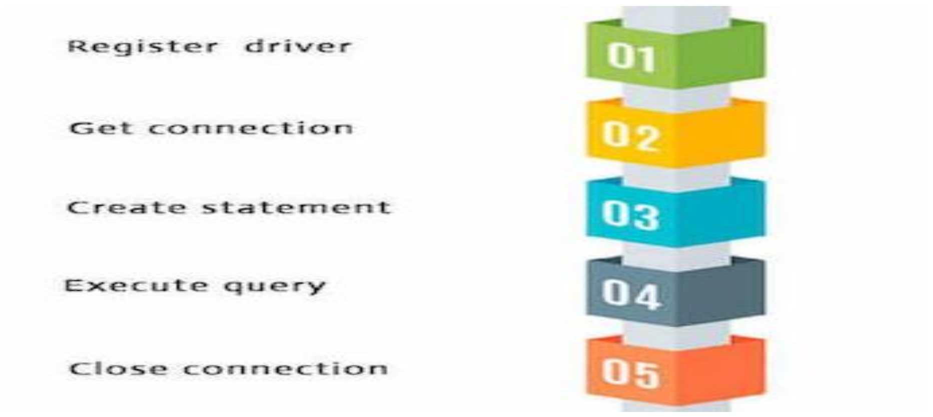


Figure 1:28 JDBC Steps

Step 1: Register the driver class

- The **forName()** method of Class class is used to register the driver class. This method is used to dynamically load the driver class.
- Syntax:

```
public static void forName(String className)throws ClassNotFoundException
```

- Example:
- Class.forName**("com.mysql.jdbc.Driver");**

DriverManager class

- can manage multiple drivers
- Each driver has to register with DriverManager class
- In JDBC we load database driver using java.lang.Class loader object
- At runtime classloader locates and loads the driver from the class path using bootstrap class loader

Step 2: Create the connection object

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

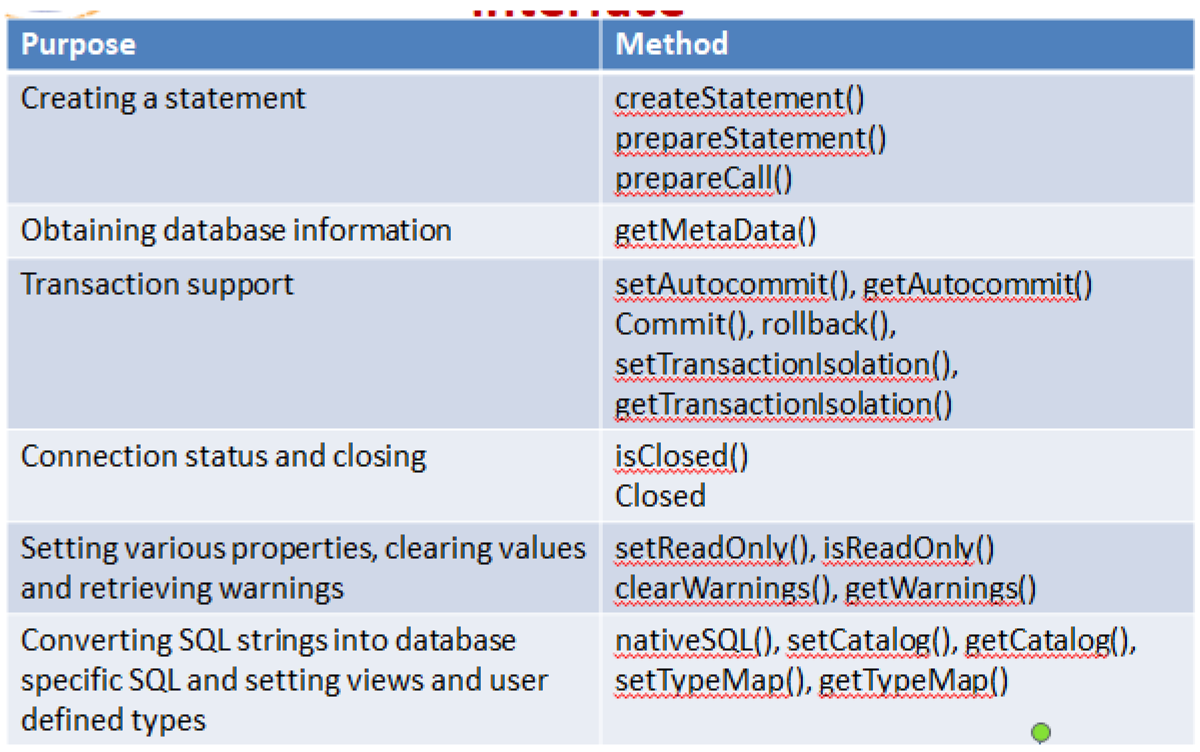
1) **public static** Connection getConnection(String url)**throws** SQLException

2) **public static** Connection getConnection(String url,String name,String password) **throws** SQLException

- Example

Connection con=DriverManager.getConnection(**"jdbc:mysql://localhost:3306/stud","root","");**

Establishing a Connection: java.sql.Connection interface:



Purpose	Method

Creating a statement	createStatement()
	prepareStatement()
	prepareCall()

Obtaining database information	getMetaData()

Transaction support	setAutocommit(), getAutocommit()
	Commit(), rollback(),
	setTransactionIsolation(),
	getTransactionIsolation()

Connection status and closing	isClosed() Closed

Setting various properties, clearing values and retrieving warnings	setReadOnly(), isReadOnly()
	clearWarnings(), getWarnings()

Converting SQL strings into database specific SQL and setting views and user defined types	nativeSQL(), setCatalog(), getCatalog(), setTypeMap(), getTypeMap()

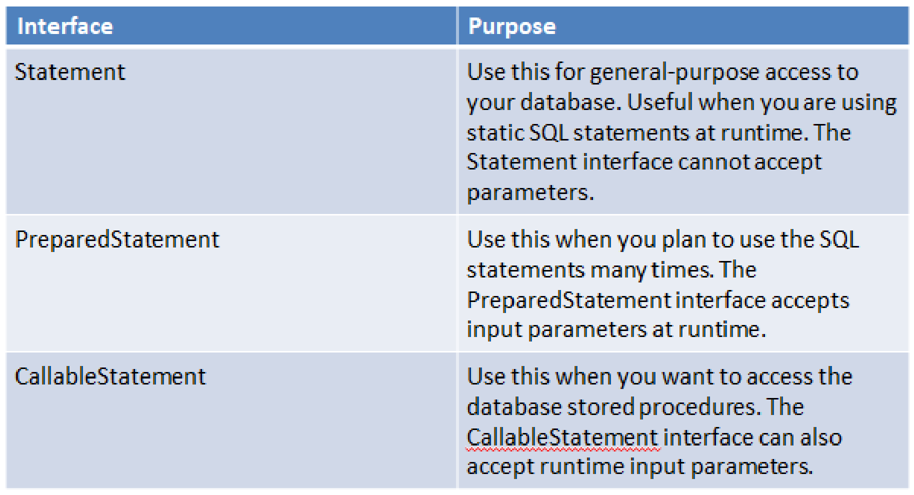
Table: 1.5 Establishing a Connection: java.sql.Connection interface

Step 3-Create the Statement object:

The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.

- Syntax:
- Example: Statement st=con.createStatement();

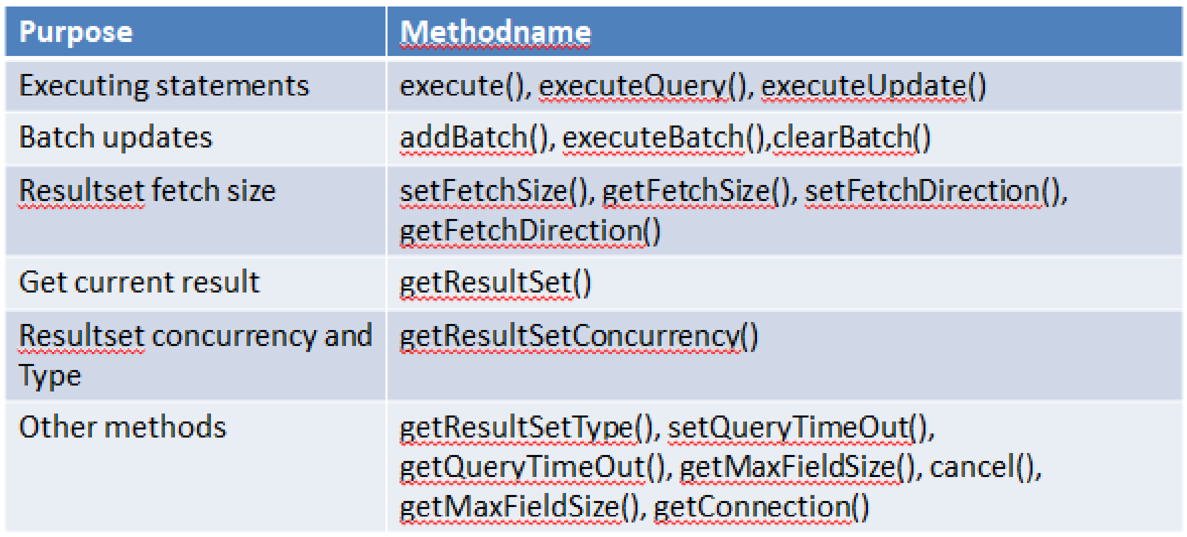
Creating and Executing SQL statement



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

Table: 1.6 Creating and Executing SQL statement

Statement interface Methods

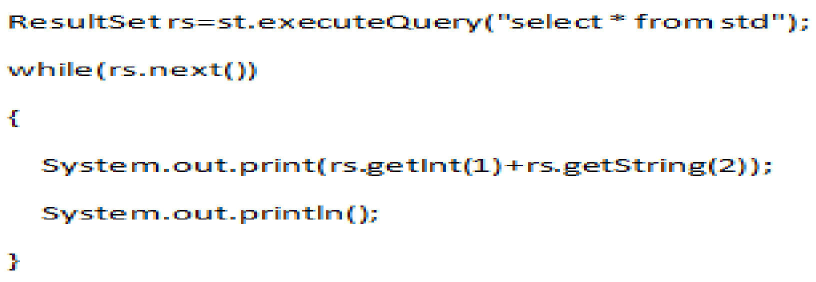


Purpose	Methodname
Executing statements	execute(), executeQuery(), executeUpdate()
Batch updates	addBatch(), executeBatch(),clearBatch()
Resultset fetch size	setFetchSize(), getFetchSize(), setFetchDirection(), getFetchDirection()
Get current result	getResultSet()
Resultset concurrency and Type	getResultSetConcurrency()
Other methods	getResultSetType(), setQueryTimeOut(), getQueryTimeOut(), getMaxFieldSize(), cancel(), getMaxFieldSize(), getConnection()

Table: 1.7 Statement interface Methods

Step 4: Execute the query

- The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.
- Syntax: **public** ResultSet executeQuery(String sql)**throws** SQLException
- Example:



```
ResultSet rs=st.executeQuery("select * from std");
while(rs.next())
{
   System.out.print(rs.getInt(1)+rs.getString(2));
   System.out.println();
}
```

Querying the database: The ResultSet interface

- Methods:
- Execute():
  - Syntax:public boolean execute(String sql) throws SQLException
- ExecuteQuery():
  - Syntax:public ResultSet executeQuery(String sql) throws SQLException

Methods to retrieve data

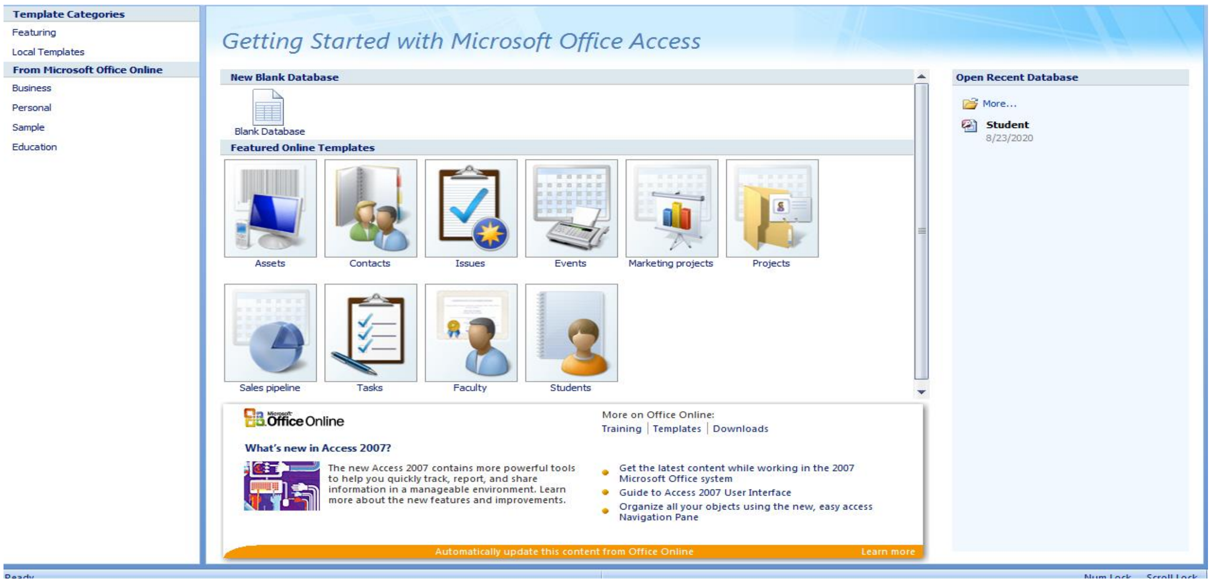
- getArray()
- getBlob()
- getBoolean()
- getInt()
- getDate()
- getByte()
- getLong()
- getString()
- getDouble()
- getObject()

getTime

Step 5 :Close the connection object

- By closing connection object statement and ResultSet will be closed automatically.
- The close() method of Connection interface is used to close the connection.
- Syntax: **public void** close()**throws** SQLException
- Example: con.close()

JDBC Connectivity using Type 1 – JDBC ODBC Driver
Create a database Student



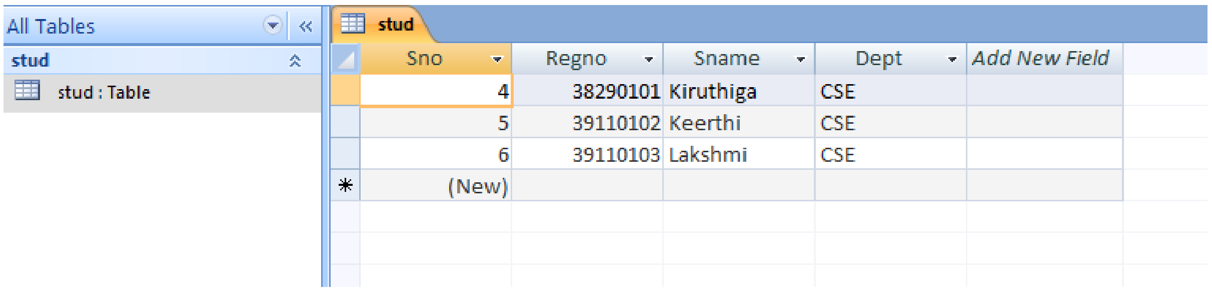
What's new in Access 2007?

The new Access 2007 contains more powerful tools to help you quickly track, report, and share information in a manageable environment. Learn more about the new features and improvements.

- Get the latest content while working in the 2007 Microsoft Office system
- Guide to Access 2007 User Interface
- Organize all your objects using the new, easy access Navigation Pane

Figure 1:29 Create a database Student

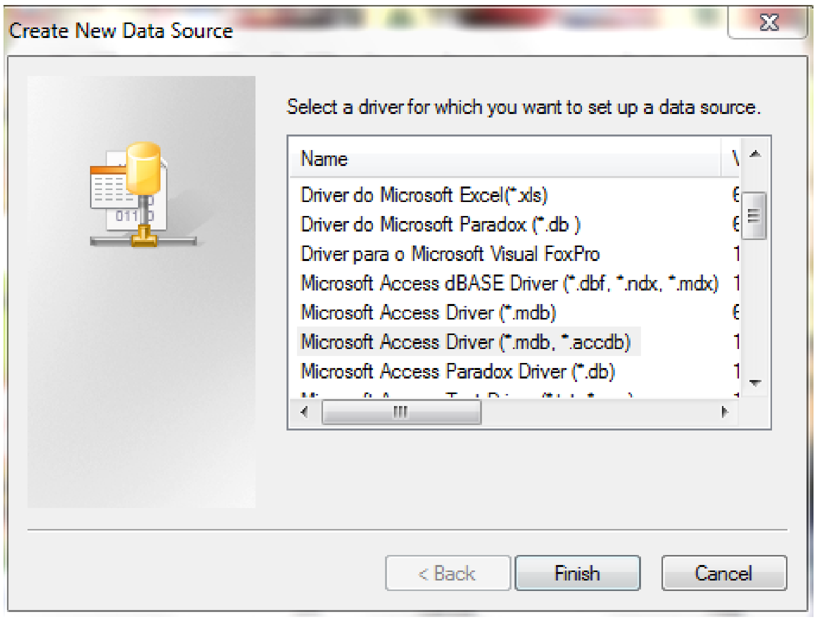
JDBC Connectivity using Type 1 – JDBC ODBC DriverCreate Table stud



Sno	Regno	Sname	Dept	Add New Field
4	38290101	Kiruthiga	CSE	
5	39110102	Keerthi	CSE	
6	39110103	Lakshmi	CSE	
(New)				

Figure 1:30 JDBC ODBC Driver Create Table stud

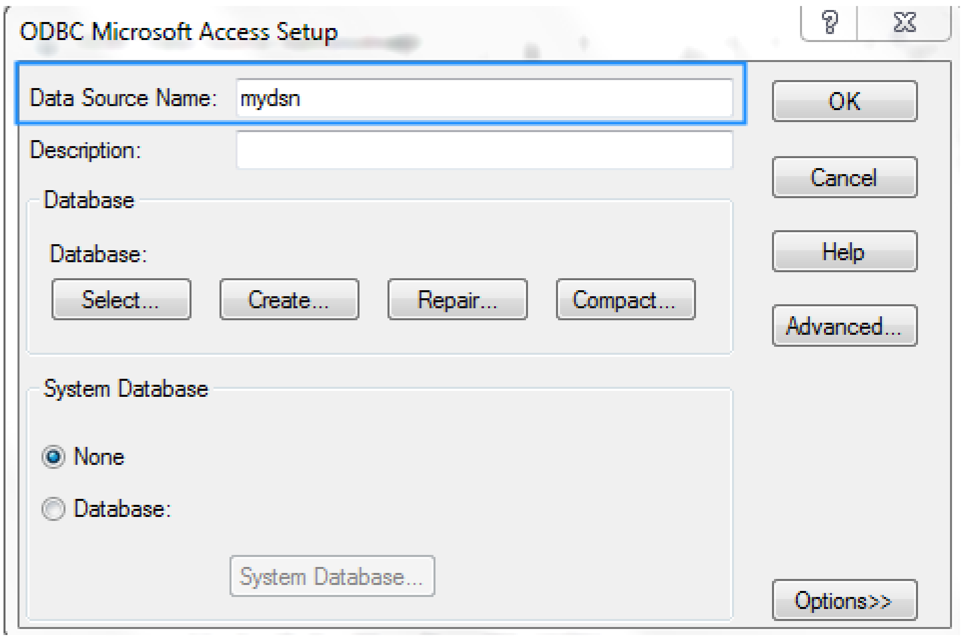
JDBC Connectivity using Type 1 – JDBC ODBC DriverCreate new data source


Create New Data Source

Select a driver for which you want to set up a data source.

Name
Driver do Microsoft Excel(*.xls)
Driver do Microsoft Paradox (*.db)
Driver para o Microsoft Visual FoxPro
Microsoft Access dBASE Driver (*.dbf, *.ndx, *.mdx)
Microsoft Access Driver (*.mdb)
Microsoft Access Driver (*.mdb, *.accdb)
Microsoft Access Paradox Driver (*.db)

Figure 1:31 – JDBC ODBC DriverCreate new data source


ODBC Microsoft Access Setup

Data Source Name: mydsn

Description:

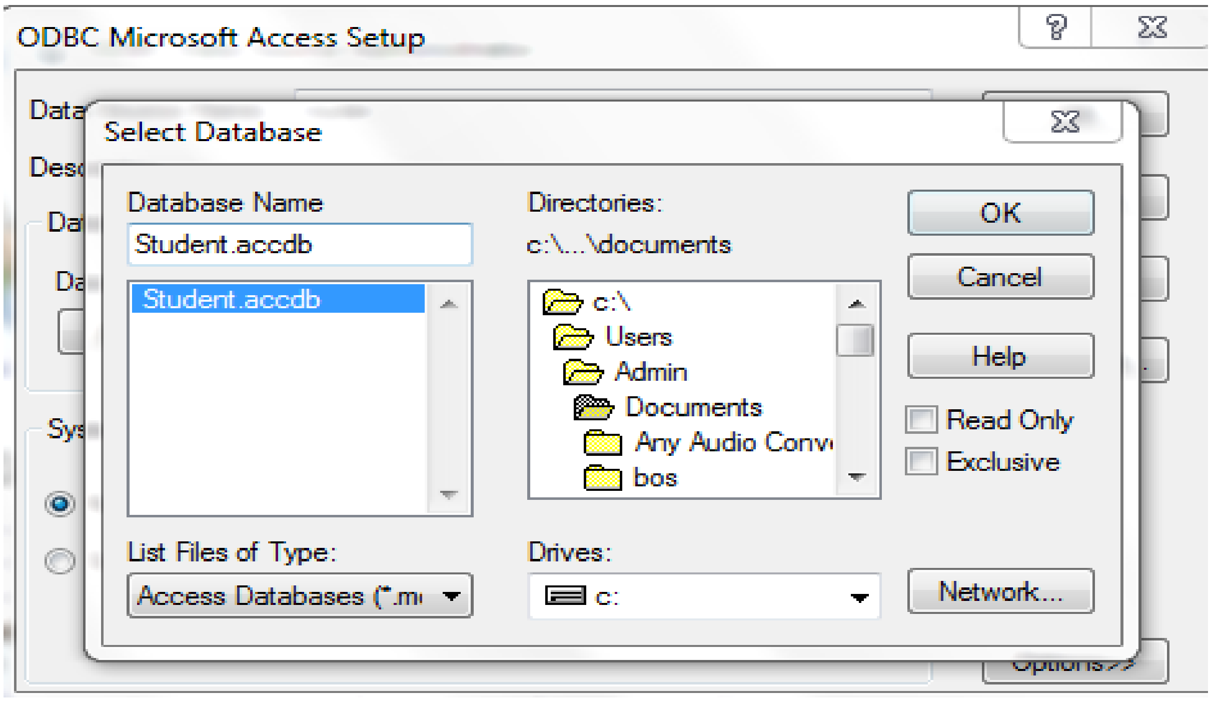
Database

Database:

System Database

(● None) Database:

Figure 1:32 – JDBC ODBC DriverCreate new data source


ODBC Microsoft Access Setup

Select Database

Database Name: Student.accdb

Student.accdb

Directories: c:\...\documents

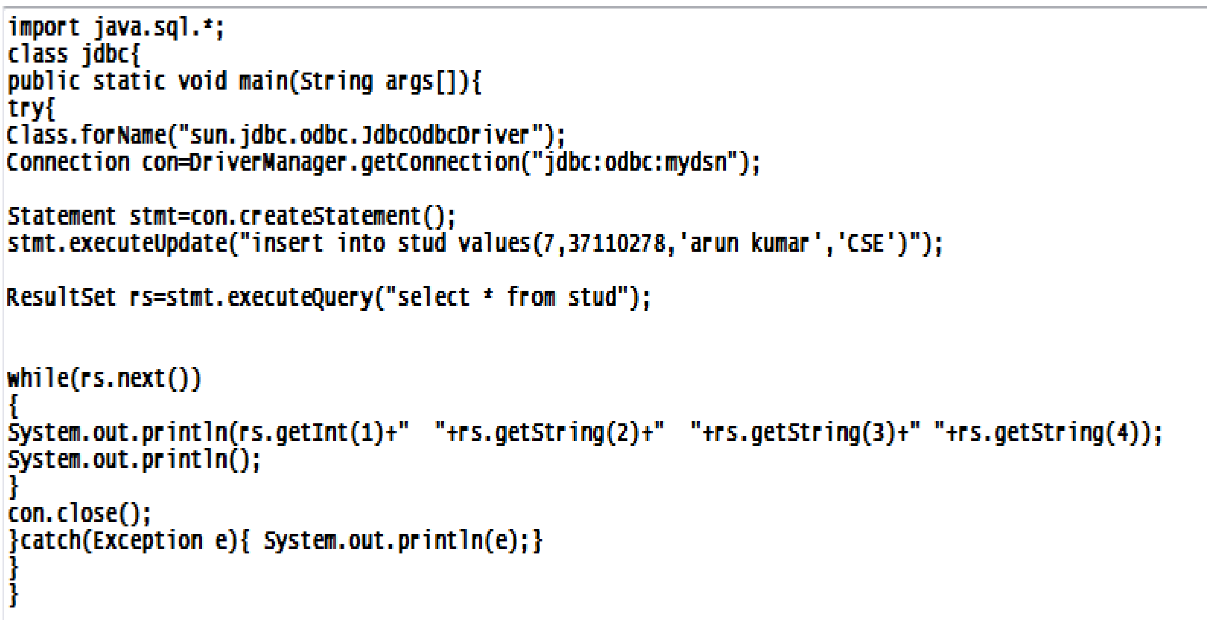
c:\
Users
Admin
Documents
Any Audio Conv
bos

List Files of Type: Access Databases (*.m

Drives: c:

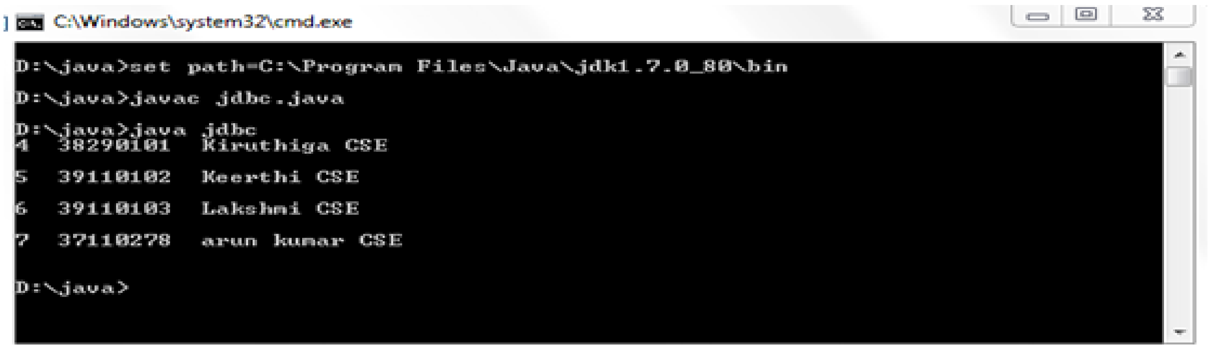
☐ Read Only
☐ Exclusive

Figure 1:33 – JDBC ODBC DriverCreate new data source



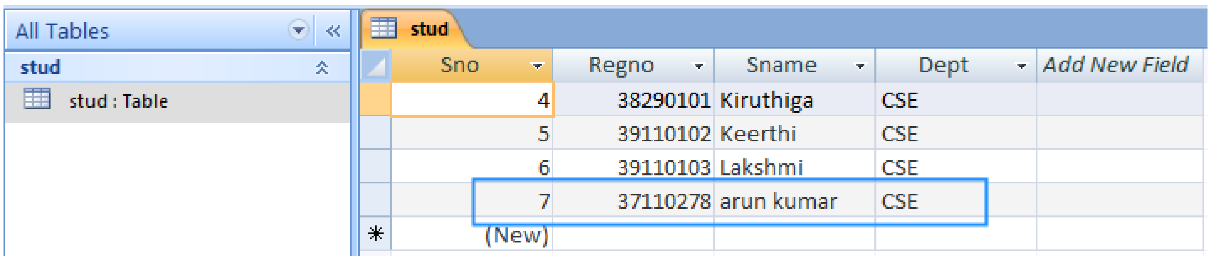
```
import java.sql.*;
class jdbc{
public static void main(String args[]){
try{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con=DriverManager.getConnection("jdbc:odbc:mydsn");

Statement stmt=con.createStatement();
stmt.executeUpdate("insert into stud values(7,37110278,'arun kumar','CSE')");

ResultSet rs=stmt.executeQuery("select * from stud");


while(rs.next())
{
System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3)+" "+rs.getString(4));
System.out.println();
}
con.close();
}catch(Exception e){ System.out.println(e);}
}
}
```

Output window after Java code execution



```
D:\java>set path=C:\Program Files\Java\jdk1.7.0_80\bin
D:\java>javac jdbc.java
D:\java>java jdbc
4  38290101  Kiruthiga CSE

5  39110102  Keerthi  CSE

6  39110103  Lakshmi  CSE

7  37110278  arun kumar  CSE

D:\java>
```

Database table update after the Java code execution



Sno	Regno	Sname	Dept	Add New Field
4	38290101	Kiruthiga	CSE	
5	39110102	Keerthi	CSE	
6	39110103	Lakshmi	CSE	
7	37110278	arun kumar	CSE	
(New)				

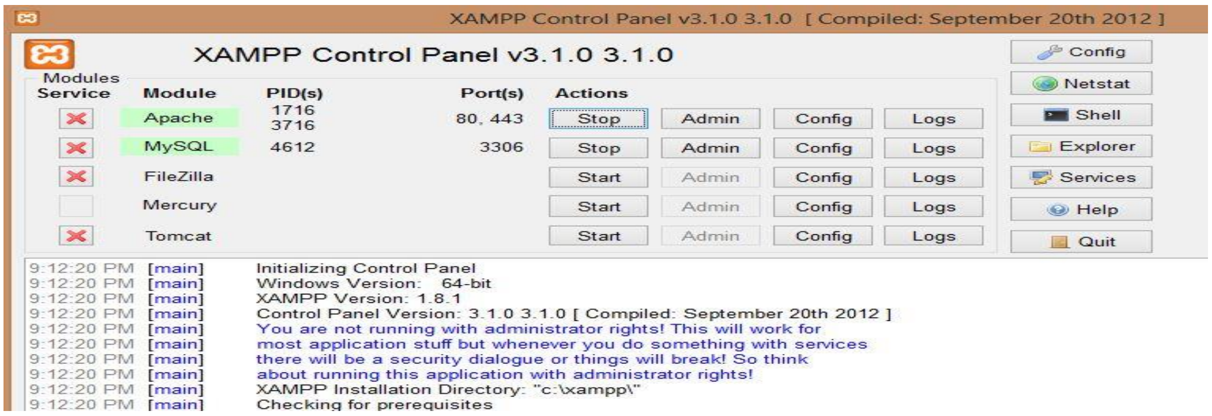
Figure 1:34 Data Base

JDBC Connectivity using Type 4 Driver

Step wise Manual:

Step 1: Install XAMPP or WAMP or MySQL server

Step 2 :Go to control panel and start Apache and MySQL.

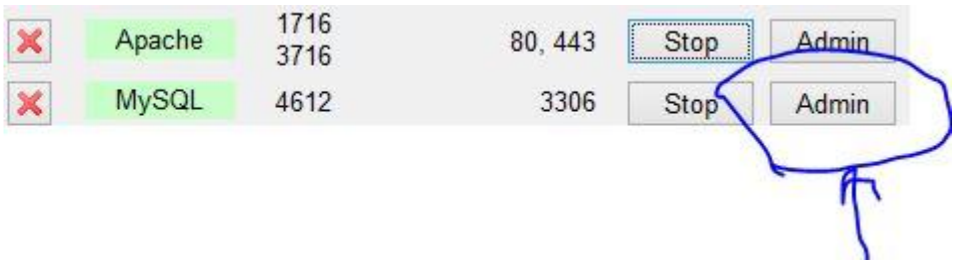

XAMPP Control Panel v3.1.0 3.1.0 [Compiled: September 20th 2012]
XAMPP Control Panel v3.1.0 3.1.0
Config
Netstat
Shell
Explorer
Services
Help
Quit

Modules Service	Module	PID(s)	Port(s)	Actions			
✗	Apache	1716 3716	80, 443	Stop	Admin	Config	Logs
✗	MySQL	4612	3306	Stop	Admin	Config	Logs
✗	FileZilla			Start	Admin	Config	Logs
	Mercury			Start	Admin	Config	Logs
✗	Tomcat			Start	Admin	Config	Logs

```
9:12:20 PM  [main]     Initializing Control Panel
9:12:20 PM  [main]     Windows Version:   64-bit
9:12:20 PM  [main]     XAMPP Version: 1.8.1
9:12:20 PM  [main]     Control Panel Version: 3.1.0 3.1.0 [ Compiled: September 20th 2012 ]
9:12:20 PM  [main]     You are not running with administrator rights! This will work for
9:12:20 PM  [main]     most application stuff but whenever you do something with services
9:12:20 PM  [main]     there will be a security dialogue or things will break! So think
9:12:20 PM  [main]     about running this application with administrator rights!
9:12:20 PM  [main]     XAMPP Installation Directory: "c:\xampp\"
9:12:20 PM  [main]     Checking for prerequisites
```

Figure 1:35 Xampp Control

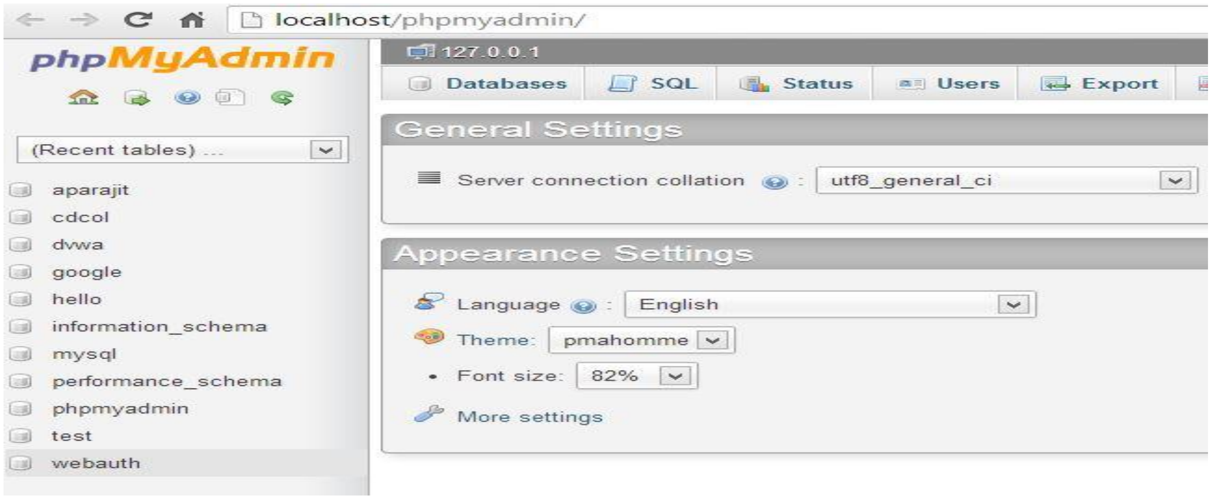
- Step 3: Click Admin Button.



✗	Apache	1716 3716	80, 443	Stop	Admin
✗	MySQL	4612	3306	Stop	Admin

Figure 1:35 Click Admin Button.

You will be getting PHPMyAdmin in the browser.



aparajit
cdcol
dvwa
google
hello
information_schema
mysql
performance_schema
phpmyadmin
test
webauth

General Settings

Server connection collation : utf8_general_ci

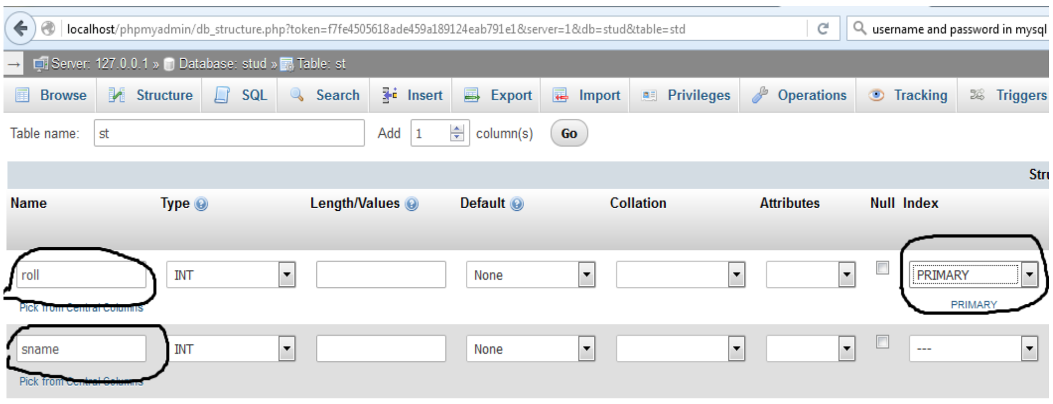
Appearance Settings

Language : English
Theme: pmahomme
Font size: 82%
More settings

Figure 1:36 PHPMyAdmin in the browser

Step 5:

Create the fields in the table and set the primary key and **save** the changes



localhost/phpmyadmin/db_structure.php?token=f7fe4505618ade459a189124eab791e1&server=1&db=stud&table=std

Server: 127.0.0.1 » Database: stud » Table: st

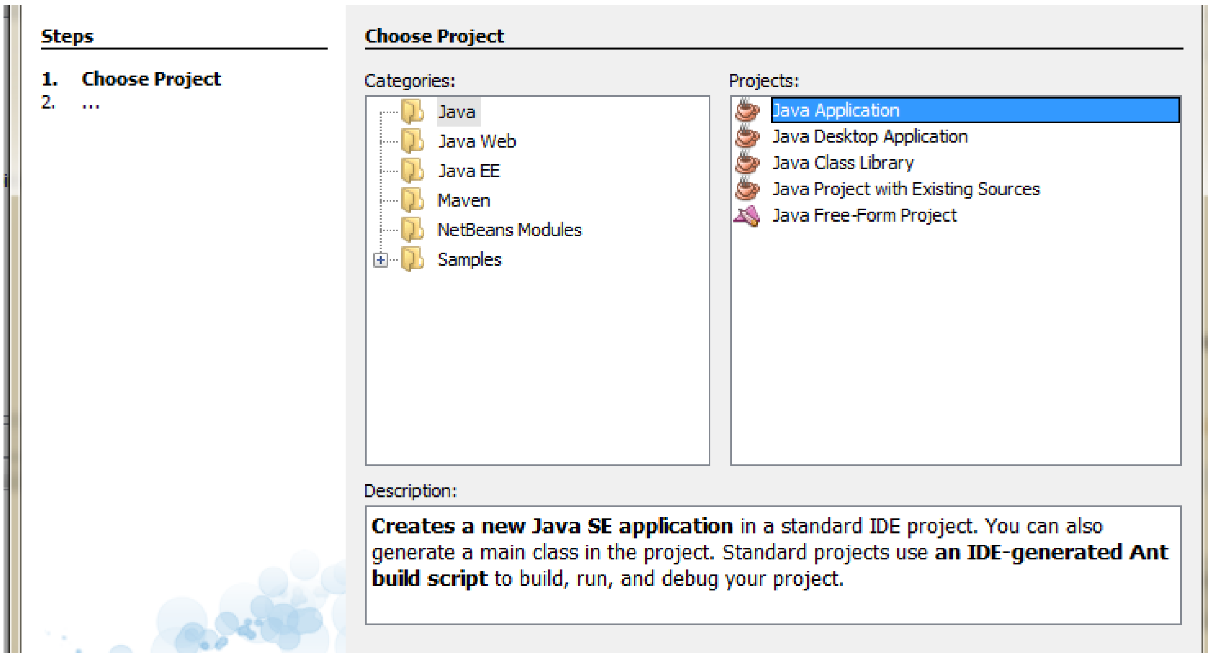
Table name: st Add 1 column(s)

Name	Type	Length/Values	Default	Collation	Attributes	Null	Index
roll	INT		None				PRIMARY
sname	INT		None				---

Figure 1:37 Create the fields in the table

- Step 6: **Open Netbeans editor and do the following:**

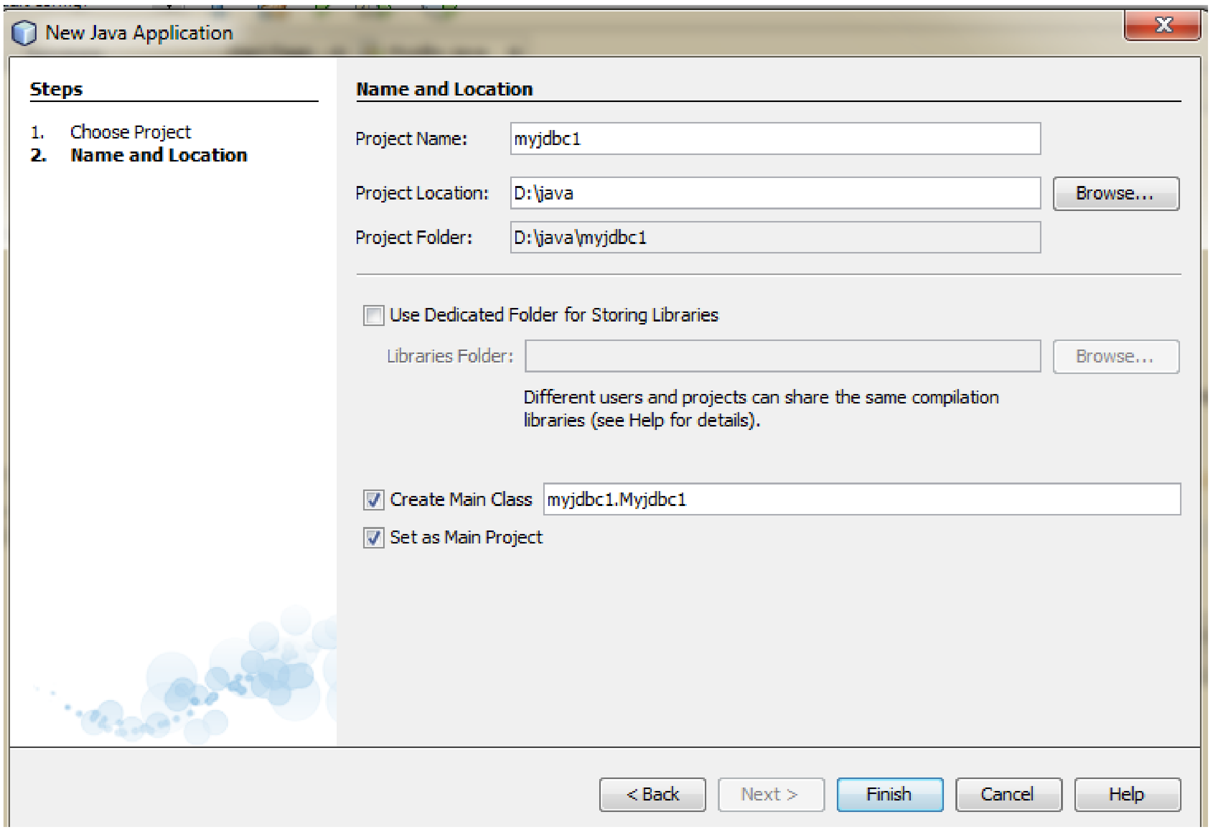
1. Create a new Project: java->java Application



Creates a new Java SE application in a standard IDE project. You can also generate a main class in the project. Standard projects use **an IDE-generated Ant build script** to build, run, and debug your project.

Figure 1:38 Open Netbeans editor

2. Click next and give any name of the project and click finish


New Java Application

Steps

1. Choose Project
2. **Name and Location**

Name and Location

Project Name: myjdbc1

Project Location: D:\java

Project Folder: D:\java\myjdbc1

☐ Use Dedicated Folder for Storing Libraries

Libraries Folder:

Different users and projects can share the same compilation libraries (see Help for details).

☑ Create Main Class myjdbc1.Myjdbc1

☑ Set as Main Project

Figure 1:39 Name and Location

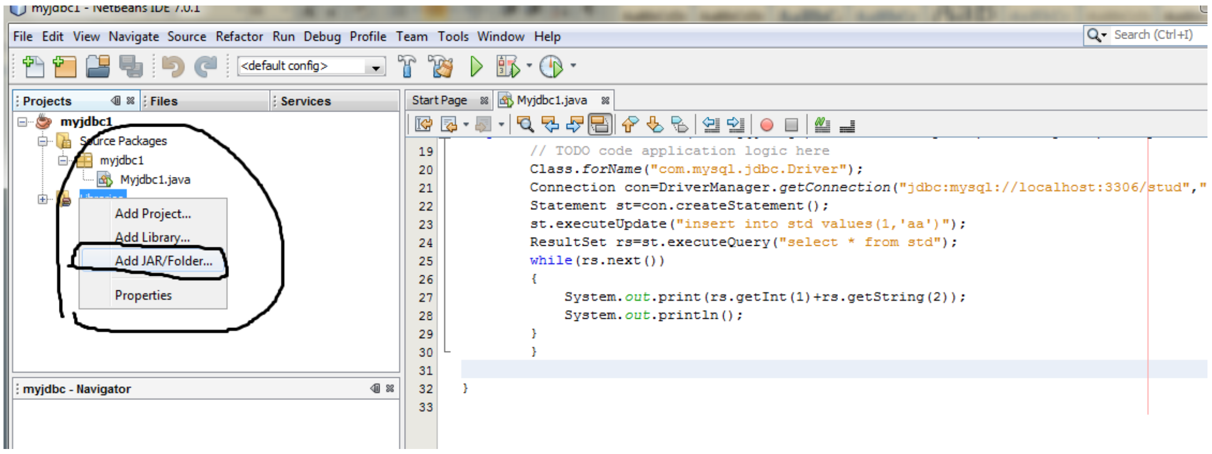
- 3. Type the JDBC code:

```
import java.io.IOException;
import java.sql.*;
public class Myjdbc1 {
    public static void main(String[] args) throws IOException, SQLException, Exception {
     Class.forName("com.mysql.jdbc.Driver");
     Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/stud","root","");
     Statement st=con.createStatement();
     st.executeUpdate("insert into std values(1,'aa')");
     ResultSet rs=st.executeQuery("select * from std");
     while(rs.next())
     {
        System.out.print(rs.getInt(1)+rs.getString(2));
        System.out.println();
     }
   con.close();
    }
}
```

4. Select the Libraries folder and right click and select Add Jar/folder and browse the path:

C:\Program Files\NetBeans 7.3.1\ide\modules\ext

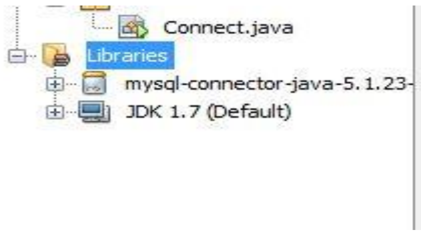
A dialog box opens Navigate to the path mentioned above and select mysql-connector-java-5.1.23-bin



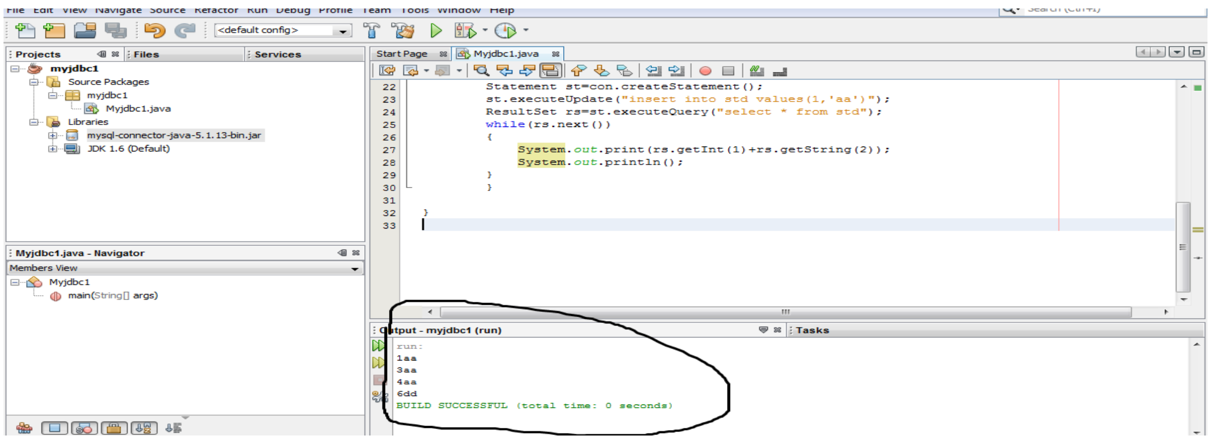
```
        // TODO code application logic here
        Class.forName("com.mysql.jdbc.Driver");
        Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/stud","
        Statement st=con.createStatement();
        st.executeUpdate("insert into std values(1,'aa')");
        ResultSet rs=st.executeQuery("select * from std");
        while(rs.next())
        {
            System.out.print(rs.getInt(1)+rs.getString(2));
            System.out.println();
        }
        }

    }
```

Figure 1:40 dialog box opens Navigate to the path

5. mysql connector is now added in the libraries folder.



5. Now press F6 and run the project. Output will be displayed in the console output window



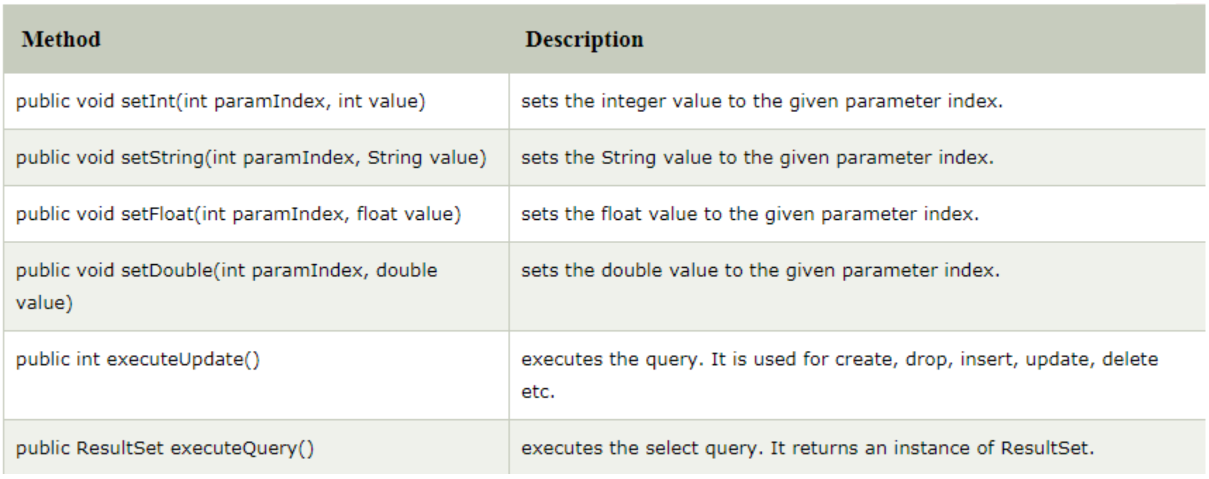
```
        Statement st=con.createStatement();
        st.executeUpdate("insert into std values(1,'aa')");
        ResultSet rs=st.executeQuery("select * from std");
        while(rs.next())
        {
            System.out.print(rs.getInt(1)+rs.getString(2));
            System.out.println();
        }
        }

    }
```

```
run:
1aa
3aa
4aa
6dd
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 1:41 Output will be displayed in the console output window

Prepared statements

- Creating parameterized statement such that data for parameters can be substituted dynamically
- Creating statement involving data values that can always be represented as character strings
- Pre-compiling SQL statement to avoid repeated compiling of same SQL statement
- Eg. Select * from <tab> where user_id=?
- If the value is not known then it is replaced by '?' during runtime
- A PreparedStatement object can hold precompiled SQL statements
- Syntax:
  public PreparedStatement prepareStatement(String query)**throws** SQLException{}



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

Table: 1.8 Prepared statements

Create a table

create table emp(id number(10),name varchar2(50));

Insert the records using PreparedStatement

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

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

PreparedStatement stmt=con.prepareStatement("insert into Emp values(?,?)");
stmt.setInt(1,101);//1 specifies the first parameter in the query
stmt.setString(2,"Ratan");

int i=stmt.executeUpdate();
System.out.println(i+" records inserted");

con.close();

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

}
}
```

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT- II Advanced Java – SBS1301

Java Script

JavaScript is a lightweight, interpreted programming language. It is designed for creating network-centric applications. It is complimentary to and integrated with Java. JavaScript is very easy to implement because it is integrated with HTML. It is open and cross-platform.

Embedding Java Script In HTML:

JavaScript is embedded into HTML and XHTML documents using the <script> element. This element can be used to embed the JavaScript directly into the web page (also known as inline), or to specify an external file that contains the JavaScript.

The <script> element is used with a number of attributes:

- **defer** - used to inform the browser that the script associated with this <script> element generates content (in other words the document.write() method is used). This can be either true or false. The default setting (i.e. if this is not specified) is false.

- **language** - This argument is used to announce the version of JavaScript that is contained within the corresponding <script> elements. This argument is now deprecated.

- **src**- Specifies URL of an external file containing the JavaScript. This argument overrides any JavaScript contained within the body of this <script> element.

- **type** - Indicates to the browser the type of content contained within the <script> body. This is typically be set to "text/javascript".

As with most HTML elements the <script> body must always be terminated with the </script> element.

Example:

```
<script type="text/javascript">
// JavaScript code goes here
</script>
```

Example Html File:

```
<script src="/j-scripts/myjscript.js" type="text/javascript">
</script>
```

Example JavaScript File:

```
document.writeln ("This is contained in an external JavaScript file")
```

Example Program:

```
<!DOCTYPE html>
<html lang="en-US">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1">
   <title>Today's Date</title>
   <script>
      let d = new Date();
      alert("Today's date is " + d);
   </script>
</head>
<body>
</body>
</html>
```

Example Program:

```
<!DOCTYPE html>
<html lang="en-US">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1">
   <title>Today's Date</title>
</head>
<body>
  <script>
     let d = new Date();
     document.body.innerHTML = "<h1>Today's date is " + d + "</h1>"
  </script>
</body> </html>
```

Variables:

Variables in JavaScript are containers which hold reusable data. It is the basic unit of storage in a program.

- The value stored in a variable can be changed during program execution.
- A variable is only a name given to a memory location, all the operations done on the variable effects that memory location.
- In JavaScript, all the variables must be declared before they can be used.

Before ES2015, JavaScript variables were solely declared using the *var* keyword followed by the name of the variable and semi-colon. Below is the syntax to create variables in JavaScript:
var var_name;
var x;
The var_name is the name of the variable which should be defined by the user and should be unique. These type of names are also known as **identifiers**. The rules for creating an identifier in JavaScript are, the name of the identifier should not be any pre-defined word(known as keywords), the first character must be a letter, an underscore (_), or a dollar sign ($). Subsequent characters may be any letter or digit or an underscore or dollar sign.

We can initialize the variables either at the time of declaration or also later when we want to use them. Below are some examples of declaring and initializing variables in JavaScript:

```
// declaring single variable

var name;

// declaring multiple variables

var name, title, num;

// initializng variables

var name = "Harsh";

name = "Rakesh";
```

Variable Scope in Javascript
Scope of a variable is the part of the program from where the variable may directly be accessible.
In JavaScript, there are two types of scopes:

1. **Global Scope** – Scope outside the outermost function attached to Window.
2. **Local Scope** – Inside the function being executed.

We have a global variable defined in first line in global scope. Then we have a local variable defined inside the function fun().

```
let globalVar = "This is a global variable";

function fun() {

let localVar = "This is a local variable";
console.log(globalVar);

console.log(localVar);

}

fun();
```

When we execute the function fun(), the output shows that both global as well as local variables are accessible inside the function as we are able to console.log them. This shows that inside the function we have access to both global variables (declared outside the function) and local variables (declared inside the function).

```
let globalVar = "This is a global variable";

function fun() {

let localVar = "This is a local variable";

}

fun();

console.log(globalVar);

console.log(localVar);
```

JavaScript Literals:

A JavaScript object literal is a comma-separated list of name-value pairs wrapped in curly braces. Object literals encapsulate data, enclosing it in a tidy package. This minimizes the use of global variables which can cause problems when combining code.

The following demonstrates an example object literal:

```
var myObject = {
   sProp: 'some string value',
   numProp: 2,
   bProp: false
};
```

Object literal property values can be of any data type, including array literals, functions, and nested object literals. Here is another object literal example with these property types:

```
var Swapper = {
   // an array literal
   images: ["smile.gif", "grim.gif", "frown.gif", "bomb.gif"],
   pos: { // nested object literal
      x: 40,
      y: 300
   },
   onSwap: function() { // function
```

```
        // code here
    }
};
```

Object Literal Syntax

Object literals are defined using the following syntax rules:

- A colon separates property name[1] from value.
- A comma separates each name-value pair from the next.
- There should be no comma after the last name-value pair.

Type Conversion

Most of the time, operators and functions automatically convert a value to the correct type. That's called "type conversion".For example, alert automatically converts any value to a string to show it. Mathematical operations convert values to numbers.There are also cases when user need to explicitly convert a value to put things right.

String Conversion in Javascript

String conversion happens when we need the string form of a value.

For example, alert(value) does it to show the value. The conversion here is done automatically. We can also use a call String(value) function for that conversion manually.

- let value = true;
- alert(typeof value); // **boolean**
- value = String(value); // now value is a string "true"
- alert(typeof value); // **string**
- String conversion is mostly obvious. A *true* becomes "*true*", *0* becomes "*0*".

Number Conversion

Numeric conversion happens in mathematical functions and expressions automatically.

For example, when division / is applied to non-numbers, which can be converted:

console.log("12"/"2"); //6 – Both strings are converted to Number

We can use a Number(value) function to explicitly convert a value to Number. Like below:

```
let str = "123";

let num = Number(str);

console.log(typeof str); //string

console.log(typeof num); //number
```

If the string to be converted is not a valid number, the result of such conversion is **NaN**, for example:

```
let str = "One two three";

let num = Number(str);

console.log(num); //NaN
```

Some other numeric conversion rules for explicit conversation are below

```
//undefined becomes null

let str;

let num = Number(str);

console.log(str); //undefined

console.log(num); //NaN

//null becomes 0

let str = null;

let num = Number(str);

console.log(str); //null

console.log(num); //0

//true/false becomes 1/0
```

```
let str1 = true;

let str2 = false;

let num1 = Number(str1);

let num2 = Number(str2);

console.log(str1); //true

console.log(num1); //1

console.log(str2); //false

console.log(num2); //0

//In strings whitespaces from the start and the end are removed. Then, if //the remaining strin
g is empty, the result is 0. Otherwise, the number is //“read” from the string. An error gives N
aN if the string cannot be //converted.

let str1 = "       ";

let str2 = "  123 ";

let str3 = "  123z ";

let num1 = Number(str1);

let num2 = Number(str2);

let num3 = Number(str3);

console.log(num1); //0

console.log(num2); //123

console.log(num3); //NaN
```

Array:

JavaScript arrays are used to store multiple values in a single variable. An array is a special variable, which can hold more than one value at a time.

Creating an Array

Using an array literal is the easiest way to create a JavaScript Array.

Syntax:

```
var array_name = [item1, item2, ...];
```

Example

```
var cars = ["Saab", "Volvo", "BMW"];
```

Example

```
var cars=[
  "Saab",
  "Volvo",
  "BMW"
];
```

Using the JavaScript Keyword new

The following example also creates an Array, and assigns values to it:

Example

```
var cars = new Array("Saab", "Volvo", "BMW");
```

Access the Elements of an Array

You access an array element by referring to the **index number**.

This statement accesses the value of the first element in `cars`:

```
var name = cars[0];
```

Array indexes start with 0.

[0] is the first element. [1] is the second element.

Changing an Array Element

This statement changes the value of the first element in `cars`:

```
cars[0] = "Opel";
```

Example

```
var cars=["Saab", "Volvo", "BMW"];
cars[0]= "Opel";
document.getElementById("demo").innerHTML = cars[0];
```

Access the Full Array

With JavaScript, the full array can be accessed by referring to the array name:

Example

```
var cars=["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = cars;
```

Java Script Statements:

JavaScript statements are the commands to tell the browser to what action to perform. Statements are separated by semicolon (;).

Example of JavaScript statement:

```
document.getElementById("demo").innerHTML = "Welcome";
```

Script Statements

Sr.No.	**Statement**	**Description**
1.	switch case	A block of statements in which execution of code depends upon different cases. The interpreter checks each case against the value of the expression until a match is found. If nothing matches, a **default** condition will be used.
2.	If else	The **if** statement is the fundamental control statement that allows JavaScript to make decisions and execute statements conditionally.
3.	While	The purpose of a while loop is to execute a statement or code block repeatedly as long as expression is true. Once expression becomes false, the loop will be exited.
4.	do while	Block of statements that are executed at least once and continues to be executed while condition is true.
5.	for	Same as while but initialization, condition and increment/decrement is done in the same line.

6.	for in	This loop is used to loop through an object's properties.
7.	continue	The continue statement tells the interpreter to immediately start the next iteration of the loop and skip remaining code block.
8.	break	The break statement is used to exit a loop early, breaking out of the enclosing curly braces.
9.	function	A function is a group of reusable code which can be called anywhere in your programme. The keyword function is used to declare a function.
10.	return	Return statement is used to return a value from a function.
11.	var	Used to declare a variable.
12.	try	A block of statements on which error handling is implemented.
13.	catch	A block of statements that are executed when an error occur.
14.	throw	Used to throw an error.

Table 2.1 Script Statements

if statement

The **if** statement is the fundamental control statement that allows JavaScript to make decisions and execute statements conditionally.

Syntax

The syntax for a basic if statement is as follows −

```
if (expression) {
  Statement(s) to be executed if expression is true
}
```

Here a JavaScript expression is evaluated. If the resulting value is true, the given statement(s) are executed. If the expression is false, then no statement would be not executed. Most of the times, you will use comparison operators while making decisions.

```
<html>
  <body>
```

```
    <script type = "text/javascript">
      <!--
        var age = 20;

        if( age > 18 ) {
          document.write("<b>Qualifies for driving</b>");
        }
      //-->
    </script>
    <p>Set the variable to different value and then try...</p>
  </body>
</html>
```

Output

Qualifies for driving
Set the variable to different value and then try...

if...else statement

The **'if...else'** statement is the next form of control statement that allows JavaScript to execute statements in a more controlled way.

Syntax

```
if (expression) {
  Statement(s) to be executed if expression is true
} else {
  Statement(s) to be executed if expression is false
}
```

```
<html>
  <body>
    <script type = "text/javascript">
      <!--
        var age = 15;

        if( age > 18 ) {
          document.write("<b>Qualifies for driving</b>");
        } else {
          document.write("<b>Does not qualify for driving</b>");
        }
      //-->
    </script>
    <p>Set the variable to different value and then try...</p>
  </body>
</html>
```

Output

Does not qualify for driving
Set the variable to different value and then try...

if...else if... statement

The **if...else if...** statement is an advanced form of **if…else** that allows JavaScript to make a correct decision out of several conditions.

Syntax

The syntax of an if-else-if statement is as follows −

```
if (expression 1) {
  Statement(s) to be executed if expression 1 is true
} else if (expression 2) {
  Statement(s) to be executed if expression 2 is true
} else if (expression 3) {
  Statement(s) to be executed if expression 3 is true
} else {
  Statement(s) to be executed if no expression is true
}
```

There is nothing special about this code. It is just a series of **if** statements, where each **if** is a part of the **else** clause of the previous statement. Statement(s) are executed based on the true condition, if none of the conditions is true, then the **else** block is executed.

```
<html>
  <body>
    <script type = "text/javascript">
      <!--
        var book = "maths";
        if( book == "history" ) {
          document.write("<b>History Book</b>");
        } else if( book == "maths" ) {
          document.write("<b>Maths Book</b>");
        } else if( book == "economics" ) {
          document.write("<b>Economics Book</b>");
        } else {
          document.write("<b>Unknown Book</b>");
        }
      //-->
    </script>
    <p>Set the variable to different value and then try...</p>
  </body>
<html>
```

Switch statement

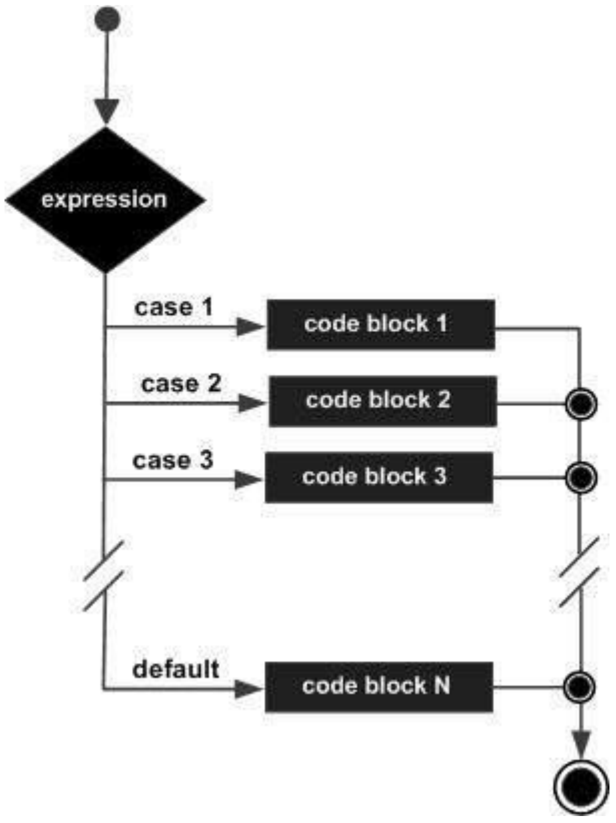


Figure 2.1 Flow Chart for Switch Statement

Syntax

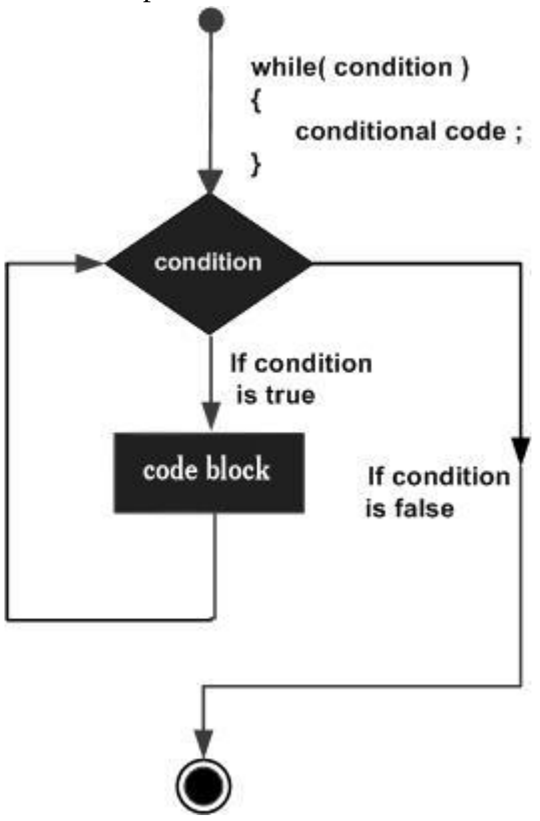
The objective of a **switch** statement is to give an expression to evaluate and several different statements to execute based on the value of the expression. The interpreter checks each **case** against the value of the expression until a match is found. If nothing matches, a **default** condition will be used.

```
switch (expression) {
  case condition 1: statement(s)
  break;

  case condition 2: statement(s)
  break;
  ...

  case condition n: statement(s)
  break;

  default: statement(s)
}
```

Example:

```
<html>
  <body>
    <script type = "text/javascript">
      <!--
        var grade = 'A';
        document.write("Entering switch block<br />");
        switch (grade) {
          case 'A': document.write("Good job<br />");
          break;

          case 'B': document.write("Pretty good<br />");
          break;

          case 'C': document.write("Passed<br />");
          break;

          case 'D': document.write("Not so good<br />");
          break;

          case 'F': document.write("Failed<br />");
          break;

          default:  document.write("Unknown grade<br />")
        }
        document.write("Exiting switch block");
      //-->
    </script>
    <p>Set the variable to different value and then try...</p>
  </body>
</html>
```

While Loop



Figurev2.2 Flow Chart for While loo

Syntax

The syntax of **while loop** in JavaScript is as follows −

```
while (expression) {
  Statement(s) to be executed if expression is true
}
```

```
<html>
  <body>

    <script type = "text/javascript">
      <!--
        var count = 0;
        document.write("Starting Loop ");

        while (count < 10) {
          document.write("Current Count : " + count + "<br />");
          count++;
        }

        document.write("Loop stopped!");
      //-->
    </script>
```

```
      <p>Set the variable to different value and then try...</p>
   </body>
</html>
```

The do...while Loop

The **do...while** loop is similar to the **while** loop except that the condition check happens at the end of the loop. This means that the loop will always be executed at least once, even if the condition is **false**.

Syntax

The syntax for **do-while** loop in JavaScript is as follows −

```
do {
   Statement(s) to be executed;
} while (expression);
```

Example:

```
<html>
   <body>
      <script type = "text/javascript">
         <!--
            var count = 0;

            document.write("Starting Loop" + "<br />");
            do {
               document.write("Current Count : " + count + "<br />");
               count++;
            }

            while (count < 5);
            document.write ("Loop stopped!");
         //-->
      </script>
      <p>Set the variable to different value and then try...</p>
   </body>
</html>
```

JavaScript Functions

JavaScript functions are used to perform operations. We can call JavaScript function many times to reuse the code.

Advantage of JavaScript function

There are mainly two advantages of JavaScript functions.

1. **Code reusability**: We can call a function several times so it save coding.
2. **Less coding**: It makes our program compact. We don't need to write many lines of code each time to perform a common task.

JavaScript Function Syntax

```
function functionName([arg1, arg2, ...argN]){
 //code to be executed
}
```

Function Example

```
<script>
function msg(){
alert("hello! this is message");
}
</script>
<input type="button" onclick="msg()" value="call function"/>
```

JavaScript Function Arguments

```
<script>
function getcube(number){
alert(number*number*number);
}
</script>
<form>
<input type="button" value="click" onclick="getcube(4)"/>
</form>
```

Function with Return Value

```
    <script>
function getInfo(){
return "hello javatpoint! How r u?";
}
</script>
<script>
document.write(getInfo());
</script>
```

JavaScript Function Methods

Method	Description
apply()	It is used to call a function contains this value and a single array of arguments.
bind()	It is used to create a new function.
call()	It is used to call a function contains this value and an argument list.
toString()	It returns the result in a form of a string.

Table 2.2 JavaScript Function Methods

JavaScript Function Object Example

```
<script>
var add=new Function("num1","num2","return num1+num2");
document.writeln(add(2,5));
</script>
```

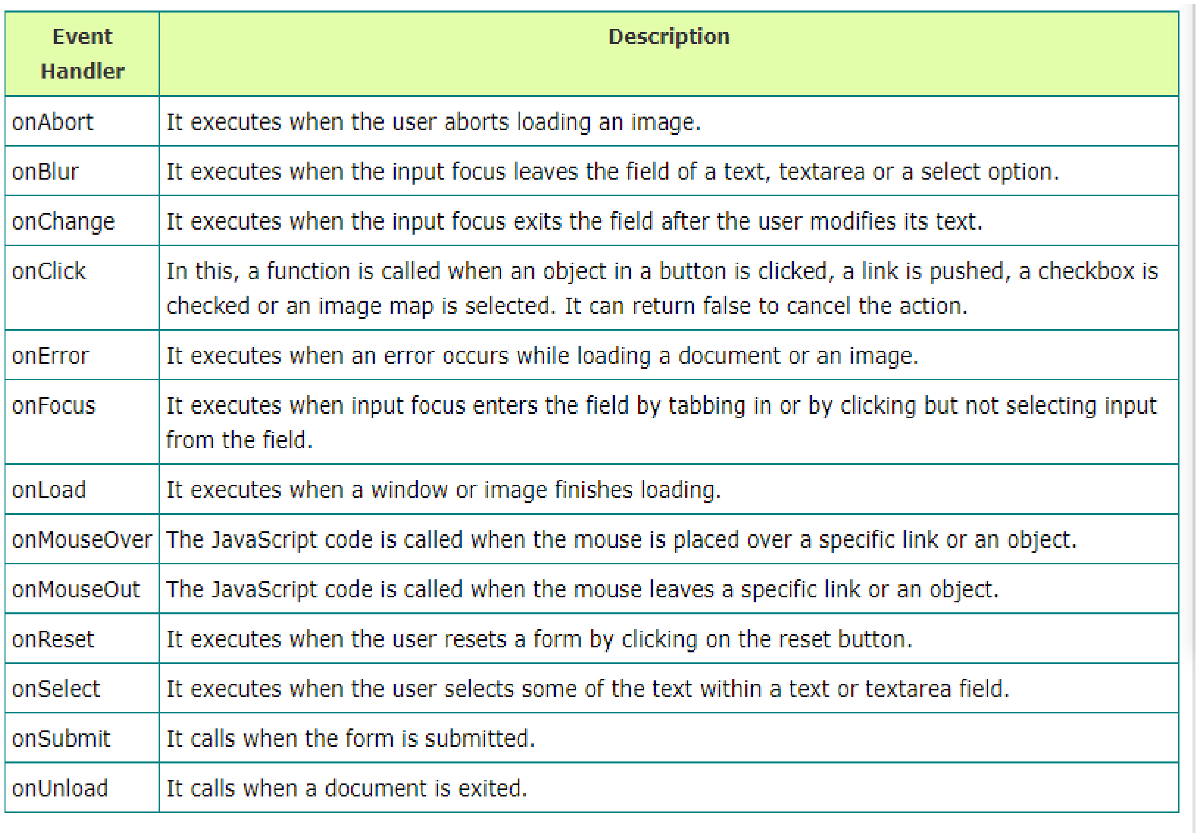
Introduction to Event Handling

- Event Handling is a software routine that processes actions, such as keystrokes and mouse movements.
- It is the receipt of an event at some event handler from an event producer and subsequent processes.

Functions of Event Handling

- Event Handling identifies where an event should be forwarded.
- It makes the forward event.
- It receives the forwarded event.
- It takes some kind of appropriate action in response, such as writing to a log, sending an error or recovery routine or sending a message.
- The event handler may ultimately forward the event to an event consumer.

Event Handlers



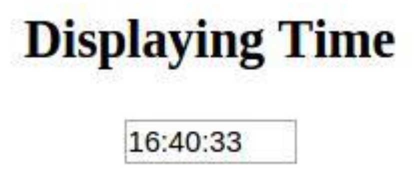
Event Handler	Description
onAbort	It executes when the user aborts loading an image.
onBlur	It executes when the input focus leaves the field of a text, textarea or a select option.
onChange	It executes when the input focus exits the field after the user modifies its text.
onClick	In this, a function is called when an object in a button is clicked, a link is pushed, a checkbox is checked or an image map is selected. It can return false to cancel the action.
onError	It executes when an error occurs while loading a document or an image.
onFocus	It executes when input focus enters the field by tabbing in or by clicking but not selecting input from the field.
onLoad	It executes when a window or image finishes loading.
onMouseOver	The JavaScript code is called when the mouse is placed over a specific link or an object.
onMouseOut	The JavaScript code is called when the mouse leaves a specific link or an object.
onReset	It executes when the user resets a form by clicking on the reset button.
onSelect	It executes when the user selects some of the text within a text or textarea field.
onSubmit	It calls when the form is submitted.
onUnload	It calls when a document is exited.

Table 2.3 Event Handlers

Example : Simple Program on onload() Event handler

```
<html>
    <head>
    <scripttype="text/javascript">
    functiontime()
    {
        vard=newDate();
        var ty = d.getHours() + ":"+d.getMinutes()+":"+d.getSeconds();
        document.frmty.timetxt.value=ty;
        setInterval("time()",1000)
    }
    </script>
    </head>
<bodyonload="time()">
    <center><h2>DisplayingTime</h2>
```

```
        <formname="frmty">
            <inputtype=textname=timetxtsize="8">
        </form>
    </center>
</body>
</html>
```


Displaying Time

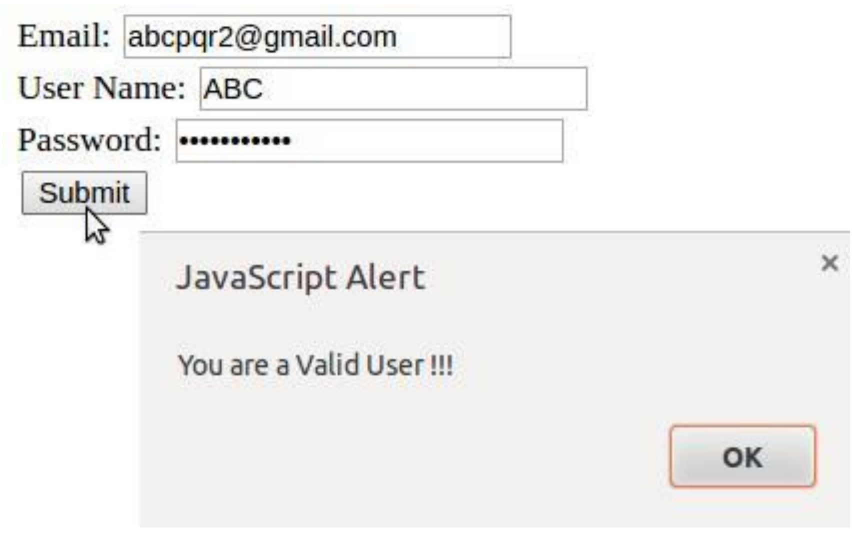
16:40:33

Example: Simple Program on onsubmit() & onfocus() Event handler

```
<html>
    <body>
        <script>
            functionvalidateform()
            {
                varuname=document.myform.name.value;
                varupassword=document.myform.password.value;
                if(uname==null||uname=="")
                {
                    alert("Namecannotbeleftblank");
                    returnfalse;
                }
                elseif(upassword.length<6)
                {
                    alert("Password must be at least 6 characters long.");
                    returnfalse;
                }
            }
            functionemailvalidation()
            {
                vara=document.myform.email.value
                if(a.indexOf("@")==-1)
                {
                    alert("Pleaseentervalidemailaddress")
                    document.myform.email.focus()
                }
            }
        </script>
    <body>
        <form name="myform" method="post" action="validpage.html" onsubmit="return
validateform()">
            Email: <inputtype="text"size="20" name="email" onblur="emailvalidation()"><br>
            UserName:<inputtype="text"name="name"><br>
            Password:<inputtype="password"name="password"><br>
            <inputtype="submit"value="Submit">
        </form>
    </body>
</html>
```

validpage.html **//Filename**

```
<html>
    <body>
        <scripttype="text/javascript">
            alert("YouareaValidUser!!!");
        </script>
    </body>
</html>
```



Email: abcpqr2@gmail.com
User Name: ABC
Password: •••••••••••

JavaScript Alert

You are a Valid User !!!

Working With Objects –

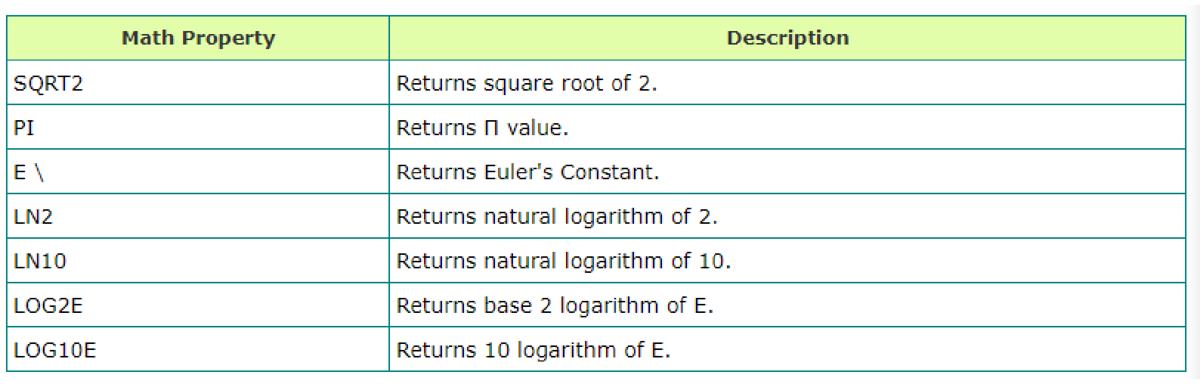
Built-in Objects

- Built-in objects are not related to any Window or DOM object model.
- These objects are used for simple data processing in the JavaScript.

1. Math Object

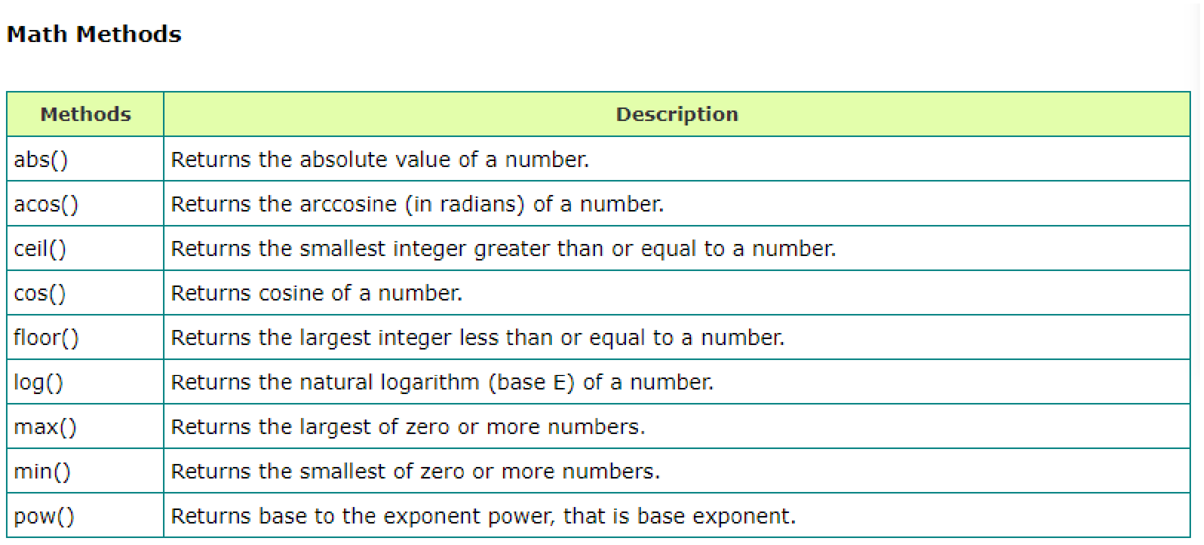
- Math object is a built-in static object.
- It is used for performing complex math operations.

Math Properties



Math Property	Description
SQRT2	Returns square root of 2.
PI	Returns Π value.
E \	Returns Euler's Constant.
LN2	Returns natural logarithm of 2.
LN10	Returns natural logarithm of 10.
LOG2E	Returns base 2 logarithm of E.
LOG10E	Returns 10 logarithm of E.

Table 3.4. Math Properties



Math Methods

Methods	Description
abs()	Returns the absolute value of a number.
acos()	Returns the arccosine (in radians) of a number.
ceil()	Returns the smallest integer greater than or equal to a number.
cos()	Returns cosine of a number.
floor()	Returns the largest integer less than or equal to a number.
log()	Returns the natural logarithm (base E) of a number.
max()	Returns the largest of zero or more numbers.
min()	Returns the smallest of zero or more numbers.
pow()	Returns base to the exponent power, that is base exponent.

Table 3.5. Math Properties

Example: Simple Program on Math Object Methods

```
<html>
    <head>
        <title>JavaScriptMathObjectMethods</title>
    </head>
    <body>
        <scripttype="text/javascript">

            varvalue=Math.abs(20);
            document.write("ABSTestValue:"+value+"<br>");

            varvalue=Math.acos(-1);
            document.write("ACOSTestValue:"+value+"<br>");

            varvalue=Math.asin(1);
            document.write("ASINTestValue:"+value+"<br>");

            varvalue=Math.atan(.5);
            document.write("ATANTestValue:"+value+"<br>");
        </script>
    </body>
</html>
```

Output

```
ABSTestValue:20
ACOSTestValue:3.141592653589793
ASINTestValue:1.5707963267948966
ATAN Test Value : 0.4636476090008061
```

Date, Math, String, Window, Calendar:

Creating Date Objects

Date objects are created with the `new Date()` constructor.

There are **4 ways** to create a new date object:

new Date()
new Date(*year, month, day, hours, minutes, seconds, milliseconds*)
new Date(*milliseconds*)
new Date(*date string*)

new Date()

`new Date()` creates a new date object with the **current date and time**:

Example

```
var d = new Date();
```

new Date(*year*, *month*, ...)

`new Date(`*year, month, ...*`)` creates a new date object with a **specified date and time**.

7 numbers specify year, month, day, hour, minute, second, and millisecond (in that order):

Example

```
var d = new Date(2018, 11, 24, 10, 33, 30, 0);
```

Example

```
var d = new Date(2018, 11, 24, 10, 33, 30);
```

Example

```
var d = new Date(2018, 11, 24, 10, 33);
```

Example

```
var d = new Date(2018, 11, 24, 10);
```

Example

```
var d = new Date(2018, 11, 24);
```

Example

```
var d = new Date(2018);
```

new Date(*dateString*)

`new Date(dateString)` creates a new date object from a **date string**:

Example

```
var d = new Date("October 13, 2014 11:13:00");
```

new Date(*milliseconds*)

`new Date(`*milliseconds*`)` creates a new date object as **zero time plus milliseconds**:

Example

var d = new Date(0);

Connecting To Search Engines.

JavaScript web apps in three main phases:

1. Crawling
2. Rendering
3. Indexing

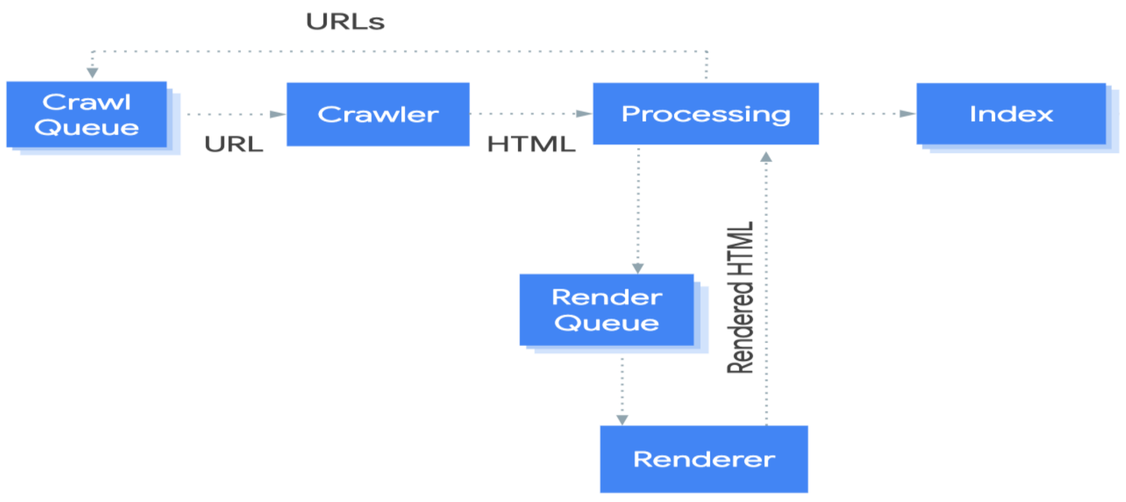


Figure 2.3 Search Engines Architecture

When Googlebot fetches a URL from the crawling queue by making an HTTP request it first checks if you allow crawling. Googlebot reads the robots.txt file. If it marks the URL as disallowed, then Googlebot skips making an HTTP request to this URL and skips the URL.

Googlebot then parses the response for other URLs in the `href` attribute of HTML links and adds the URLs to the crawl queue. To prevent link discovery, use the nofollow mechanism.

Crawling a URL and parsing the HTML response works well for classical websites or server-side rendered pages where the HTML in the HTTP response contains all content. Some JavaScript sites may use the app shell model where the initial HTML does not contain the actual content and Googlebot needs to execute JavaScript before being able to see the actual page content that JavaScript generates.

Googlebot queues all pages for rendering, unless a robots meta tag or header tells Googlebot not to index the page. The page may stay on this queue for a few seconds, but it can take longer than that. Once Googlebot's resources allow, a headless Chromium renders the page and executes the JavaScript. Googlebot parses the rendered HTML for links again and queues the URLs it finds for crawling. Googlebot also uses the rendered HTML to index the page.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT- III Advanced Java – SBS1301

Servlets

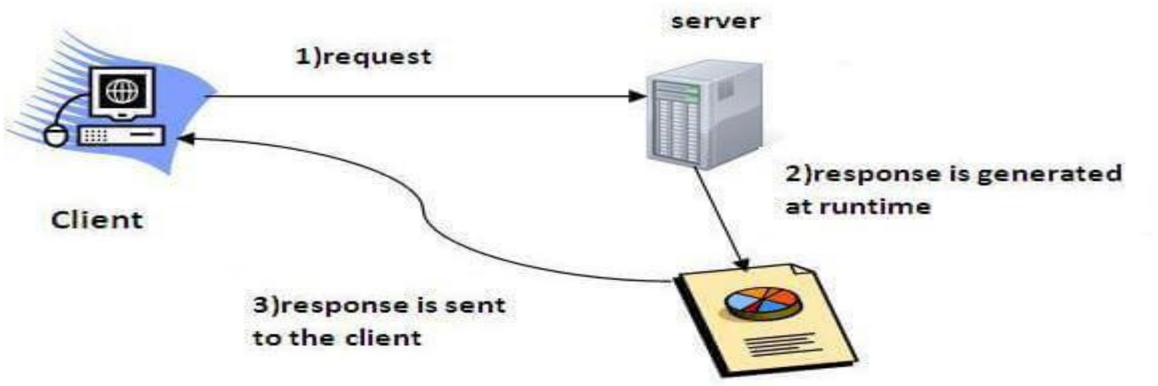
Servlet technology is used to create a web application (resides at server side and generates a dynamic web page).

Servlet technology is robust and scalable because of java language. Before Servlet, CGI (Common Gateway Interface) scripting language was common as a server-side programming language. However, there were many disadvantages to this technology. We have discussed these disadvantages below.

There are many interfaces and classes in the Servlet API such as Servlet, GenericServlet, HttpServlet, ServletRequest, ServletResponse, etc.

Servlet can be described in many ways, depending on the context.

- Servlet is a technology which is used to create a web application.
- Servlet is an API that provides many interfaces and classes including documentation.
- Servlet is an interface that must be implemented for creating any Servlet.
- Servlet is a class that extends the capabilities of the servers and responds to the incoming requests. It can respond to any requests.
- Servlet is a web component that is deployed on the server to create a dynamic web page.


server
1)request
2)response is generated at runtime
Client
3)response is sent to the client

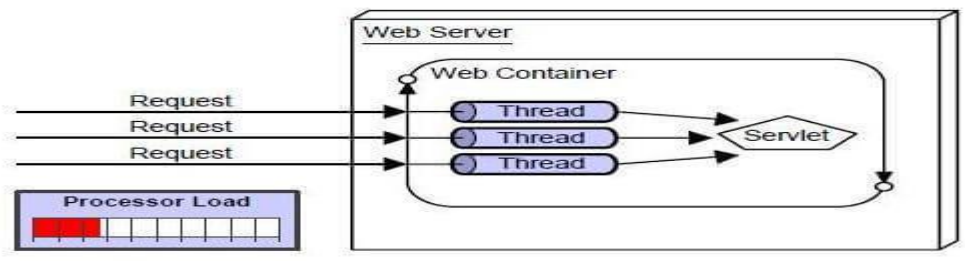


Figure 3.1 Servlet Technology

The advantages of Servlet are as follows:

1. Better performance: because it creates a thread for each request, not process.
2. Portability: because it uses Java language.
3. Robust: JVM manages Servlets, so we don't need to worry about the memory leak, garbage collection, etc.
4. Secure: because it uses java language.

Life Cycle of a Servlet:

The web container maintains the life cycle of a servlet instance. Let's see the life cycle of the servlet:

1. Servlet class is loaded.
2. Servlet instance is created.
3. init method is invoked.
4. service method is invoked.
5. destroy method is invoked.

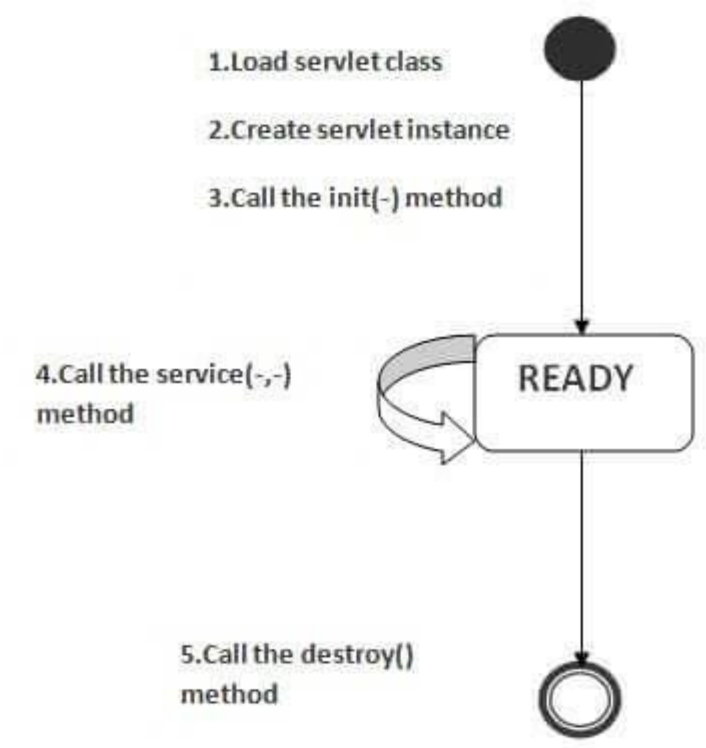

1.Load servlet class
2.Create servlet instance
3.Call the init(-) method
4.Call the service(-,-) method
READY
5.Call the destroy() method

Figure 3.2 Life Cycle of a Servlet

1) Servlet class is loaded

The classloader is responsible to load the servlet class. The servlet class is loaded when the first request for the servlet is received by the web container.

2) Servlet instance is created

The web container creates the instance of a servlet after loading the servlet class. The servlet instance is created only once in the servlet life cycle.

3) init method is invoked

The web container calls the init method only once after creating the servlet instance.

The init method is used to initialize the servlet.

It is the life cycle method of the javax.servlet.Servlet interface.

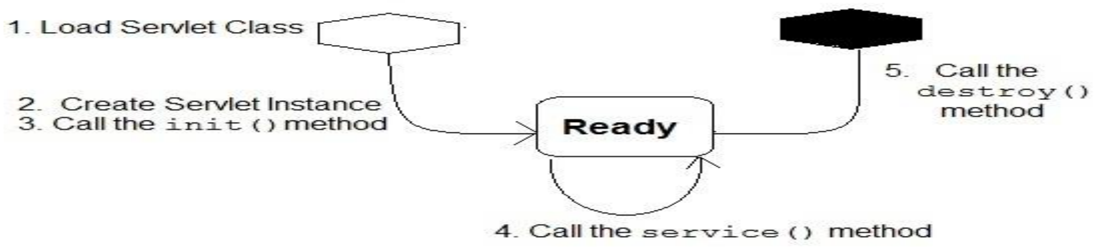
1. public void init(ServletConfig config) throws ServletException

The web container calls the service method each time when request for the servlet is received. If servlet is not initialized, it follows the first three steps as described above then calls the service method. If servlet is initialized, it calls the service method. Notice that servlet is initialized only once. The syntax of the service method of the Servlet interface is given below:

1. public void service(ServletRequest request, ServletResponse response)
2. throws ServletException, IOException .

5) destroy method is invoked

The web container calls the destroy method before removing the servlet instance from the service. It gives the servlet an opportunity to clean up any resource for example memory, thread


1. Load Servlet Class
2. Create Servlet Instance
3. Call the init() method
Ready
5. Call the destroy() method
4. Call the service() method

public void destroy()

HTTP is a protocol which allows the fetching of resources, such as HTML documents. It is the foundation of any data exchange on the Web and it is a client-server protocol, which

HTTP protocol

HTTP is a protocol which allows the fetching of resources, such as HTML documents. It is the foundation of any data exchange on the Web and it is a client-server protocol, which means requests are initiated by the recipient, usually the Web browser. A complete document is reconstructed from the different sub-documents fetched, for instance text, layout description, images, videos, scripts, and more.

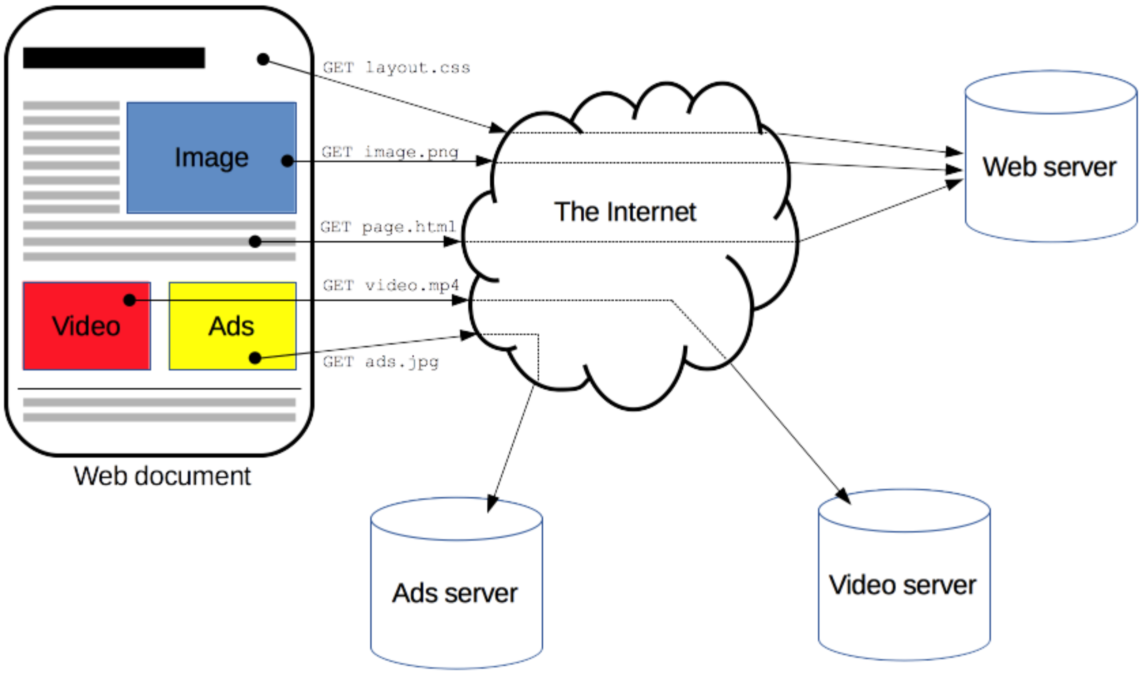


Figure 3.4 HTTP protocol

Clients and servers communicate by exchanging individual messages (as opposed to a stream of data). The messages sent by the client, usually a Web browser, are called requests and the messages sent by the server as an answer are called responses.

Basic Features

There are three basic features that make HTTP a simple but powerful protocol:

- **HTTP is connectionless:** The HTTP client, i.e., a browser initiates an HTTP request and after a request is made, the client waits for the response. The server processes the request and sends a response back after which client disconnect the connection. So client and server knows about each other during current request and response only. Further requests are made on new connection like client and server are new to each other.
- **HTTP is media independent:** It means, any type of data can be sent by HTTP as long as both the client and the server know how to handle the data content. It is required for the client as well as the server to specify the content type using appropriate MIME-type.
- **HTTP is stateless:** As mentioned above, HTTP is connectionless and it is a direct result of HTTP being a stateless protocol. The server and client are aware of each other only during a current request. Afterwards, both of them forget about each

other. Due to this nature of the protocol, neither the client nor the browser can retain information between different requests across the web pages.

HTTP Servlet

If you creating Http Servlet you must extend javax.servlet.http.HttpServlet class, which is an abstract class. Unlike Generic Servlet, the HTTP Servlet doesn't override the service() method. Instead it overrides one or more of the following methods. It must override at least one method from the list below:

- **doGet()** – This method is called by servlet service method to handle the HTTP GET request from client. The Get method is used for getting information from the server
- **doPost()** – Used for posting information to the Server
- **doPut()** – This method is similar to doPost method but unlike doPost method where we send information to the server, this method sends file to the server, this is similar to the FTP operation from client to server
- **doDelete()** – allows a client to delete a document, webpage or information from the server
- **init() and destroy()** – Used for managing resources that are held for the life of the servlet
- **getServletInfo()** – Returns information about the servlet, such as author, version, and copyright.

In Http Servlet there is no need to override the service() method as this method dispatches the Http Requests to the correct method handler, for example if it receives HTTP GET Request it dispatches the request to the doGet() method.

Basic Architecture

The following diagram shows a very basic architecture of a web application and depicts where HTTP sits:

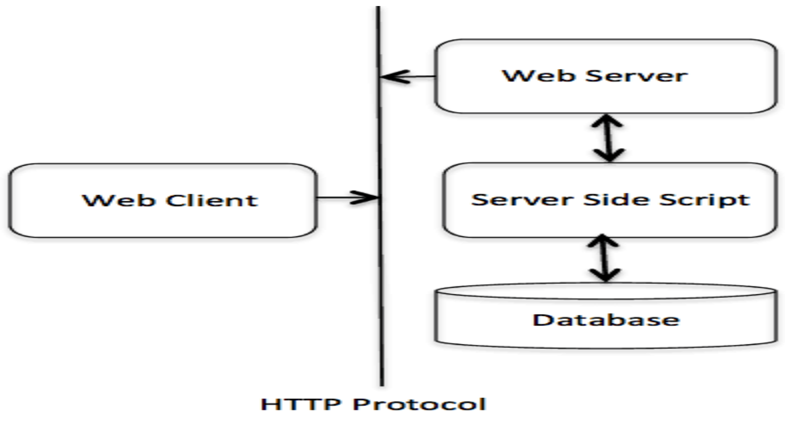


Figure 3.4 HTTP Protocol

The HTTP protocol is a request/response protocol based on the client/server based architecture where web browsers, robots and search engines, etc. act like HTTP clients, and the Web server acts as a server.

Client

The HTTP client sends a request to the server in the form of a request method, URI, and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content over a TCP/IP connection.

Server

The HTTP server responds with a status line, including the message's protocol version and a success or error code, followed by a MIME-like message containing server information, entity meta information, and possible entity-body content.

The Servlet API:

The javax.servlet and javax.servlet.http packages represent interfaces and classes for servlet api.

The **javax.servlet** package contains many interfaces and classes that are used by the servlet or web container. These are not specific to any protocol.

The **javax.servlet.http** package contains interfaces and classes that are responsible for http requests only.

Interfaces in javax.servlet package

There are many interfaces in javax.servlet package. They are as follows:

1. Servlet
2. ServletRequest
3. ServletResponse
4. RequestDispatcher
5. ServletConfig
6. ServletContext
7. SingleThreadModel
8. Filter
9. FilterConfig
10. FilterChain
11. ServletRequestListener
12. ServletRequestAttributeListener
13. ServletContextListener
14. ServletContextAttributeListener

Classes in javax.servlet package

There are many classes in javax.servlet package. They are as follows:

1. GenericServlet
2. ServletInputStream
3. ServletOutputStream
4. ServletRequestWrapper
5. ServletResponseWrapper
6. ServletRequestEvent
7. ServletContextEvent
8. ServletRequestAttributeEvent
9. ServletContextAttributeEvent
10. ServletException
11. UnavailableException

Interfaces in javax.servlet.http package

There are many interfaces in javax.servlet.http package. They are as follows:

1. HttpServletRequest
2. HttpServletResponse
3. HttpSession
4. HttpSessionListener
5. HttpSessionAttributeListener
6. HttpSessionBindingListener
7. HttpSessionActivationListener
8. HttpSessionContext (deprecated now)

Classes in javax.servlet.http package

There are many classes in javax.servlet.http package. They are as follows:

1. HttpServlet
2. Cookie
3. HttpServletRequestWrapper
4. HttpServletResponseWrapper
5. HttpSessionEvent
6. HttpSessionBindingEvent
7. HttpUtils (deprecated now)

Generic and Http Servlet:

GenericServlet class implements Servlet, ServletConfig and Serializable interfaces. It provides the implementation of all the methods of these interfaces except the service method.

GenericServlet class can handle any type of request so it is protocol-independent.

Methods of GenericServlet class

There are many methods in GenericServlet class. They are as follows:

1. **public void init(ServletConfig config)** is used to initialize the servlet.
2. **public abstract void service(ServletRequest request, ServletResponse response)** provides service for the incoming request. It is invoked at each time when user requests for a servlet.
3. **public void destroy()** is invoked only once throughout the life cycle and indicates that servlet is being destroyed.
4. **public ServletConfig getServletConfig()** returns the object of ServletConfig.
5. **public String getServletInfo()** returns information about servlet such as writer, copyright, version etc.
6. **public void init()** it is a convenient method for the servlet programmers, now there is no need to call super.init(config)
7. **public ServletContext getServletContext()** returns the object of ServletContext.
8. **public String getInitParameter(String name)** returns the parameter value for the given parameter name.
9. **public Enumeration getInitParameterNames()** returns all the parameters defined in the web.xml file.
10. **public String getServletName()** returns the name of the servlet object.
11. **public void log(String msg)** writes the given message in the servlet log file.
12. **public void log(String msg,Throwable t)** writes the explanatory message in the servlet log file and a stack trace.

Following figure shows the hierarchy of Servlet vs GenericServlet vs HttpServlet and to know from where HttpServlet comes.

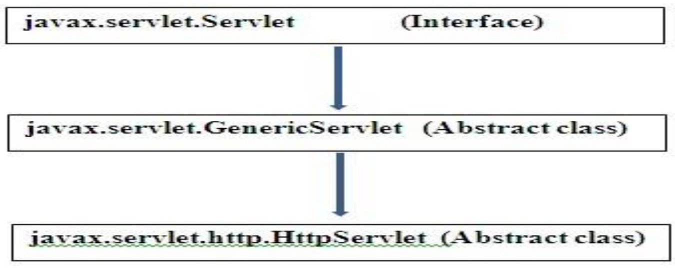

javax.servlet.Servlet (Interface)
javax.servlet.GenericServlet (Abstract class)
javax.servlet.http.HttpServlet (Abstract class)

Figure 4.5 Servlet vs GenericServlet vs HttpServlet

ServletConfig and ServletContext:

An object of ServletConfig is created by the web container for each servlet. This object can be used to get configuration information from web.xml file. If the configuration information is modified from the web.xml file, we don't need to change the servlet. So it is easier to manage the web application if any specific content is modified from time to time.

Advantage of ServletConfig

The core advantage of ServletConfig is that you don't need to edit the servlet file if information is modified from the web.xml file.

Methods of ServletConfig interface

1. **public String getInitParameter(String name):**Returns the parameter value for the specified parameter name.
2. **public Enumeration getInitParameterNames():**Returns an enumeration of all the initialization parameter names.
3. **public String getServletName():**Returns the name of the servlet.
4. **public ServletContext getServletContext():**Returns an object of ServletContext.

How to get the object of ServletConfig

1. **getServletConfig() method** of Servlet interface returns the object of ServletConfig.

Syntax of getServletConfig() method

public ServletConfig getServletConfig();

Example of getServletConfig() method

ServletConfig config=getServletConfig();

//Now we can call the methods of ServletConfig interface

Syntax to provide the initialization parameter for a servlet

The init-param sub-element of servlet is used to specify the initialization parameter for a servlet.

```
<web-app>
    <servlet>
      ......

       <init-param>
         <param-name>parametername</param-name>
         <param-value>parametervalue</param-value>
       </init-param>
       ......
     </servlet>
    </web-app>
```

Example of ServletConfig to get initialization parameter

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

public class DemoServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    ServletConfig config=getServletConfig();
    String driver=config.getInitParameter("driver");
    out.print("Driver is: "+driver);

    out.close();
    }

}
```

web.xml

```
<web-app>

<servlet>
<servlet-name>DemoServlet</servlet-name>
<servlet-class>DemoServlet</servlet-class>

<init-param>
<param-name>driver</param-name>
<param-value>sun.jdbc.odbc.JdbcOdbcDriver</param-value>
</init-param>

</servlet>

<servlet-mapping>
<servlet-name>DemoServlet</servlet-name>
<url-pattern>/servlet1</url-pattern>
</servlet-mapping>

</web-app>
```

ServletContext::

object of ServletContext is created by the web container at time of deploying the project. This object can be used to get configuration information from web.xml file. There is only one ServletContext object per web application. If any information is shared to many servlet, it is better to provide it from the web.xml file using the **<context-param>** element.

Advantage of ServletContext

Easy to maintain if any information is shared to all the servlet, it is better to make it available for all the servlet. We provide this information from the web.xml file, so if the information is changed, we don't need to modify the servlet. Thus it removes maintenance problem.

Usage of ServletContext Interface

There can be a lot of usage of ServletContext object. Some of them are as follows:

1. The object of ServletContext provides an interface between the container and servlet.
2. The ServletContext object can be used to get configuration information from the web.xml file.
3. The ServletContext object can be used to set, get or remove attribute from the web.xml file.
4. The ServletContext object can be used to provide inter-application communication.

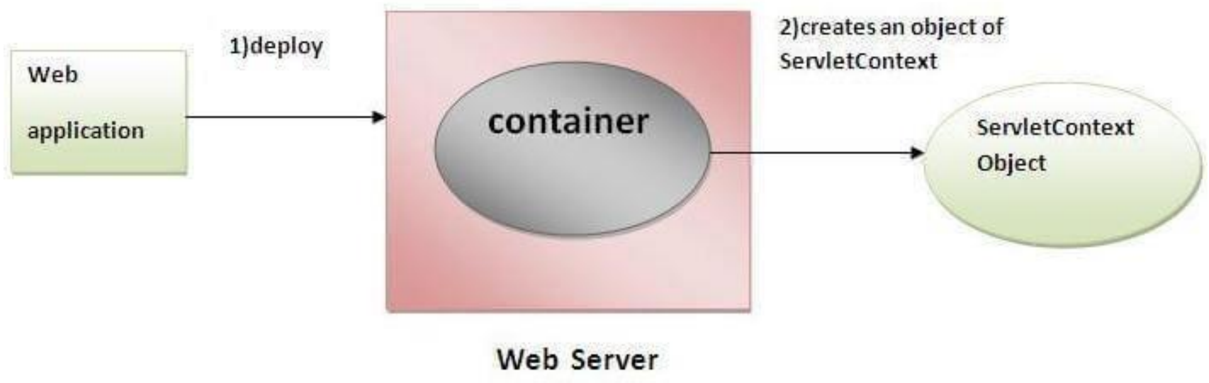


Figure 5.6 ServletContext Interface

Commonly used methods of ServletContext interface

There is given some commonly used methods of ServletContext interface.

1. **public String getInitParameter(String name):**Returns the parameter value for the specified parameter name.
2. **public Enumeration getInitParameterNames():**Returns the names of the context's initialization parameters.

3. **public void setAttribute(String name,Object object):**sets the given object in the application scope.
4. **public Object getAttribute(String name):**Returns the attribute for the specified name.
5. **public Enumeration getInitParameterNames():**Returns the names of the context's initialization parameters as an Enumeration of String objects.
6. **public void removeAttribute(String name):**Removes the attribute with the given name from the servlet context.

How to get the object of ServletContext interface

1. **getServletContext() method** of ServletConfig interface returns the object of ServletContext.
2. **getServletContext() method** of GenericServlet class returns the object of ServletContext.

Syntax of getServletContext() method
public ServletContext getServletContext()

Example of getServletContext() method
//We can get the ServletContext object from ServletConfig object
ServletContext application=getServletConfig().getServletContext();

//Another convenient way to get the ServletContext object
ServletContext application=getServletContext();

Difference between ServletConfig and ServletContext in Java Servlet

ServletConfig and **ServletContext**, both are objects created at the time of servlet initialization and used to provide some initial parameters or configuration information to the servlet. But, the difference lies in the fact that information shared by ServletConfig is for a specific servlet, while information shared by ServletContext is available for all servlets in the web application.

Handling HTTP Requests and Responses:

HTTP Requests

The request sent by the computer to a web server, contains all sorts of potentially interesting information; it is known as HTTP requests.

The HTTP client sends the request to the server in the form of request message which includes following information:

- The Request-line

- The analysis of source IP address, proxy and port
- The analysis of destination IP address, protocol, port and host
- The Requested URI (Uniform Resource Identifier)
- The Request method and Content
- The User-Agent header
- The Connection control header
- The Cache control header

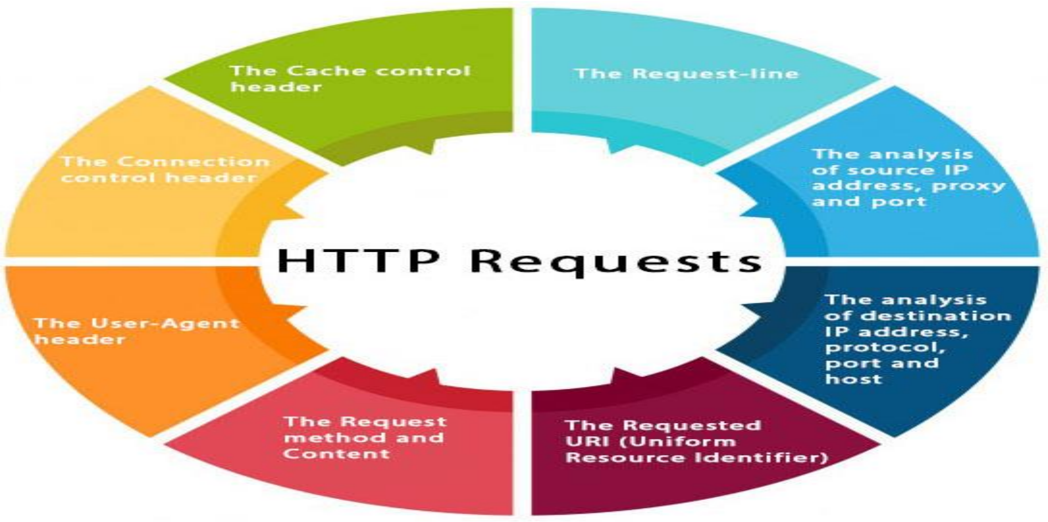
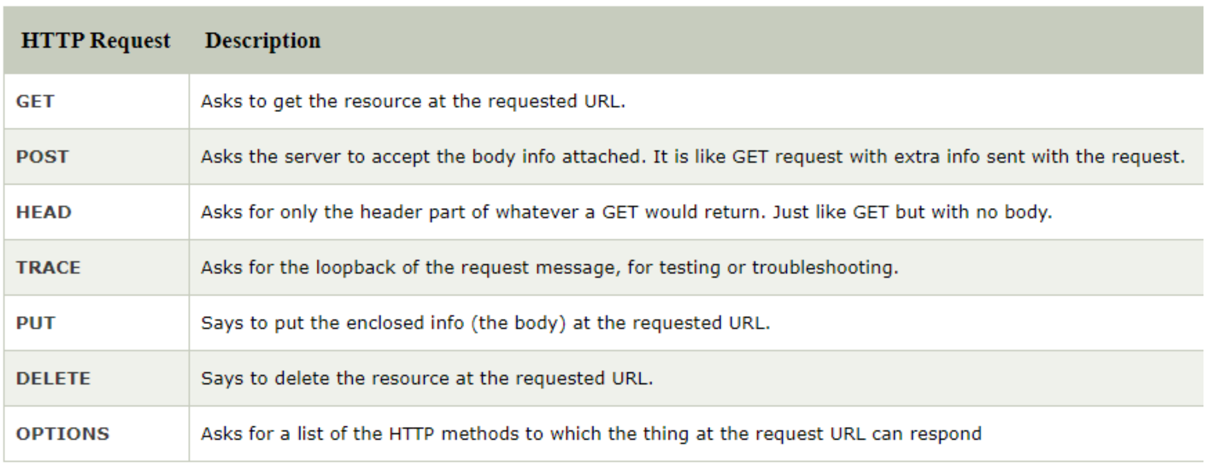

HTTP Requests
The Request-line
The analysis of source IP address, proxy and port
The analysis of destination IP address, protocol, port and host
The Requested URI (Uniform Resource Identifier)
The Request method and Content
The User-Agent header
The Connection control header
The Cache control header

Figure 3.7 HTTP Request

The HTTP request method indicates the method to be performed on the resource identified by the **Requested URI (Uniform Resource Identifier)**. This method is case-sensitive and should be used in uppercase.

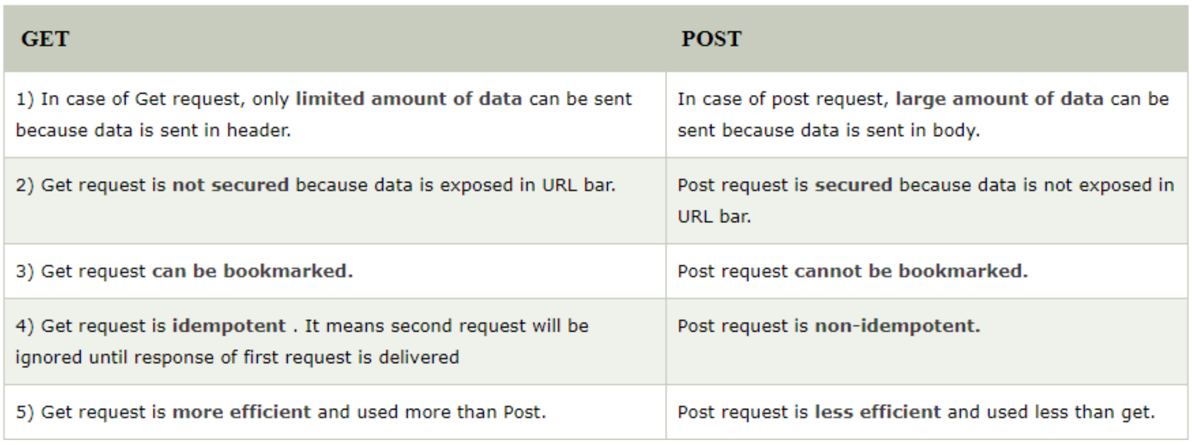
The HTTP request methods are



HTTP Request	Description
GET	Asks to get the resource at the requested URL.
POST	Asks the server to accept the body info attached. It is like GET request with extra info sent with the request.
HEAD	Asks for only the header part of whatever a GET would return. Just like GET but with no body.
TRACE	Asks for the loopback of the request message, for testing or troubleshooting.
PUT	Says to put the enclosed info (the body) at the requested URL.
DELETE	Says to delete the resource at the requested URL.
OPTIONS	Asks for a list of the HTTP methods to which the thing at the request URL can respond

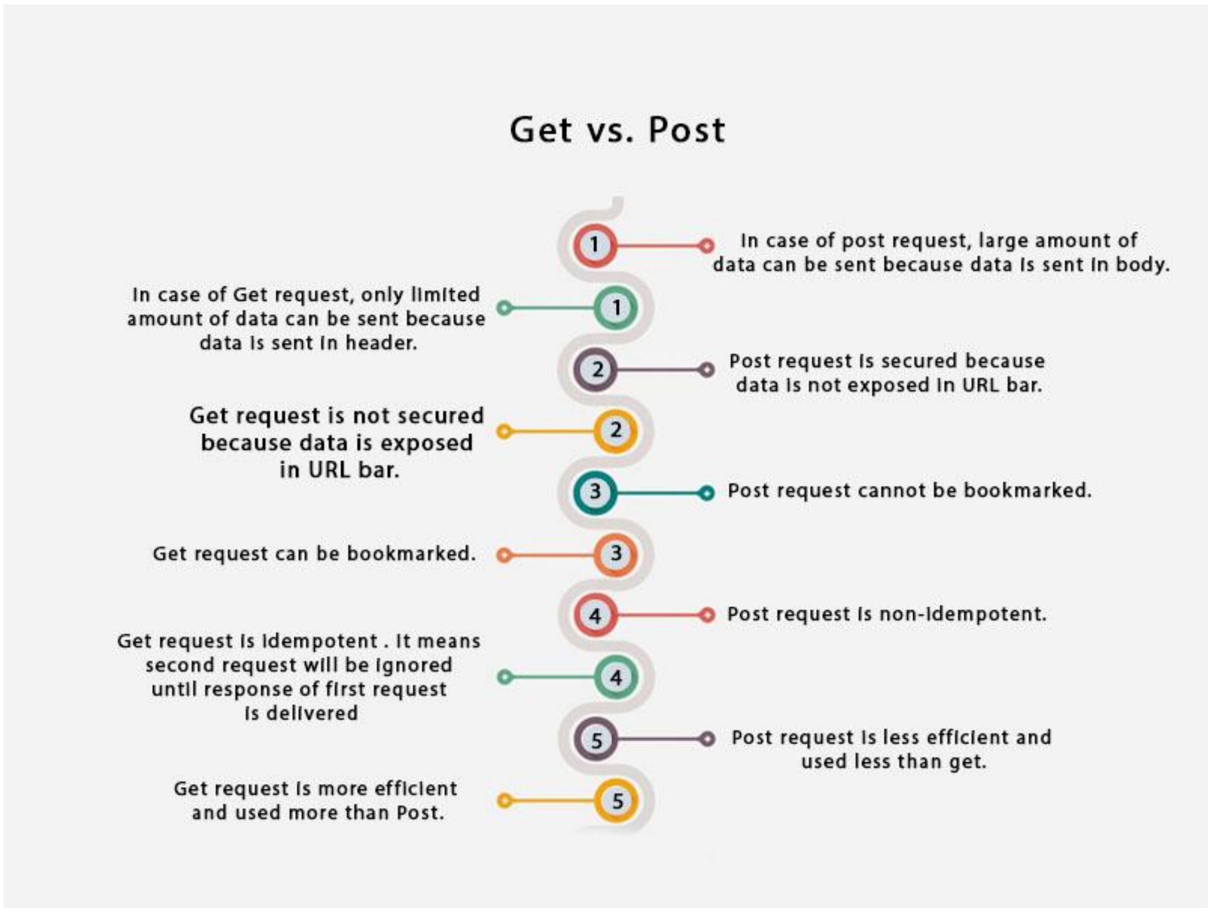
Table 3.1 HTTP Request Methods

Get vs. Post
Differences between the Get and Post request



GET	POST
1) In case of Get request, only **limited amount of data** can be sent because data is sent in header.	In case of post request, **large amount of data** can be sent because data is sent in body.
2) Get request is **not secured** because data is exposed in URL bar.	Post request is **secured** because data is not exposed in URL bar.
3) Get request **can be bookmarked.**	Post request **cannot be bookmarked.**
4) Get request is **idempotent** . It means second request will be ignored until response of first request is delivered	Post request is **non-idempotent.**
5) Get request is **more efficient** and used more than Post.	Post request is **less efficient** and used less than get.

Table 3.2 Differences between the Get and Post request



Get vs. Post

In case of post request, large amount of data can be sent because data is sent in body.

In case of Get request, only limited amount of data can be sent because data is sent in header.

Post request is secured because data is not exposed in URL bar.

Get request is not secured because data is exposed in URL bar.

Post request cannot be bookmarked.

Get request can be bookmarked.

Post request is non-idempotent.

Get request is idempotent . It means second request will be ignored until response of first request is delivered

Post request is less efficient and used less than get.

Get request is more efficient and used more than Post.

Figure 3.8 GET vs POST

Some other features of GET requests are:

- It remains in the browser history
- It can be bookmarked
- It can be cached
- It have length restrictions
- It should never be used when dealing with sensitive data
- It should only be used for retrieving the data

Handling form data with get and post request:

There are two ways the browser client can send information to the web server.

- The GET Method
- The POST Method

Before the browser sends the information, it encodes it using a scheme called URL encoding. In this scheme, name/value pairs are joined with equal signs and different pairs are separated by the ampersand.

name1=value1&name2=value2&name3=value3

Spaces are removed and replaced with the + character and any other nonalphanumeric characters are replaced with a hexadecimal values. After the information is encoded it is sent to the server.

The GET Method

The GET method sends the encoded user information appended to the page request. The page and the encoded information are separated by the **?** character.

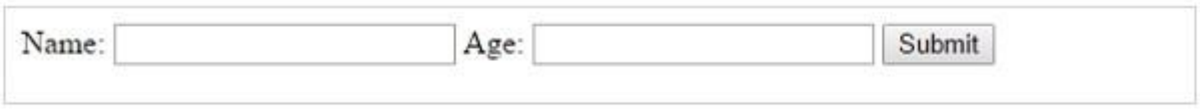
http://www.test.com/index.htm?name1=value1&name2=value2

- The GET method produces a long string that appears in your server logs, in the browser's Location: box.
- The GET method is restricted to send upto 1024 characters only.
- Never use GET method if you have password or other sensitive information to be sent to the server.
- GET can't be used to send binary data, like images or word documents, to the server.
- The data sent by GET method can be accessed using QUERY_STRING environment variable.
- The PHP provides **$_GET** associative array to access all the sent information using GET method.

Try out following example by putting the source code in test.php script.

```
<?php
   if( $_GET["name"] || $_GET["age"] ) {
      echo "Welcome ". $_GET['name']. "<br />";
      echo "You are ". $_GET['age']. " years old.";
      
      exit();
   }
?>
<html>
   <body>
   
      <form action = "<?php $_PHP_SELF ?>" method = "GET">
         Name: <input type = "text" name = "name" />
         Age: <input type = "text" name = "age" />
         <input type = "submit" />
      </form>
      
   </body>
</html>
```

It will produce the following result –



The POST Method

The POST method transfers information via HTTP headers. The information is encoded as described in case of GET method and put into a header called QUERY_STRING.

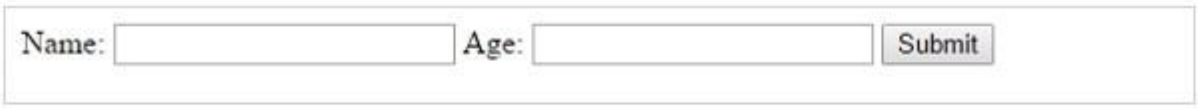
- The POST method does not have any restriction on data size to be sent.
- The POST method can be used to send ASCII as well as binary data.
- The data sent by POST method goes through HTTP header so security depends on HTTP protocol. By using Secure HTTP you can make sure that your information is secure.
- The PHP provides **$_POST** associative array to access all the sent information using POST method.

Try out following example by putting the source code in test.php script.

```
<?php
   if( $_POST["name"] || $_POST["age"] ) {
      if (preg_match("/[^A-Za-z'-]/",$_POST['name'] )) {
         die ("invalid name and name should be alpha");
      }
      echo "Welcome ". $_POST['name']. "<br />";
      echo "You are ". $_POST['age']. " years old.";
      
      exit();
   }
```

```
?>
<html>
   <body>

      <form action = "<?php $_PHP_SELF ?>" method = "POST">
         Name: <input type = "text" name = "name" />
         Age: <input type = "text" name = "age" />
         <input type = "submit" />
      </form>

   </body>
</html>
```

It will produce the following result



SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

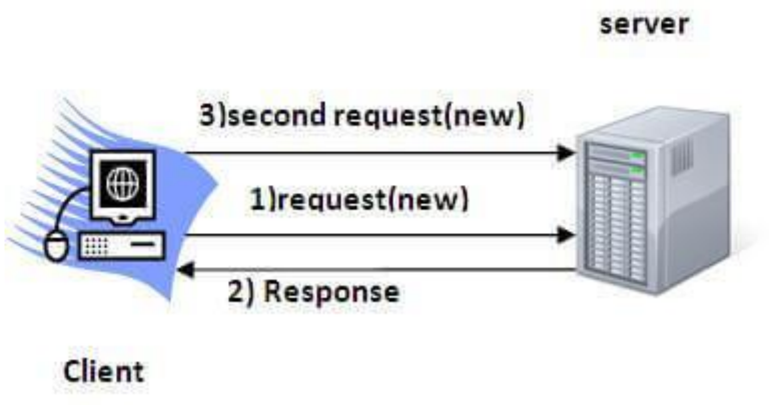
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT- 4 Advanced Java – SBS1301

Advanced Servlets

Approach to Session Tracking

Session simply means a particular interval of time.Session Tracking is a way to maintain state (data) of an user. It is also known as session management in servlet.Http protocol is a stateless so we need to maintain state using session tracking techniques. Each time user requests to the server, server treats the request as the new request. So we need to maintain the state of an user to recognize to particular user.HTTP is stateless that means each request is considered as the new request. It is shown in the figure given below:



Why use Session Tracking?

To recognize the user. It is used to recognize the particular user.

Session Tracking Techniques

There are four techniques used in Session tracking:

1. Cookies
2. Hidden Form Field
3. URL Rewriting
4. HttpSession

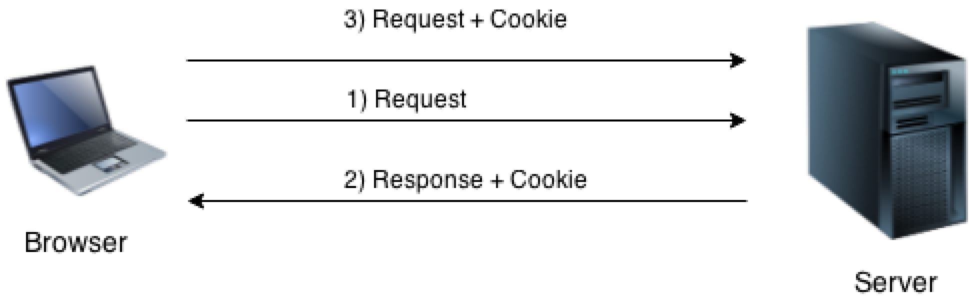
Cookies in Servlet

A **cookie** is a small piece of information that is persisted between the multiple client requests.

A cookie has a name, a single value, and optional attributes such as a comment, path and domain qualifiers, a maximum age, and a version number.

How Cookie works

By default, each request is considered as a new request. In cookies technique, we add cookie with response from the servlet. So cookie is stored in the cache of the browser. After that if request is sent by the user, cookie is added with request by default.



Types of Cookie

There are 2 types of cookies in servlets.

1. Non-persistent cookie
2. Persistent cookie

Non-persistent cookie

It is **valid for single session** only. It is removed each time when user closes the browser.

Persistent cookie

It is **valid for multiple session** . It is not removed each time when user closes the browser. It is removed only if user logout or signout.

Advantage of Cookies

1. Simplest technique of maintaining the state.
2. Cookies are maintained at client side.

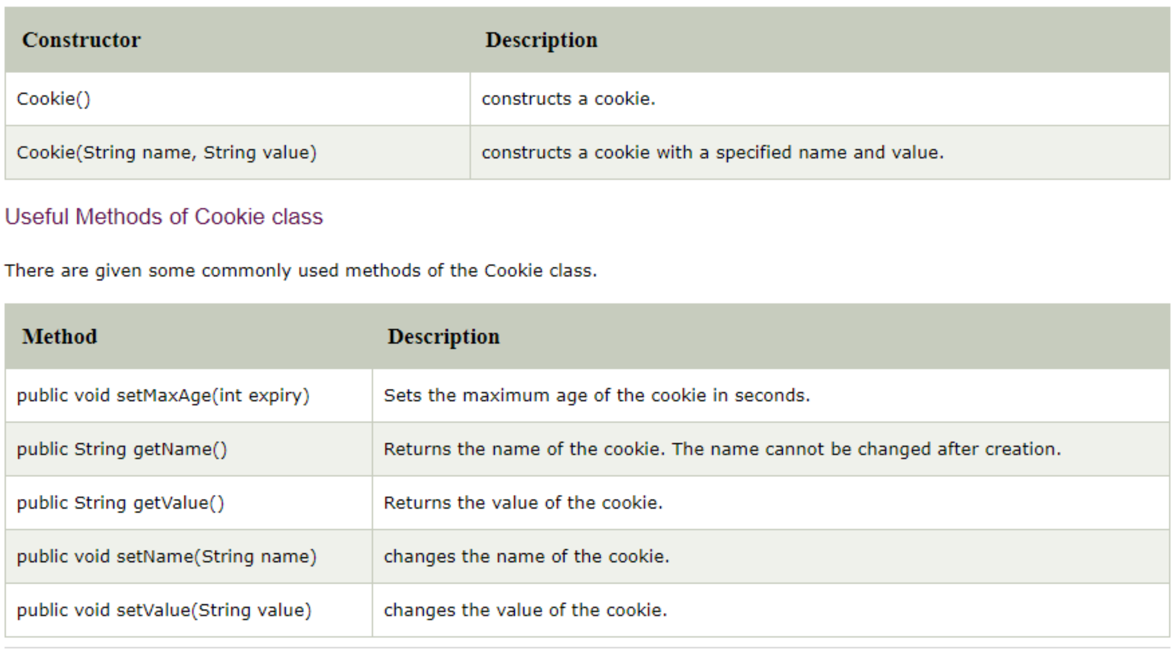
Disadvantage of Cookies

1. It will not work if cookie is disabled from the browser.
2. Only textual information can be set in Cookie object.

Cookie class

javax.servlet.http.Cookie class provides the functionality of using cookies. It provides a lot of useful methods for cookies.

Constructor of Cookie class



Constructor	Description
Cookie()	constructs a cookie.
Cookie(String name, String value)	constructs a cookie with a specified name and value.

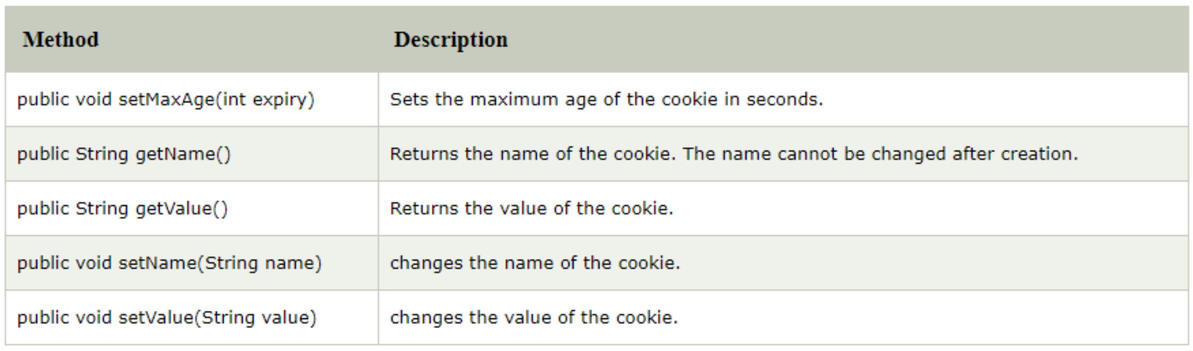
Useful Methods of Cookie class

There are given some commonly used methods of the Cookie class.

Method	Description
public void setMaxAge(int expiry)	Sets the maximum age of the cookie in seconds.
public String getName()	Returns the name of the cookie. The name cannot be changed after creation.
public String getValue()	Returns the value of the cookie.
public void setName(String name)	changes the name of the cookie.
public void setValue(String value)	changes the value of the cookie.

Useful Methods of Cookie class

There are given some commonly used methods of the Cookie class.



Method	Description
public void setMaxAge(int expiry)	Sets the maximum age of the cookie in seconds.
public String getName()	Returns the name of the cookie. The name cannot be changed after creation.
public String getValue()	Returns the value of the cookie.
public void setName(String name)	changes the name of the cookie.
public void setValue(String value)	changes the value of the cookie.

Example:
index.html

```
<form action="servlet1" method="post">
Name:<input type="text" name="userName"/><br/>
<input type="submit" value="go"/>
</form>
```

FirstServlet.java

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


public class FirstServlet extends HttpServlet {
  public void doPost(HttpServletRequest request, HttpServletResponse response){
    try{

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    String n=request.getParameter("userName");
    out.print("Welcome "+n);

    Cookie ck=new Cookie("uname",n);//creating cookie object
    response.addCookie(ck);//adding cookie in the response

    //creating submit button
    out.print("<form action='servlet2'>");
    out.print("<input type='submit' value='go'>");
    out.print("</form>");

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

SecondServlet.java

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

 public class SecondServlet extends HttpServlet {

 public void doPost(HttpServletRequest request, HttpServletResponse response){
    try{

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    Cookie ck[]=request.getCookies();
    out.print("Hello "+ck[0].getValue());

    out.close();

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

```
}
```

web.xml

```
<web-app>

<servlet>
<servlet-name>s1</servlet-name>
<servlet-class>FirstServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>s1</servlet-name>
<url-pattern>/servlet1</url-pattern>
</servlet-mapping>

<servlet>
<servlet-name>s2</servlet-name>
<servlet-class>SecondServlet</servlet-class>
</servlet>

servlet-mapping>
<servlet-name>s2</servlet-name>
<url-pattern>/servlet2</url-pattern>
</servlet-mapping>

</web-app>
```

2) Hidden Form Field

In case of Hidden Form Field **a hidden (invisible) textfield** is used for maintaining the state of an user.In such case, we store the information in the hidden field and get it from another servlet. This approach is better if we have to submit form in all the pages and we don't want to depend on the browser.

```
<input type="hidden" name="uname" value="Vimal Jaiswal">
```

Here, uname is the hidden field name and Vimal Jaiswal is the hidden field value.

Real application of hidden form field

It is widely used in comment form of a website. In such case, we store page id or page name in the hidden field so that each page can be uniquely identified.

Advantage of Hidden Form Field

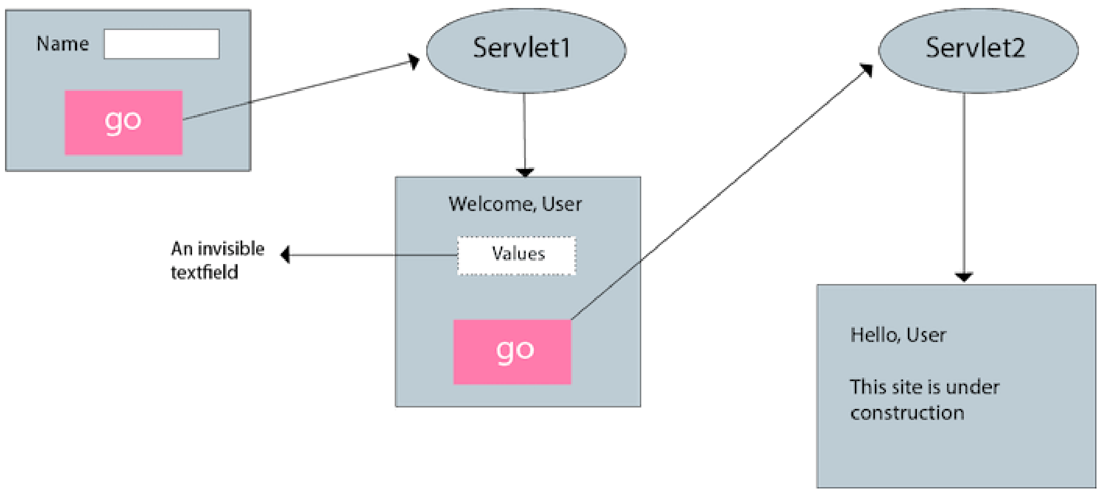
1. It will always work whether cookie is disabled or not.

Disadvantage of Hidden Form Field:

1. It is maintained at server side.
2. Extra form submission is required on each pages.
3. Only textual information can be used.

Example of using Hidden Form Field

In this example, we are storing the name of the user in a hidden textfield and getting that value from another servlet.



index.html

```
<form action="servlet1">
Name:<input type="text" name="userName"/><br/>
<input type="submit" value="go"/>
</form>
```

FirstServlet.java

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

public class FirstServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response){
        try{

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String n=request.getParameter("userName");
        out.print("Welcome "+n);

        //creating form that have invisible textfield
        out.print("<form action='servlet2'>");
        out.print("<input type='hidden' name='uname' value='"+n+"'>");
        out.print("<input type='submit' value='go'>");
        out.print("</form>");
        out.close();

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

 }
```

SecondServlet.java

```
 import java.io.*;
 import javax.servlet.*;
 import javax.servlet.http.*;
 public class SecondServlet extends HttpServlet {
 public void doGet(HttpServletRequest request, HttpServletResponse response)
        try{
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        //Getting the value from the hidden field
        String n=request.getParameter("uname");
        out.print("Hello "+n);

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

web.xml

```
 <web-app>

 <servlet>
 <servlet-name>s1</servlet-name>
 <servlet-class>FirstServlet</servlet-class>
 </servlet>
```

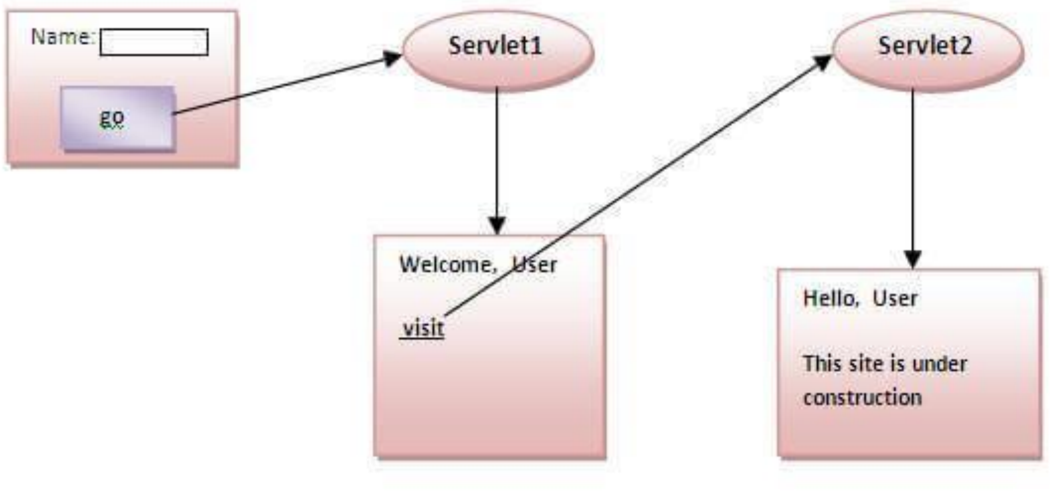
```
<servlet-mapping>
<servlet-name>s1</servlet-name>
<url-pattern>/servlet1</url-pattern>
</servlet-mapping>

<servlet>
<servlet-name>s2</servlet-name>
<servlet-class>SecondServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>s2</servlet-name>
<url-pattern>/servlet2</url-pattern>
</servlet-mapping>

</web-app>
```

3)URL Rewriting

In URL rewriting, we append a token or identifier to the URL of the next Servlet or the next resource. We can send parameter name/value pairs using the following format:

url?name1=value1&name2=value2&??

A name and a value is separated using an equal = sign, a parameter name/value pair is separated from another parameter using the ampersand(&). When the user clicks the hyperlink, the parameter name/value pairs will be passed to the server. From a Servlet, we can use getParameter() method to obtain a parameter value.



Advantage of URL Rewriting

1. It will always work whether cookie is disabled or not (browser independent).

2. Extra form submission is not required on each pages.

Disadvantage of URL Rewriting

1. It will work only with links.
2. It can send Only textual information.

Example of using URL Rewriting

In this example, we are maintaning the state of the user using link. For this purpose, we are appending the name of the user in the query string and getting the value from the query string in another page.

index.html

```
<form action="servlet1">
Name:<input type="text" name="userName"/><br/>
<input type="submit" value="go"/>
</form>
```

FirstServlet.java

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


public class FirstServlet extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response){
        try{

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        String n=request.getParameter("userName");
        out.print("Welcome "+n);

        //appending the username in the query string
        out.print("<a href='servlet2?uname="+n+"'>visit</a>");

        out.close();

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

}
```

SecondServlet.java

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

public class SecondServlet extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response)
        try{

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        //getting value from the query string
        String n=request.getParameter("uname");
        out.print("Hello "+n);

        out.close();

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


}
```

web.xml

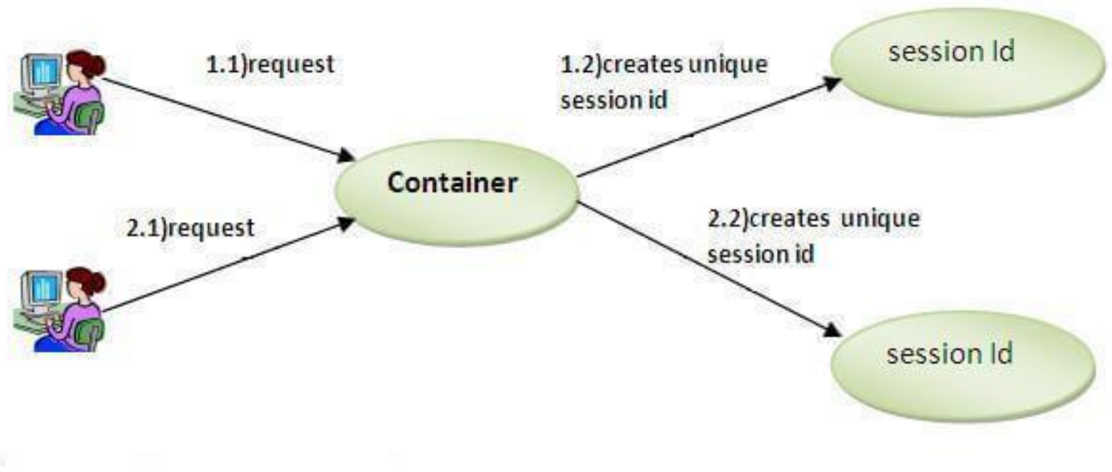
```
<web-app>

<servlet>
<servlet-name>s1</servlet-name>
<servlet-class>FirstServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>s1</servlet-name>
<url-pattern>/servlet1</url-pattern>
</servlet-mapping>

<servlet>
<servlet-name>s2</servlet-name>
<servlet-class>SecondServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>s2</servlet-name>
```

```
<url-pattern>/servlet2</url-pattern>
</servlet-mapping>

</web-app>
```

4) HttpSession interface

In such case, container creates a session id for each user.The container uses this id to identify the particular user.An object of HttpSession can be used to perform two tasks:

1. bind objects
2. view and manipulate information about a session, such as the session identifier, creation time, and last accessed time.



How to get the HttpSession object ?

The HttpServletRequest interface provides two methods to get the object of HttpSession:

1. **public HttpSession getSession():**Returns the current session associated with this request, or if the request does not have a session, creates one.
2. **public HttpSession getSession(boolean create):**Returns the current HttpSession associated with this request or, if there is no current session and create is true, returns a new session.

Commonly used methods of HttpSession interface

1. **public String getId():**Returns a string containing the unique identifier value.
2. **public long getCreationTime():**Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
3. **public long getLastAccessedTime():**Returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT.

4. **public void invalidate():**Invalidates this session then unbinds any objects bound to it.

Example of using HttpSession

In this example, we are setting the attribute in the session scope in one servlet and getting that value from the session scope in another servlet. To set the attribute in the session scope, we have used the setAttribute() method of HttpSession interface and to get the attribute, we have used the getAttribute method.

index.html

```
<form action="servlet1">
Name:<input type="text" name="userName"/><br/>
<input type="submit" value="go"/>
</form>
```

FirstServlet.java

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


  public class FirstServlet extends HttpServlet {

  public void doGet(HttpServletRequest request, HttpServletResponse response){
        try{

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        String n=request.getParameter("userName");
        out.print("Welcome "+n);

        HttpSession session=request.getSession();
        session.setAttribute("uname",n);

        out.print("<a href='servlet2'>visit</a>");

        out.close();

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

  }
```

SecondServlet.java

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

public class SecondServlet extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response)
        try{

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        HttpSession session=request.getSession(false);
        String n=(String)session.getAttribute("uname");
        out.print("Hello "+n);

        out.close();

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


}
```

web.xml

```
<web-app>

<servlet>
<servlet-name>s1</servlet-name>
<servlet-class>FirstServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>s1</servlet-name>
<url-pattern>/servlet1</url-pattern>
</servlet-mapping>

<servlet>
<servlet-name>s2</servlet-name>
<servlet-class>SecondServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>s2</servlet-name>
<url-pattern>/servlet2</url-pattern>
</servlet-mapping>
```

```
</web-app>
```

Servlet Context Interface:

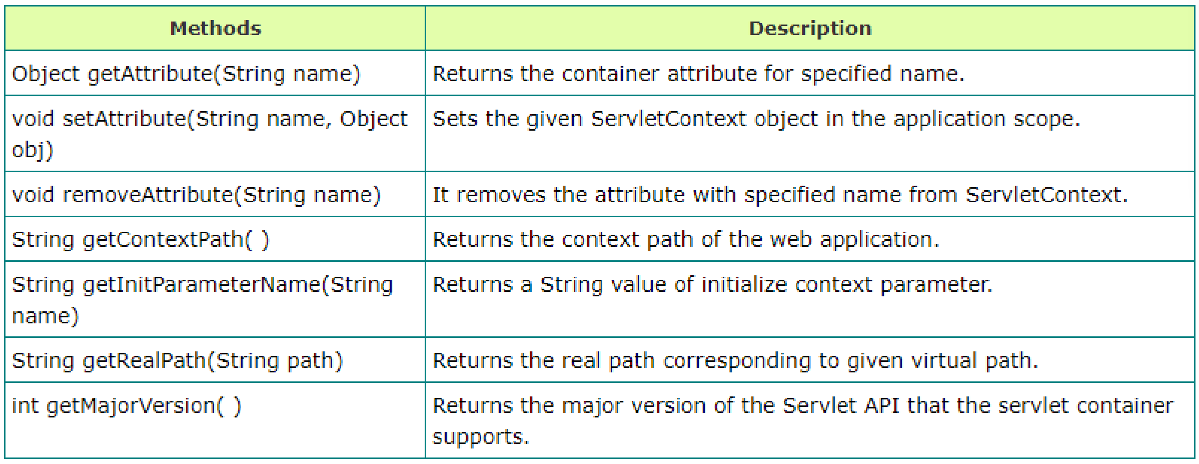
The object of ServletContext is automatically created by the container when the web application is deployed. There is only one **ServletContext** object per web application. It is available on **javax.servlet.*** package.

Uses of ServletContext Interface

- It helps to establish communication between container and servlets.
- The ServletContext object also provides inter-application communication.
- ServletContext has init parameter that can be used to get configuration information from web.xml.
- It provides the accessibility of application level parameter.

ServletContext Interface Methods

Following are the important methods of ServletContext interface:



Methods	Description
Object getAttribute(String name)	Returns the container attribute for specified name.
void setAttribute(String name, Object obj)	Sets the given ServletContext object in the application scope.
void removeAttribute(String name)	It removes the attribute with specified name from ServletContext.
String getContextPath( )	Returns the context path of the web application.
String getInitParameterName(String name)	Returns a String value of initialize context parameter.
String getRealPath(String path)	Returns the real path corresponding to given virtual path.
int getMajorVersion( )	Returns the major version of the Servlet API that the servlet container supports.

Getting Object of ServletContext Interface

- The ServletConfig interface provides a **getServletContext( )** method to return the object of ServletContext.

- The GenericServlet class also provides the **getServletContext( )** method to return the object of ServletContext.

Example

ServletContext app = getServletConfig( ).getServletContext( );

Or

ServletContext app = getServletContext( );

Servlet Context Lifecycle:

Step 1: Servlet container reads the DD (Deployment Descriptor – web.xml) and creates the name/value string pair for each <context-param> when web application is getting started.

Step 2: Container creates the new Instance of ServletContext.

Step 3: Servlet container gives the ServletContext a reference to each name/value pair of the context init parameter.

Step 4: Every servlet and JSP in the same web application will now has access to this ServletContext.

Servlet context lifecycle events

This category of events corresponds to the event receivers on the javax.servlet.ServletContextListener interface. The event propagated is a javax.servlet.ServletContext (not a javax.servlet.ServletContextEvent, since the ServletContext is the only relevant information this event provides).

There are two qualifiers provided in the org.jboss.solder.servlet.event package (@Initialized and @Destroyed) that can be used to observe a specific lifecycle phase of the servlet context.

The servlet context lifecycle events are documented in the table below.

Qualifier	Type	Description
@Default (optional)	javax.servlet.ServletContext	The servlet context is initialized or destroyed
@Initialized	javax.servlet.ServletContext	The servlet context is initialized
@Destroyed	javax.servlet.ServletContext	The servlet context is destroyed

If you want to listen to both lifecycle events, leave out the qualifiers on the observer method:

```
public void observeServletContext(@Observes ServletContext ctx) {

   System.out.println(ctx.getServletContextName() + " initialized or destroyed");

}
```

If you are interested in only a particular lifecycle phase, use one of the provided qualifiers:

```
public void observeServletContextInitialized(@Observes @Initialized ServletContext ctx) {

   System.out.println(ctx.getServletContextName() + " initialized");

}
```

Event Handling :

Events are basically occurrence of something. Changing the state of an object is known as an event.

We can perform some important tasks at the occurrence of these exceptions, such as counting total and current logged-in users, creating tables of the database at time of deploying the project, creating database connection object etc.

There are many Event classes and Listener interfaces in the javax.servlet and javax.servlet.http packages.

Event classes

The event classes are as follows:

1. ServletRequestEvent
2. ServletContextEvent
3. ServletRequestAttributeEvent
4. ServletContextAttributeEvent
5. HttpSessionEvent
6. HttpSessionBindingEvent

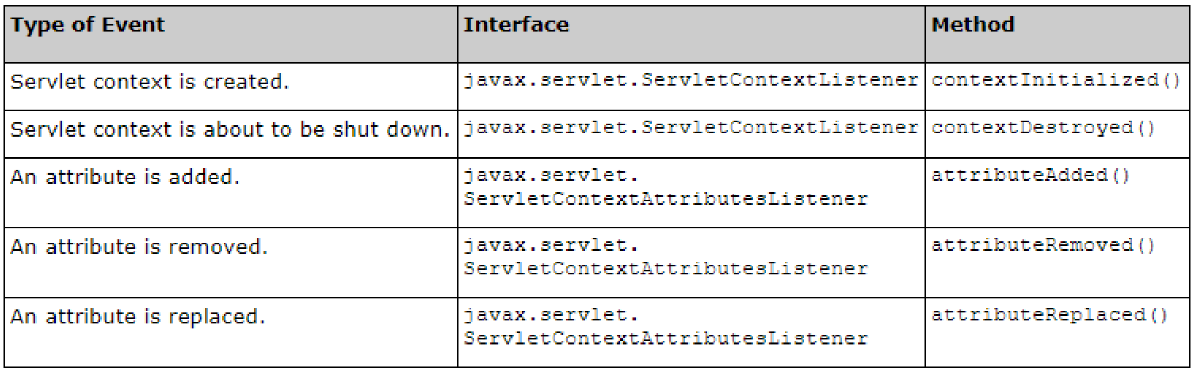
Event interfaces

The event interfaces are as follows:

1. ServletRequestListener
2. ServletRequestAttributeListener
3. ServletContextListener
4. ServletContextAttributeListener
5. HttpSessionListener

6. HttpSessionAttributeListener
7. HttpSessionBindingListener
8. HttpSessionActivationListener

Servlet Context Events



Type of Event	Interface	Method
Servlet context is created.	javax.servlet.ServletContextListener	contextInitialized()
Servlet context is about to be shut down.	javax.servlet.ServletContextListener	contextDestroyed()
An attribute is added.	javax.servlet. ServletContextAttributesListener	attributeAdded()
An attribute is removed.	javax.servlet. ServletContextAttributesListener	attributeRemoved()
An attribute is replaced.	javax.servlet. ServletContextAttributesListener	attributeReplaced()

Servlet Collaboration:

The Servlet collaboration is all about sharing information among the servlets. Collaborating servlets is to pass the common information that is to be shared directly by one servlet to another through various invocations of the methods. To perform these operations, each servlet need to know the other servlet with which it is collaborated. Here are several ways to communicate with one another:

- Using RequestDispatchers *include()* and *forward()* method;
- Using HttpServletResponse *sendRedirect()* method;
- Using ServletContext *setAttribute()* and *getAttribute()* methods;
- Using Java's system-wide *Properties* list;
- Using singleton class object.

Example of using RequestDispatcher for Servlet Collaboration
The following example explains how to use RequestDispatcher interface to achieve Servlet Collaboration:

index.html
```
<html>

<head>

<body>

<form action="login" method="post">

Name:<input type="text" name="userName"/><br/>

Password:<input type="password" name="userPass"/><br/>
```

```
<input type="submit" value="login"/>
</form>
</body>
</html>
```

Login.java

```
// First java servlet that calls another resource
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Login extends HttpServlet {

	public void doPost(HttpServletRequest req,
					HttpServletResponse res)
throws ServletException, IOException
	{
		// The method to receive client requests
		// which are sent using 'post'

		res.setContentType("text/html");
		PrintWriter out = response.getWriter();

		// fetches username
		String n = request.getParameter("userName");

		// fetches password
		String p = request.getParameter("userPass");
	if(p.equals("Thanos"){
			RequestDispatcher rd = request.getRequestDispatcher("servlet2");
			// Getting RequestDispatcher object
```

```
                    // for collaborating with servlet2

                    // forwarding the request to servlet2
                    rd.forward(request, response);
        }
        else{
                    out.print("Password mismatch");
                    RequestDispatcher rd = request.getRequestDispatcher("/index.html");

                rd.include(request, response);

                }
        }

}
```

Welcome.java

```
// Called servlet in case password matches
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Welcome extends HttpServlet {

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

                response.setContentType("text/html");
                PrintWriter out = response.getWriter();

                // fetches username
                String n = request.getParameter("userName");

                // prints the message
                out.print("Welcome " + n);
        }
}
```

web.xml

```
<web-app>
<servlet>
        <servlet-name>Login</servlet-name>
        <servlet-class>Login</servlet-class>
</servlet>
<servlet>
        <servlet-name>WelcomeServlet</servlet-name>
        <servlet-class>Welcome</servlet-class>
</servlet>



<servlet-mapping>
        <servlet-name>Login</servlet-name>
        <url-pattern>/servlet1</url-pattern>
</servlet-mapping>
<servlet-mapping>
        <servlet-name>WelcomeServlet</servlet-name>
        <url-pattern>/servlet2</url-pattern>
</servlet-mapping>
```

```
<welcome-file-list>

<welcome-file>index.html</welcome-file>

</welcome-file-list>

</web-app>
```

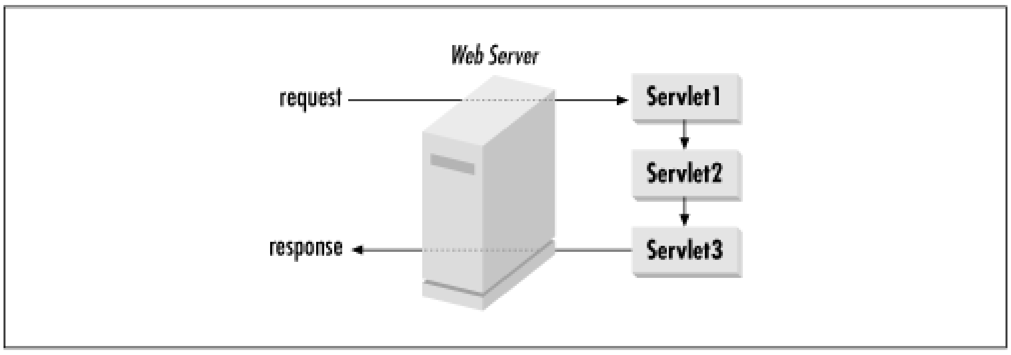
Servlet Chaining

Servlet Chaining means the output of one servlet act as a input to another servlet. Servlet Aliasing allows us to invoke more than one servlet in sequence when the URL is opened with a common servlet alias. The output from first Servlet is sent as input to other Servlet and so on. The Output from the last Servlet is sent back to the browser. The entire process is called Servlet Chaining.

Example

```
public class Test1 extends HttpServlet {
String name ;
ServletConfig config;

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

res.setContentType("text/plain");

PrintWriter out = res.getWriter();
name = req.getParameter("name");
RequestDispatcher rd = config.
getServletContext().getRequestDispatcher("SecondServlet");

if(name!=null) {

request.setAttribute("MyName",name);
rd.forward(req , res);
// Forward this value to another Servlet
} else {
res.sendError(res.SC_BAD_REQUEST,
"MyName Required");
}
}
}

public class Test2 extends HttpServlet {
public void doGet(HttpServletRequest req ,
HttpServletResponse res)
throws ServletException , IOException {
```

```
res.setContentType("text/plain");
PrintWriter out = res.getWriter();
String myname = (String)req.getAttribute("MyName");

// Extracting the value which is set in Test1
out.println("MyName is "+ myname);
}
}
```



Servlet chaining

Request Dispatching. :

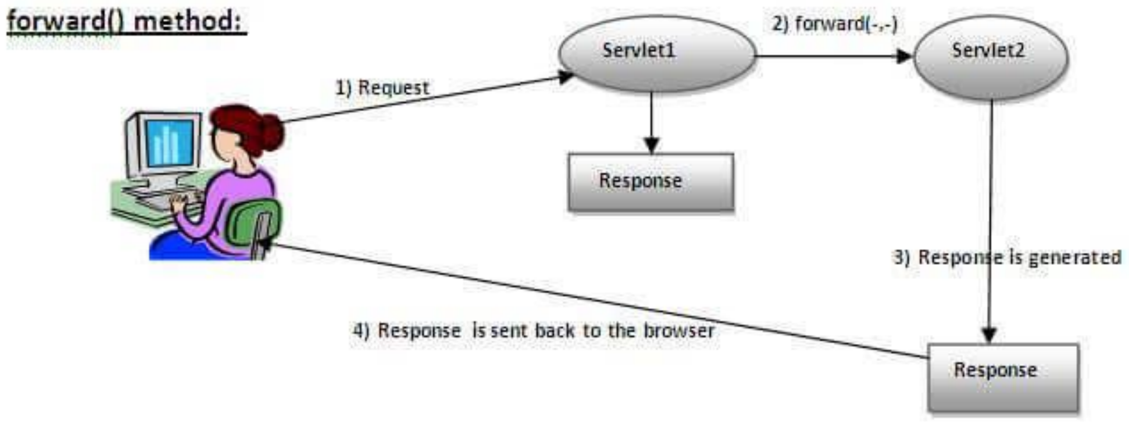
The RequestDispatcher interface provides the facility of dispatching the request to another resource it may be html, servlet or jsp. This interface can also be used to include the content of another resource also. It is one of the way of servlet collaboration.

There are two methods defined in the RequestDispatcher interface.

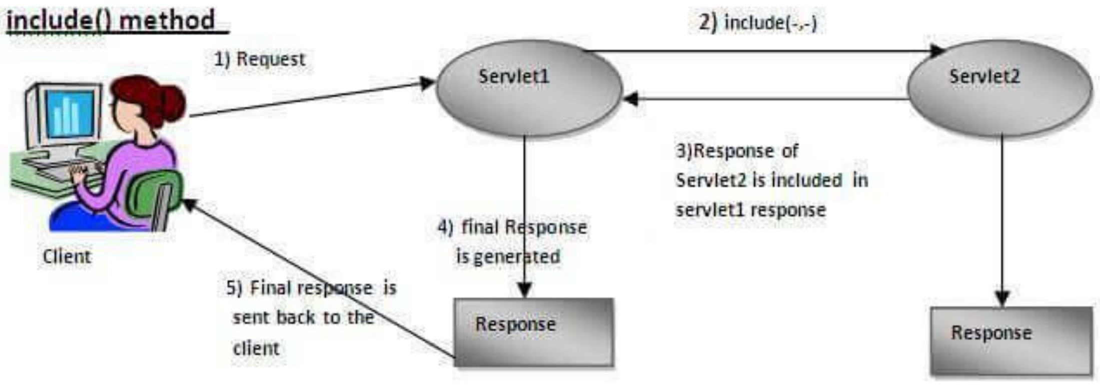
Methods of RequestDispatcher interface

The RequestDispatcher interface provides two methods. They are:

1. **public void forward(ServletRequest request,ServletResponse response)throws ServletException,java.io.IOException:**Forwards a request from a servlet to another resource (servlet, JSP file, or HTML file) on the server.
2. **public void include(ServletRequest request,ServletResponse response)throws ServletException,java.io.IOException:**Includes the content of a resource (servlet, JSP page, or HTML file) in the response



Response of second servlet is sent to the client. Response of the first servlet is not displayed to the user.


include() method
1) Request
Servlet1
2) include(-,-)
Servlet2
3)Response of Servlet2 is included in servlet1 response
4) final Response is generated
5) Final response is sent back to the client
Client
Response
Response

How to get the object of RequestDispatcher

The getRequestDispatcher() method of ServletRequest interface returns the object of RequestDispatcher. Syntax:

Syntax of getRequestDispatcher method

public RequestDispatcher getRequestDispatcher(String resource);

Example of using getRequestDispatcher method

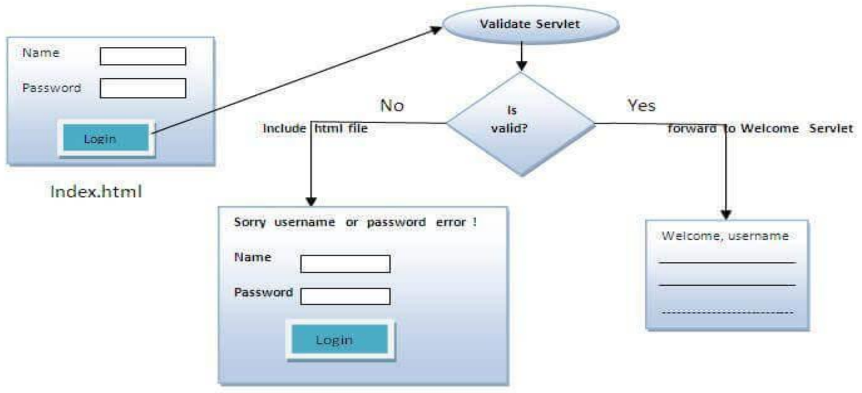
```
RequestDispatcher rd=request.getRequestDispatcher("servlet2");
//servlet2 is the url-pattern of the second servlet

rd.forward(request, response);//method may be include or forward
```

Example of RequestDispatcher interface

In this example, we are validating the password entered by the user. If password is servlet, it will forward the request to the WelcomeServlet, otherwise will show an error message: sorry username or password error!. In this program, we are cheking for hardcoded information

In this example, we have created following files:

- **index.html file:** for getting input from the user.
- **Login.java file:** a servlet class for processing the response. If password is servet, it will forward the request to the welcome servlet.
- **WelcomeServlet.java file:** a servlet class for displaying the welcome message.
- **web.xml file:** a deployment descriptor file that contains the information about the servlet.



index.html

```
<form action="servlet1" method="post">
Name:<input type="text" name="userName"/><br/>
Password:<input type="password" name="userPass"/><br/>
<input type="submit" value="login"/>
</form>
```

Login.java

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

```
public class Login extends HttpServlet {

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

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    String n=request.getParameter("userName");
    String p=request.getParameter("userPass");

    if(p.equals("servlet"){
        RequestDispatcher rd=request.getRequestDispatcher("servlet2");
        rd.forward(request, response);
    }
    else{
        out.print("Sorry UserName or Password Error!");
        RequestDispatcher rd=request.getRequestDispatcher("/index.html");
        rd.include(request, response);

        }
    }

}
```

WelcomeServlet.java

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

public class WelcomeServlet extends HttpServlet {

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

  response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    String n=request.getParameter("userName");
    out.print("Welcome "+n);
    }
```

```
      }
web.xml
      <web-app>
       <servlet>
          <servlet-name>Login</servlet-name>
          <servlet-class>Login</servlet-class>
        </servlet>
        <servlet>
          <servlet-name>WelcomeServlet</servlet-name>
          <servlet-class>WelcomeServlet</servlet-class>
        </servlet>


        <servlet-mapping>
          <servlet-name>Login</servlet-name>
          <url-pattern>/servlet1</url-pattern>
        </servlet-mapping>
        <servlet-mapping>
          <servlet-name>WelcomeServlet</servlet-name>
          <url-pattern>/servlet2</url-pattern>
        </servlet-mapping>

        <welcome-file-list>
         <welcome-file>index.html</welcome-file>
  </welcome-file-list>
web-app>
```

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT- 5 Advanced Java – SBS1301

Java Server Pages

JSP technology is used to create web application just like Servlet technology. It can be thought of as an extension to Servlet because it provides more functionality than servlet such as expression language, JSTL, etc.

A JSP page consists of HTML tags and JSP tags. The JSP pages are easier to maintain than Servlet because we can separate designing and development. It provides some additional features such as Expression Language, Custom Tags, etc.

Advantages of JSP over Servlet

There are many advantages of JSP over the Servlet. They are as follows:

1) Extension to Servlet

JSP technology is the extension to Servlet technology. We can use all the features of the Servlet in JSP. In addition to, we can use implicit objects, predefined tags, expression language and Custom tags in JSP, that makes JSP development easy.

2) Easy to maintain

JSP can be easily managed because we can easily separate our business logic with presentation logic. In Servlet technology, we mix our business logic with the presentation logic.

3) Fast Development: No need to recompile and redeploy

If JSP page is modified, we don't need to recompile and redeploy the project. The Servlet code needs to be updated and recompiled if we have to change the look and feel of the application.

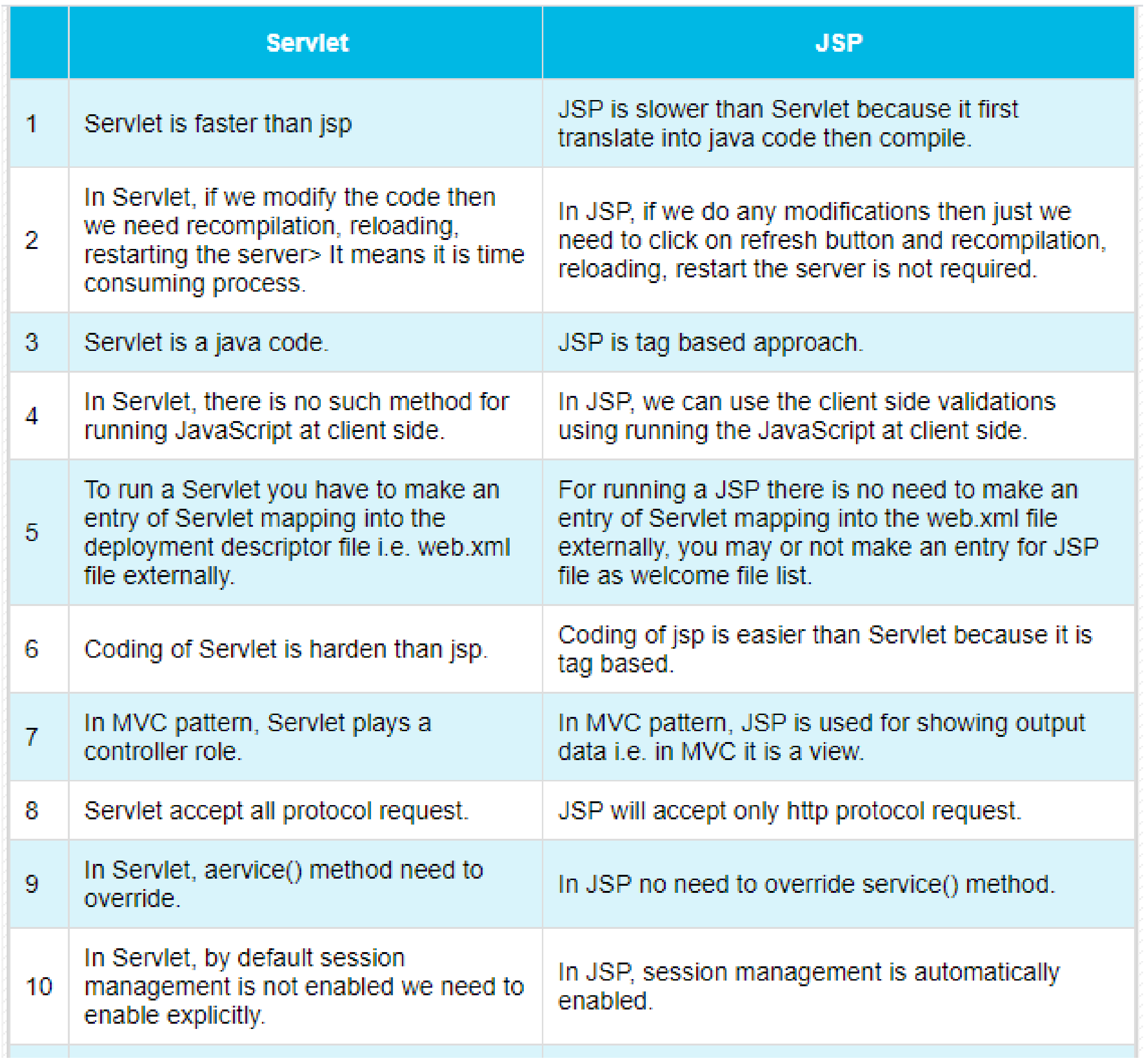
4) Less code than Servlet

In JSP, we can use many tags such as action tags, JSTL, custom tags, etc. that reduces the code. Moreover, we can use EL, implicit objects, etc.

Features of JSP

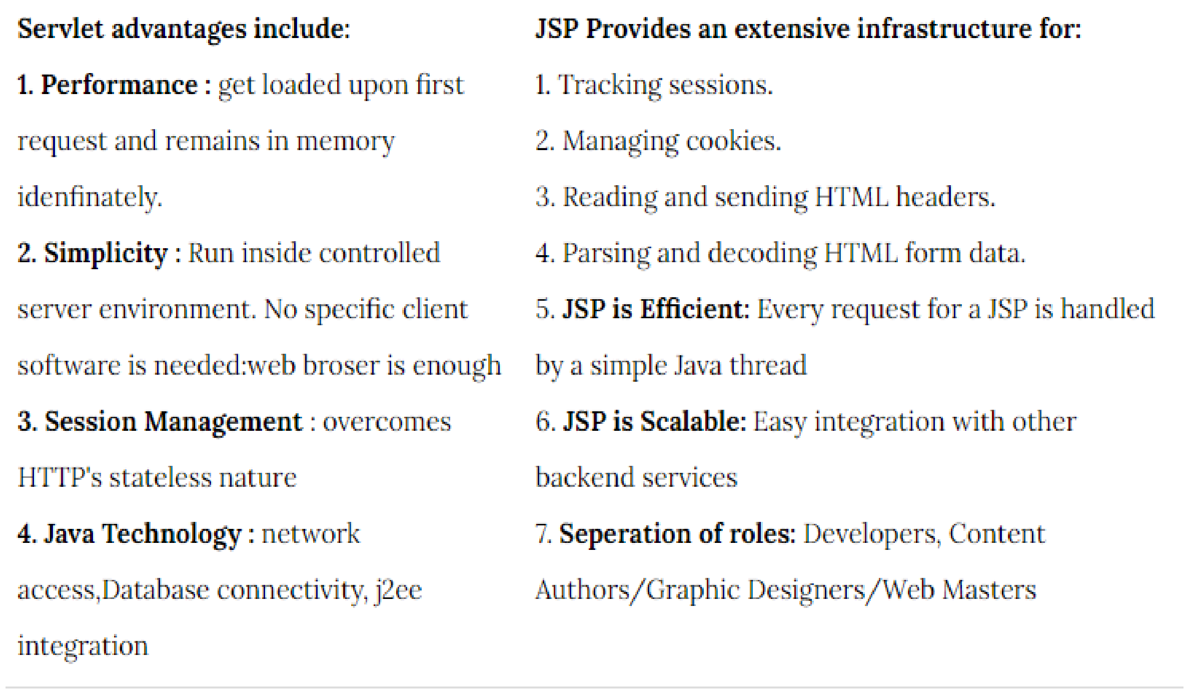
- **Coding in JSP is easy** :- As it is just adding JAVA code to HTML/XML.
- **Reduction in the length of Code** :- In JSP we use action tags, custom tags etc.
- **Connection to Database is easier** :-It is easier to connect website to database and allows to read or write data easily to the database.
- **Make Interactive websites** :- In this we can create dynamic web pages which helps user to interact in real time environment.
- **Portable, Powerful, flexible and easy to maintain** :- as these are browser and server independent.
- **No Redeployment and No Re-Compilation** :- It is dynamic, secure and platform independent so no need to re-compilation.
- **Extension to Servlet** :- as it has all features of servlets, implicit objects and custom tags

Difference between Servlets and JSP:



	Servlet	JSP
1	Servlet is faster than jsp	JSP is slower than Servlet because it first translate into java code then compile.
2	In Servlet, if we modify the code then we need recompilation, reloading, restarting the server> It means it is time consuming process.	In JSP, if we do any modifications then just we need to click on refresh button and recompilation, reloading, restart the server is not required.
3	Servlet is a java code.	JSP is tag based approach.
4	In Servlet, there is no such method for running JavaScript at client side.	In JSP, we can use the client side validations using running the JavaScript at client side.
5	To run a Servlet you have to make an entry of Servlet mapping into the deployment descriptor file i.e. web.xml file externally.	For running a JSP there is no need to make an entry of Servlet mapping into the web.xml file externally, you may or not make an entry for JSP file as welcome file list.
6	Coding of Servlet is harden than jsp.	Coding of jsp is easier than Servlet because it is tag based.
7	In MVC pattern, Servlet plays a controller role.	In MVC pattern, JSP is used for showing output data i.e. in MVC it is a view.
8	Servlet accept all protocol request.	JSP will accept only http protocol request.
9	In Servlet, aervice() method need to override.	In JSP no need to override service() method.
10	In Servlet, by default session management is not enabled we need to enable explicitly.	In JSP, session management is automatically enabled.

Table 5.1 Difference between Servlets and JSP:



Servlet advantages include:	**JSP Provides an extensive infrastructure for:**
1. **Performance :** get loaded upon first request and remains in memory idenfinately.	1. Tracking sessions.
2. **Simplicity :** Run inside controlled server environment. No specific client software is needed:web broser is enough	2. Managing cookies.
3. **Session Management :** overcomes HTTP's stateless nature	3. Reading and sending HTML headers.
4. **Java Technology :** network access,Database connectivity, j2ee integration	4. Parsing and decoding HTML form data.
	5. **JSP is Efficient:** Every request for a JSP is handled by a simple Java thread
	6. **JSP is Scalable:** Easy integration with other backend services
	7. **Seperation of roles:** Developers, Content Authors/Graphic Designers/Web Masters

Table 5.2 Difference between Servlets and JSP:

A JSP is just another servlet, and like HTTP servlets, a JSP is a server-side Web component that can be used to generate dynamic Web pages. The fundamental difference between servlets and JSPs is

- Servlets generate HTML from Java code.
- JSPs embed Java code in static HTML.

A JavaServer Pages (JSP) component is a type of Java servlet that is designed to fulfill the role of a user interface for a Java web application. Web developers write JSPs as text files that combine HTML or XHTML code, XML elements, and embedded JSP actions and commands. JSPs were originally designed around the model of embedded server-side scripting tools such as Microsoft Corporation's ASP technology; however, JSPs have evolved to focus on XML elements, including custom-designed elements, or *custom tags* , as the principal method of generating dynamic web content.

JSP files typically have a .jsp extension, as in *mypage.jsp*. When a client requests the JSP page for the first time, or if the developer precompiles the JSP the web container translates the textual document into a servlet.

A JSP compiler (such as Tomcat's Jasper component) automatically converts the text-based document into a servlet. The web container creates an instance of the servlet and makes the servlet available to handle requests. These tasks are transparent to the developer, who never has to handle the translated servlet source code

The developer focuses on the JSP's dynamic behavior and which JSP elements or custom-designed tags she uses to generate the response.

A JSP file that displays the date

```
<%-- use the 'taglib' directive to make the JSTL 1.0 core tags available; use the uri "http://java.sun.com/jsp/jstl/core" for JSTL 1.1 --%>
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>

<%-- use the 'jsp:useBean' standard action to create the Date object;  the object is set as an attribute in page scope --%>
<jsp:useBean id="date" class="java.util.Date" />
<html>
<head><title>First JSP</title></head>
<body>
<h2>Here is today's date</h2>
<c:out value="${date}" />
</body>
</html>
```

JSP Syntax and Structure

Before writing your first JSP, you need to gain an understanding of the syntax and the structure of a JSP.

As you have seen, JSP elements are embedded in static HTML. Like HTML, all JSP elements are enclosed in open and close angle brackets (< >). Unlike HTML, but like XML, all JSP elements are case sensitive.

JSP elements are distinguished from HTML tags by beginning with either <% or <jsp:.

JSPs follow XML syntax, they all have a start tag (which includes the element name) and a matching end tag. Like XML tags, a JSP tag with an empty body can combine the start and end tags into a single tag. The following is an empty body tag:

```
<jsp:useBean id=”agency” class=”web.AgencyBean”>
```

```
</jsp:useBean>
```

The following tag is equivalent to the previous example:

```
<jsp:useBean id=”agency” class=”web.AgencyBean”/>
```

Optionally, a JSP element may have attributes and a body. You will see examples of all these types during today’s lesson. See Appendix C, “An Overview of XML,” for more information on XML and the syntax of XML elements.

JSP Elements

The basic JSP elements are summarised in following Table

Element Type	JSP Syntax	Description
Directives	<%@Directive…%>	Information used to control the translation of the JSP text into Java code
Scripting	<% %>	Embedded Java code
Actions	<jsp: >	JSP-specific tags primarily used to support JavaBeans

Table 5.3 The basic JSP elements are summarised **Action**

Actions are specific tags that affect the run-time behaviour of the JSP and affect the response sent back to the client.

- <jsp:useBean>
- <jsp:param>
- <jsp:setProperty>
- <jsp:getProperty>
- <jsp:include>
- <jsp:forward>
- <jsp:plugin>

- jsp:setProperty:

This standard action is used in conjunction with the useBean action described in the preceding section and sets the values of simple and included properties in a bean.

```
<jsp:setProperty name=”beanname” propertydetail/>
```

attribute name: The name of the bean instance defined by a <jsp:useBean> tag.

attribute property: The name of the bean property whose value is being set.

- jsp:include:

This action allows a static or dynamic remote to be included in the current jsp at request time. The resource is specified using the URL format described in the earlier section on the include directive.

<jsp:include page=”FILE NAME” flush=”true”/>

true: The buffer in the output stream is flushed.

- jsp:forward

This action allows a request to be forward to another jsp, a servlet, or a static resource.

<jsp:forward page=”URL”/>

- jsp:plugin

This action is used to generate client browser specific HTML tags that ensure the JavaPlugin software in available, followed by execution of the applet or JavaBean component specified in the tag.

Scripting Elements

Scripting elements contain the code logic. It is these elements that get translated into a Java class and compiled. There are three types of scripting elements—declarations,

scriptlets, and expressions. They all start with <% and end with %>.

1. Expressions of the form <%= expression %>, which are evaluated and inserted into the servlet’s output

2. Scriptlets of the form <% code %>, which are inserted into the servlet’s _jspService method (called by service)

3. Declarations of the form <%! code %>, which are inserted into the body of the servlet class, outside of any existing methodsDeclarations

Declarations are used to introduce one or more variable or method declarations, each one separated by semicolons. A variable must be declared before it is used on a JSP page.

Declarations are differentiated from other scripting elements with a <%! start tag. An example declaration that defines two variables is as follows:

<%! String color = “blue”; int i = 42; %>

You can have as many declarations as you need. Variables and methods defined in declarations are declared as instance variables outside of any methods in the class.

Expressions

JSP expressions are single statements that are evaluated, and the result is cast into a string and placed on the HTML page. An expression is introduced with <%= and must not be terminated with a semi-colon. The following is an expression that will put the contents of the i element in the items array on the output page.

<%= items[i] %>

JSP expressions can be used as values for attributes in JSP tags. The following example shows how the i element in the items array can be used as the value for a submit button on a form:

<INPUT type=submit value=”<%= items[i] %>”>

Scriptlets

Scriptlets contain code fragments that are processed when a request is received by the JSP. Scriptlets are processed in the order they appear in the JSP. They need not produce output. Scriptlets can be used to create local variables, for example

<% int i = 42;%>

<BIG>The answer is <%= i %></BIG>

The difference between scriptlet variables and declarations is that scriptlet variables are scoped for each request. Variables created in declarations can retain their values between requests (they are instance variables).

JSP scriptlets let you insert arbitrary code into the servlet's _jspService method (which is called by service). Scriptlets have the following form:

```
<% Java Code %>
```

Scriptlets have access to the same automatically defined variables as expressions (request, response, session, out, etc.; see Section 10.5). So, for example, if you want output to appear in the resultant page, you would use the out variable, as in the following example.

```
<%
String queryData = request.getQueryString();
out.println("Attached GET data: " + queryData);
%>
```

In this particular instance, you could have accomplished the same effect more easily by using the following JSP expression:

```
Attached GET data: <%= request.getQueryString() %>
```

In general, however, scriptlets can perform a number of tasks that cannot be accomplished with expressions alone. These tasks include setting response headers and status codes, invoking side effects such as writing to the server log or updating a database, or executing code that contains loops, conditionals, or other complex constructs. For instance, the following snippet specifies that the current page is sent to the client as plain text, not as HTML (which is the default).

```
<% response.setContentType("text/plain"); %>
```

JSP Comments

There are three types of comments in a JSP page. The first type is called a JSP comment. JSP comments are used to document the JSP page. A JSP comment is completely ignored; it is not included in the generated code. A JSP comment looks like the following:

```
<%-- this is a JSP comment --%>
```

An alternative way to comment a JSP is to use the comment mechanism of the scripting language, as in the following:

```
<% /* this is a java comment */ %>
```

This comment will be placed in the generated Java code. The third mechanism for adding comments to a JSP is to use HTML comments.

```
<!-- this is an HTML comment -->
```

HTML comments are passed through to the client as part of the response. As a result, this form can be used to document the generated HTML document. Dynamic information can be included in HTML comments as shown in the following:

```
<!-- comment <%= expression %> comment -->
```

JSP Declarations

A JSP declaration lets you define methods or fields that get inserted into the main body of the servlet class (outside of the _jspService method that is called by service to process the request). A declaration has the following form:

```
<%! Java Code %>
```

Since declarations do not generate any output, they are normally used in conjunction with JSP expressions or scriptlets. For example, here is a JSP fragment that prints the number of times the current page has been requested since the server was booted (or the servlet class was changed and reloaded). Recall that multiple client requests to the same servlet result only in multiple threads calling the service method of a single servlet instance.

They do not result in the creation of multiple servlet instances except possibly when the servlet implements SingleThreadModel. For a discussion of SingleThreadModel,

Instance variables (fields) of a servlet are shared by multiple requests and accessCount does not have to be declared static below.

```
<%! private int accessCount = 0; %>
Accesses to page since server reboot:
<%= ++accessCount %>
```

JSP Directives

Directives are used to define information about your page to the translator, they do not produce any HTML output. All directives have the following syntax:

```
<%@ directive [ attr="value" ] %>
```

where directive can be page, include, or taglib.

The include Directive

You use the include directive to insert the contents of another file into the JSP. The included file can contain HTML or JSP tags or both. It is a useful mechanism for including the same page directives in all your JSPs or reusing small pieces of HTML to create common look and feel.

If the include file is itself a JSP, it is standard practice to use .jsf or .jspf, as suggested in the JSP specification, to indicate that the file contains a JSP fragment. These extensions show that the file is to be used in an include directive (and does not create a wellformed HTML page). ".jsp" should be reserved to refer to standalone JSPs.

dateBanner.jsp

```
<HTML> <HEAD>
<TITLE>JSP Date Example with common banner</TITLE>
</HEAD> <BODY>
<%@ include file="banner.html" %>
<BIG> Today's date is <%= new java.util.Date() %> </BIG>
```

```
</BODY> </HTML>
```

banner.html

```
<TABLE border=”0” width=”600” cellspacing=”0” cellpadding=”0”>

<TR>  <TD width=”350”><H1>Temporal Information </H1> </TD>

<TD align=”right” width=”250”><IMG src=”clock.gif”> </TD>  </TR>

 </TABLE> <BR>
```

The page Directive

Page directives are used to define page-dependent properties. You can have more than one page directive in the JSP. A page directive applies to the whole JSP, together with any files incorporated via the include directive.

TABLE JSP Page Directives

Directive Example Effect info

<%@ page info=”my first JSP Example” %>

Defines text string that is placed in the Servlet.getServletInfo() method in the translated code LISTING 13.4 Continued import <%@ page import=” java.math.*” %> A comma-separated list of package names to be imported for this JSP. The default

import list is java.lang.*,

javax.servlet.*,

javax.servlet.jsp.*, and

javax.servlet.http.*.

isThreadSafe <%@ page isThreadSafe=”true” %> If set to true, this indicates that <%@ page isThreadSafe=”false” %> this page can be run multithreaded.

This is the default, so you should ensure that access to shared objects (such as

instance variables) is synchronized.

errorPage <%@ page errorPage=”/agency/error.jsp” %> The client will be redirected to the specified URL when an exception occurs that is not caught by the current page.

isErrorPage <%@ page isErrorPage=”true” %> Indicates whether this page is <%@ page isErrorPage=”false” %> the target URL for an errorPage directive. If true, an implicit scripting variable called “exception” is defined and references the exception thrown in the source JSP. The default is false.

name.jsp

```
<%@page import=”java.util.*, javax.naming.*, agency.*” %>
<%@page errorPage=”errorPage.jsp” %>
<HTML> <TITLE>Agency Name</TITLE> <BODY>
<% InitialContext ic = null;
   ic = new InitialContext();
AgencyHome agencyHome = (AgencyHome)ic.lookup(“java:comp/env/ejb/Agency”);
Agency agency = agencyHome.create(); %>
<H1><%= agency.getAgencyName() %> </H1>
</BODY> </HTML>
```

Accessing HTTP Servlet Variables

The JSP pages you write are translated into servlets that process the HTTP GET and POST requests. The JSP code can access servlet information using implicit objects defined for each page. These implicit objects are pre-declared variables that you can reference from the Java code on your JSP.

JSP Implicit Objects

Reference Name	Class	Description
config	javax.servlet.ServletConfig	The servlet configuration information for the page
request	A subclass of javax.servlet.ServletRequest	Request information for the current HTTP request

session	javax.servlet.http.HttpSession	The servlet session object for the client
out	javax.servlet.jsp.JspWriter A subclass of java.io.Writer	that is used to output text for inclusion on the Web page
pageContext	javax.servlet.jsp.PageContext	The JSP page context used primarily when implementing custom tags
application	javax.servlet.ServletContext	The context for all Web components in the same application

Table 5.4 JSP Implicit Objects

These implicit objects can be used on any JSP page. Using the date JSP shown in Listing 13.3 as an example, an alternative way of writing the date to the page is as follows:

<BIG> Today's date is <% out.print(new java.util.Date()); %> </BIG>

Predefined Variables

To simplify code in JSP expressions and scriptlets, you are supplied with eight automatically defined variables, sometimes called implicit objects. Since JSP declarations (see Section 10.4) result in code that appears outside of the _jspService method, these variables are not accessible in declarations. The available variables are request, response, out, session, application, config, pageContext, and page. Details for each are given below.

request

This variable is the HttpServletRequest associated with the request; it gives you access to the request parameters, the request type (e.g., GET or POST), and the incoming HTTP headers (e.g., cookies). Strictly speaking, if the protocol in the request is something other than HTTP, request is allowed to be a subclass of ServletRequest other than HttpServletRequest. However, few, if any, JSP servers currently support non-HTTP servlets.

response

This variable is the HttpServletResponse associated with the response to the client. Note that since the output stream (see out) is normally buffered, it is legal to set HTTP status codes and response headers in JSP pages, even though the setting of headers or status codes is not permitted in servlets once any output has been sent to the client.

out

This is the PrintWriter used to send output to the client. However, to make the response object useful, this is a buffered version of Print-Writer called JspWriter. You can adjust the buffer size through use of the buffer attribute of the page directive that out is used almost exclusively in scriptlets, since JSP expressions are automatically placed in the output stream and thus rarely need to refer to out explicitly.

session

This variable is the HttpSession object associated with the request. Recall that sessions are created automatically, so this variable is bound even if there is no incoming session reference. The one exception is if you use the session attribute of the page directive (see Section 11.4) to turn sessions off. In that case, attempts to reference the session variable cause errors at the time the JSP page is translated into a servlet.

application

This variable is the ServletContext as obtained via

getServletConfig().

getContext().

Servlets and JSP pages can store persistent data in the ServletContext object rather than in instance variables. ServletContext has setAttribute and getAttribute methods that let you store arbitrary data associated with specified keys. The difference between storing data in instance variables and storing it in the Servlet-Context is that the ServletContext is shared by all servlets in the servlet engine (or in the Web application, if your server supports such a capability).

config

This variable is the ServletConfig object for this page.

pageContext

JSP introduced a new class called PageContext to give a single point of access to many of the page attributes and to provide a convenient place to store shared data. The pageContext variable stores the value of the PageContext object associated with the current page.

page

This variable is simply a synonym for this and is not very useful in the Java programming language. It was created as a place holder for the time when the scripting language could be something other than Java.

Using HTTP Request Parameters

The next requirement for many JSP pages is to be able to use request parameters to configure the behavior of the page. Using the Agency case study as an example, you will develop a simple JSP to display the contents of a named database table.

The first step is to define a simple form to allow the user to select the table to display.

tableForm.jsp

```
<HTML> <TITLE>Agency Tables</TITLE> <BODY>

<FORM action=table>

Select a table to display:

<SELECT name=table>

<OPTION>Applicant          <OPTION>ApplicantSkill          <OPTION>Customer

<OPTION>Job                <OPTION>JobSKill                <OPTION>Location

<OPTION>Matched            <OPTION>Skill                   </SELECT><P>

<INPUT type=submit> </FORM>    </BODY>     </HTML>
```

You will need to add this JSP to the simple Web application and define an alias of /tableForm to use it. The actual JSP to display the table is shown in Listing 13.8. LISTING 13.8 Full Text of table.jsp

```
<%@page import=”java.util.*, javax.naming.*, agency.*” %>

<%@page errorPage=”errorPage.jsp” %>

<% String table=request.getParameter(“table”); %>

<HTML> <TITLE>Agency Table: <%= table %></TITLE>

<BODY> <H1>Data for table <%= table %> </H1>
```

```
<TABLE border=1>
<%InitialContext ic = null;
ic = new InitialContext();
AgencyHome agencyHome = (AgencyHome)ic.lookup("java:comp/env/ejb/Agency");
Agency agency = agencyHome.create();
Collection rows = agency.select(table);
Iterator it = rows.iterator();
while (it.hasNext()) {
out.print("<TR>");
String[] row = (String[])it.next();
for (int i=0; i<row.length; i++)
out.print("<TD>"+row[i]+"</TD>");
out.print("</TR>");
 } %>
</TABLE> </BODY> </HTML>
```

JSP Problems

There are three types of errors you can make with JSP pages:

- JSP errors causing the translation to fail
- Java errors causing the compilation to fail
- HTML errors causing the page to display incorrectly

Finding and correcting these errors can be quite problematic because the information you need to discover the error is not readily available. Before looking at resolving errors, you will need to understand the JSP lifecycle.

JSP Lifecycle

As has already been stated, JSPs go through a translation and compilation phase prior to processing their first request.

The Web server automatically translates and compiles a JSP; you do not have to manually run any utility to do this. JSP translation and compilation can occur at any time prior to the JSP first being accessed. It is implementation dependent when this translation and compilation occurs but it is usually either

• On deployment

• When the first request for the JSP is received

If the latter strategy is used, not only is there a delay in processing the first request because the page is translated and compiled, but if the compilation fails, the client will be presented with some unintelligible error. If your server uses this strategy, ensure that you always force the translation and compilation of your JSP, either by making the first page request after it has been deployed or by forcing the page to be pre-compiled. With J2EE RI, the translation and compilation only takes place when the page is first accessed. You can find the translated JSP in <*J2EE installation*>\repository\ <*machine name*>\web\<context root>\. You may find it useful to refer to the translated

JSP to understand any compilation errors.

Detecting and Correcting JSP Errors

Realistically, you are going to make errors when writing JSPs. These errors can be quite

difficult to comprehend because of the way they are detected and reported. There are three categories of error:

• JSP translation

• Servlet compilation

• HTML presentation

The first two categories of error are detected by the Web server and sent back to the client browser instead of the requested page. The last type of error (HTML) is detected by the Web browser. Correcting each category of error requires a different technique

Translation Errors

If you mistype the JSP tags or fail to use the correct attributes for the tags, you will get a translation error returned to your browser. With the simple date example, missing the closing % sign from the JSP expression, as in the following code

Today’s date is <%= new java.util.Date() >

It will generate a translation error. Using the Web browser to report errors is an expedient solution to the problem of reporting errors, but this approach is not used by all Web servers. Some simply write the error to a log file and return an HTTP error to the browser. The JSP specification simply requires the Web server to report an HTTP 500 problem if there is an error on the JSP.

This shows all of the useful information for determining the error. The first part of the line tells you the exception that occurred:

org.apache.jasper.compiler.ParseException:

In this case, a generic parsing exception reported by the JSP translator. The J2EE RI includes a version of the Apache Tomcat Web server and it is the Jasper parser of Tomcat that has reported the error.

The second part of the error identifies the JSP page: /date.jsp and the third part specifies the line and column number: (8,0)

You know that the error is on line 8 of the date.jsp page. The column number is often misleading and is best ignored when looking for the error. The final part of the error message is a brief description of the problem:

Unterminated <%= tag

The rest of the error information returned to the Web browser is a stack trace of where the exception occurred in the Jasper translator. This is of no practical use to you and can be ignored. From the error information you should be able to identify the problem on the original JSP. Depending on the nature of the error, you may need to look at JSP lines prior to the one with the reported error. Sometimes errors are not reported until much later in the JSP.

The worst scenario is when the error is reported on the very last line because this means the error could be practically anywhere in the JSP.

Compilation Errors

Compilation errors can occur when you mistype the Java code in a Java scripting element or when you omit necessary page directives, such as import package lists.

Compilation errors are shown on the page returned to the browser and show the line number in error in the generated file. Figure 13.7 shows compilation error that occurs if you mistype Date as Datex in the date example show in Listing 13.3. The following is the error line:

Today’s date is <%= new java.util.Datex() %>

The information provided identifies the line in error in the JSP file and the corresponding line in error in the generated Java file. If you cannot determine the error from the JSP file, you will need to examine the generated file.

As stated earlier, the J2EE RI saves the generated Java file in the repository directory in the J2EE installation directory. The actual location is in a directory hierarchy named after the current workstation, the application name, and the Web application name. The filename is generated from the original JSP name.

In the example error, if the current host is ABC123, the file will be stored as <*J2EE home*>\repository\ABC123\web\simple\0002fdate_jsp.java The following code fragment shows the generated code containing the Java error:

```
application = pageContext.getServletContext();

config = pageContext.getServletConfig();

session = pageContext.getSession();

out = pageContext.getOut();

// HTML // begin [file=”/date.jsp”;from=(0,0);to=(4,20)]

out.write(”<HTML>\r\n<TITLE>JSP Date Example</TITLE>\r\n<BODY>\r\n <BIG>\r\n
Today’s date is “);

// end
// begin [file=”/date.jsp”;from=(4,23);to=(4,46)]

out.print( new java.util.Datex() );

// end

// HTML // begin [file=”/date.jsp”;from=(4,48);to=(8,0)]

out.write(“\r\n </BIG>\r\n</BODY>\r\n</HTML>\r\n”);

// end
```

As you can see, comments are inserted into the generated code to tie the Java code back to the original JSP code.

Session Tracking

It is similar to servlet session tracking. Take example as shoping cart prog.

Scope

The scope of JSP object- JavaBeans, and implicit objects

1 Pages scope

It is bounded to **javax.servlet.jsp.PageContext** and invoking **getAttribute()** on implicit request object.

2 Request scope

It is bounded to **javax.servlet.ServletRequest**

3 Session scope

It is bounded to **javax.servlet.jsp.PageContext** and invoking **setAttribute()** on implicit session object.

4 Application scope

It is bounded to **javax.servlet.ServletContext** and invoking **getAttribute()** on implicit application object.

EJB(Enterprise JavaBeans)

The Enterprise JavaBeans architecture is a component architecture for the development and deployment of component-based distributed business applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional, and multiuser secure. These applications may be written once, and then deployed on any server platform that supports the Enterprise JavaBeans specification.3

Enterprise JavaBeans (EJB) defines a server-side component model that allows business objects to be developed and moved from one brand of EJB container to another. A component (an enterprise bean) presents a simple programming model that allows the developer to focus on its business purpose. An EJB server is responsible for making the component a distributed object and for managing services such as transactions, persistence, concurrency, and security. In addition to defining the bean’s business logic, the developer defines the bean’s runtime attributes in a way that is similar to choosing the display properties of visual widgets. The transactional, persis tence, and security behaviors of a component can be defined by choosing from a list of properties. The end result is that Enterprise JavaBeans makes developing distributed component systems that are managed in a robust transactional environment much easier. For developers and corporate IT shops that have struggled with the complexities of delivering mission-critical, high-performance distributed systems using CORBA, DCOM, or Java RMI, Enterprise JavaBeans provides a far simpler and more productive platform on which to base development efforts.

Server-Side Components

A server-side component model may define an architecture for developing *distributed business objects*. They combine the accessibility of distributed object systems with the fluidity of objectified business logic. Server-side component models are used on the middle-tier application servers, which manage the components at runtime and make them available to remote clients. They provide a baseline of functionality that makes it easy to develop distributed business objects and assemble them into business solutions.

Server-side components can also be used to model other aspects of a business system, such as presentation and routing. The Java Servlet for example is a server-side component that is used to generate HTML and XML data for presentation layer of a three-tier architecture. The EJB 2.0 message-driven beans, which are discussed later, are aserver-side components that is used for routing asynchronous messages from one source to another.Server-side components, like other components, can be bought and sold as independent pieces of executable software.

There are three types of Enterprise Java Beans namely,

a) Session beans

b) Entity beans

c) Message-driven beans

Thus , we have two interfaces and a class in any EJB program. The **'Remote Interface'** for describing the job to be done. The **'Home Interface** ' for locating the manager for creating the bean

The **'bean' class** which provides the object for carrying out the job. EJB can be thought of as Enterprise-level RMI/RMI-IIOP. If the EJB is invoked by using a servlet and the end-user is using a browser , there are no questions about platform and language of the client, because the client for the EJB is only the servlet.

But, if the EJB is being invoked from a standalone program, the client may be a java-program or non-java program.If the client is not a java program, the question of lient's platform raises its head. So, once again we need the OMG-IDL for our ejb service. incorporates concepts of **RMI, Session-tracking servlet and plain Javabean.**

An instance of Entity bean can be thought of as a record(row) in a table of a relational database. Such **Entity beans** can be of two types.

a) **CMP** ....(EntityBean with **Container-Manged Persistence**)

b) **BMP .....(**EntityBean with **Bean-Managed Persistence)**

In **CMP**, we do not need to write any sql-realted code for **adding** a new record, **editing** a record or **deleting** a record. or **querying** a table. The creation of the table,and appropriate sql statements is automatically managed by the ejb server itself. This is not only very simple but is recommended as well because, it abstracts the details about the underlying database.

On the other hand, we also saw that a javabean may be simply a class with some general functionality. That part of the bean is offered as **'Session bean'** in EJB. Session beans can be either **Stateless** beans or **Stateful** beans.

'Stateless session beans' are very much like RMI.

'Stateful session beans' are like session-tracking servlets but in RMI style.!

We will now consider simplest demo for the following types., one by one.

a) stateless session bean

b) stateful session bean

c) container-managed Entity bean

d) bean managed entity bean

EXAMPLE FOR STATELESS SESSION BEAN

====================================

Reverting back to our greeter example, it is a typical candidate for 'Stateless Session bean'.

A stateless session bean is just some encapsulated functionality but invoked in RMI style.

***It should not have any general variables .All the data required for the method should be passed as parameters only!**

We will create three files .

a) The Remote-Interface

b) The Home-interface

c) The Bean.

We will follow the naming convention as:

a) greeterRemote.java

b) greeterHome.java

c) greeterBean.java

As in RMI, we should always begin with the remote-interface, then write the home-interface and only then write the implementation as a bean.

greeterRemote.java

================

```
import javax.ejb.*;

import java.rmi.*;

public interface greeterRemote  extends EJBObject

{

public String greetme (String s) throws RemoteException;

}
```

greeterHome.java

==============

```
import java.io.*;

import java.rmi.*;

import javax.ejb.*;

public intrerface greeterHome  extends  EJBHome

 {

    greeterRemote  create() throws RemoteException, CreateException;

 }
```

greeterBean.java

===============

```
  import java.rmi.*;

  import javax.ejb.*;

 public class greeterBean implements SessionBean
```

```
      {
         public String greetme(String  s)
          {
           return   "How are you?...." + s;
          }
     public greeterBean(  )     {   }
      public void ejbCreate(  )  {   }
      public void ejbRemove(  )  {   }
      public void ejbActivate(  ) {   }
      public void ejbPassivate(  ){   }
      public void setSessionContext(SessionContext  sc) {  }
 }
```

greeterClient.java (CONSOLE-MODE)

================================

```
import javax.ejb.*;
import java.rmi.*;
import javax.rmi.*;
import javax.naming.*;
import java.util.*;
import helloRemote;
import helloHome;
import java.io.*;
public class helloClient
{
 public static void main(String args[])
```

```
{
 try
  {
  System.out.println("please wait!");
 Properties   props = new Properties();
 props.put(Context.INITIAL_CONTEXT_FACTORY,
                 "weblogic.jndi.WLInitialContextFactory");
 props.put(Context.PROVIDER_URL,
                  "t3://127.0.0.1:7001");
   Context           ctx =             new InitialContext(props);
   greeterhome           home  = (greeterHome)ctx  .  lookup("helloJndi");
    greeterRemote    remote = home.create();
  String  s1= remote.greetme(args[0]);
  System.out.println(s1);
  }
 catch(Exception e1)
 {
 System.out.println(" " + e1);
 }
}
```

ENTITY BEANS

=============

Entity beans are characterized by the following 3 features.

a) They are **'Persistent'**. (they are stored in hard-disk)

b) They are **shared** by many clients.

c) They have , **'Primary key'**.

As already mentioned ,Entity beans can be thought of as a record (or row) in a table of a **relational** database. (This is just for easy understanding because, the database can also be **Object Database**, **XML** database etc.)

Let us consider a simple Java class named **'customer'**. Let this class have just three attributes ,namely,**'key'**, **'Name'** and **'Place'**. In a javabean, we would provide **accessor methods,** such as **'getName()'** & **'setName(**String s)etc. for each attribute. The same method is employed in Entity bean. (**Roughly**).

Thus, we deal with Java idiom only and this is more intuitive.

If we have an account bean, we can write code for deposit as follows:

```
 int  n  = account1.  getBalance();
n=n+400;
account1.setBalance(n);
```

Doubtless, this is much simpler and easier than writing sql code.

Entity beans **'persist'** (ie) they are stored in Enterprise server's **hard disk** and so even if there is a shutdown of the server, the beans 'survive' and can be created again from the hard disk storage.[A **session bean** is **not** stored in hard disk].

A **Session bean** , whether it is stateless or stateful is meant for a **single client.** But Entity bean , being a record in a table of a database, is likely to be accessed by a number of clients . So, they are typically **shared** by a number of clients. For the same reason, entity beans should work within

'Transaction'.management as specified. in the Deployment descriptor.

But, typically in an enterprise situation, a database will be accessed by thousands of clients concurrently, and the very rationale for the development of EJB is to tackle the problems which arise then. That is why, Entity beans are the correct choice for Enterprise situations.

If we think of an entity bean instance as a record in a table of database, it automatically follows that it should have **a primary key for unique identification** of the record..[Many books provide a 'primary key class'. But it is not atall necessary.] But carefully note that it should be a serializable java class. So, if we provide a primary key as **'int'** type, we will have to provide a wrapper class (ie) **Integer**. This is clumsy. The best and easiest method is to provide a string type as the primary key. (String class). This is the method that we will be following in our illustarations. So, in our example, we are having an Access database named **'customer'**.

This database has a table known as **'table1'**. The table has three columns .

a) **'key'** **(primary key field)**

b) **'name'**

c) **'place'**

(all of them are of **String** type)

We create a table like this without any entries and then register it in ODBC. (this is the most familiar and easy approach.We can also use other types of jdbc drivers.)

Entity beans can have two types of Persistence.

a) **Container-managed Persistence** **(CMP**)

b) **Bean-managed Persistence type** **(BMP)**

We can declare the type of persistence required by us in the 'Deployment Descriptor'.

In CMP, the bean designer does not have to write any sql-related code atall. The necessary sql statements are automatically generated by the container.

The container takes care of synchronizing the entity bean's attributes with the corresponding columns in the table of the database. Such variables are referred to as **'container-managed fields'.**

This requirement also is declared by us in the Deployment descriptor.

With CMP, the entity bean class does not contain the code that connects to a database. . So, we are able to get flexibility by simply editing the **deployment descriptors** and **weblogic.properties files**, without editing and recompiling the java class files.

What are the advantages of CMP?

CMP beans have two advantages:

i) less code.

ii) the code is independent of the type of data store such as Relational database.

What are the limitations of CMP?

If we want to have **complex joins between different tables**, CMP is not suitable. In such cases, we should use BMP .

EXAMPLE FOR CMP -ENTITY BEAN

================================

As before we begin with the Remote Interface file.

```
        //    **********customerRemote.java**********

import javax.ejb.*;

import java.rmi.*;

public interface customerRemote extends EJBObject

{

   public String  getName()     throws RemoteException;
```

```
    public void    setName(String   s)     throws RemoteException;

    public String  getPlace()    throws RemoteException;

    public  void   setPlace(String s) throws RemoteException;

}
```

Next we write the home interfcae.

```
            //     **********customerHome.java ******************

import javax.ejb.*;

import java.rmi.*;

public  interface   customerHome  extends  EJBHome

{

public  customerRemote  create(String a, String b, String c) throws

RemoteException, CreateException;

  public  customerRemote    findByPrimaryKey(String a) throws

  RemoteException, FinderException;

}
```

```
                          //   ********** customerBean.java**********

 import javax.ejb.*;

import java.rmi.*;

public       class       customerBean       implements          EntityBean

{

   public           String                key;

   public           String               name;
```

```
  public            String                place;

  public       String       getName()

      {

      return   name;

      }

      public      String        getPlace()

      {

      return   place;

      }

     //---------------------------

     public        void           setName(String b)

             {

             name=b;

             }

      public       void          setPlace(String c)

             {

             place=c;

             }

       //------------------------------

        public  String  ejbCreate(String  a, String  b, String  c)
throws CreateException

        {

          this.key  =  a;

          this.name=  b;

          this.place = c;
          return null;

        }

        public void ejbPostCreate(String a,String b,String c)throws
CreateException {}

        public  void   ejbActivate()      {}

        public  void   ejbPassivate()     {}

        public  void   ejbRemove()        {}

        public  void   ejbLoad()          {}

        public  void   ejbStore()         {}

        public void    setEntityContext(EntityContext ec)  { }

        public void    unsetEntityContext()              { }

}
```

We now create three xml files as given below.

1) **ejb-jar. xml**

2) weblogic-ejb-jar.xml

3) weblogic-cmp-rdbms-jar.xml

These three files are very important and should be created with utmost care. Remember that XML is case-sensitive and the DTD (Deployment descriptor) for each file expects the correct structure of the document. So type exactly as given.(No formatting..shown here for clarity only)

```
//  customerClient.java

import customerRemote;

import customerHome;

import javax.ejb.*;
import java.rmi.*;

import javax.rmi.*;

import javax.naming.*;

import java.util.*;

import java.io.*;

public class customerclient

{

 public static void main(String args[])

 {

   try

    {

    System.out.println("please wait!");

    Properties      props =         new Properties();

    props.put(Context.INITIAL_CONTEXT_FACTORY,

                 "weblogic.jndi.WLInitialContextFactory");

    props.put(Context.PROVIDER_URL,

                   "t3://127.0.0.1:7001");

    Context ctx = new InitialContext(props);

    customerHome home =

                (customerHome)ctx.lookup("customerJndi");

    DataInputStream  ins =

                 new DataInputStream(System.in);

    String s;

    do
```

```
{

System.out.println("add or find or update or delete?");

s=ins.readLine();

if(s.equals("add"))

{

 System.out.println("what key");

 String a=ins.readLine();

 System.out.println("what name");

 String b=ins.readLine();

 System.out.println("what place");

 String c=ins.readLine();

 home.create(a,b,c);

 System.out.println("record created");

 System.out.println("=============");

}

if(s.equals("find"))

{

  System.out.println("what key?");

  String pk=ins.readLine();

  customerRemote remote  =

                        home.findByPrimaryKey(pk);

 System.out.println(remote.getName());

 System.out.println(remote.getPlace());

 System.out.println("===============");
}
if(s.equals("update"))
 {
  System.out.println("what key?");
  String   pk=ins.readLine();
  customerRemote     remote=
                     home.findByPrimaryKey(pk);
  System.out.println("you want to change name or place?");
  String v=ins.readLine();
  if(v.equals("name"))
      {
  System.out.println("what new name?");
  String   b = ins.readLine();
  remote.setName(b);
      }
   if(v.equals("place"))
         {
          System.out.println("what new place?");
          String c = ins.readLine();
          remote.setPlace(c);
         }
 System.out.println("record updated!");
  System.out.println("==============");
 }
```

```
      if(s.equals("delete"))
        {
          System.out.println("what key?");
          String   pk=ins.readLine();
           customerRemote     remote=
                        home.findByPrimaryKey(pk);
           remote.remove();
        System.out.println("record removed");
        System.out.println("============");
        }
      }while(!s.equals("over"));
    }   catch(Exception e1){ System.out.println(""+e1); }
  }
}
```

ENTITY BEAN WITH BEAN-MANAGED PERSISTENCE

==

Thus far, we acquainted ourselves with **Sessionbeans** (both stateless & stateful) and also **CMP Entity bean**.In this instalment, we take up the most difficult type, (ie) **Bean-Managed Persistent Entity bean.(BMP)**

Example for **Bean-Managed Entity Bean**

====================================

We have to write the SQL code for Persistence ,ourselves, in the case of Bean-Managed persistence. In our example,If we consider a single instance of the bean, it exists as an object in **memory** with 3 **attributes** (key,name,place) At the same time, it should also be persisted as a **row**, with corresponding **fields**, in '**table1'** of 'customer' **database** in hard disk.They should match. This is known as **'Synchronization'**.

The mechanism by which Synchronization is achieved by various vendors may vary. It is known as **'Object-relational mapping'**.

Whether, it is CMP or BMP, the process of synchronising is done by the **container**.The only difference is that , in CMP , we do not write any **SQL** for Storing (memory to hard disk)or for 'Loading' (hard disk to memory).In BMP, we have to write the **SQL** for these tasks.

However, the client cannot explicitly , call either 'load' or 'store' functions. This job is taken care of by the Container.Depending on a number of factors such as **Transaction monitoring**, the container chooses the appropriate moment to synchronize the data in memory with the data in table of hard disk storage.

This process, is **transparent** ,(ie) the programmer need not know about it. The methods are **'callback'** methods. Such methods are called by the container by itself without explicit user program or interaction.

In CMP, we just deal with the objects in memory. The task of persistence is automatically done by the container, at the appropriate time. The container itself generates the required SQL code to perform this task. But, in BMP , we should write the necessary SQL code for Store and Load.

This is the difference between CMP and BMP.

Which is better? Opinions differ.One camp claims that CMP is better because,it abstracts the details about the underlying database and deals only with objects in memory.

The other camp feels that CMP is useful only for very simple cases and most of the real life applications are complex, requiring joins from different tables and CMP is not suitable for real life situations.

It will be safer for us to be conversant with BMP , in case it is needed by our application.

As before (as in the case of CMP), we begin with :

i) customerRemote.java

ii) customerHome.java

A DEMO FOR BEAN-MANAGED PERSISTENT ENTITY-BEAN

==

Access Database name : Customer

table name : table1

Three fields (key, name, place) (all String type).

(Primary key field is 'key').

For our example, remember to register the database 'customer' with ODBC.Let us create our source files in the following folder:

c:\weblogic\beans\bmpdemo

As before remember to set JAVA_HOME, WL_HOME.

Also, set path & classpath;

```
c:\weblogic\beans\bmpdemo>set JAVA_HOME=C:\JDK1.3
 ...bmpdemo>set WL_HOME=C:\WEBLOGIC
 ...bmpdemo>set path=c:\windows\command;
                 c:\jdk1.3\bin;
                 c:\weblogic\bin
 ...bmpdemo>set classpath=c:\weblogic\beans\bmpdemo;
              c:\weblogic\classes;
              c:\weblogic\lib\weblogicaux.jar;
```

customerRemote.java

==================

```
import javax.ejb.*;
import java.rmi.*;

public interface customerRemote extends EJBObject
{
    public String  getName() throws RemoteException;
    public void    setName(String s)throws RemoteException;
    public String  getPlace() throws RemoteException;
    public  void   setPlace(String s) throws RemoteException;
}
```

customerHome.java

=================

```
import javax.ejb.*;
import java.rmi.*;

public interface customerHome extends EJBHome
{

  public customerRemote create(String a, String b, String c) throws RemoteException,
CreateException;

  public customerRemote    findByPrimaryKey(String a) throws RemoteException,
FinderException;
}
```

We now take up customerBean.java

As this is a lenthy file, it is always good practice to list the sequence in which the functions appear in code file:(This is just for convenience in code-reading).

1) setName(

2) getName()

3) setPlace(

4) getPlace()

5) **getConnection ()   (this is defined by us)**

6) ejbCreate()

7) ejbFindByPrimaryKey()

8) ejbLoad()

9) ejbStore()

10) ejbRemove()

11) ejbPostCreate(

12) ejbSetEntityContext(

13) ejbUnsetEntityContext()

14) ejbActivate()

15) ejbPassivate()

It will necessary to give very brief note on the purpose of these various methods.

a) The important function is '**getConnection**'. This is written by us.The purpose is to to get connection to the database.

b) '**ejbCreate**' method is called when the client invokes 'create' method. Its purpose is to create a new object in memory with the parameters passed by the client and also create the corresponding row in the database table.

c) **'ejbPostCreate'** is called after 'ejbCreate' by the container.This can be used for any desired tasks.It should have the same parameters as the 'ejbCreate' method.

d) **'ejbLoad'** is meant for bringing a row from table into the memory.depending on the primary key in the current context.

(**context.getPrimaryKey()**).and setting the attributes of the object accordingly. This is a 'callback' method (ie) it is invoked at the appropriate time by the container.

e) **'ejbStore'** is meant for storing the state of the bean specified by the primary key in the current context and updating the row in the table accordingly.This also is a callback method.

f) **'ejbActivate'** is a callback method which brings a bean into the bean-pool.

g) **`ejbPassivate'** is a callback method which removes a bean from the bean-pool.

Remote Method Invocation (RMI)

Java is a distributed application.

<u>Distributed application</u>

A distributed application is an application whose processing is distributed across multiple computer networks.

Rmi Architecture

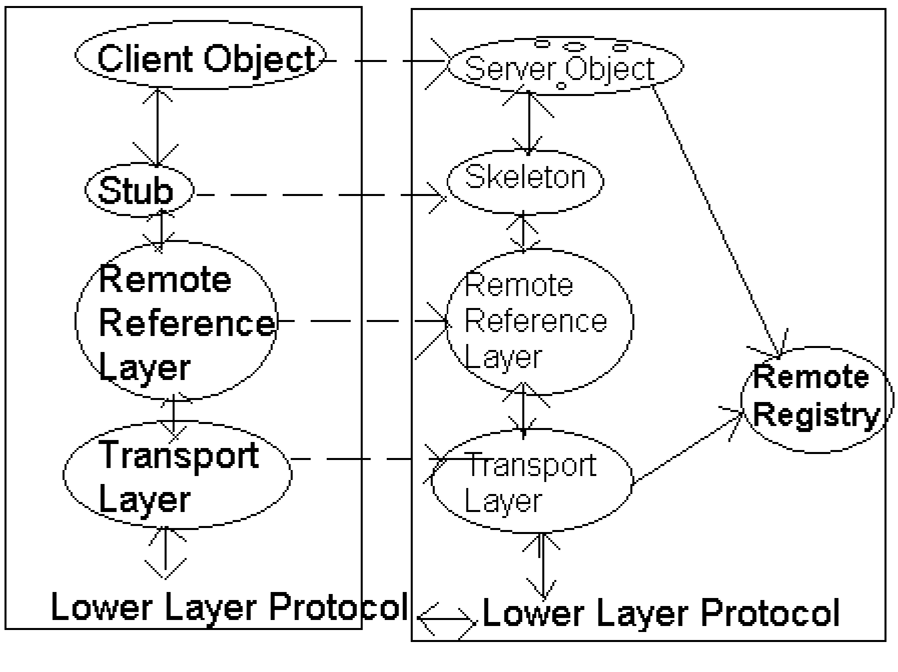


Figure 5.1 Rmi Architecture

Server object:

Server object is a object whose methods are invoked by client object.

This distributed object model used by java allows objects that execute in JVM to invoke the methods of object that execute in other JVMS.

These other JVMS may execute as separate process on the same computer or other remote computers. The object making the method invocation is referred to as client application. The object whose methods are being invoked is referred as server object. The client object is local object and remote object is called server object.

In this model a client object never references a remote object directly. Instead, it references a remote interface i.e., implemented by the remote object. The use of remote interface allows server objects to differentiate between their local and remote interfaces.

In addition to a remote interface this model makes use of stub and skeleton classes. Stub class serves as local proxies for the remote object skeleton acts a remote proxies. Both stub and skeleton implemented the remote interface the server object. The client interface object invokes the methods of the local stub object. The local stub communicates this method invocation to the skeleton via remote reference layer. The remote reference layer is used to active the server object when they are invoked remotely. The remote reference layer on the local host communicates with the remote reference layer on the remote host via the RMI transport layer. The transport layer sets up & manages the connection between the address spaces of the local & remote host, keeps track of object that can be accessed remotely & determines when connections have been timed out. The transport layer uses the TCP Sockets by default to communicate between local & remote host. The transport layer on the remote host identifies the server object instance by using remote registry it forwards method invocation to the skeleton via remote reference layer. Then the skeleton forwards this call to the server object. The server object process the function and results back to the client object via stub and skeletons.

The remote registry is process which maintains database objects and names by which these objects can be referenced.

Courses.java

```
import java.rmi.*;

public interface Courses extends Remote

{
 public String PGDCA() throws RemoteException;
}
//Client.java
import java.rmi.*;
import java.rmi.server.*;
class Client
{
 public static void main(String args[])
 {
  try
  {
   Courses c=(Courses)Naming.lookup("//Sunil/Courses");
   System.out.println("\n Subjects in PGDCA are : " + c.PGDCA());
  }
  catch(Exception e)
  {
   System.out.println(e);
  }
 }
}
//Server.java
import java.rmi.*;
import java.rmi.server.*;
public class SunilServer extends UnicastRemoteObject
```

```
   implements Courses
{
 SunilServer() throws RemoteException
 {
  super();
 }
 public String PGDCA() throws RemoteException
 {
  return "Ms-office\t C with DS, Java, VB, Oracle, Unix, Web Designing";
 }
 public static void main(String args[])
 {
  try
  {
   SunilServer gs=new SunilServer();
   Naming.rebind("//Sunil/Courses",gs);
   System.out.println("\n SunilServer is Ready.");
  }
  catch(Exception e)
  {   System.out.println(e);
  }
 }
}
```