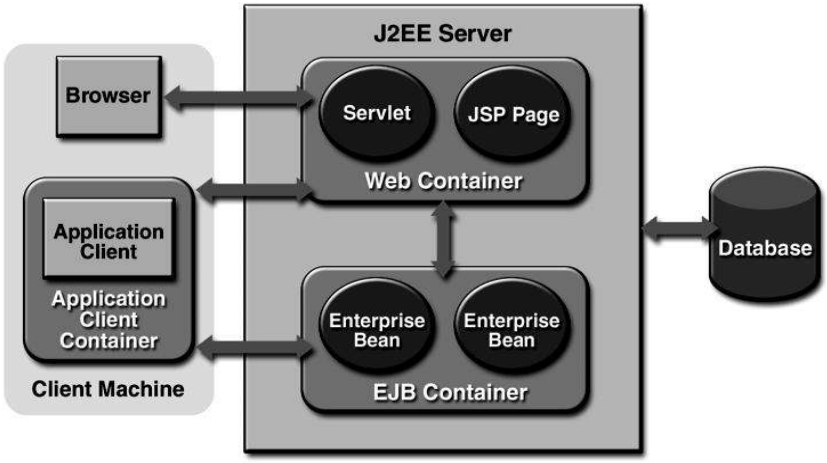
J2EE ARCHITECTURE

J2EE is a platform for building server-side applications. It contains one or more containers. A J2EE container is a runtime to manage application components developed using the API specifications, and to provide access to the J2EE APIs

J2EE Containers



There are 4 containers in the J2EE architecture.

1. **Web Container:** Manages the execution of JSP page and servlet components for J2EE applications. Web components and their container run on the J2EE server.
2. **EJB Container:** Manages the execution of enterprise beans for J2EE applications. Enterprise beans and their container run on the J2EE server.
3. **Applet Container**: Manages the execution of application client components. Application clients and their container run on the client.
4. **Application client container**: Manages the execution of applets. Consists of a Web browser and Java Plug-in running on the client together.

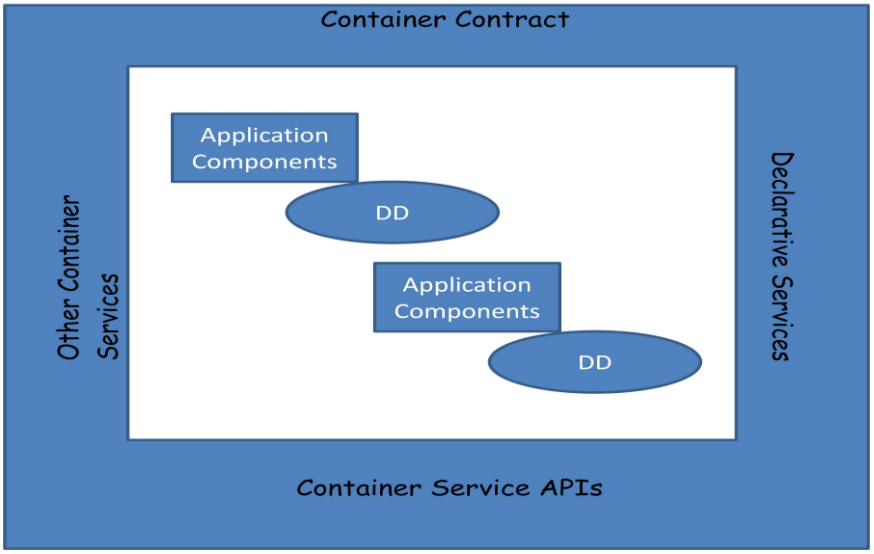
There are primarily two types of clients:

1. **Web clients:** A Web client consists of two parts: dynamic Web pages containing various types of markup language (HTML, XML, and so on), which are generated by Web components running in the Web tier, and a Web browser, which renders the pages received from the server.

   A Web client is sometimes called a *thin client*. Thin clients usually do not do things like query databases, execute complex business rules, or connect to legacy applications. When you use a thin client, heavyweight operations like these are off-loaded to enterprise beans executing on the J2EE server where they can leverage the security, speed, services, and reliability of J2EE server-side technologies.

2. **EJB clients:** These are applications that access the EJB components in EJB containers

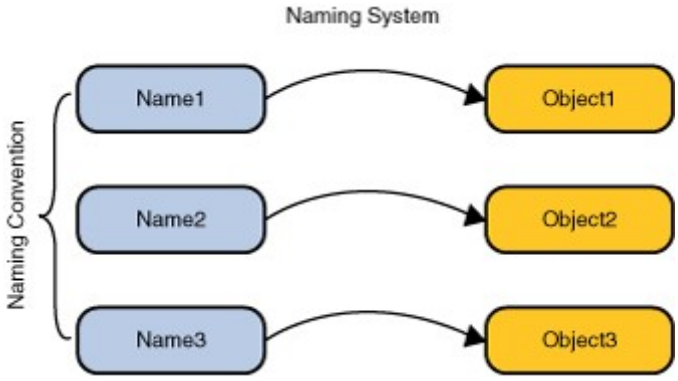
CONTAINER ARCHITECTURE



- Application Components: These include Servlets, JSP pages, EJBs, etc
- Deployment descriptors: It is an XML file that describes the application components.It also includes additional information required to effectively manage the application components.
- Component contract: A set of APIs specified by the container that ACs need to implement or extend
- Container services APIs: Additional services commonly required for all the applications in the container.
- Declarative services: Services that the container interposes on the application based on the DD such as security, transactions, etc
- Other container services: Life cycle management components, Resource pooling, garbage collection, etc

NAMING AND DIRECTORY CONCEPTS

Naming Concepts: A fundamental facility in any computing system is the *naming service*--the means by which names are associated with objects and objects are found based on their names. When using any computer program or system, we are always naming one object or another. For example, when you use an electronic mail system, we must provide the name of the recipient. To access a file in the computer, supply its name. A naming service allows us to look up an object given its name.



A naming service's primary function is to map people friendly names to objects, such as addresses, identifiers, or objects typically used by computer programs.

For example, the Internet Domain Name System (DNS) maps machine names to IP Addresses:
www.example.com ==> 192.0.2.5

A file system maps a filename to a file reference that a program can use to access the contents of the file.
c:\bin\autoexec.bat ==> File Reference

These two examples also illustrate the wide range of scale at which naming services exist--from naming an object on the Internet to naming a file on the local file system.

Names: To look up an object in a naming system, you supply it the *name* of the object. The naming system determines the syntax that the name must follow. This syntax is sometimes called the naming systems *naming convention*. A name is made up components. A name's representation consist of a component separator marking the components of the name.

Naming System	Component Separator	Names
UNIX file system	"/"	/usr/hello
DNS	"."	sales.Wiz.COM
LDAP	"," and "="	cn=Rosanna Lee, o=Sun, c=US

The UNIX file system's naming convention is that a file is named from its path relative to the root of the file system, with each component in the path separated from left to right using the forward slash character ("/"). The UNIX *pathname*, /usr/hello, for example, names a file hello in the file directory usr, which is located in the root of the file system.

DNS naming convention calls for components in the DNS name to be ordered from right to left and delimited by the dot character (".). Thus the DNS name sales.Wiz.COM names a DNS entry with the name sales, relative to the DNS entry Wiz.COM. The DNS entry Wiz.COM, in turn, names an entry with the name Wiz in the COM entry.

The Lightweight Directory Access Protocol (LDAP) naming convention orders components from right to left, delimited by the comma character (","). Thus the LDAP name cn=Rosanna Lee, o=Sun, c=US names an LDAP entry cn=Rosanna Lee, relative to the entry o=Sun, which in turn, is relative to c=us. LDAP has the further rule that each component of the name must be a name/value pair with the name and value separated by an equals character ("=").

Bindings: The association of a name with an object is called a *binding*. A file name is *bound* to a file.The DNS contains bindings that map machine names to IP addresses. An LDAP name is bound to an LDAP entry.

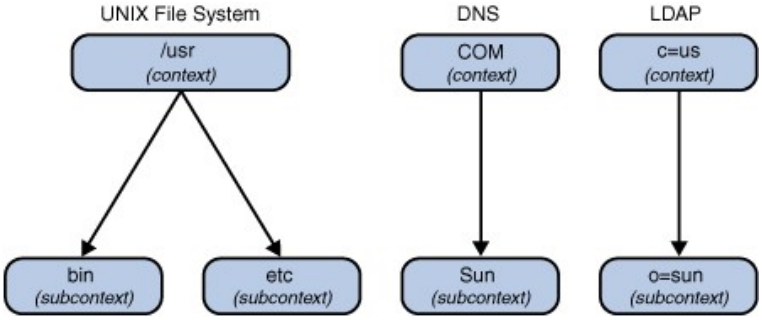
References and Addresses: Depending on the naming service, some objects cannot be stored directly by the naming service; that is, a copy of the object cannot be placed inside the naming service. Instead, they must be stored by reference; that is, a *pointer* or *reference* to the object is placed inside the naming service. A reference represents information about how to access an object. Typically, it is a compact representation that can be used to communicate with the object, while the object itself might contain more state information. Using the reference, you can contact the object and obtain more information about the object.

For example, an airplane object might contain a list of the airplane's passengers and crew, its flight plan, and fuel and instrument status, and its flight number and departure time. By contrast, an airplane object reference might contain only its flight number and departure time.

The reference is a much more compact representation of information about the airplane object and can be used to obtain additional information. A file object, for example, is accessed using a *file reference*. A printer object, for example, might contain the state of the printer, such as its current queue and the amount of paper in the paper tray. A printer object reference, on the other hand, might contain only information on how to reach the printer, such as its print server name and printing protocol.

Although in general a reference can contain any arbitrary information, it is useful to refer to its contents as *addresses* (or communication end points): specific information about how to access the object.

Context: A *context* is a set of name-to-object bindings. Every context has an associated naming convention. A context always provides a lookup (*resolution*) operation that returns the object, it typically also provides operations such as those for binding names, unbinding names, and listing bound names. A name in one context object can be bound to another context object (called a *subcontext*) that has the same naming convention.



A file directory, such as /usr, in the UNIX file system represents a context. A file directory named relative to another file directory represents a subcontext (UNIX users refer to this as a*subdirectory*). That is, in a file directory /usr/bin, the directory bin is a subcontext of usr.

A DNS domain, such as COM, represents a context. A DNS domain named relative to another DNS domain represents a subcontext. For the DNS domain Sun.COM, the DNS domain Sun is a subcontext of COM.

Finally, an LDAP entry, such as c=us, represents a context. An LDAP entry named relative to another LDAP entry represents a subcontext. For the LDAP entry o=sun,c=us, the entry o=sun is a subcontext of c=us.

Naming Systems and Namespaces: A *naming system* is a connected set of contexts of the same type (they have the same naming convention) and provides a common set of operations.
A system that implements the DNS is a naming system. A system that communicates using the LDAP is a naming system.

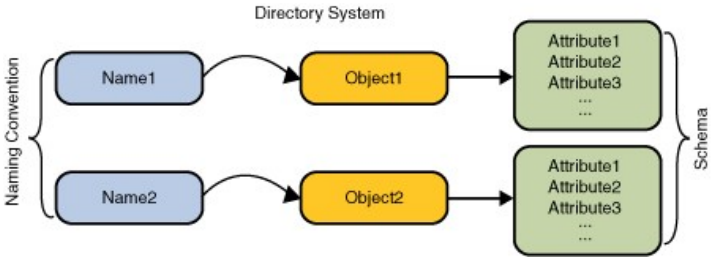
A naming system provides a *naming service* to its customers for performing naming-related operations. A naming service is accessed through its own interface. The DNS offers a naming service that maps machine names to IP addresses. LDAP offers a naming service that maps LDAP names to LDAP entries. A file system offers a naming service that maps filenames to files and directories.

A *namespace* is the set of all possible names in a naming system. The UNIX file system has a namespace consisting of all of the names of files and directories in that file system. The DNS namespace contains names of DNS domains and entries. The LDAP namespace contains names of LDAP entries.

Directory Concepts: Many naming services are extended with a *directory service*. A directory service associates names with objects and also associates such objects with *attributes*.

directory service = naming service + objects containing attributes

We not only can look up an object by its name but also get the object's attributes or *search* for the object based on its attributes.



An example is the telephone company's directory service. It maps a subscriber's name to his address and phone number. A computer's directory service is very much like a telephone company's directory service in

directory service is much more powerful, however, because it is available online and can be used to store a variety of information that can be utilized by users, programs, and even the computer itself and other computers.

A *directory object* represents an object in a computing environment. A directory object can be used, for example, to represent a printer, a person, a computer, or a network. A directory object contains *attributes* that describe the object that it represents.

Attributes: A directory object can have *attributes*. For example, a printer might be represented by a directory object that has as attributes its speed, resolution, and color. A user might be represented by a directory object that has as attributes the user's e-mail address, various telephone numbers, postal mail address, and computer account information.

An attribute has an *attribute identifier* and a set of *attribute values*. An attribute identifier is a token that identifies an attribute independent of its values. For example, two different computer accounts might have a "mail" attribute; "mail" is the attribute identifier. An attribute value is the contents of the attribute. The email address, for example, might have:
Attribute Identifier : Attribute Value
mail john.smith@example.com

Directories and Directory Services: A *directory* is a connected set of directory objects. A *directory service* is a service that provides operations for creating, adding, removing, and modifying the attributes associated with objects in a directory. The service is accessed through its own interface. Many examples of directory services are possible.

- **Network Information Service (NIS):** NIS is a directory service available on the UNIX operating system for storing system-related information, such as that relating to machines, networks, printers, and users.
- **Oracle Directory Server:** The Oracle Directory Server is a general-purpose directory service based on the Internet standard LDAP.

Search Service: We can look up a directory object by supplying its name to the directory service. Alternatively, many directories, such as those based on the LDAP, support the notion of *searches*. When you search, you can supply not a name but a *query* consisting of a logical expression in which you specify the attributes that the object or objects must have. The query is called a *search filter*. This style of searching is sometimes called *reverse lookup* or *content-based searching*. The directory service searches for and returns the objects that satisfy the search filter.

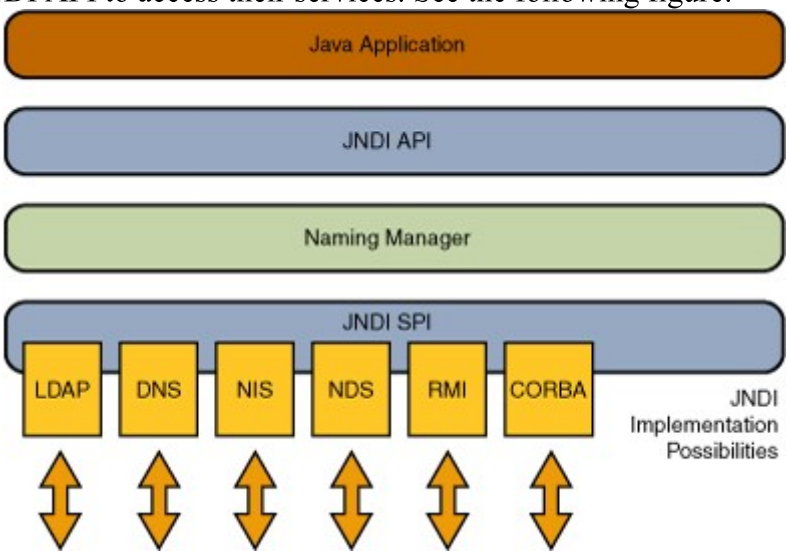
For example, you can query the directory service to find:
- all users that have the attribute "age" greater than 40 years.
- all machines whose IP address starts with "192.113.50".

Combining Naming and Directory Services: Directories often arrange their objects in a hierarchy. For example, the LDAP arranges all directory objects in a tree, called a *directory information tree (DIT)*. Within the DIT, an organization object, for example, might contain group objects that might in turn contain person objects. When directory objects are arranged in this way, they play the role of naming contexts in addition to that of containers of attributes.

OVERVIEW OF JNDI

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.

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. See the following figure:



Naming Services: In the above figure it shows that JNDI supports plug-in Service Providers for several well-known naming services, including the following:

- ***Lightweight Directory Access Protocol*** (LDAP) is the approved standard for an Internet naming service. LDAP is a true directory service and supports attributes as well as names for objects. LDAP is fast becoming the *de facto* directory service for the enterprise.
- ***Domain Name System*** (DNS) is the Internet naming service for identifying machines on a network.
- ***Novell Directory Services*** (NDS) from Novell provides information about network services, such as files and printers. NDS is found primarily in environments where the main networking software is Novell.
- ***Network Information Service*** (NIS) from Sun Microsystems provides system-wide information about machines, files, users, printers, and networks. NIS is primarily found on Solaris systems, but Linux and some other Unix platforms also support it.

JNDI also supports some more specialized naming systems. For example, CORBA for distributed component programming and RMI for distributed Java programming.

Although there is no named Service Provider for Windows Active Directory within JNDI, it is supported. Windows Active Directory supports an LDAP interface, and you can access it via the JNDI LDAP Service Provider Interface.

JNDI PACKAGES

The JNDI is divided into five packages:

- javax.naming
- javax.naming.directory
- javax.naming.ldap
- javax.naming.event
- javax.naming.spi

Naming Package(javax.naming): The javax.naming package contains classes and interfaces for accessing naming services.

- **Context:** The ***javax.naming*** package defines a Context interface, which is the core interface for

- **Lookup:**The most commonly used operation is lookup(). You supply lookup() the name of the object you want to look up, and it returns the object bound to that name.
- **Bindings:**listBindings() returns an enumeration of name-to-object bindings. A binding is a tuple containing the name of the bound object, the name of the object's class, and the object itself.
- **List:** list() is similar to listBindings(), except that it returns an enumeration of names containing an object's name and the name of the object's class. list() is useful for applications such as browsers that want to discover information about the objects bound within a context but that don't need all of the actual objects. Although listBindings() provides all of the same information, it is potentially a much more expensive operation.

- **Name:** Name is an interface that represents a generic name--an ordered sequence of zero or more components. The Naming Systems use this interface to define the names that follow its conventions.

- **References:** Objects are stored in naming and directory services in different ways. A reference might be a very compact representation of an object.

  The JNDI defines the Reference class to represent reference. A reference contains information on how to construct a copy of the object. The JNDI will attempt to turn references looked up from the directory into the Java objects that they represent so that JNDI clients have the illusion that what is stored in the directory are Java objects.

- **The Initial Context:** In the JNDI, all naming and directory operations are performed relative to a context. There are no absolute roots. Therefore the JNDI defines an InitialContext, which provides a starting point for naming and directory operations. Once you have an initial context, you can use it to look up other contexts and objects.
- **Exceptions:** The JNDI defines a class hierarchy for exceptions that can be thrown in the course of performing naming and directory operations. The root of this class hierarchy is NamingException. Programs interested in dealing with a particular exception can catch the corresponding subclass of the exception. Otherwise, they should catch NamingException.

Directory (javax.naming.directory)and LDAP Packages(javax.naming.ldap)

Directory Package: The javax.naming.directory package extends the javax.naming package to provide functionality for accessing directory services in addition to naming services. This package allows applications to retrieve associated with objects stored in the directory and to search for objects using specified attributes.

- **The Directory Context:** The ***DirContext*** interface represents a *directory context*. ***DirContext*** also behaves as a naming context by extending the Context interface. This means that any directory object can also provide a naming context. It defines methods for examining and updating attributes associated with a directory entry.
- **Attributes:** We use ***getAttributes()*** method to retrieve the attributes associated with a directory entry (for which we supply the name). Attributes are modified using ***modifyAttributes***() method. You can add, replace, or remove attributes and/or attribute values using this operation.
- **Searches:** DirContext contains methods for performing content based searching of the directory. In the simplest and most common form of usage, the application specifies a set of attributes possibly with specific values to match and submits this attribute set to the ***search***() method. Other overloaded forms of ***search( )*** support more sophisticated search filters.

LDAP Package: The ***javax.naming.ldap*** package contains classes and interfaces for using features that are specific to the LDAP v3 that are not already covered by the more generic javax.naming.directorypackage. In fact, most JNDI applications that use the LDAP will find the `javax.naming.directory` package sufficient and will not need to use the javax.naming.ldap package at all. This package is primarily for those applications that need to use "extended" operations, controls, or unsolicited notifications.

- **"Extended" Operation:** In addition to specifying well defined operations such as search and modify, the LDAP v3 (RFC 2251) specifies a way to transmit yet-to-be defined operations between the LDAP client and the server. These operations are called *"extended" operations*. An "extended" operation may be defined by a standards organization such as the Internet Engineering Task Force (IETF) or by a vendor.
- **Controls:** The LDAP v3 allows any request or response to be augmented by yet-to-be defined modifiers, called *controls* . A control sent with a request is a *request control* and a control sent with a response is a *response control* . A control may be defined by a standards organization such as the IETF or by a vendor. Request controls and response controls are not necessarily paired, that is, there need not be a response control for each request control sent, and vice versa.
- **Unsolicited Notifications:** In addition to the normal request/response style of interaction between the client and server, the LDAP v3 also specifies *unsolicited notifications*--messages that are sent from the server to the client asynchronously and not in response to any client request.
- **The LDAP Context:** The LDAPContext interface represents a *context* for performing "extended" operations, sending request controls, and receiving response controls.

Event and Service Provider Packages

Event Package: The javax.naming.event package contains classes and interfaces for supporting event notification in naming and directory services.

- **Events:** A NamingEvent represents an event that is generated by a naming/directory service. The event contains a *type* that identifies the type of event. For example, event types are categorized into those that affect the namespace, such as "object added," and those that do not, such as "object changed."

- **Listeners:** A NamingListener is an object that listens for NamingEvents. Each category of event type has a corresponding type of NamingListener. For example, a NamespaceChangeListener represents a listener interested in namespace change events and an ObjectChangeListener represents a listener interested in object change events.

  To receive event notifications, a listener must be registered with either an EventContext or an EventDirContext. Once registered, the listener will receive event notifications when the corresponding changes occur in the naming/directory service.

Service Provider Package: The javax.naming.spi package provides the means by which developers of different naming/directory service providers can develop and hook up their implementations so that the corresponding services are accessible from applications that use the JNDI.

- **Plug-In Architecture:** The javax.naming.spi package allows different implementations to be plugged in dynamically. These implementations include those for the initial context and for contexts that can be reached from the initial context.

- **Java Object Support:** The javax.naming.spi package supports implementors of lookup and related methods to return Java objects that are natural and intuitive for the Java programmer. For example, if you look up a printer name from the directory, then you likely would expect to get back a printer object on which to operate. This support is provided in the form of object factories.

  This package also provides support for doing the reverse. That is, implementors of Context.bind() and related methods can accept Java objects and store the objects in a format acceptable to the underlying naming/directory service. This support is provided in the form of state factories.

- **Multiple Naming Systems (Federation):** JNDI operations allow applications to supply names that span multiple naming systems. In the process of completing an operation, one service provider might

in the next naming system. This package provides support for different providers to cooperate to complete JNDI operations.

CREATING AN INITIAL CONTEXT

Before performing any operation on the naming or directory service, we need to acquire an initial context. The initial context is the starting point into the namespace. This is important because all methods on naming and directory services are performed relative to some context

To get an initial context, three steps are to be followed:

1. Select the service provider of the corresponding service you want to access.
2. Specify any configurations that the initial context needs.
3. Call the InitialContext constructor

1. **Select the Service Provider for the Initial Context :** We can specify the service provider to use for the initial context by creating a set of environment properties (a Hashtable) and adding the name of the service provider class to it.

   For example, if we are using the LDAP service provider from Sun Microsystems, then code would look like the following.
   Hashtable env = new Hashtable();
   env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");

   To specify the file system service provider from Sun Microsystems, you would write code that looks like the following.
   Hashtable env = new Hashtable();
   env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.fscontext.RefFSContextFactory");

2. **Specify any configurations that the initial context needs.**
   Clients of different directories might need various information for contacting the directory. For example, it might need to specify on which machine the server is running and what information is needed to identify the user to the directory. Such information is passed to the service provider via environment properties.

   The JNDI specifies some generic environment properties that service providers can use.

   For example, suppose that the program is using the LDAP service provider. This provider requires that the program specify the location of the LDAP server, as well as user identity information.

   To provide this information, we write code that looks as follows.

   env.put(Context.PROVIDER_URL, "ldap://ldap.wiz.com:389");
   env.put(Context.SECURITY_PRINCIPAL, "joeuser");
   env.put(Context.SECURITY_CREDENTIALS, "joepassword");

3. **Creating the Initial Context**
   To create the initial context, pass to the *InitialContext* constructor environment properties that we created previously:
   Context ctx = new InitialContext(env);
   Now we have a reference to a Context object, we can begin to access the naming service.

   To perform directory operations use an InitialDirContext. To do that, use one of its constructors:
   DirContext ctx = new InitialDirContext(env);

   This statement returns a reference to a DirContext object for performing directory operations.

NAMING OPERATIONS

The following naming operations can be performed by using the JNDI

1. Looking up an object
2. Listing the contents of a context
3. Adding, overwriting and removing a binding
4. Renaming an object
5. Creating and destroying subcontexts.

1. Looking up an object

To look up an object from the naming service, we use the Context.lookup() method.This method takes as parameter, the name of the object which we want to look up.

Object obj = ctx.lookup("report.txt");

This returns the report.txt object. It can be explicitly cast to a file as

```
Import  java.io.File;
File f = (File)ctx.lookup("report.txt");
```

2. Listing the contents of a context

Sometimes we would like to lookup an entire context instead of single objects. In such situations, we can use two methods: list() and listbindings(). They take as parameter the context name.
The context.list() method returns an enumeration of nameclasspair. Each nameclasspair consists of the object's name and its class name.

```
NamingEnumeration list = ctx.list("awt");
While(list.hasMore())
{
NameClassPair nc = (NameClassPair) list.next();
System.out.println(nc);
}
```

The Context.listBindings() returns an enumeration of binding. Binding is a subclass of NameClassPair. Binding contains not only the object's name and class name, but also the object.

```
NamingEnumeration bindings = ctx.listBindings("awt");
While(bindings.hasMore())
{
Binding bd =(Binding)bindings.next();
System.out.println(bd.getName()+ ":" + bd.getObject());
}
```

list( ) is intended for browser style applications. *listBindings( )* is intended for applications that need to perform operations on the objects in the context.

3. Adding, overwriting and removing a binding

The *Context.Bind*( ) method is used to add a binding to a context. It accepts as arguments the name of the object and the object to be bound.

```
Fruit fruit = new Fruit("orange");
Ctx.bind("favourite", fruit);
```

The *rebind*() method is used to add or replace a binding. It is similar to bind, but if the name is already bound, it will be unbound and the newly given object will be bound.

```
Fruit fruit = new Fruit("lemon");
Ctx.rebind("favourite", fruit);
```

The *unbind*( ) method is used to remove a binding.

```
Ctx.unbind("favourite");
```

4. **Renaming an object**

The *rename*() method is used to rename an object in the context.

```
Ctx.rename("report.txt", "old_report.txt");
```

5. **Creating and destroying subcontexts.**

The Context interface contains methods for creating and destroying a subcontext. A subcontext is a context bound in another context of the same type.

The methods for creating a subcontext is *createSubcontext( )*

```
Context result = new createsubcontext("new");
```

The method for destroying a subcontext is *Destroysubcontext()*

```
Ctx.destroysubcontext("new");
```

Remote Method Invocation (RMI)

INTRODUCTION:

RMI's purpose is to make objects in separate JVM's look alike and act like local objects. The JVM that calls the remote object is usually referred to as a client and the JVM that contains the remote object is the server.

One of the most important aspects of the RMI design is its intended transparency. Applications do not know whether an object is remote or local. A method invocation on the remote object has the same syntax as a method invocation on a local object, though under the hood there is a lot more going on.

In RMI the term "Server" does not refers to a physical server or application but to a single remote object having methods that can be remotely invoked. Similarly the Term "Client" does not refer to a client m/c but actually refers to the object invoking a remote method on a remote object.

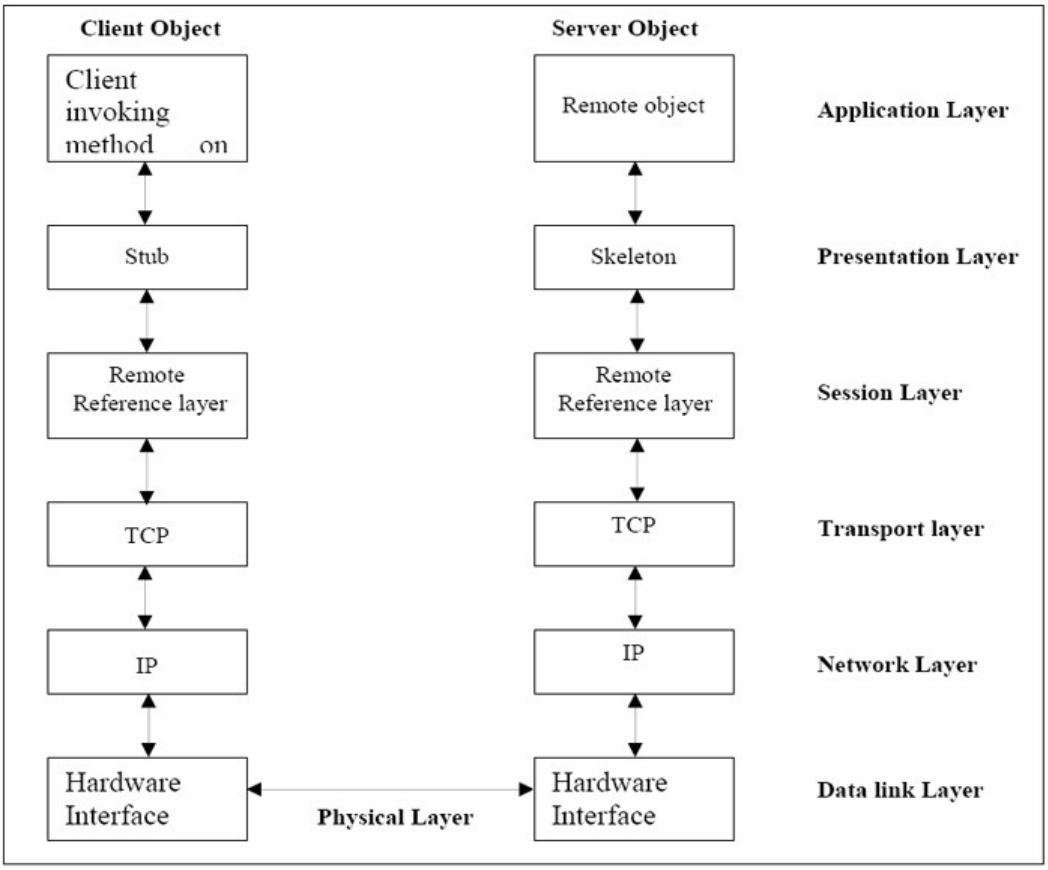
- The same object can be both a client and a server. Although obtaining a reference to a remote object is somewhat different from doing so for local objects. Once we have the reference, we use the remote object as if it were local. The RMI infrastructure will automatically intercept the method call, find the remote object and process the request remotely. This location transparency even includes garbage collection.
- A remote object is always accessed via its remote interface. In other words the client invokes methods on the object only after casting the reference to the remote interface.

The RMI implementation is essentially built from three abstraction layers:

- The Stub/Skeleton Layer
- The Remote Reference Layer
- The Transport Layer

RMI ARCHITECTURE

The following diagram shows the RMI Architecture:



RMI Architecture

The Stub/Skeleton Layer: This layer intercepts method calls made by the client to the interface reference and redirects these calls to a remote object. Stubs are specific to the client side, whereas skeletons are found on the server side.

To achieve location transparency, RMI introduces two special kinds of objects known as stubs and skeletons that serve as an interface between an application and rest of the RMI system. This Layer's purpose is to transfer data to the Remote Reference Layer via marshalling and unmarshalling.

Marshalling refers to the process of converting the data or object being transferred into a byte stream and unmarshalling is the reverse – converting the stream into an object or data. This conversion is achieved via object serialization.

The Stub/ Skeleton layer of the RMI lies just below the actual application and is based on the proxy design pattern. In the RMI use of the proxy pattern, the stub class plays the role of the proxy for the remote service implementation.

The skeleton is a helper class that is generated by RMI to help the object communicate with the stub; it reads the parameters for the method call from the link, makes the call to the remote service implementation object, accepts the return value and then writes the return value back to the stub.

In short, the proxy pattern forces method calls to occur through a proxy that acts as a surrogate, delegating all calls to the actual object in a manner transparent to the original caller.

Stub: The stub is a client-side object that represents (or acts as a proxy for) the remote object. The stub has the same interface, or list of methods, as the remote object. However when the client calls a stub method, the stub forwards the request via the RMI infrastructure to the remote object (via the skeleton), which actually executes it.

Sequence of events performed by the stub:

- Initiates a connection with the remote VM containing the remote object
- Marshals (writes and transmits) the parameters to the remote VM.
- Waits for the result of the method invocation.
- Unmarshals (reads) the return value or exception returned.
- Return the value to the caller

In the remote VM, each remote object may have a corresponding skeleton.

Skeleton: On the server side, the skeleton object takes care of all the details of "remoteness" so that the actual remote object does not need to worry about them. In other words we can pretty much code a remote object the same way as if it were local; the skeleton insulates the remote object from the RMI infrastructure.

Sequence of events performed by the skeleton:

- Unmarshals (reads) the parameters for the remote method (remember that these were marshaled by the stub on the client side)
- Invokes the method on the actual remote object implementation.
- Marshals (writes and transmits) the result (return value or exception) to the caller (which is then unmarshalled by the stub)

The Remote Reference Layer: The remote reference layer defines and supports the invocation semantics of the RMI connection. This layer maintains the session during the method call.

The Transport Layer: The Transport layer makes the stream-based network connections over TCP/IP between the JVMs, and is responsible for setting and managing those connections. Even if two JVMs are running on the same physical computer, they connect through their host computers TCP/IP network protocol stack. RMI uses a protocol called JRMP (Java Remote Method Protocol) on top of TCP/IP (an analogy is HTTP over TCP/IP).

From the JDK 1.2 the JRMP protocol was modified to eliminate the need for skeletons and instead use reflection to make connections to the remote services objects. Thus we only need to generate stub classes in system implementation compatible with jdk 1.2 and above. To generate stubs we use the Version 1.2 option with rmic.

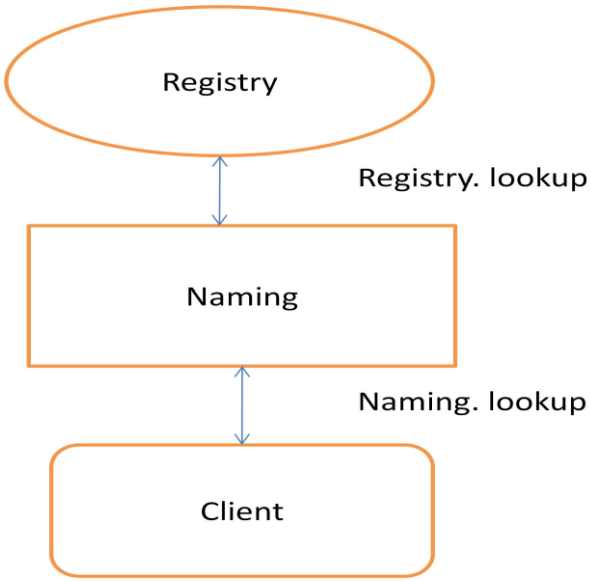
The RMI transport layer is designed to make a connection between clients and server, even in the face of networking obstacles. The transport layer in the current RMI implementation is TCP-based, but again a UDPbased transport layer could be substituted in a different implementation.

LOCATING REMOTE OBJECTS

Clients find remote services by using a naming or directory service. The host system and port no. of the service is known to the client. The RMI naming service, a registry, is a remote object that serves as a directory service for clients by keeping a Hashtable like mapping of names to other remote objects.

The behaviour of the registry is defined by the interface ***java.rmi.registry.Registry***. RMI itself includes a simple implementation of this interface called the RMI registry. Its default port is 1099.

A remote object is associated with a name in the registry. The client obtains a reference to a remote object by looking up its name in the registry. The lookup returns a stub to the object. RMI also provides another class called ***java.rmi.Naming*** to interact with the object serving as the registry.



VARIOUS METHODS USED IN RMI APPLICATIONS

- Public static void bind(String name, Remote obj)
- Public static String[] list(String name)
- Public static Remote lookup(String name)
- Public static void rebind(String name, Remote obj)
- Public static void unbind(String name)

The methods in the Naming and Registry interface have identical signatures and throw a variety of exceptions.

When the client first invokes a lookup for a particular URL in the naming class, a socket connection is opened to the host on the specified port. Next a stub to the registry is returned from the host. Using this stub, a registry lookup is performed for the remote object. The stub for the remote object is returned. The client then directly interacts with the remote object on the port to which it was exported.

The url takes the form: *Rmi://<host_name>[:<name_service_port]/ <service_name/>*

To facilitate this, the server program performs the following:

1. Creates a local object
2. Exports that object to create a listening service, that waits for clients to connect and request the service
3. Registers the object in the rmi registry under a public name.

Hierarchy for Package java.rmi

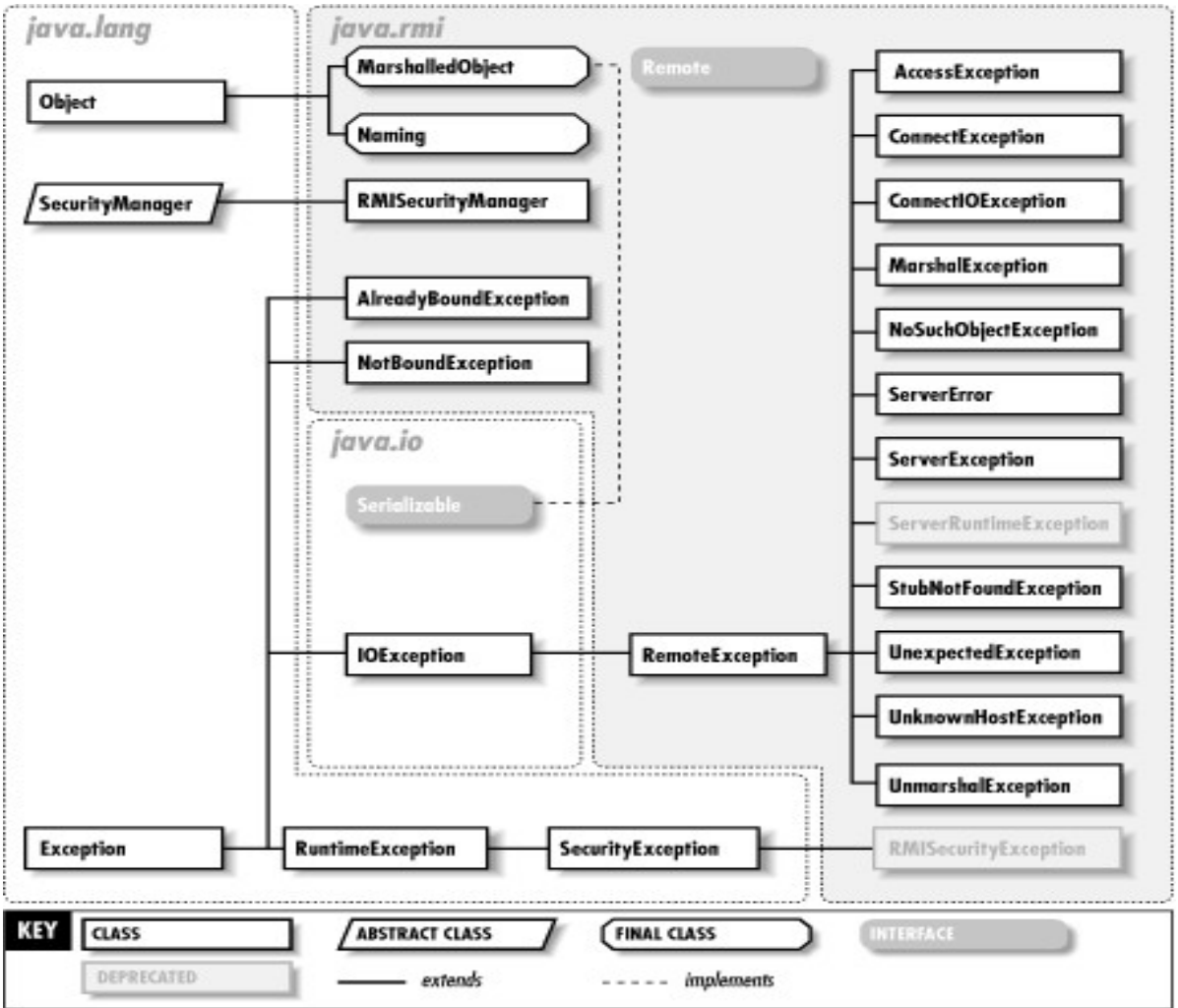
Class Hierarchy

- java.lang.**Object**
  - java.rmi.**MarshalledObject**<T> (implements java.io.Serializable)
  - java.rmi.**Naming**
  - java.lang.**SecurityManager**
    - java.rmi.**RMISecurityManager**
  - java.lang.**Throwable** (implements java.io.Serializable)

- java.lang.**Exception**
  - java.rmi.**AlreadyBoundException**
  - java.io.**IOException**
    - java.rmi.**RemoteException**
      - java.rmi.**AccessException**
      - java.rmi.**ConnectException**
      - java.rmi.**ConnectIOException**
      - java.rmi.**MarshalException**
      - java.rmi.**NoSuchObjectException**
      - java.rmi.**ServerError**
      - java.rmi.**ServerException**
      - java.rmi.**ServerRuntimeException**
      - java.rmi.**StubNotFoundException**
      - java.rmi.**UnexpectedException**
      - java.rmi.**UnknownHostException**
      - java.rmi.**UnmarshalException**
  - java.rmi.**NotBoundException**
  - java.lang.**RuntimeException**
    - java.lang.**SecurityException**
      - java.rmi.**RMISecurityException**

Interface Hierarchy

- java.rmi.**Remote**



RMI EXCEPTION CLASSES LIST

Exception	Description
AccessException	An AccessException is thrown by certain methods of the java.rmi.Naming class (specifically bind, rebind, and unbind) and methods of the java.rmi.activation.ActivationSystem interface to indicate that the caller does not have permission to perform the action requested by the method call.
AlreadyBoundException	An AlreadyBoundException is thrown if an attempt is made to bind an object in the registry to a name that already has an associated binding.
ConnectException	A ConnectException is thrown if a connection is refused to the remote host for a remote method call.
ConnectIOException	A ConnectIOException is thrown if an IOException occurs while making a connection to the remote host for a remote method call.
MarshalException	A MarshalException is thrown if a java.io.IOException occurs while marshalling the remote call header, arguments or return value for a remote method call.
NoSuchObjectException	A NoSuchObjectException is thrown if an attempt is made to invoke a method on an object that no longer exists in the remote virtual machine.
NotBoundException	A NotBoundException is thrown if an attempt is made to lookup or unbind in the registry a name that has no associated binding.
RemoteException	A RemoteException is the common superclass for a number of communication-related exceptions that may occur during the execution of a remote method call.
RMISecurityException	Deprecated *Use **SecurityException** instead.*
ServerError	A ServerError is thrown as a result of a remote method invocation when an Error is thrown while processing the invocation on the server, either while unmarshalling the arguments, executing the remote method itself, or marshalling the return value.
ServerException	A ServerException is thrown as a result of a remote method invocation when a RemoteException is thrown while processing the invocation on the server, either while unmarshalling the arguments or executing the remote method itself.
ServerRuntimeException	Deprecated *no replacement*
StubNotFoundException	A StubNotFoundException is thrown if a valid stub class could not be found for a remote object when it is exported.
UnexpectedException	An UnexpectedException is thrown if the client of a remote method call receives, as a result of the call, a checked exception that is not among the checked exception types declared in the throws clause of the method in the remote interface.
UnknownHostException	An UnknownHostException is thrown if a java.net.UnknownHostException occurs while creating a connection to the remote host for a remote method call.
UnmarshalException	An UnmarshalException can be thrown while unmarshalling the parameters or results of a remote method call if any of the following conditions occur: if an exception occurs while unmarshalling the call header if the protocol for the return value is invalid if a java.io.IOException occurs unmarshalling parameters (on the server side) or the return value (on the client side).

DEVELOPING APPLICATIONS WITH RMI

1. Defining a remote interface

A remote interface by definition is the set of methods that can be invoked remotely by a client.

1. The remote interface must be declared public
2. It must extend the *java.rmi.Remote* interface
3. Each method must throw a *java.rmi.RemoteException.*
4. If the remote methods have any remote objects as parameters or return types, they must be interface types not the implementation classes.

Note: java.rmi.remote is a marker interface i.e., it does not have any methods.

Example:

Public interface hellointerface extends java.rmi.Remote {

Public string sayhello( ) throws java.rmi.RemoteException }

2. Implementing the remote interface

The implementation class provides the actual implementation of the remote interface. The *java.rmi.server.RemoteObject* provides this functionality by overriding the *equals*(), *hashCode*() and the *toString*() methods of the *java.lang.Object* class

The abstract subclass *java.rmi.RemoteServer* describes the behaviour associated with the server implementation and provides the basic semantics to support remote references.

java.rmi.RemoteServer has two subclasses:

- ***java.rmi.server.UnicastRemoteObject:*** it defines a non replicated remote object whose references are valid only while the server process is alive.
- ***java.rmi.activation.Activatable:*** it is a concrete class that defines behaviour for on demand instantiation of remote objects.
- The class extends ***java.rmi.RemoteServer*** or any of its subclasses. But the superclass constructor must be called so that it can be exported.
- The class explicitly exports itself by passing itself ("this") to different forms of *UnicastRemoteObject.exportObject( )* methods

Example:

```
import java.io.*;
Import java.rmi.*;
Import java.rmi.server.*;
Import java.util.Date;
Public class HelloServer extends UnicastRemoteObject implements HelloInterface {
Public HelloServer() throws RemoteException {
Super();
}
Public String sayHello() throws RemoteException {
Return "Hello world, the current system time is "+new Date();
}
}
```

3. **Writing the client that uses the remote objects**

The client performs a lookup on the registry on the host and obtains a reference to the remote object.

```
import java.rmi.*;
public class HelloClient {
public static void main(String[] args)
{
if (System.getSecurityManager()==null) {
System.setSecurityManager (new RMISecurityManager ());
}
Try {
HelloInterface obj=( HelloInterface)Naming.lookup("/HelloServer");
String message=obj.sayHello();
System.out.println(message);
} catch(Exception e) {
System.out.println("helloclient exception"+e);
} } } }
```

4. **Generating the stubs and skeletons**

The stubs and skeletons can be generated with *rmic* tool after the compilation of the classes

Note: The classpath must be set

Type the following line in the command prompt

rmic –v1.2 HelloServer

5. **Starting the registry and registering the object**

The object has to be made available to clients by binding it to a registry. The stubs and skeletons are needed for registration. There are two methods in java.rmi.Naming that bind an object in the registry bind( ) and rebind the object in the registry rebind( ).

The registry must be running for the object to bind.

Rmiregistry –J-Djava.security.policy=registerit.policy

Example:

```
import java.rmi.*;
public class RegisterIt
{
public static void main(String[] args)
{
try {
HelloServer obj=new HelloServer();
System.out println("Object instantiated: +obj);
Naming.rebind("/HelloServer,obj);
System.out.println("HelloServer bound in registry");
}catch(Exception e) {
System.out.println(e);
}}}
```

6. Running the server and client.

To run the client, we should first compile and run the ***RegisterIt.java*** file .Open another command window which runs the RMI registry. A third window runs the client ***HelloClient*** as *Java -Djava.security.policy=registerit.policy.HelloClient*

Java Database Connectivity(JDBC)

Introduction:

Java facilitates client-server computing and allows it to integrate with popular commercial DBMS. Java Database Connectivity (JDBC) provides a program level interface for communicating with databases using Java Programming language. A relational database is usually the primary data resource in an enterprise application. JDBC is a part of JDK software so no need to install separate software for JDBCAPI.

Using the JDBC API, developers can create a client that can connect to a database, execute SQL statements and process the result of those statements.
Database applications consist of several components such as:

- Presentation logic
- Data access logic
- Business logic

In client-server architecture these tasks can be distributed between client and server in a number of ways.

- **2-Tier architecture:** Presentation and Business logic at the client side. Data access logic at the server side. It is also called Thick Client
- **3-Tier architecture:** Presentation logic at the client side. Business logic and Data access logic at the Application server. The DBMS may reside on a third system called the Database Server. It is also called Thin Client.

JDBC ARCHITECTURE

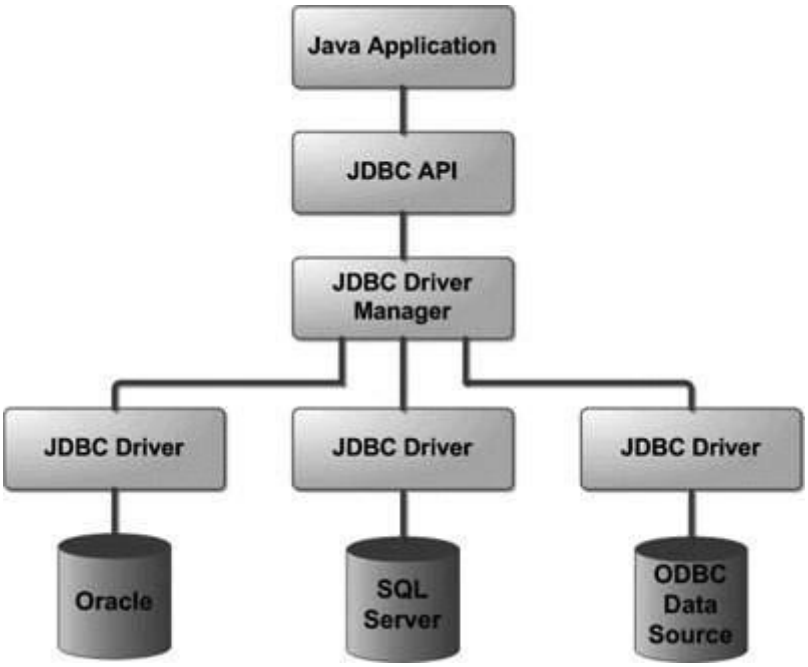
The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers −

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

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

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

The figure shown below is the architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application



Common JDBC Components

The JDBC API provides the following interfaces and classes −

- **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.

- **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.

- **Connection:** This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.

- **Statement:** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.

- **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.

- **SQLException:** This class handles any errors that occur in a database application.

JDBC API

JDBCAPI is a java API that can access any kind of tabular data and specific data stored in relational database management system (RDBMS). JDBC API consists of two packages

Packages

- java.sql package
- javax.sql package

Classes in JDBC

- DriverManager
- SQLException
- Types

- Time

Interfaces in JDBC

- Connection
- Statement
- PreparedStatement
- CallableStatementResultset
- ResultSetMetaData
- DatabaseMetaData
- Driver
- Blob
- Clob

Note: All JDBC Interfaces are implemented in JDBC Driver

Data Source Name: DSN (Data Source Name) is a file which stores a database name. An ODBC driver uses a DSN file and reads a database name then connects with that database.

Types of DSN

1. System DSN: It will be stored in a sharable location of registry and this type of DSN is accessible for multiple user accounts of the system.
2. User DSN: It will be stored in a non-sharable location of registry and this type of DSN is only accessible for one user account of the system.
3. File DSN: It is like user DSN, but the DSN will be stored in a user given location of hard disk.

STEPS TO WRITE JDBC PROGRAM

There are 6 steps to connect any java application with the database using JDBC. They are as follows:

1. Load the JDBCdriver class or register the JDBCdriver.
2. Establish the connection
3. Create a statement
4. Execute the sql commands on database and get the result
5. Print the result
6. Close the connection

1. **Register the driver class:** In this step we load the JDBC driver class into JVM. This step is also called as registering the JDBC driver. The *forName*() method of ***Class*** class is used to register the driver class. This method is used to dynamically load the driver class. This step can be completed in two ways.
   - Class.forName("fully qualified classname")
   - DriveManager.registerDriver(object of driver class)

Syntax of forName() method

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

Sun.Jdbc.Odbc.JdbcOdbcDriver is a driver class provided by Sun MicroSystem and it can be loaded into JVM like the following.

Syntax

```
class.forName("Sun.Jdbc.Odbc.JdbcOdbcDriver");
```

Syntax

```
Sun.Jdbc.Odbc.JdbcOdbcDriver jod=new Sun.Jdbc.Odbc.JdbcOdbcDriver();
DriverManager.registerDriver(jod);
```

2. **Create the connection object:** In this step connection between a java program and a database will be opened. To open the connection, we call *getConnection*() method of ***DriverManager*** class. For *getConnection*() method we need to pass three parameters.
   - **url:** url is used to select one register JDBCdriver among multiple registered driver by DriverManager class.
   - **username and password:** username and password are used for authentication purpose.

Syntax of getConnection() method

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

Example to establish connection with the Oracle database

Connection con=**new** DriverManager.getConnection(url, username, password);
Example:
Connection con=**new** DriverManager.getConnection(Jdbc:Odbc:< dsn >", "scott","tiger");

3. **Create the Statement object :** To transfer SQL commands from java program to database we need statement object. To create a statement object we call *createStatement*() method of connection interface. The *createStatement*( ) method of ***Connection*** interface is used to create statement. The object of statement is responsible to execute queries with the database.

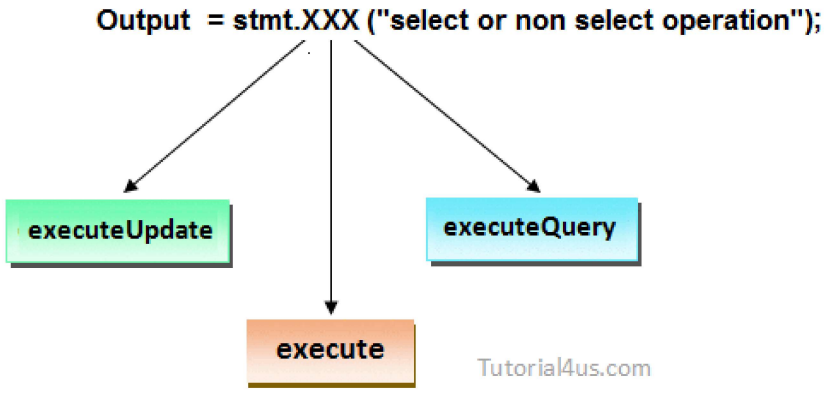
Syntax of createStatement() method

public Statement createStatement()**throws** SQLException

Example to create the statement object

Statement stmt=**new** createStatement();

4. **Executing queries:** In this step we call any one of the following three methods of ***Statement*** interface is used to execute queries to the database and to get the output.
   - **executeUpdate( ):** Used for non-select operations.
   - **executeQuery( ):** Used for select operation.
   - **execute( ):** Used for both select or non-select operation.



5. **Print the result:** In this step we print the retrieved results using ***System.out*** class *println*( ) method.

Syntax

System.**out**.println(output);

6. **Closing connection:** Close the connection. By closing connection object statement and ***ResultSet*** will be closed automatically. The *close*( ) method of ***Connection*** interface is used to close the connection.

Syntax of close() method

```
public void close()throws SQLException
```

Example for close connection

```
con.close();
```

Example

```
import java.sql.*;
class CreateTable
{
public static void main(String[] args) throws Exception
{
//step-1

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
System.out.println("driver is laoded");

//step-2
Connection con=DriverManager.getConnection("jdbc:odbc:ramadsn","system","system");
System.out.println("connection is established");

//step-3
Statement stmt=con.createStatement();
System.out.println("statement object is cretaed");

//step-4
int i=stmt.executeUpdate("create table student(sid number(3),sname varchar2(10),marks number(5))");
//step-5

System.out.println("Result is="+i);
System.out.println("table is created");

//step-6
stmt.close();
con .close();
}
}
```

JDBC DRIVER

JDBCAPI contains a set of classes and Interfaces where classes definition is provided by Sun Micro System and Interfaces Implementation are not provided. For interface implementation classes will be provided by vendors, these set of classes are called **JDBCDriver Software**.

A Driver software contains set of classes among these classes one class is driver class. Driver class is a mediator class between a java application and a database.

Java Application use JDBCAPI and this API connect to driver class, then driver class connect to database.

Types of JDBC Drivers

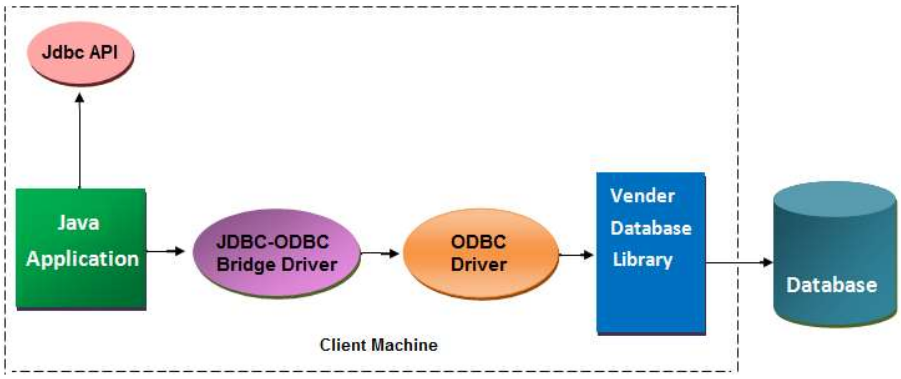
JDBC Driver is a software component that enables java application to communicate with the database. There are 4 types of JDBC drivers, they are:

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)

Short Description of JDBCdrivers

- Type 1 driver are database independent and platform dependent.
- Type 2 driver are database dependent and platform dependent.
- Type 3 driver are database independent and platform independent.
- Type 4 driver are database dependent and platform independent.

1. **JDBCodbc bridge driver:** This driver connect a java program with a database using ODBC driver. It is install automatically along with JDK software. It is provided by Sun MicroSystem for testing purpose this driver cannot be used in real time application. This driver convert JDBC calls into ODBC calls(function) So this is called a **bridge driver**.



JDBC-ODBC bridge driver

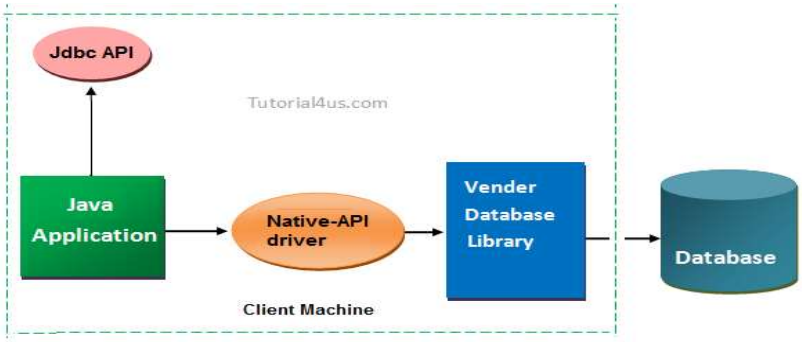
Advantage of bridge driver

- Easy to use
- It is a database independent driver
- Can be easily connected to any database.
- This driver software is built-in with JDK so no need to install separately.

Disadvantage of bridge driver

- It is a slow driver so not used in real time application
- Because of ODBC connectivity it is a platform dependent driver.
- It is not a portable driver.
- It is not suitable for applet to connect with database.

2. **Native API Driver:** Native API driver uses native API to connect a java program directly to the database. Native API id s C, C++ library, which contains a set of function used to connect with database directly. Native API will be different from one database to another database. So this Native API driver is a database dependent driver.



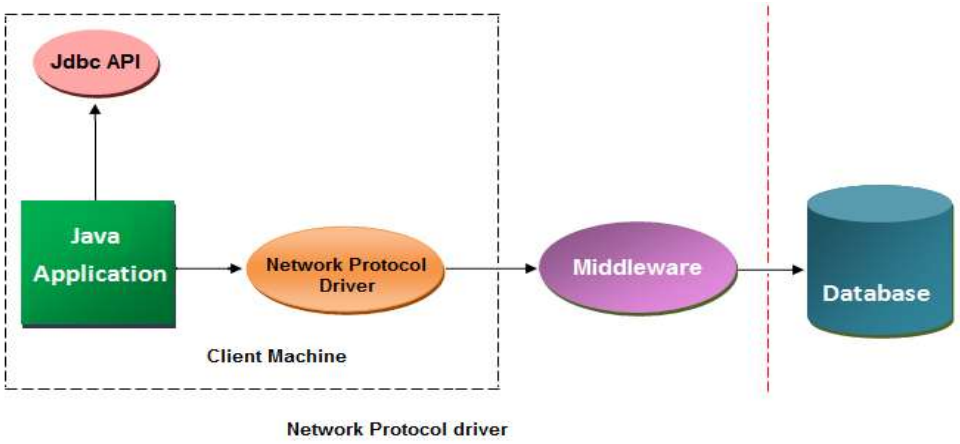
Advantage of Thin driver

- Native API driver comparatively faster than JDBC-ODBC bridge driver.

Disadvantage of Thin driver

- Native API driver is database dependent and also platform dependent because of Native API.

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



Network Protocol driver

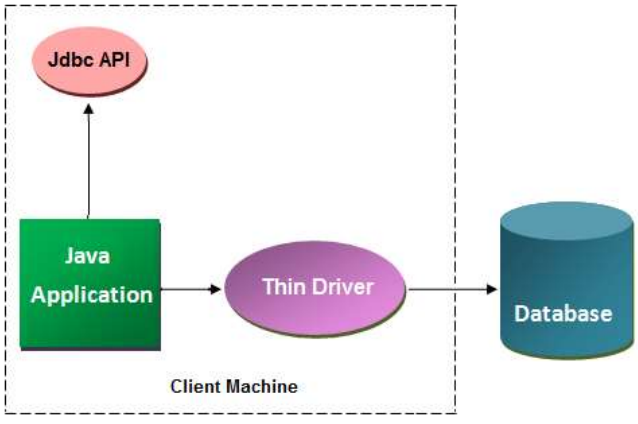
Advantage of Network Protocol driver

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.
- This driver is both database and platform independent driver

Disadvantage of Network Protocol driver

- 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.

4. **Thin 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. This thin driver uses the following three information to connect with a database.
   - Ip address of a machine (system), where the database server is running.
   - Port number of the database server.
   - Database name, also called SID (service ID).



Thin driver

Advantage of Thin driver

- Thin driver is the fastest driver among all JDBC drivers.
- No software is required at client side or server side.
- It is portable driver because it is platform independent.
- It can be used to connect an applet with the database.

Disadvantage of Thin driver

Thin driver connect with database

Oracle Corporation provides two JDBC drivers software for connecting a java application to a database of oracle server.

- Oracle oci driver.
- Oracle thin driver.

The following are the connection properties of oracle thin driver.

driver name:	**Oracle.Jdbc.OracleDriver**
url:	Jdbc:Oracle:thin@ipaddress:sid
name	System
password	Tiger

CONNECT JAVA WITH ORACLE DATABASE

Oracle Corporation has provided two JDBC driver software for connection java application to a database of oracle server.

- Oracle oci driver.
- Oracle thin driver.

For connecting java application with the oracle database, we need to follow below steps. In this example we are using Oracle 10g as the database. So we need to know following information for the oracle database.

The following are the connection properties of oracle thin driver.

Driver Class:	Oracle.Jdbc.OracleDriver
Connection url:	jdbc:oracle:thin:@localhost:1521:xe
Username:	The default username for the oracle database is system.
Password	Password is given by the user at the time of installing the oracle database.

jdbc:oracle:thin:@localhost:1521:x

- **JDBC**is the API
- **oracle** is the database
- **thin** is the driver
- **localhost** is the server name on which oracle is running, we may also use IP address
- **1521** is the port number and
- **XE** is the Oracle service name.

Note: You may get all above information from the tnsnames.ora file.

Create a table in oracle database

```
create table student(roll number(10),name varchar2(40));
```

Example to Connect Java Application with Oracle database

```
import java.sql.*;
class OracleCon
{
public static void main(String args[]){
try
{
//step1 load the driver class
```

```
//step2 create  the connection object
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

//step3 create the statement object
Statement stmt=con.createStatement();

//step4 execute query
ResultSet rs=stmt.executeQuery("select * from student");
while(rs.next())
System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));

//step5 close the connection object
con.close();

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

CONNECTION INTERFACE

A Connection is the session between java application and database. The Connection interface is a factory of Statement, PreparedStatement, and DatabaseMetaData i.e. object of Connection can be used to get the object of Statement and DatabaseMetaData. The Connection interface provide many methods for transaction management like commit(),rollback() etc.

Method of Connection Interface

	method	Description
1	public Statement createStatement()	creates a statement object that can be used to execute SQL queries.
2	public Statement createStatement(int resultSetType,int resultSetConcurrency)	Creates a Statement object that will generate ResultSet objects with the given type and concurrency.
3	public void setAutoCommit(boolean status)	Used to set the commit status.By default it is true.
4	public void commit()	saves the changes made since the previous commit/rollback permanent.
5	public void rollback()	Drops all changes made since the previous commit/rollback.
6	public void close():	closes the connection and Releases a JDBCresources immediately.

DRIVERMANAGER CLASS

The DriverManager class acts as an interface between user and drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver. The DriverManager class maintains a list of Driver classes that have registered themselves by calling the method DriverManager.registerDriver().

Methods of DriverManager Class

	method	Description
1	public static void registerDriver(Driver driver)	Used to register the given driver with DriverManager.
2	public static void deregisterDriver(Driver driver):	Used to registered the given driver (drop the driver from the list) with DriverManager.
3	public static Connection getConnection(String url):	Used to establish the connection with the specified url.
4	public static Connection getConnection(String url,String userName,String password):	Used to establish the connection with the specified url, username and password.

STATEMENT INTERFACE

The Statement interface provides methods to execute queries with the database.

Method of Statement Interface

	method	Description
1	public ResultSet executeQuery(String sql)	Used to execute SELECT query. It returns the object of ResultSet.
2	public int executeUpdate(String sql)	Used to execute specified query, it may be create, drop, insert, update, delete etc.
3	public boolean execute(String sql)	Used to execute queries that may return multiple results.
4	public int[] executeBatch()	Used to execute batch of commands.

RESULTSET INTERFACE

The object of ResultSet maintains a cursor pointing to a particular row of data. Initially, cursor points to before the first row.

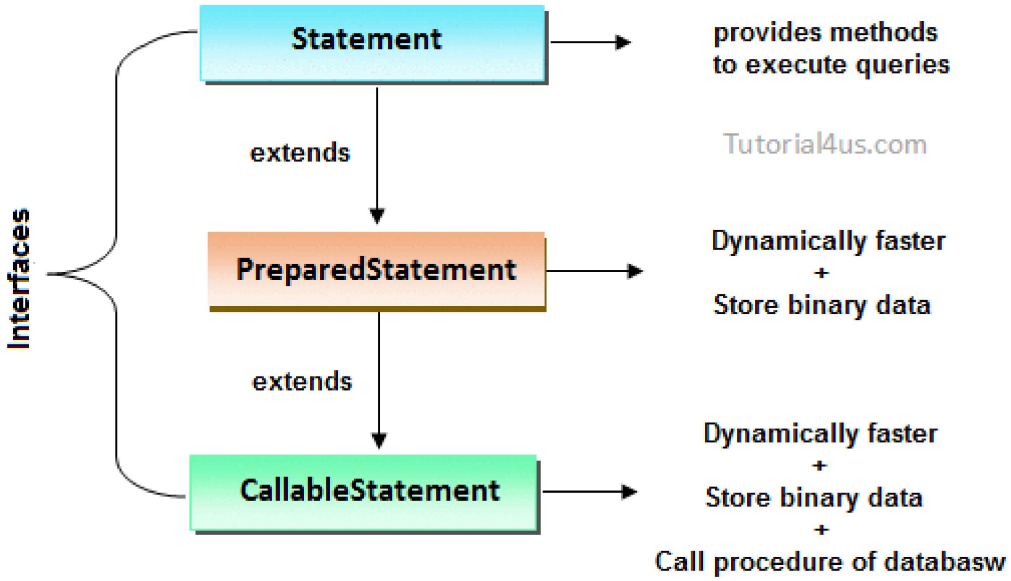
Method of ResultSet Interface

	method	Description
1	public boolean next()	Used to move the cursor to the one row next from the current position.
2	public boolean previous()	Used to move the cursor to the one row previous from the current position.
3	public boolean first()	Used to move the cursor to the first row in result set object.
4	public boolean last()	Used to move the cursor to the last row in result set object.
5	public boolean last()	Used to move the cursor to the last row in result set object.

PREPAREDSTATEMENT IN JDBC

If the SQL command is same then actually no need to compiling it for each time before it is executed. So

PreparedStatement Interface is derived Interface of Statement and CallableStatement is derived Interface of PreparedStatement.



We know that when working with **Statement Interface** of JDBC the SQL command will be compiled first and then it is executed at database side even through the same SQL command is execute repeatedly but each time the command is compiled and then executed at database. Due to this performance of application will be decreased. To overcome this problem we use PreparedStatement. In PreparedStatement, if the SQL command is same then no need to compiling it for each time before it is executed.

In case of preparedStatement

- First sql command is send to database for compilation and then compiled code will be stored in preparedStatement object.
- The compiled code will be executed for n number of time without recompiling the sql command.
- The criteria to use preparedStatement is when we want to execute same sql query for multiple times with different set of values.
- Comparatively preparedStatement is faster than Statement Interface.

DIFFERENCE BETWEEN PREPAREDSTATEMENT AND STATEMENT

	Statement	PreparedStatement
1	Statement interface is slow because it compile the program for each execution	PreparedStatement interface is faster, because its compile the command for once.
2	We can not use ? symbol in sql command so setting dynamic value into the command is complex	We can use ? symbol in sql command, so setting dynamic value is simple.
3	We can not use statement for writing or reading binary data (picture)	We can use PreparedStatement for reading or writing binary data.

Create an object of PreparedStatement

Syntax

```
Connection con; // con is reference of connection
PreparedStatement pstmt=con.prepareStatement("sql command");
```

To pre-compile a command, syntax of the command is required so we can use '?' symbol for value in the command. '?' symbol is called parameter or replacement operator or place-resolution operator.

Syntax

```
PreparedStatement pstmt=con.prepareStatement("Insert into student_table value(?, ?, ?)");
```

Note:

- In PreparedStatement only '?' symbol are allow, no other symbols are allowed.
- '?' is only for replacing value but not for table name or column names.
- '?' symbol are not allowed in DDL operation.

SETTING VALUE: We call setxxx() methods to set the value in place of ? symbols, before executing the command. Here we pass two parameters for setxxx(), where first parameter is index and second is value. XXX means any data type.

Syntax

```
pstmt.setInt(1,102);
```

GETTING VALUE: The method getXX() is used to access the columns in a row that are being pointed to by the cursor. Here XX is the data type of the column.

Example:

```
ResultSet rs = stmt.executeQuery("select * from Emp")
while(rs.next())
{
String name = rs.getString(1);
int age = rs.getInt(2);
S.o.p(name + "   "+age);
}
```

Example of PreparedStatement

```
import java.sql.*;
import javax.sql.*;//PreparedStatement;
import java.util.*;
class PrepardTest1
{
Connection con;
void openConnection()throws Exception
{
Class.forName("oracle.jdbc.OracleDriver");
System.out.println("driver is loaded");
con=DriverManager.getConnection("jdbc:oracle:thin:@John-pc:1521:xe","system","system");
System.out.println("connection is opend");
}
```

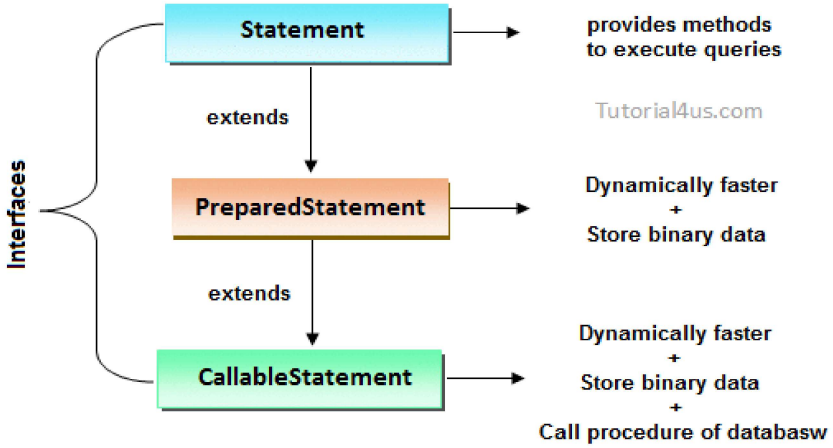
```
void insertTest()throws Exception
{
PreparedStatement pstmt=con.prepareStatement("insert into student values(?,?,?) ");
Scanner s=new Scanner(System.in);
String Choice="yes";
```

```
while(Choice.equals("yes"))
{
System.out.println("enter student id");
int sid=s.nextInt();
System.out.println("enter student name");
String sname=s.next();
System.out.println("enter Student marks");
int marks=s.nextInt();

//setting the values
pstmt.setInt(1,sid);
pstmt.setString(2,sname);
pstmt.setInt(3,marks);
int i=pstmt.executeUpdate();
System.out.println(i+"Row inserted");
System.out.println("do you want to inset another row(Yes/no)");
Choice=s.next();
}//end while
pstmt.close();
}
void closeConnection()throws Exception
{
con.close();
System.out.println("connection is closed");
}
public static void main(String[] args) throws Exception
{
PrepardTest1 pt=new PrepardTest1();
pt.openConnection();
pt.insertTest();
pt.closeConnection(); } }
```

CALLABLE STATEMENT

To call the procedures and functions of a database, CallableStatement interface is used. It is a derived Interface of preparedStatement. It has one additional feature over PreparedStatement that is calling procedures and function of a database. Because of CallableStatement is inherited from PreparedStatement all the features of PreparedStatement are also available with CallableStatement.



Create object of CallableStatement: The *prepareCall*() method of Connection interface returns the instance of ***CallableStatement***. To create a reference of ***CallableStatement*** we have two syntaxes one with command and the other is with calling procedure or function.

Syntax of prepareCall() method

```
public CallableStatement prepareCall("{ call procedurename(?,?...?)}");
```

Syntax

```
CallableStatement cstmt=con.prepareCall("sql command");
```

Syntax

```
CallableStatement cstmt=con.prepareCall("{call procedure name or function name}");
```

Syntax

```
CallableStatement stmt=con.prepareCall("{call myprocedure(?,?)}");
```

RESULTSETMETADATA INTERFACE

The metadata means data about data, in case of database we get metadata of a table like total number of column, column name, column type etc. While executing a select operation on a database if the table structure is already known for the programmer, then a programmer of JDBCcan read the data from ResutlSet object directly. If the table structure is unknown then a JDBCprogrammer has to take the help of **ResultSetMetadata**. A ResultSetMetaData reference stores the metadata of the data selected into a ResutlSet object.

To obtain a object of ***ResultSetMetaData***, we need to call *getMetaData*() method of ***ResultSet*** object..

Syntax

```
ResultSetMetaData rsmd=rs.getMetaData();
```

Methods of ResultSetMetaData

The following are the methods called on ResultSetMetaData reference.

	method	Description
1	getColumnCount()	To find the number of columns in a ResultSet
2	getColumnName()	To find the column name of a column index.
3	getColumnTypeName()	To find data type of a column.
4	getColumnDisplaySize()	To find size of a column.

Example of ResultSetMetaData

```
import java.sql.*;
class ResmataDataTest
{
public static void main(String[] args)throws Exception
{
Class.forName("oracle.jdbc.OracleDriver");
Connection con =DriverManager.getConnection("jdbc:oracle:thin:@John-
pc:1521:xe","system","system");
```

```
System.out.println("driver is loaded");
Statement stmt=con.createStatement();
ResultSet rs=stmt.executeQuery("select * from student");
ResultSetMetaData rsmd=rs.getMetaData();
//find the no of columns
int count=rsmd.getColumnCount();
for(int i=1;i <=count;i++)
{
System.out.println("column no :"+i);
System.out.println("column name :"+rsmd.getColumnName(i));
System.out.println("column type :"+rsmd.getColumnTypeName(i));
System.out.println("column size :"+rsmd.getColumnDisplaySize(i));
System.out.println("-----------");
}
rs.close();
stmt.close();
con.close();
} }
```

DATABASEMETADATA

This metadata interface is used for reading metadata about a database. Metadata about database is nothing but reading database server, version, driver name and its version, maximum number of columns allowed in a table etc.

To obtain a object of DatabaseMetaData, we need to call getMetaData() method of Connection object.

Syntax

```
DatabaseMetaData dbms=con.getMetaData();
```

Methods of DatabaseMetaData

The methods to read some metadata information are:

	method	Description
1	getDatabaseProductName()	To read database server name.
2	getDatabaseProductVersion()	To read database server version.
3	getDriverName()	To read driver software name.
4	getColumnInTable()	To find maximum number of columns allowed in the table..

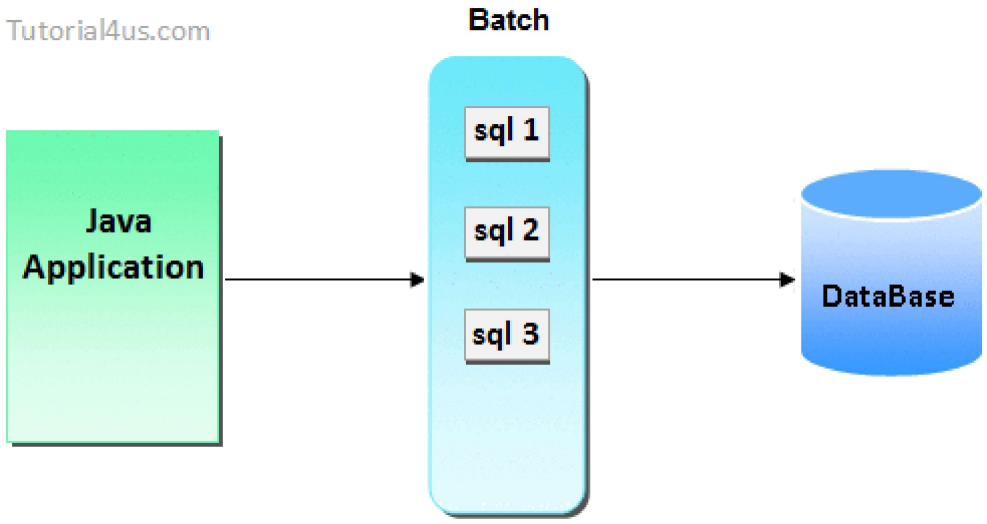
Example of DatabaseMetaData

```
import java.sql.*;
class DBMetaDeta
{
public static void main(String[] args) throws Exception
{
Class.forName("oracle.jdbc.OracleDriver");
Connection con=DriverManager.getConnection("jdbc:oracle:thin:@rama
                                                    pc:1521:xe","system","system");
System.out.println("conection is opened");
```

```
DatabaseMetaData dbmd=con.getMetaData();
System.out.println("database server name:"+dbmd.getDatabaseProductName());
System.out.println("database server version:"+dbmd.getDatabaseProductVersion());
System.out.println("driver server version:"+dbmd.getDriverVersion());
System.out.println("driver server name:"+dbmd.getDriverName());
System.out.println("max columns:"+dbmd.getMaxColumnsInTable());
stmt.close();
con.close();
}
}}
```

BATCH PROCESSING

Instead of executing a single query, we can execute a batch (group) of queries using **batch processing**. In a JDBC program if multiple SQL operations are there, then each operation is individually transferred to the database. This approach will increase the number of round trips between java program (application) and database. If the number of round trips is increased between an application and database, then it will reduce the performance of an application. To overcome these problems we use Batch Processing.



Advantage of Batch Processing: Increase the performance of an application.

Method of Batch Processing

The following two methods are used for performing Batch Processing. These methods are given by Statement Interface.

	Method	Description
1	void addBatch(String query)	It adds query into batch.
2	int[] executeBatch()	It executes the batch of queries.

Note: In Batch Processing only non-select operations are allowed select operation is not allowed. If any operation failed in batch processing then java.sql.BatchUpdate Exception will be thrown. When we want to cancel all the operation of the batch then apply batch processing with transaction management.

Example of Batch Processing

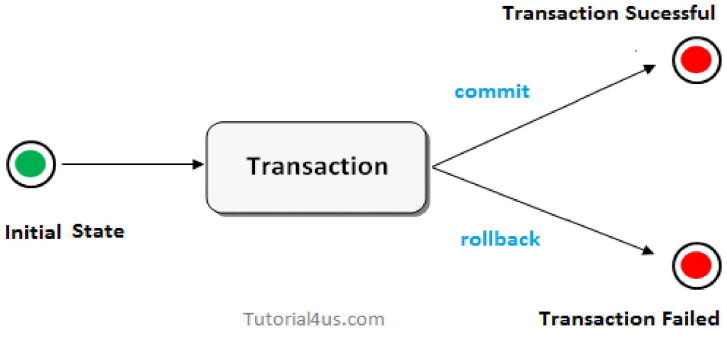
```
import java.sql.*;
class BatchTest
{
public static void main(String[] args) throws Exception
{
Class.forName("oracle.jdbc.OracleDriver");
Connection con=DriverManager.getConnection("jdbc:oracle:thin:@rama-
pc:1521:xe","system","system");
Statement stmt=con.createStatement();

//create batch
stmt.addBatch("insert into student values(901,'PQR',788)");
stmt.addBatch("update emp_info set esal=8888 where eno=1012");
stmt.addBatch("delete from customer where custid=111");

//disabl auto-commit mode
con.setAutoCommit(false);
try
{
 int i[]=stmt.executeBatch();
 con.commit();
 System.out.println("batch is successfully executed");
}
catch (Exception e)
{
try
{
con.rollback();
System.out.println("batch is failed");
System.out.println("Exception is"+e);
}
catch (Exception e1)
{
}
}//end of outer catch
//cleanup
stmt.close();
con.close(); }  }
```

TRANSACTION MANAGEMENT IN JDBC

A transaction is a group of operation used to be perform as one task. If all operations in the group are success then the task is finished and the transaction is successfully completed. If any one operation in the group is failed then the task is failed and the transaction is failed.

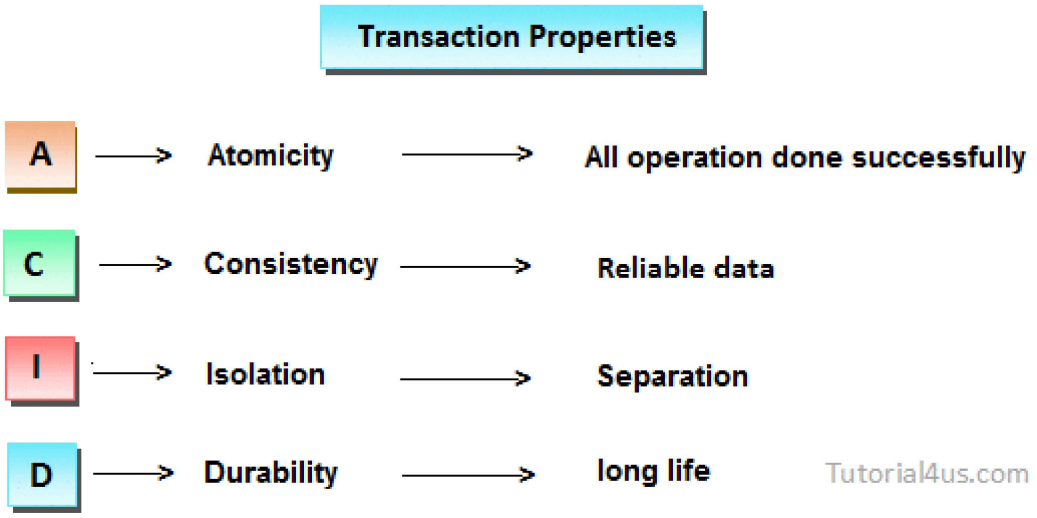


Example:

- Verify the seats
- Reserve the seats
- Payment
- Issue tickets

If all the above four operations are done successfully then a transaction is finished successfully. In the middle, if any one operation is failed then all operation are cancelled and finally a transaction is failed.

Properties of Transaction managements: Every transaction follows some transaction properties these are called **ACID** properties.



- **Atomicity:** Atomicity of a transaction is nothing but in a transaction either all operations can be done or all operation can be undone, but some operations are done and some operation are undone should not occur.
- **Consistency:** Consistency means, after a transaction completed with successful, the data in the data store should be a reliable data this reliable data is also called as consistent data.
- **Isolation:** Isolation means, if two transaction are going on same data then one transaction will not disturb another transaction.
- **Durability:** Durability means, after a transaction is completed the data in the data store will be permanent until another transaction is going to be performed on that data.

Advantage of Transaction Management

- Fast performance: It makes the performance fast because database is hit at the time of commit.

Types of Transaction

- Local Transaction: A local transaction means, all operation in a transaction are executed against one database.
  For example; If transfer money from first account to second account belongs to same bank then transaction is local transaction.
- Distributed or global transaction: A global transaction means, all operations in a transaction are executed against multiple database.
  For Example; If transfer money from first account to second account belongs to different banks then the transaction is a global transaction.

Note: JDBC technology perform only local transactions. For global transaction in java we need use either EJB or spring framework.

Things required for transaction in JDBC

Step 1: Disable auto commit mode of JDBC
Step 2: Put all operation of a transaction in try block.
Step 3: If all operations are done successfully then commit in try block, otherwise rollback in catch block.

By default in JDBC autocommit mode is enabled but we need to disable it. To disable call *setAutoCommit*() method of connection Interface.

Example

```
con.setAutoCommit(false);
```

To commit a transaction, call *commit*() and to rollback a transaction, call *rollback*() method of ***Connection*** Interface respectively.

Example

```
con.commit();
con.rollback();
```

Note:

- In transaction management DDL operation are not allowed.
- The operation in a transaction management may be executed on same table or different table but database should be same.

Example

```
import java.sql.*;
class TrxaExample
{
public static void main(String[] args)throws Exception
{
Class.forName("oracle.jdbc.OracleDriver");
Connection con =DriverManager.getConnection("jdbc:oracle:thin:@rama-
pc:1521:xe","system","system");
System.out.println("driver is loaded");
Statement stmt=con.createStatement();
con.setAutoCommit(false);
try
{
int i1=stmt.executeUpdate("insert into student values(110,'rama',685)");
int i2=stmt.executeUpdate("update customer set custadd='Hyderabad'where custid=111");
int i3=stmt.executeUpdate("delete from student where sid=101");
con.commit();
System.out.println("Transaction is success");
}//end of try
catch (Exception e)
{
try
{
con.rollback();
System.out.println("Trasaction is failed");
}
catch (Exception ex)
```

```
{
System.out.println(ex);
}
}//end of catch
stmt.close();
con.close();
System.out.println("connection is closed");
} //end of main
} //end of class
```

WEB CONTAINERS

Introduction:

There are two types of clients in the J2EE architecture: application clients and web clients. Application clients are also called as fat clients as they handle most of the functionality. With the advent of the Internet, web clients replaced the traditional application clients.Web clients are also called as thin clients.

Features of web clients:

- A web browser which manages the user interaction; this is the client layer
- HTML (with JavaScript and/or DHTML) or XML (with XSLT) which creates the user interface
- HTTP(S) is the information exchange protocol used by the clients and applications.
    - The application programs on the server side execute the application logic on behalf of the browser clients.

What does a web server do?

A web server takes a client request and gives something back to the client. A web browser lets a user request a resource. The web server gets the request, finds the resource and returns something to the user. The resource can be a HTML page, a picture, audio, video, etc. The programs written at the server are called server side programs.

What does a web client do?

A web client lets the user request something on the server, and shows the user the result of the request. The human user and the browser are called the clients. The browser is a piece of software that knows how to communicate with the server. Its other important job is to interpret the HTML code and render the web page.

What is the HTTP protocol?

HTTP is a network protocol that has web specific features, but it depends on TCP/IP to get the complete request and response from one place to another. The structure of an HTTP conversation is a simple.

Request/Response sequence: A browser **requests** and a server **responds**

Key elements of request and response streams

Request Stream	Response Stream
HTTP method(the action to be performed)	A Status code (for whether the request was successful)
The Page to access(URL)	Content-type(Text,picture, HTML, etc)
Form parameters(like the arguments to a method)	The content (the actual HTML,image,etc)

HTTP request methods

As an application level protocol, HTTP defines the types of requests that clients can send to servers and the types of responses servers can send to clients. The protocol also specifies how these requests and responses are structured. There are many request methods like GET, POST, HEAD, OPTIONS, PUT, TRACE, etc. But the most commonly used are GET and POST.

The *get* request method

The GET request is the simplest and most frequently used request method. It is typically used to access static resources such as HTML documents and images. The GET request can sometimes carry query parameters in the request. But it is generally not used for this purpose as the total amount of characters in the GET is very limited and the data which is sent with the GET is appended to the URL and can be seen in the browser bar.

The *POST* request method

POST method is a very powerful request method. It is commonly used for accessing dynamic resources. It is used to transmit information that is request dependent. It is also used when large amount of information is to be sent to the server. The POST request allows the encapsulation of multi part messages into the request body.

HTTP response methods

When a server receives an HTTP request, it responds with the **status** of the response and some **meta-information** describing the response. These are part of the response header. The server also sends the content that corresponds to the resource specified in the request. The content header fields in the response contain useful information that clients may want to check in certain conditions. Typical field include: Date, Content-Type and Expiries. Servers and clients that communicate using HTTP use MIME to indicate the type of content in request and response bodies.

Web applications

Web applications are server side applications. The requirements for server side applications are:

- A programming model and an API
- Server side runtime support
- Deployment support
    - Deployment is the process of installing the application on the server.

J2EE specifications

To fulfill the requirements, J2EE provides the following specifications.

- Java Servlets and JSPs: These are the building blocks of the web applications. They are also called web components.
- Web application: Collection of Servlets, JSPs, helper classes, class libraries, HTML, XHTML, XML, images, etc
- Web container: It is essentially a Java runtime that provides an implementation of the Servlets and JSPs. It is responsible for initializing, invoking, and managing the lifecycle of the Servlets and JSPs
- Packaging structure and Deployment Descriptors

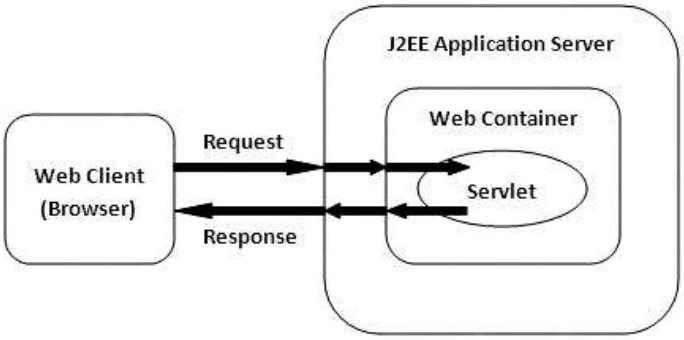
JAVA SERVLETS

INTRODUCTION:

The web server can only handle static content. But sometimes the response may involve some kind of processed information. To handle this dynamic content, the web server makes use of a helper application that provides this support. This helper application is a web container. The programs which do the processing are the servlets.

SERVLETS: Java Servlets are small, platform-independent server-side programs that programmatically extend the functionality of the web server. When a client sends a request for dynamic content to a web server, the web server passes on the request to the web container. The web container calls the corresponding servlet. The servlet processes the request, generates the response and sends it to the web container. The web container then passes on the response to the web server application. This response is returned to the client and is displayed on the browser.

SERVLET PROCESS

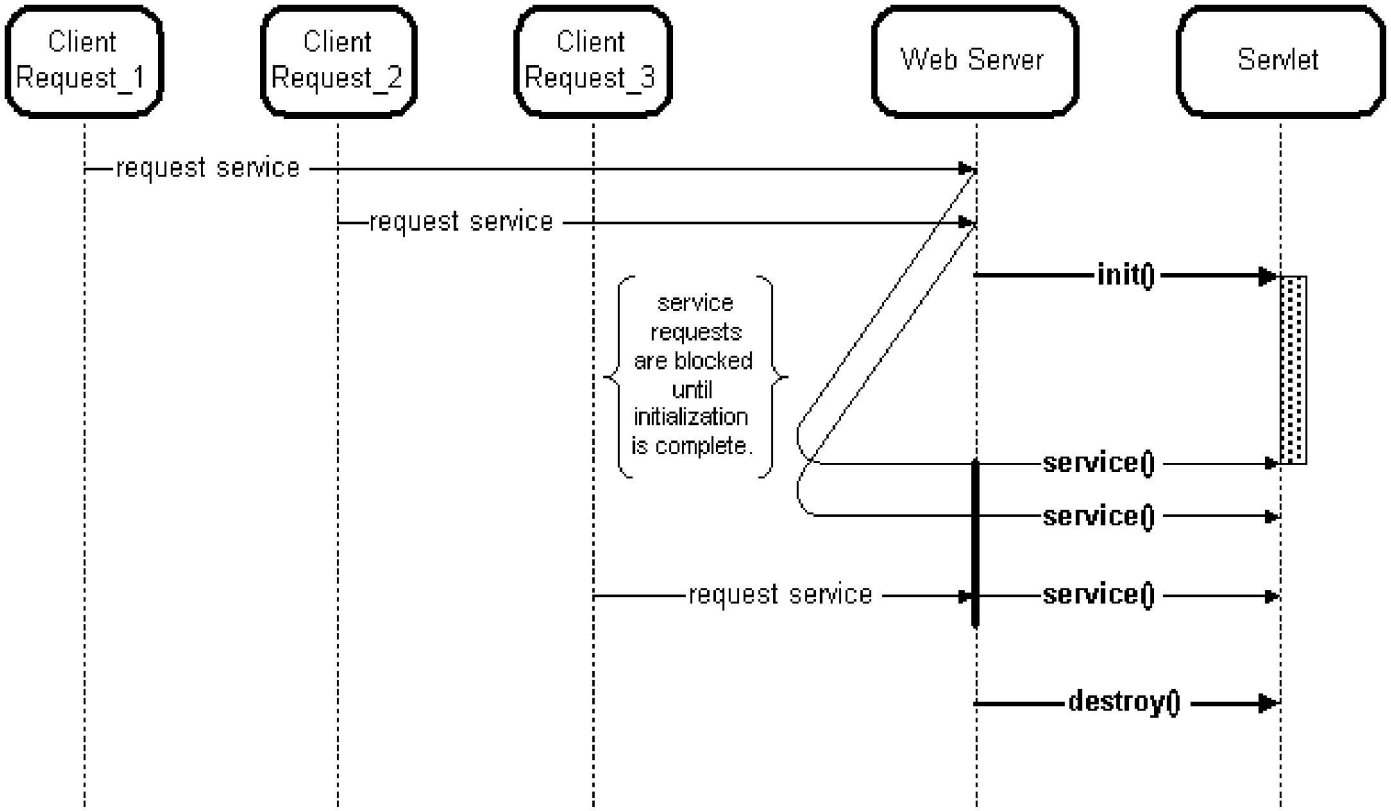


Servlet is a class which is deployed in a container. Every servlet is identified by a unique URL. Servlet class is present in a package called **javax.servlet.** When a user submits a query, the information is carried to the HTTP Server. The HTTP server then transfers this request to the container which holds the servlet

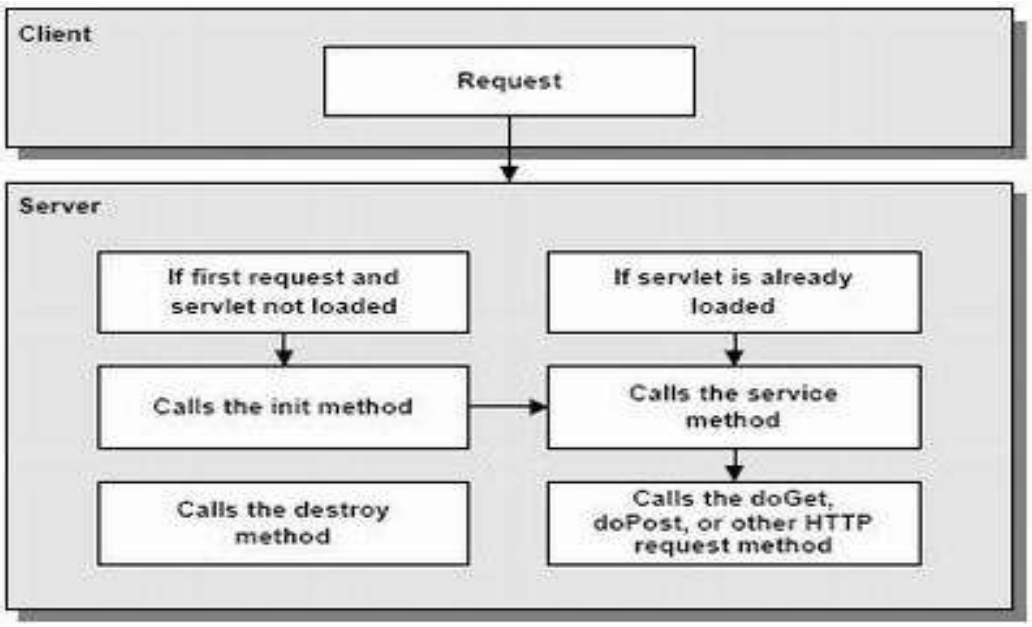
SERVLET LIFECYCLE

If the servlet has been deployed but never used, the container instantiates the servlet i.e. it creates the servlet object. This is done by calling the **init()** method. The status of the servlet now changes from deployed to initialized. The container now creates a servlet thread which executes the **service()** method . The body of this method contains the code of what the servlet is supposed to do. Once a servlet is initialized, every incoming request will be given a thread which calls the service() method. To delete a servlet, the servlet object calls the **destroy()** method. After this the servlet is garbage collected.

In the lifecycle of the servlet, the init() and destroy() methods are called only once. The service() method is called many times.



Servlet processing:



SERVLET CLASS HIERARCHY

GenericServlet is an abstract class which is derived from the Servlet interface. It has the definitions of all methods except the service() method. Another powerful class called the HttpServlet class extends from the GenericServlet class. This class contains all methods of the GenericServlet class plus methods of its own.

Service( ) method

The service() method takes two parameters which are the request object and the response object. The request object contains the request parameters. The response object contains the final response.

The syntax of the service method is

public abstract void service(ServletRequest request, ServletResponse response) throws ServletException, IOException

DEPLOYMENT DESCRIPTOR

Along with every servlet, there is a file called Deployment Descriptor (DD). This file enables the servlet to be deployed on the container. The DD is an XML file and it must always be saved with the name web.xml.

The general structure of the DD is as follows

```
<web-app>
<servlet>
<servlet –name>Servlet_Name</servlet-name>
<servlet-class>Servlet_ClassName</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name> Servlet_Name </servlet-name>
<url-pattern>/Pattern-word/*<url-pattern>
</servlet –mapping>
</web-app>
```

Steps for deployment of an application

- Create a directory with the same name as the application.
- Save the servlet java file into this directory
- Within this directory create a folder called WEB-INF
- Save the web.xml file here
- Within the WEB-INF folder, create another folder called classes and save the servlet's .class file there
- Create a war file of this application as
  jar –cvf app.war
- This file is to be deployed onto the server.
- The Servlet is executed as
  http://localhost:8080/<application>/<servlet>

Creating Servlet using Generic Servlet class

```
import javax.servlet.*;
import javax.io.*;
public class MyServlet extends GenericServlet
{
public void service(ServletRequest req, ServletResponse res) throws IOException, ServletException
{
PrintWriter out=res.getWriter();
out.println("Welcome to Servlets");
}
}
```

HTTP Servlet: The HttpServlet class is a very powerful class. It is present in **javax.servlet.http.*.** In this class, the service() method is overridden. The service method either calls doGet() / doPost().

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

- public void doPost(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException

Add.html

```
<html>
  <head>
    <title>add</title>
  </head>
  <body>
   <form action="http://localhost:8084/WebApplication1/AdditionServlet">
      Num1:<input type="text" name="num1"><br>
      Num2:<input type="text" name="num2"><br>
      <input type="submit" value="add">
   </form>
  </body>
</html>
```

AdditionServlet.java

```
import javax.servlet.*;
import javax.servlet.http.*;
public class AdditionServlet extends HttpServlet {
        public void doGet(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException  {
          PrintWriter out = response.getWriter();
       int i=Integer.parseInt(request.getParameter("num1"));
       int j=Integer.parseInt(request.getParameter("num2"));
       out.println("<html>");
       out.println("<head>");
       out.println("<title>Servlet AdditionServlet</title>");
       out.println("</head>");
       out.println("<body>");
       out.println("the result is "+(i+j));
       out.println("</body>");
       out.println("</html>");

       }
}
```