CMR INSTITUTE OF TECHNOLOGY

USN | | | | | | | | | | |

Internal Assessment Test 2–April 2018

Sub:	Advanced Java Programming						Code:	16MCA 41
Date:	16.04.2018	*Duration:*	*90 mins*	*Max Marks:*	*50*	*Sem:* 4	*Branch:*	*MCA*

1. Explain different types of session tracking techniques with example

Hidden Form:

<INPUT TYPE="HIDDEN" NAME="session" VALUE="a1234">

This entry means that, when the form is submitted, the specified name and valare automatically included in the GET or POST data. This hidden field can be used tostore information about the session but has the major disadvantage that it only works if every page is dynamically generated by a form submission. Clicking on a regular hypertext link does not result in a form submission, so hidden form fields cannot support general session tracking, only tracking within a specific series ooperations such as checking out at a store.

Cookies

Cookies are small bits of textual information that a web server sends to a browser and that the browser later returns unchanged when visiting the same web site or domain

Sending cookies to the client:

1.Creating a cookie object

- Cookie():constructs a cookie.
- Cookie(String name, String value)constructs a cookie with a specified name and value.

EX:

Cookie ck=new Cookie("user",”mca");

2.Setting the maximum age

setMaxAge() is used to specify how long (in seconds) the cookie should be valid.

Ex:cookie.setMaxAge(60*60*24);

3.Placing the cookie into the HTTP response headers.

We use **response.addCookie** to add cookies in the HTTP response header as follows:

response.addCookie(cookie);

Reading cookies from the client:

1. Call request.getCookies(). This yields an array of cookie objects.
2. Loop down the array, calling getName on each one until you find the cookie of interest.

Ex:

```
 String cookieName=“userID”;
Cookie[] cookies=request.getCookies();
If(cookies!=null)
{
            for(int i=0;i<cookies.length;i++){
      Cookie cookie=cookies[i];
       if(cookieName.equals(cookie.getName())){
          doSomethingwith(cookie.getValue());
}}}
```

Session Tracking:

1. Accessing the session object associated with the currentrequest.

Call request.getSession to get an HttpSessionobject, which is a simple hash table for storing user-specific data.

2. Looking up information associated with a session.

Call getAttribute on the HttpSession object, cast the return value to the appropriate type, and check whether the result is null.

3.Storing information in a session.

Use setAttribute with a key and a value.

4.Discarding session data.

Call removeAttribute to discard a specific value. Call invalidate to discard an entire session. Call logout to log the client out of the Web server and invalidate all sessions associated with that user.

URL Rewriting

- With URL rewriting, every local URL the user might client on is dynamically modified or rewritten, to include extra information.
- The extra info. can be in the form of extra path information, added parameters or some custom, server-specific URL change.

URL?name1=value1&name2=value2

Advantages

- IT will always work whether cookie is disable or not, so it is broser independent
- Extra form submission is of required on each page

Disadvantage:

- It will work only with links
- It can send only textual information

2. List and discuss the basic data types and advanced data types of JDBC.

1. CHAR, VARCHAR, and LONGVARCHAR

CHAR

Represents a small, fixed-length character string

SQL CHAR type corresponding to JDBC CHAR is defined in SQL-92 and is supported by all the major databases

CHAR(12) defines a 12-character string.

All the major databases support CHAR lengths up to at least 254 characters. To retrieve the data from CHAR, ResultSet.getString method will be used.

VARCHAR

VARCHAR represents a small, variable-length character string

It takes a parameter that specifies the maximum length of the string.

VARCHAR(12) defines a string whose length may be up to 12 characters.

All the major databases support VARCHAR lengths up to 254 characters.

When a string value is assigned to a VARCHAR variable, the database remembers the length of the assigned string and on a SELECT, it will return the exact original string.

To retrieve the data from VARCHAR, ResultSet.getString method will be used.

LONGVARCHAR

LONGVARCHAR represents a large, variable-length character string No consistent SQL mapping for the JDBC LONGVARCHAR type.

All the major databases support some kind of very large variable-length string supporting up to at least a gigabyte of data, but the SQL type names vary

These methods are getAsciiStream and getCharacterStream, which deliver the data stored in a LONGVARCHAR column as a stream of ASCII or Unicode characters.

2. BINARY, VARBINARY, and LONGVARBINARY

BINARY
Represents a small, fixed-length binary value

SQL BINARY type corresponding to JDBC BINARY is defined in SQL-92 and is supported by all the major databases

BINARY(12) defines a 12-byte binary value.

To retrieve the data from BINARY, ResultSet.getBytes method will be used.

VARBINARY
VARBINARY represents a small, variable-length binary value

It takes a parameter that specifies the maximum binary bytes.
BINARY(12) defines a 12-byte binary type.

BINARY values are limited to 254 bytes.

To retrieve the data from BINARY, ResultSet.getBytes method will be used

LONGVARBINARY
LONGVARBINARY represents a large, variable-length byte value

No consistent SQL mapping for the JDBC LONGVARBINARY type.

JDBC LONGVARBINARY stores a byte array that is many megabytes long, however, the method getBinaryStream is recommended

3. BIT

The JDBC type BIT represents a single bit value that can be zero or one.
SQL-92 defines an SQL BIT type.

Portable code may use the JDBC SMALLINT type, which is widely supported.

The recommended Java mapping for the JDBC BIT type is as a Java boolean.

4. TINYINT

The JDBC type TINYINT represents an 8-bit integer value between 0 and 255 that may be signed or unsigned.

Portable code may use the JDBCSMALLINT type, which is widely supported.

Java mapping for the JDBC TINYINT type is as either a Java byte or a Java short.
Represents a signed value from -128to 127

16-bit Java short will always be able to hold all TINYINT values.

5. SMALLINT
Represents a 16-bit signed integer value between -32768 and 32767.

The recommended Java mapping for the JDBC SMALLINT type is as a Java short.

6. INTEGER
Represents a 32-bit signed integer value ranging between -2147483648 and 2147483647.

SQL type, INTEGER, is defined in SQL-92 and is widely supported by all the major databases.

Recommended Java mapping for the INTEGER type is as a Java int.\

7. BIGINT

Represents a 64-bit signed integer value between - 9223372036854775808 and 9223372036854775807.

The corresponding SQL type BIGINT is a nonstandard extension to SQL.
Recommended Java mapping for the BIGINT type is as a Java long.

8. REAL

The JDBC type REAL represents a "single precision" floating point number that supports 7 digits of mantissa.

4 bytes(32 bits) will be allocated. 1 bit for sign, 8 bits for exponent and 23 for fraction.
Recommended Java mapping for the REAL type is as a Java float.

9. DOUBLE

The JDBC type DOUBLE represents a "double precision" floating point number that supports 15 digits of mantissa.

8 bytes (64 bits) will be allocated. 1 bit for sign, 11 bits for exponent and 52 for fraction.
Recommended Java mapping for the DOUBLE type is as a Java double.

10. FLOAT

The JDBC type FLOAT is basically equivalent to the JDBC type DOUBLE.

FLOAT represents a "double precision" floating point number that supports 15 digits of mantissa.

Both FLOAT and DOUBLE in a possibly misguided attempt at consistency with previous database APIs.

Use the JDBC DOUBLE type in preference to FLOAT.

11. DECIMAL and NUMERIC

The JDBC types DECIMAL and NUMERIC are very similar.
They both represent fixed-precision decimal values.

The precision is the total number of decimal digits supported
The scale is the number of decimal digits after the decimal point.

For example, the value "12.345" has a precision of 5 and a scale of 3, and the value ".11" has a precision of 2 and a scale of 2.

NUMERIC(12,4) will always be represented with exactly 12 digits

DECIMAL(12,4) might be represented by some larger number of digits.

Java mapping for the DECIMAL and NUMERIC types is java.math.BigDecimal.

The method recommended for retrieving DECIMAL and NUMERIC values is ResultSet.getBigDecimal

12. DATE, TIME, and TIMESTAMP

There are three JDBC types relating to time:

The JDBC DATE type represents a date consisting of day, month, and year. The corresponding SQL DATE type is defined

The JDBC TIME type represents a time consisting of hours, minutes, and seconds. The corresponding SQL TIME type is defined

JDBC TIMESTAMP type represents DATE plus TIME plus a nanosecond field. The corresponding SQL TIMESTAMP type is defined

Advanced JDBC Data Types

1. BLOB

- The JDBC type `BLOB` represents an SQL3 `BLOB` (Binary Large Object).
- A JDBC `BLOB` value is mapped to an instance of the `Blob` interface in the Java programming language.
- A `Blob` object logically points to the BLOB value on the server rather than containing its binary data, greatly improving efficiency.
- The `Blob` interface provides methods for materializing the `BLOB` data on the client when that is desired.

2. CLOB

- The JDBC type `CLOB` represents the SQL3 type `CLOB` (Character Large Object).

- A JDBC `CLOB` value is mapped to an instance of the `Clob` interface in the Java programming language.
- A `Clob` object logically points to the `CLOB` value on the server rather than containing its character data, greatly improving efficiency.
- Two of the methods on the `Clob` interface materialize the data of a `CLOB` object on the client.

3. ARRAY

- The JDBC type `ARRAY` represents the SQL3 type `ARRAY`.
- An `ARRAY` value is mapped to an instance of the `Array` interface in the Java programming language.
- An `Array` object logically points to an `ARRAY` value on the server rather than containing the elements of the `ARRAY` object, which can greatly increase efficiency.
- The `Array` interface contains methods for materializing the elements of the `ARRAY` object on the client in the form of either an array or a `ResultSet` object.

Example :
```
ResultSet rs = stmt.executeQuery("SELECT NAMES FROM STUDENT");
rs.next();
Array stud_name=rs.getArray("NAMES");
```

4. DISTINCT

- The JDBC type DISTINCT represents the SQL3 type `DISTINCT`.
- For example, a `DISTINCT` type based on a `CHAR` would be mapped to a `String` object, and a `DISTINCT` type based on an SQL `INTEGER` would be mapped to an `int`.
- The `DISTINCT` type may optionally have a custom mapping to a class in the Java programming language.
- A custom mapping consists of a class that implements the interface `SQLData` and an entry in a `java.util.Map` object.

5. STRUCT

- The JDBC type `STRUCT` represents the SQL3 structured type.
- An SQL structured type, which is defined by a user with a `CREATE TYPE` statement, consists of one or more attributes. These attributes may be any SQL data type, built-in or user-defined.
- A `Struct` object contains a value for each attribute of the`STRUCT` value it represents.
- A custom mapping consists of a class that implements the interface `SQLData` and an entry in a `java.util.Map` object.

6. REF

- The JDBC type `REF` represents an SQL3 type `REF<structured type>`.
- An SQL `REF` references (logically points to) an instance of an SQL structured type, which the `REF` persistently and uniquely identifies.
- In the Java programming language, the interface `Ref` represents an SQL `REF`.

7. JAVA_OBJECT

- The JDBC type `JAVA_OBJECT`, makes it easier to use objects in the Java programming language as values in a database.
- `JAVA_OBJECT` is simply a type code for an instance of a class defined in the Java programming language that is stored as a database object.
- The `JAVA_OBJECT` value may be stored as a serialized Java object, or it may be stored in some vendor-specific format.

- The type `JAVA_OBJECT` is one of the possible values for the column `DATA_TYPE` in the `ResultSet` objects returned by various `DatabaseMetaData` methods, including `getTypeInfo`, `getColumns`, and `getUDTs`.
- Values of type `JAVA_OBJECT` are stored in a database table using the method `PreparedStatement.setObject`.
- They are retrieved with They are retrived with the methods `ResultSet.getObject` or `CallableStatement.getObject` and updated with the `ResultSet.updateObject` method.

For example, assuming that instances of the class `Engineer` are stored in the column `ENGINEERS` in the table `PERSONNEL`, the following code fragment, in which stmt is a `Statement` object, prints out the names of all of the engineers.

3. Write a short note about
a)Implementation class

The contract, implemented as interfaces in Java, defines what our service will do, and leaves it up to the implementation classes to decide how it's done. Remember that the same interface cannot be used for both @Local and @Remote, so we'll make some common base that may be extended.

```
public interface CalculatorCommonBusiness {
 /** * Adds all arguments * *
 @return The sum of all arguments */
int add(int... arguments);
 }
public class CalculatorBeanBase implements CalculatorCommonBusiness {
 /** *
 {
@link CalculatorCommonBusiness#
add(int...)
}
 */ @Override
 public int add(final int... arguments)
{ // Initialize int result = 0;
// Add all arguments for (final int arg : arguments)
{
result += arg;
}
// Return return result;
}
 }
```

This contains the required implementation of CalculatorCommonBusiness. add(int...). The bean implementation class therefore has very little work to do.

```
 import javax.ejb.LocalBean;
 import javax.ejb.Stateless;
@Stateless
@LocalBean
public class SimpleCalculatorBean extends CalculatorBeanBase
 {
/* * Implementation supplied by common base class */
}
```

The function of our bean implementation class here is to bring everything together and define the EJB metadata. Compilation will embed two important bits into the resultant .class file. First, we have an SLSB, as noted by the @Stateless annotation. And second, we're exposing a no-interface view

b) Integration Testing of EJB

There are three steps involved in performing integration testing upon an EJB.
First, we must package the sources and any descriptors into a standard Java Archive
Next, the resultant deployable must be placed into the container according to a vendor-specific mechanism.
Finally, we need a standalone client to obtain the proxy references from the Container and invoke upon them.

Packaging
A standard jar tool that can be used to assemble classes, resources, and other metadata into a unified JAR file, which will both compress and encapsulate its contents.

Deployment into the Container The EJB Specification intentionally leaves the issue of deployment up to the vendor's discretion.

The client
Instead of creating POJOs via the new operator, we'll look up true EJB references via JNDI. JNDI is a simple store from which we may request objects keyed to some known address.

4. Explain the container services provided by component model of EJB

In the case of EJB, the Component Model defines our interchangeable parts, and the Container Services are specialists that perform work upon them. The Specification provides:

1. Dependency injection
2. Concurrency
3. Instance pooling/caching
4. Transactions - *Transaction management*—Declarative transactions provide a means for the developer to easily delegate the creation and control of transactions to the container.
5. Security - declarative security provides a means for the developer to easily delegate the enforcement of security to the container.
6. Timers
7. Naming and object stores - The EJB container and server will provide the EJB with access to naming services. These services are used by local and remote clients to look up the EJB and by the EJB itself to look up resources it may need.
8. Interoperability
9. Lifecycle callbacks
10. Interceptors
11. Java Enterprise Platform integration

1.Dependency Injection (DI)

- A component based approach to software design brings with it the complication of inter-module communication.
- Tightly coupling discrete units together violates module independence and separation of concerns,
- While using common look up code leads to the maintenance of more plumbing
- As EJB is a component-centric architecture, it provides a means to reference dependent modules in decoupled fashion.
- The container provide the implementation at deployment time.

In pseudocode, this looks like:

```
prototype UserModule
{
  // Instance Member
  @DependentModule
  MailModule mail;

  // A function
  function mailUser()
  {
    mail.sendMail("me@ejb.somedomain");
  }
}
```

The @DependentModule annotation serves two purposes:
• Defines a dependency upon some service of type MailModule. UserModule may not deploy until this dependency is satisfied.
• Marks the instance member mail as a candidate for injection. The container will populate this field during deployment.

2. Concurrency

- **Assuming each Service is represeznted by one instance,** dependency injection alone is a fine solution for a single-threaded application; only one client may be accessing a resource at a given time.
- However, this quickly becomes a problem in situations where a centralized server is fit to serve many simultaneous requests.
- Deadlocks, livelocks, and race conditions are some of the possible nightmares arising out of an environment in which threads may compete for shared resources.
- These are hard to anticipate, harder to debug, and are prone to first exposing themselves in production!
- EJB allows the application developer to sidestep the problem entirely thanks to a series of concurrency policies.

3. Instance Pooling/Caching

Because of the strict concurrency rules enforced by the Container, an intentional bottleneck is often introduced where a service instance may not be available for processing until some other request has completed.

If the service was restricted to a singular instance, all subsequent requests would have to queue up until their turn was reached (see Figure 3-1).

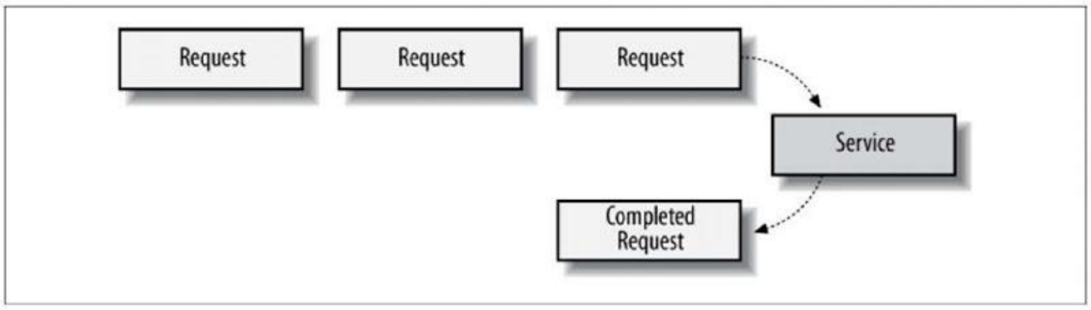


Figure 3-1. Client requests queuing for service

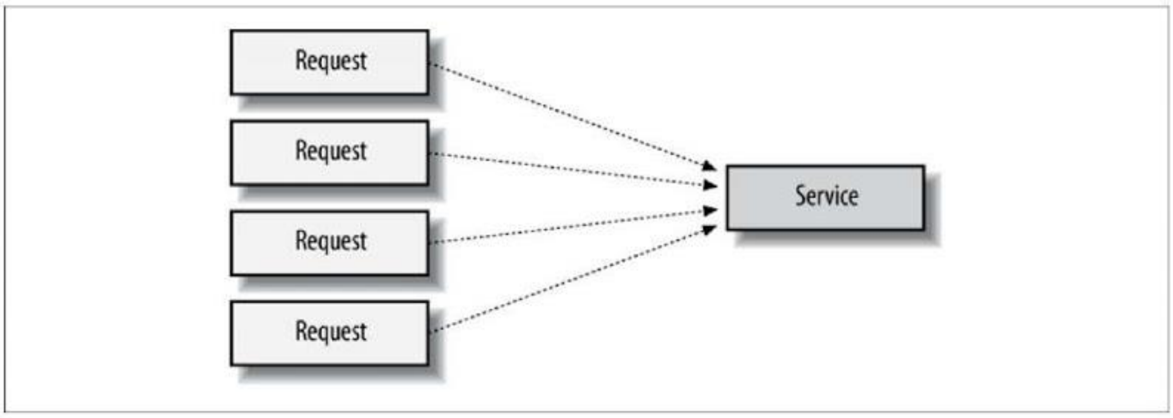


Figure 3-2. Many invocations executing concurrently with no queuing policy

EJB addresses this problem through a technique called instance pooling, in which each module is allocated some number of instances with which to serve incoming requests (Figure 3-3).
Many vendors provide configuration options to allocate pool sizes appropriate to the work being performed, providing the compromise needed to achieve optimal throughput.

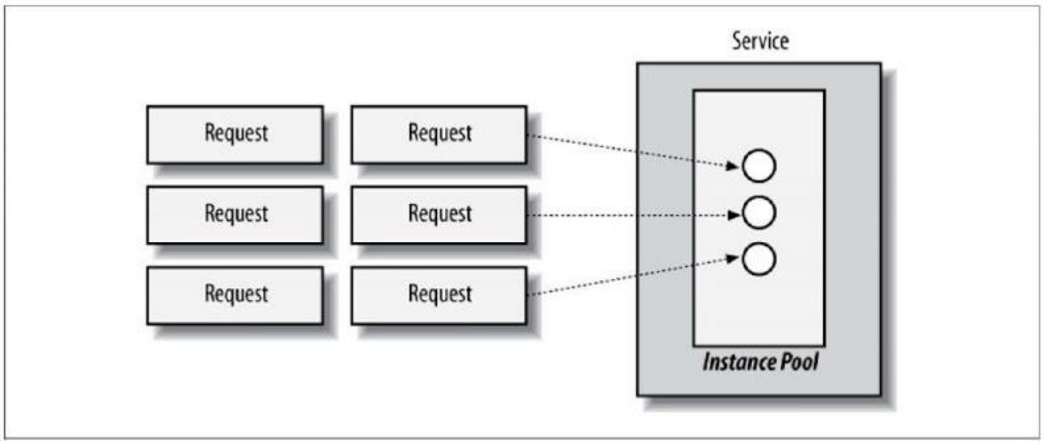


Figure 3-3. A hybrid approach using a pool

4. Transactions

Transactions provide a means for the developer to easily delegate the creation and control of transactions to the container.

- When a bean calls createTimer(), the operation is performed in the scope of the current transaction. If the transaction rolls back, the timer is undone and it’s not created
- The timeout callback method on beans should have a transaction attribute of RequiresNew.

- This ensures that the work performed by the callback method is in the scope of container-initiated transactions.

5. Security

Most enterprise applications are designed to serve a large number of clients, and users are not necessarily equal in terms of their access rights.
An administrator might require hooks into the configuration of the system, whereas unknown guests may be allowed a read-only view of data.

If we group users into categories with defined roles, we can then allow or restrict access to the role itself, as illustrated in Figure 15-1.

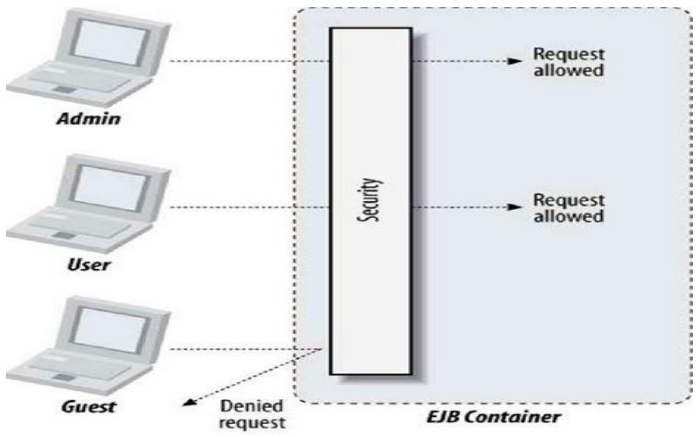


Figure 15-1. EJB security permitting access based upon the caller’s role
This allows the application developer to explicitly allow or deny access at a fine-grained level based upon the caller’s identity

6.Timers

We dealt exclusively with client-initiated requests. While this may handle the bulk of an application’s requirements, it doesn’t account for scheduled jobs:
• A ticket purchasing system must release unclaimed tickets after some timeout of inactivity.
• An auction house must end auctions on time.
• A cellular provider should close and mail statements each month.
The EJB Timer Service may be leveraged to trigger these events and has been enhanced in the 3.1 specification with a natural-language expression syntax.

7. Naming and Object Stores

- They provide clients with a mechanism for locating distributed objects or resources.
- To accomplish this, a naming service must fulfill two requirements:
- Object Binding - Object binding is the association of a distributed object with a natural language name or identifier
- Lookup API - A lookup API provides the client with an interface to the naming system; it simply allows us to connect with a distributed service and request a remote reference to a specific object.

- A Enterprise JavaBeans mandates the use of Java Naming and Directory Interface as lookup API on Java clients.
- JNDI supports just about any kind of naming and directory service.
- Java client applications can use JNDI to initiate a connection to an EJB server and locate a specific EJB.

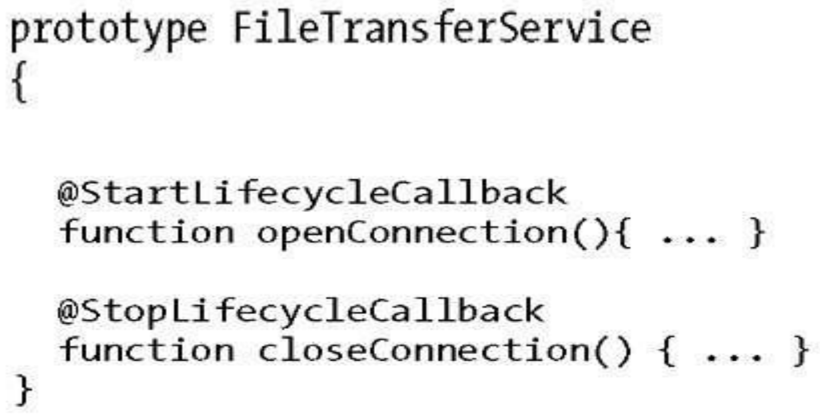
8. Interoperability

- Our application may want to consume data from or provide services to other programs, perhaps written in different implementation languages.
- There are a variety of open standards that address this inter-process communication, and EJB leverages these.
- **Interoperability is a vital part of EJB.**
- The specification includes the required support for Java RMI-IIOP for remote method invocation and provides for transaction, naming, and security interoperability.
- EJB also requires support for JAX-WS, JAX-RPC, Web Services for Java EE, and Web Services Metadata for the Java Platform specifications

9.Lifecycle Callbacks

- **Some services require some initialization** or cleanup to be used properly.
- For example, a file transfer module may want to open a connection to a remote server before processing requests to transfer files and should safely release all resources before being brought out of service.
- For component types that have a lifecycle, EJB allows for callback notifications, which act as a hook for the bean provider to receive these events.

In the case of our file transfer bean, this may look like:



```
prototype FileTransferService
{


  @StartLifecycleCallback
  function openConnection(){ ... }

  @StopLifecycleCallback
  function closeConnection() { ... }
}
```

Here we've annotated functions to open and close connections as callbacks; they'll be invoked by the container as their corresponding lifecycle states are reached.

10.Interceptors

- EJB provides aspectised handling of many of the container services the specification cannot possibly identify all cross cutting concerns facing your project
- EJB makes to possible to define custom inceptors upon business methods and lifecycle callbacks
- Example : We want to measure the execution time of all invocations to a particular method
  We would write an interceptor

```
prototype MetricsInterceptor
{
        Function intercept(Invocation invocation)
        {
                Time startTime = getTime();
                Invocation.continue();
                Time endTime=getTime();
                log("Took : "+ (endTime-startTime));
        }
}
```

\
Then we could apply this to methods as we would like

@ApplyInterceptor(MetricsInterceptor.class)
Function myLoginMethod{……}

@ApplyInterceptor(MetricsInterceptor.class)
Function myLogoutMethod{……}

11.Platform Integration

As a key technology within the Java Enterprise Edition (JEE) 6, EJB aggregates many of the other platform frameworks and APIs:

- Java Transaction Service
- Java Persistence API
- Java Naming and Directory Interface (JNDI)
- Security Services
- Web Services

- **In most cases,** the EJB metadata used to define this integration is a simplified view of the underlying services.
- This gives bean providers a set of powerful constructs right out of the box, without need for additional configuration.
- EJB is also one of the target models recognized by the new Java Contexts and Dependency Injection specification**.**
- This ads a unified binding to both Java Enterprise and Standard editions across many different component types.

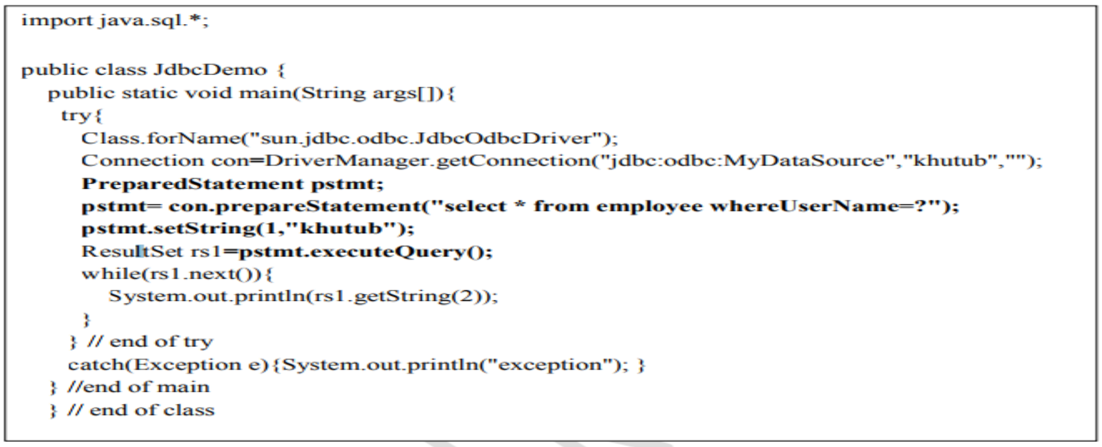
5. a. What is the use and advantages of prepared statement? Illustrate it with code snippets.

The preparedStatement object allows you to execute parameterized queries.

A SQL query can be precompiled and executed by using the PreparedStatement object.

• Ex: Select * from publishers where pub_id=?

Here a query is created as usual, but a question mark is used as a placeholder for a value• that is inserted into the query after the query is com



```
import java.sql.*;

public class JdbcDemo {
   public static void main(String args[]){
     try{
       Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
       Connection con=DriverManager.getConnection("jdbc:odbc:MyDataSource","khutub","");
       PreparedStatement pstmt;
       pstmt= con.prepareStatement("select * from employee whereUserName=?");
       pstmt.setString(1,"khutub");
       ResultSet rs1=pstmt.executeQuery();
       while(rs1.next()){
          System.out.println(rs1.getString(2));
       }
     } // end of try
     catch(Exception e){System.out.println("exception"); }
   } //end of main
   } // end of class
```

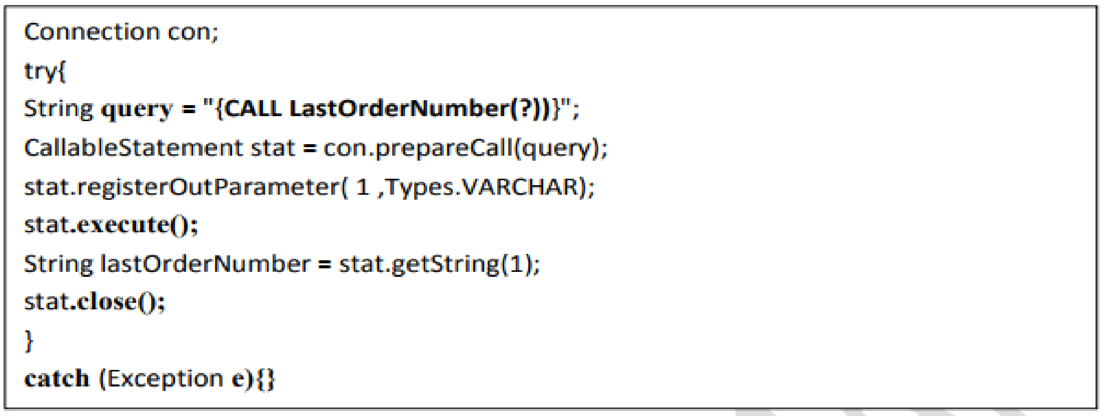
piled.

The preparedStatement() method of Connection object is called to return the• PreparedStatement object.Ex: PreparedStatement stat; stat= con.prepareStatement("select * from publisher wherpub_id=?")

b. Explain how do you call a procedure using Callable statement with code snippets.

Callable Statement:

The CallableStatement object is used to call a stored procedure from within a J2EE object. A Stored procedure is a block of code and is identified by a unique name.
The type and style of code depends on the DBMS vendor and can be written in PL/SQL Transact-SQL, C, or other programming languages.
IN, OUT and INOUT are the three parameters used by the CallableStatement object to call a stored procedure.
The IN parameter contains any data that needs to be passed to the stored procedure and• whose value is assigned using the setxxx() method.
 The OUT parameter contains the value returned by the stored procedures. The OUT parameters must be registered using the registerOutParameter() method, later retrieved by using the getxxx()
The INOUT parameter is a single parameter that is used to pass information to the stored procedure and retrieve information from the stored procedure.



```
Connection con;
try{
String query = "{CALL LastOrderNumber(?))}";
CallableStatement stat = con.prepareCall(query);
stat.registerOutParameter( 1 ,Types.VARCHAR);
stat.execute();
String lastOrderNumber = stat.getString(1);
stat.close();
}
catch (Exception e){}
```

6.a. Explain about Batch Updates with example

Batch Processing allows you to group related SQL statements into a batch and submit them with one call to the database.
When you send several SQL statements to the database at once, you reduce the amount of communication overhead, thereby improving performance.
JDBC drivers are not required to support this feature. You should use the Database MetaData.supports BatchUpdates() method to determine if the target database supports batch update processing. The method returns true if your JDBC driver supports this feature.
The addBatch() method of Statement, PreparedStatement, and CallableStatement is used to add individual statements to the batch. The executeBatch() is used to start the execution of all the statements grouped together.
The executeBatch() returns an array of integers, and each element of the array represents the update count for the respective update statement.
Just as you can add statements to a batch for processing, you can remove them with the clearBatch() method. This method removes all the statements you added with the addBatch() method. However, you cannot selectively choose which statement to remove.
Here is a typical sequence of steps to use Batch Processing with Statement Object −
Create a Statement object using either createStatement() methods.
Set auto-commit to false using setAutoCommit().
Add as many as SQL statements you like into batch using addBatch() method on created statement object.
Execute all the SQL statements using executeBatch() method on created statement object.
Finally, commit all the changes using commit() method.

EX:

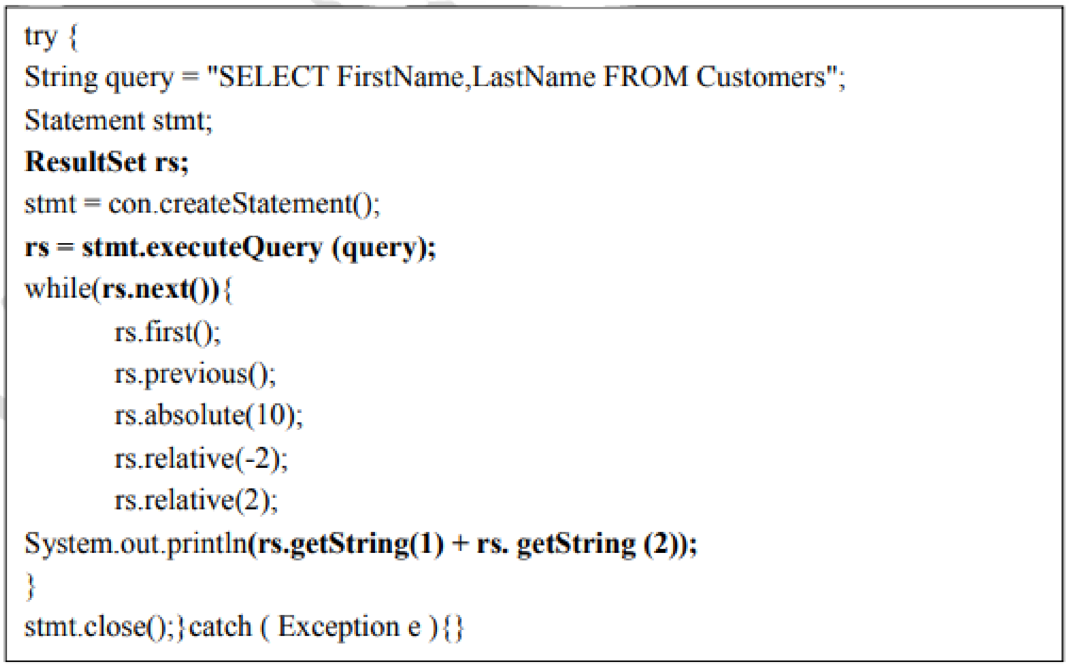
```
// Create statement object
Statement stmt = conn.createStatement();

// Set auto-commit to false
conn.setAutoCommit(false);

// Create SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) " +
             "VALUES(200,'Zia', 'Ali', 30)";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);

// Create one more SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) " +
             "VALUES(201,'Raj', 'Kumar', 35)";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);

// Create one more SQL statement
String SQL = "UPDATE Employees SET age = 35 " +
             "WHERE id = 100";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);

// Create an int[] to hold returned values
int[] count = stmt.executeBatch();

//Explicitly commit statements to apply changes
conn.commit();
```

b) Mention the methods of Result Set and discuss their usage with code snippets.

The ResultSet object contains methods that are used to copy data from the ResultSet into a Java collection object or variable for further processing.

 Data in a ResultSet is logically organized into a virtual table consisting of rows and columns.

 The ResultSet uses a virtual cursor to point to a row of the virtual table.
 The virtual cursor is positioned above the first row of data when the ResultSet is returned by the executeQuery(). This means the virtual cursor must be moved to the first row using the next() method.
 The next() returns a boolean true if the row contains data, else false.
 Once the virtual cursor points to a row, the getxxx() is used to copy data from the row to a collection, object or a variable.
Some of the methods of Resultset are
 They are first(), last(), previous(), absolute(), relative() and getrow().
 first() Moves the virtual cursor to the first row in the Resultset.
 last() Positions the virtual cursor at the last row in the Resultset
 previous() Moves the virtual cursor to the previous row.
 absolute() Positions the virtual cursor to a specified row by the an integer value passed to the method.
 relative() Moves the virtual cursor the specified number of rows contained in the parameter. The parameter can be positive or negative integer.
 getRow() Returns an integer that represents the number of the current row in the Resultset.
 To handle the scrollable ResultSet , a constant value is passed to the Statement object that is created using the createStatement(). Three constants.
 TYPE_FORWARD_ONLY restricts the virtual cursor to downward movement
TYPE_SCROLL_INSENSITIVE and TYPE_SCROLL_SENSITIVE (Permits the virtual cursor to Move in any direction)



```
try {
String query = "SELECT FirstName,LastName FROM Customers";
Statement stmt;
ResultSet rs;
stmt = con.createStatement();
rs = stmt.executeQuery (query);
while(rs.next()){
        rs.first();
        rs.previous();
        rs.absolute(10);
        rs.relative(-2);
        rs.relative(2);
System.out.println(rs.getString(1) + rs. getString (2));
}
stmt.close();}catch ( Exception e ){}
```

7. Write a JAVA Program to insert data into Student DATA BASE and retrieve info based on particular queries(For example update, delete, search etc…).

```
package j2ee.p9;
import java.sql.*;
import java.io.*;

public class Studentdata {
	public static void main(String[] args) {
		Connection con;
		PreparedStatement pstmt;
		Statement stmt;
		ResultSet rs;
		String uname, pword;
		Integer marks,count;
		try
		{
			Class.forName("com.mysql.jdbc.Driver"); // type1 driver

			try{

	con=DriverManager.getConnection("jdbc:mysql://127.0.0.1/mca","root","system"); //  type1
access connection
				BufferedReader br=new BufferedReader(new
InputStreamReader(System.in));
			   do
			   {


				System.out.println("\n1. Insert.\n2. Select.\n3. Update.\n4. Delete.\n5.
Exit.\nEnter your choice:");
				int choice=Integer.parseInt(br.readLine());
				switch(choice)
				{
					case 1: System.out.print("Enter UserName :");
						uname=br.readLine();
						System.out.print("Enter Password :");
						pword=br.readLine();
						pstmt=con.prepareStatement("insert into student
values(?,?)");

						pstmt.setString(1,uname);
						pstmt.setString(2,pword);
						pstmt.execute();
						System.out.println("\nRecord Inserted successfully.");
					break;
					case 2:
						stmt=con.createStatement();
						rs=stmt.executeQuery("select *from student");
						if(rs.next())
						{
						System.out.println("User Name\tPassword\n------------
--------------------");

						do
						{
							uname=rs.getString(1);
							pword=rs.getString(2);

							System.out.println(uname+"\t"+pword);
						}while(rs.next());
						}
						else
```

```
                                                        System.out.println("Record(s) are not
available in database.");
                                                break;
                                                case 3:
                                                        System.out.println("Enter User Name to
update :");
                                                        uname=br.readLine();
                                                        System.out.println("Enter new password :");
                                                        pword=br.readLine();
                                                        stmt=con.createStatement();
                                                        count=stmt.executeUpdate("update student set
password='"+pword+"'where username='"+uname+"'");
                                                        System.out.println("\n"+count+" Record
Updated.");
                                                break;
                                                case 4: System.out.println("Enter User Name to delete
record:");
                                                        uname=br.readLine();
                                                        stmt=con.createStatement();
                                                        count=stmt.executeUpdate("delete from
student where username='"+uname+"'");


                                                        if(count!=0)
                                                                System.out.println("\nRecord
"+uname+" has deleted.");
                                                        else
                                                                System.out.println("\nInvalid USN,
Try again.");
                                                break;

                                                case 5: con.close(); System.exit(0);
                                                default: System.out.println("Invalid choice, Try
again.");
                                        }//close of switch
                                        }while(true);
                                        }//close of nested try
                                        catch(SQLException e2)
                                        {
                                                System.out.println(e2);
                                        }
                                        catch(IOException e3)
                                        {
                                                System.out.println(e3);
                                        }
                        }//close of outer try
                        catch(ClassNotFoundException e1)
                        {
                                System.out.println(e1);
                        }
                }
}
```

8. List and describe the built in annotations in detail.

Built in Annotations:

Java defines many built-in annotations.
These four are the annotations imported from **java.lang.annotation**: **@Retention**, **@Documented**, **@Target**,and **@Inherited**.
@Override, **@Deprecated**, and **@SuppressWarnings** are included in **java.lang**.
@Retention
@Retention is designed to be used only as an annotation to another annotation. It specifies the retention policy.
@Documented
The **@Documented** annotation is a marker interface that tells a tool that an annotation is to be documented. It is designed to be used only as an annotation to an annotation declaration.
@Target
The **@Target** annotation specifies the types of declarations to which an annotation can be applied. It is designed to be used only as an annotation to another annotation. **@Target** takes one argument, which must be a constant from the **ElementType** enumeration. This argument specifies the types of declarations to which the annotation can be applied. The constants are shown here along with the type of declaration to which they correspond.

Target Constant	Annotation Can Be Applied To
ANNOTATION_TYPE	Another annotation
CONSTRUCTOR	Constructor
FIELD	Field
LOCAL_VARIABLE	Local variable
METHOD	Method
PACKAGE	Package
PARAMETER	Parameter
TYPE	Class, interface, or enumeration

we can specify one or more of these values in a **@Target** annotation. To specify multiple values, we must specify them within a braces-delimited list. For example, to specify that an annotation applies only to fields and local variables, we can use this **@Target** annotation:
@Target({ ElementType.FIELD, ElementType.LOCAL_VARIABLE })
@Inherited
@Inherited is a marker annotation that can be used only on another annotation declaration. it affects only annotations that will be used on class declarations. **@Inherited** causes the annotation for a superclass to be inherited by a subclass. Therefore, when a request for a specific annotation is made to the subclass, if that annotation is not present in the subclass,then its superclass is checked. If that annotation is present in the superclass, and if it is annotated with **@Inherited**, then that annotation will be returned.
@Override
@Override is a marker annotation that can be used only on methods. A method annotated with **@Override** must override a method from a superclass. If it doesn't, a compile-time error will result. It is used to ensure that a superclass method is actually overridden, and not simply overloaded.
@Deprecated
@Deprecated is a marker annotation. It indicates that a declaration is obsolete and has been replaced by a newer form.
@SuppressWarnings
@SuppressWarnings specifies that one or more warnings that might be issued by the compiler are to be suppressed. The warnings to suppress are specified by name, in string form. This annotation can be applied to any type of declaration..

9. What is session bean? Explain in detail about the types of session bean

Session Beans If EJB is a grammar, session beans are the verbs.• Session beans contain business methods.• The client does not access the EJB directly, which allows the Container to perform all sorts of• magic before a request finally hits the target method. It's this separation that allows for the client to be completely unaware of the location of the server,• concurrency policies, or queuing of requests to manage resources.

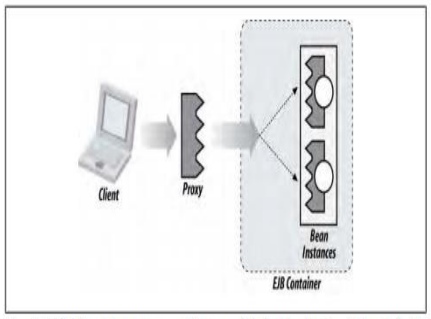


Figure 2-1. Client invoking upon a proxy object, responsible for delegating the call along to the EJB Container

Types of Session Bean
There are 3 types of session bean.
1) Stateless Session Bean: It doesn't maintain state of a client between multiple method calls.
2) Stateful Session Bean: It maintains state of a client across multiple requests.
3) Singleton Session Bean: One instance per application, it is shared between clients and supports concurrent access.
Stateless session beans (SLSBs) Stateless session beans are useful for functions in which state does not need to be carried from• invocation to invocation. The Container will often create and destroy instances however it feels will be most efficient• How a Container chooses the target instance is left to the vendor's discretion.• Because there's no rule linking an invocation to a particular target bean instance, these instances may be used interchangeably and shared by many clients. This allows the Container to hold a much smaller number of objects in service, hence keeping• memory footprint down.

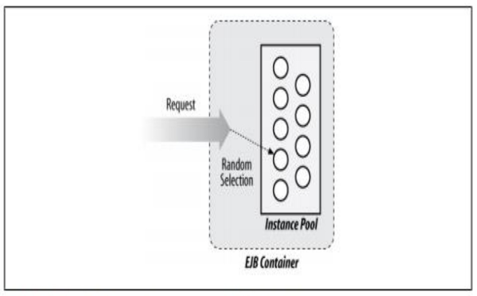


Figure 2-2. An SLSB Instance Selector picking an instance at random

Stateful session beans (SFSBs) Stateful session beans differ from SLSBs in that every request upon a given proxy reference is• guaranteed to ultimately invoke upon the same bean instance. SFSB invocations share conversational state.• Each SFSB proxy object has an isolated session context, so calls to one session will not affect• another. Stateful sessions, and their corresponding bean instances, are created sometime before the first• invocation upon a proxy is made to its target instance (Figure 2-3). They live until the client invokes a method that the bean provider has marked as a remove event,• or until the Container decides to remove the session

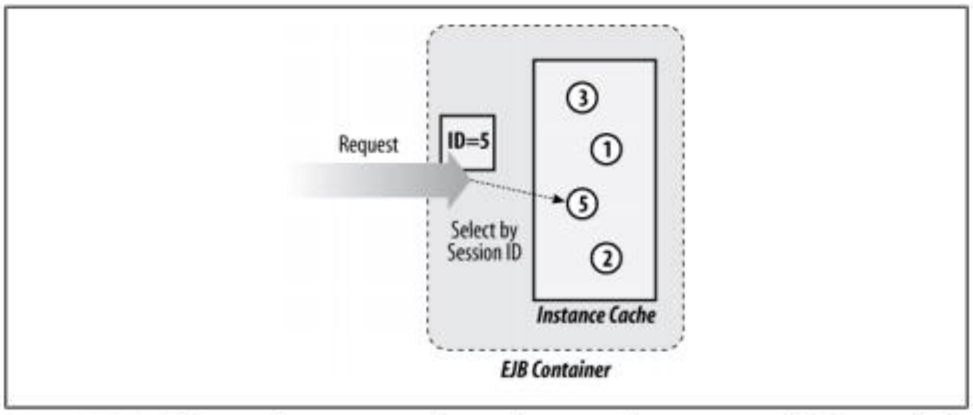


Figure 2-3. Stateful session bean creating and using the correct client instance, which lives inside the EJB Container, to carry out the invocation

Singleton beans Sometimes we don't need any more than one backing instance for our business objects.• All requests upon a singleton are destined for the same bean instance,• The Container doesn't have much work to do in choosing the target (Figure 2-4).• The singleton session bean may be marked to eagerly load when an application is deployed;• therefore, it may be leveraged to fire application lifecycle events. This draws a relationship where deploying a singleton bean implicitly leads to the invocation of its• lifecycle callbacks. We'll put this to good use when we discuss singleton beans.

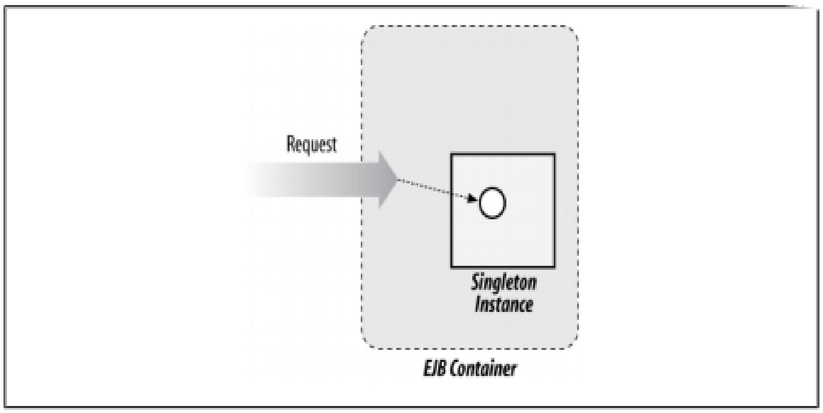


Figure 2-4. Conceptual diagram of a singleton session bean with only one backing bean instance

10. Write an EJB application that demonstrates Session Bean.

<u>Calculator.java</u>

```
package package1;
import javax.ejb.Stateless;
@Stateless
public class Calculator implements CalculatorLocal
{
   @Override
   public Integer Addition(int a, int b) {
      return a+b;
   }
   @Override
   public Integer Subtract(int a, int b) {
      return a-b;
   }
```

```
    @Override
    public Integer Multiply(int a, int b) {
        return a*b;
    }
    @Override
     public Integer Division(int a, int b) {
        return a/b;
    }
}
```

CalculatorLocal.java

```
package package1;
import javax.ejb.Local;
public interface CalculatorLocal {
        Integer Addition(int a, int b);
        Integer Subtract(int a, int b);
        Integer Multiply(int a, int b);
        Integer Division(int a, int b);
}
```

Servlet1.java

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import package1.CalculatorLocal;

public class Servlet1 extends HttpServlet {
    @EJB
    private CalculatorLocal calculator;
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException
    {
            response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter())
        {
            out.println("Output :  "+ "<br/>");
            int a;
            a = Integer.parseInt(request.getParameter("num1"));
            int b;
            b=Integer.parseInt(request.getParameter("num2"));
            out.println("Number1 :  " + a + "<br/>");
            out.println("Number2 :  " + b+ "<br/>");
            out.println("Addition  : " + calculator.Addition(a, b)+ "<br/>");
            out.println("Subtraction  :" + calculator.Subtract(a, b)+ "<br/>");
            out.println("Multiplication  :"+calculator.Multiply(a, b)+ "<br/>");
            out.println("Division  :"+calculator.Division(a, b)+ "<br/>");

        }
    }
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException
    {
        processRequest(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException
    {
        processRequest(request, response);
    }

    @Override
    public String getServletInfo()
    {
        return "Short description";
    }
}
```

index.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Calculator</title>
    </head>
    <body>
        <form method="get" action="Servlet1">
            Enter Number1 :  <input type="text" name="num1"/><br/>
            Enter Number2 :  <input type="text" name="num2"/><br/>
            <input type="submit" value="Submit"/>
        </form>
    </body>
</html>
```