**Java EE**

**Work book**

Prepared By: Java Delivery Team

Date:

Reviewed By:

Approved By:

**Review Tracker:-**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **S.no** | **Reviewer Name** | **Date** | **Major Changes Made** | **Before Review V.Number** | **After Review V.Number** | **Signature** |
|  |  |  |  |  |  |  |

Table of Contents

[1. Basics of Web Application 6](#_Toc202264715)

[2. JDBC-Introduction 13](#_Toc202264716)

[ **JDBC Architecture :** 15](#_Toc202264717)

[Code Listing 1 37](#_Toc202264718)

[Code Listing 2 38](#_Toc202264719)

[Code Listing 3 38](#_Toc202264720)

[Code Listing 4 39](#_Toc202264721)

[Code Listing 5 40](#_Toc202264722)

[Code Listing 6 41](#_Toc202264723)

[3. JDBC Prepared Statement & Callable Statement 42](#_Toc202264724)

[ L3.4.3.3 Creating a Data Source 53](#_Toc202264725)

[\*DataDirect Connect *for* JDBC - Supports Oracle, SQL Server, DB2, Sybase, \*Informix, and MySQL 53](#_Toc202264726)

[ L3.4.3.4 Connecting to a Data Source 56](#_Toc202264727)

[ L3.6.2 Using CallableStatement methods to call stored procedures 59](#_Toc202264728)

[http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.doc/ad/tjvcscsp.htm 59](#_Toc202264729)

[ L3.7.2 Setting and Rolling Back to a Savepoint 62](#_Toc202264730)

[ L3.7.3 Releasing a Savepoint 62](#_Toc202264731)

[o The method Connection.releaseSavepoint takes a Savepoint object as a parameter and removes it from the current transaction. 62](#_Toc202264732)

[o Once a savepoint has been released, attempting to reference it in a rollback operation causes an SQLException to be thrown. 62](#_Toc202264733)

[o Any savepoints that have been created in a transaction are automatically released and become invalid when the transaction is committed, or when the entire transaction is rolled back. 62](#_Toc202264734)

[o Rolling a transaction back to a savepoint automatically releases and makes invalid any other savepoints that were created after the savepoint in question. 62](#_Toc202264735)

[JDBC 3.0 63](#_Toc202264736)

[Auto-Generated Keys 64](#_Toc202264737)

[Conclusion 64](#_Toc202264738)

[Code Listing 1 65](#_Toc202264739)

[Code Listing 2 66](#_Toc202264740)

[Code Listing 3 67](#_Toc202264741)

[Code Listing 4 68](#_Toc202264742)

[Code Listing 5 69](#_Toc202264743)

[Code Listing 6 70](#_Toc202264744)

[4.JDBC-Advanced concepts 72](#_Toc202264745)

[Code Listing 1 81](#_Toc202264746)

[Code Listing 2 81](#_Toc202264747)

[Code Listing 3 81](#_Toc202264748)

[Code Listing 4 83](#_Toc202264749)

[5. Servlets Basics 85](#_Toc202264750)

[Code Listing 1 108](#_Toc202264751)

[Code Listing 2 111](#_Toc202264752)

[**Application structure** 124](#_Toc202264753)

[**Deployment descriptor** 125](#_Toc202264754)

[**Specifying the servlet details** 125](#_Toc202264755)

[**Servlet mappings** 126](#_Toc202264756)

[**Using URL paths** 126](#_Toc202264757)

[7. Servlets – JDBC & Session Tracking 129](#_Toc202264758)

[8.JSP Introduction and Scripting Elements 143](#_Toc202264759)

[9.JSP DIRECTIVES AND JSP ACTIONS 150](#_Toc202264760)

[SOLUTIONS 161](#_Toc202264761)

[java.sql.SQLException: [Oracle][ODBC][Ora]ORA-01756: quoted string not properly terminated. 161](#_Toc202264762)

[java.sql.SQLException: [Oracle][ODBC]Invalid transaction state. 162](#_Toc202264763)

[java.sql.SQLException: [Oracle][ODBC][Ora]ORA-00913: too many values. 162](#_Toc202264764)

[java.sql.SQLException: [Oracle][ODBC][Ora]ORA-01722: invalid number 162](#_Toc202264765)

[unexpected error EXCEPTION\_ACCESS\_VIOLATION 162](#_Toc202264766)

[When we close the connection and then try using it or don’t create a connection then we may get this error 162](#_Toc202264767)

[java.sql.SQLException: [Oracle][ODBC]Invalid column number <7>. 163](#_Toc202264768)

[When we try to access an index of a table which does not exist then we get such a kind of error. 163](#_Toc202264769)

[sun.jdbc.odbc.JdbcOdbcBatchUpdateException: [Oracle][ODBC][Ora]ORA-00001: unique constraint (SCOTT.SYS\_C0060855) violated 163](#_Toc202264770)

[When we try to update a particular value in a table which is already defined as primary key then we get such kind of exception 163](#_Toc202264771)

[java.sql.SQLException: [Oracle][ODBC][Ora]ORA-06550: wrong number or types of arguments in call to 'MYPROCIN' 163](#_Toc202264772)

[java.sql.SQLException: [Oracle][ODBC][Ora]ORA-06550: object SCOTT.MYPROCOUT is invalid 163](#_Toc202264773)

[java.sql.SQLException: [Oracle][ODBC][Ora]ORA-06550: object SCOTT.MYPROCINOUT is invalid 163](#_Toc202264774)

[Variable outParam might not have been initialized 163](#_Toc202264775)

[When we try to call a particular procedure in oracle with wrong type of arguments as parameters then we get such kind of exception 163](#_Toc202264776)

[java.sql.SQLException: [Microsoft][ODBC Driver Manager] Invalid cursor state 163](#_Toc202264777)

[Since the cursor is at the last position it can't go to the next position 163](#_Toc202264778)

[Instead of using .next() method use .previous() 163](#_Toc202264779)

[java.sql.SQLException: Cursor position (0) is invalid 163](#_Toc202264780)

[As the exception says the cursor position is invalid. 163](#_Toc202264781)

[use a valid cursor position 163](#_Toc202264782)

[java.sql.SQLException: Column not found 164](#_Toc202264783)

[As the exception says the column is invalid. 164](#_Toc202264784)

[java.sql.SQLException: Invalid Cursor Type: 1003 164](#_Toc202264785)

[As the exception says the column is invalid. 164](#_Toc202264786)

[Exception in thread "main" java.lang.NullPointerException 164](#_Toc202264787)

[A NullPointerException (NPE) is thrown when your code tries to access a member (either method or variable) of an Object that has not been instantiated 164](#_Toc202264788)

[javax.servlet.ServletException: Class com.NewServlet is not a Servlet 165](#_Toc202264789)

[When we dont extend the created class with either HttpServlet or GenericServlet then we come across such a kind of error 165](#_Toc202264790)

[Extend the servlet class with either GenericServlet or HttpServlet to make it a servlet. 165](#_Toc202264791)

[unreported exception java.io.IOException; must be caught or declared to be thrown 165](#_Toc202264792)

[HTTP Status 405 - HTTP method GET is not supported by this URL 165](#_Toc202264793)

[The specified HTTP method is not allowed for the requested resource (HTTP method GET is not supported by this URL) as in the html file we have specified the GET method but we are using the POST method in the servlet 165](#_Toc202264794)

[There is no error. But There is an BUG cause this application will always cause the Login failure to get executed 165](#_Toc202264795)

[As we have used the SERVICE method in the Servlet, but the method used in the Html is POST so the values are not passed into the getparametter hence the login is always a failure. 165](#_Toc202264796)

[HTTP Status 404 - Login 165](#_Toc202264797)

[The requested resource is not available. 165](#_Toc202264798)

[The 404 or Not Found error message is an HTTP standard response code indicating that the 166](#_Toc202264799)

[client was able to communicate with the server but either the server could 0not find what 166](#_Toc202264800)

[was requested, or it was configured not to fulfill the request and not reveal the reason 166](#_Toc202264801)

[why. 404 errors should not be confused with "server not found" or similar errors, in which 166](#_Toc202264802)

[a connection to the destination server cannot be made at all. 166](#_Toc202264803)

[Check the deployment descriptor if the server mapping is missing then it may give rise 166](#_Toc202264804)

# 1. Basics of Web Application

##### Objectives:

* Understand the Web applications architecture
* Understand J2EE platform

## L1.1 FOUNDATION

##### L1.1.1 Web page & Web site

**Web pages**

* Computer document available on the world-wide-web, written in HTML.
* Web pages are linked together by hyperlinks and are often collected under one broad address or website.

**Web site**

* A set of interconnected web pages, stored on the world-wide-web sharing a common *domain name.*
* Each website is usually hosted on the same server (computer) and is maintained by the same individual, group or organization.

##### L1.1.2 Static & Dynamic Web pages

**Static Page**

* A web page that always comprises the same information in response to *all* download requests from *all* users.
* Contrast with Dynamic web page.
* It displays the same information for all users, from all contexts, providing the classical hypertext, where navigation is performed through "static" documents.

**Advantages**

Quick and easy to put together, even by someone who doesn't have much experience.Ideal for demonstrating how a site will look.Cache friendly, one copy can be shown to many people.

**Disadvantages**

* Difficult to maintain when a site gets large.
* Difficult to keep consistent and up to date.
* Offers little visitor personalization (all would have to be client side).

**Dynamic Page**

* Classical hypertext navigation occurs among "static" documents.
* However, web navigation also provides an *interactive experience* that is termed "*dynamic*". There are two ways to create this kind of interactivity:
* Using client-side scripting to change interface behaviors **within** a specific web page, in response to mouse or keyboard actions or at specified timing events. In this case the dynamic behavior occurs within the presentation.
* Using server-side scripting to change the supplied page source **between** pages. Server responses may be determined by such conditions as data in a posted HTML form, parameters in the URL, the type of browser being used, the passage of time, or a database or server state.

##### L1.1.3 Protocol- URL & URI

**Protocol**

* A protocol is a convention or standard that controls or enables the connection, communication, and data transfer between two computing endpoints.
* The rules governing the syntax, semantics, and synchronization of communication.
* Object-oriented programming has extended the use of the term to include the programming protocols available for connections and communication between objects.

**URL**

* Uniform Resource Locator is address of a resource on the Internet. The resource can be any type of file stored on a server, such as a Web page, a text file, a graphics file, or an application program.
* The address contains three elements:
  + the type of protocol used to access the file
  + the domain name or IP address of the server where the file resides
  + the pathname to the file

**URI**

* A Uniform Resource Identifier is a compact string of characters used to identify or name a resource.
* The main purpose of this identification is to enable interaction with representations of the resource over a network.
* URIs are defined in schemes defining a specific syntax and associated protocols.

##### L1.1.4 3/n Tier Architecture

*N-tier architecture* is a client-server architecture in which an application is executed by more than one distinct software agent. The most widespread use of "multi-tier architecture" refers to ***three-tier architecture***

* Three-tier architecture
* '*Three-tier'* is client-server architecture.
* In this the user interface, functional process logic ("business rules"), computer data storage and data access are developed and maintained as independent modules, most often on separate platforms.
* The 3-Tier architecture has the following three tiers:
* **Presentation Tier :** 
  + The top most level of the application.
  + It communicates with other tiers by outputting results to the browser/client tier and all other tiers in the network.
* **Application Tier (Business Logic/Logic Tier)** 
  + The logic tier is pulled out from the presentation tier.
  + It has its own layer, to control an application’s functionality by performing detailed processing.
* **Data Tier** 
  + This tier consists of Database Servers.
  + Here information is stored and retrieved.
  + This tier keeps data neutral and independent from application servers or business logic.
  + Giving data its own tier also improves scalability and performance.

## L1.2 JAVA TECHNOLOGY

##### L1.2.1 Flavors of JAVA

* **Java Platform, Standard Edition(J2SE)**
* Java Platform, Standard Edition (also known as Java 2 Platform) lets you develop and deploy Java applications on desktops and servers, as well as today's demanding [embedded](http://java.sun.com/j2se/embedded/) and Real-Time environments.
* Java SE includes classes that support the development of Java Web Services and provides the foundation for Java Platform, Enterprise Edition (Java EE).
* There are two principal products in the Java SE platform family: Java SE Runtime Environment (JRE) and Java Development Kit (JDK).

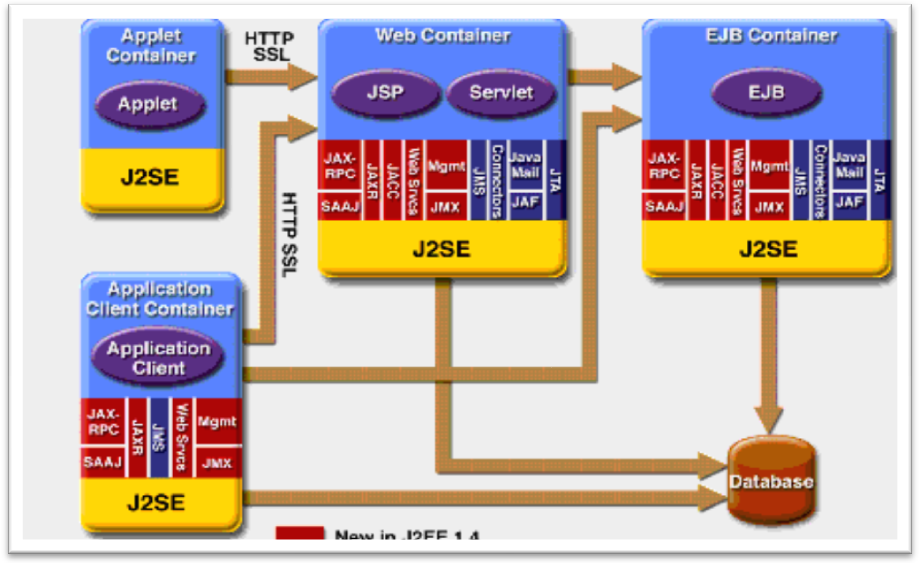
**Java Runtime Environment (JRE) :**

* + - The Java Runtime Environment (JRE) provides the libraries, the Java Virtual Machine, and other components to run applets and applications written in the Java programming language.
    - It is also the foundation for the technologies in the Java 2 Platform, Enterprise Edition (J2EE) for enterprise software development and deployment
    - The JRE does not contain tools and utilities such as compilers or debuggers for developing applets and applications.

**Java Development Kit (JDK) :**

* + - The JDK is a superset of the JRE, and contains everything that is in the JRE, plus tools such as the compilers and debuggers necessary for developing applets and applications.
* **Java Platform, Enterprise Edition(J2EE)**
  + Java Platform, Enterprise Edition (Java EE) builds on the solid foundation of Java Platform, Standard Edition (Java SE).
  + Java EE 5 focuses on making development easier, yet retains the richness of the J2EE 1.4 platform.
  + It consists of Java Servlet, JSP, EJB, JMS, JDBC, JNDI, JTA, JavaMail, JAAS, XML, etc.
* **Java 2 Platform Micro Edition(J2ME)**
  + It is very new platform with highly optimized JRE.
  + It is used in devices with limited memory and/or processor power.
  + It uses virtual machine’s internal format instead of the normal class-file format some that replace similar J2SE classes and others that provide new functionality.
  + It defines the minimum requirements to run Java on a family of devices that have similar characteristics

##### L1.2.2 J2EE Big Picture



## L1.3 CASE STUDY

**JIM:** this was warm up session; now want to play some game

**JOE:**

* **SUMMARY**
* Web page is a computer document available on the world-wide-web, written in HTML
* Web Site is a set of interconnected web pages, stored on the world-wide-web sharing a common domain name.
* A web page that always comprises the same information in response to *all* download requests from *all* users is a Static web page.
* Static web pages are difficult to maintain, keep consistent and up to date.
* Web pages that provide navigation experiencing client server interaction are dynamic pages.
* A protocol is a convention or standard that controls or enables the connection, communication, and data transfer between two computing endpoints.
* URL is address of a resource on the Internet, while URI is a compact string of characters used to identify or name a resource.
* N-tier architecture is a client-server architecture in which an application is executed by more than one distinct software agent.
* The 3-Tier architecture has the three tiers: Presentation Tier, Application Tier (Business Logic/Logic Tier) and Data Tier.
* Java Platform, Standard Edition (also known as Java 2 Platform) lets you develop and deploy Java applications on desktops and servers.
* There are two principal products in the Java SE platform family: Java SE Runtime Environment (JRE) and Java Development Kit (JDK).
* J2EE builds on the solid foundation of J2SE, consisting of Java Servlet, JSP, EJB, JMS, JDBC, JNDI, JTA, JavaMail, JAAS, XML, etc.
* **REVIEW**
* **CROSS WORD PUZZLE**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2. JDBC-Introduction  * **Course Objectives :**   Upon completion of this Aurora Information Systems course, students will be able to   * + - * Connect to a database using JDBC with all types of Drivers.       * Create and alter database tables       * Access and modify data contained in a database       * DatabaseMetaData methods. * **Prerequisites :**   Experience in Java Programming is essential. The student must be comfortable with object-oriented Java, interfaces and abstract classes. Understanding of basic SQL usage or other work with relational databases will be helpful.   * **What is jdbc :**   Short for *Java Database Connectivity,* a Java API that enables Java programs to execute SQL statements. This allows Java programs to interact with any SQL-compliant database. Since nearly all relational database management systems (DBMSs) support SQL, and because Java itself runs on most platforms, JDBC makes it possible to write a single database application that can run on different platforms and interact with different DBMSs.   * **What is Java Application :**   It can be a stand alone java program , which uses the jdbc API To get connected and perform operations on the database data.   * **What is jdbc API :**   It is a set of classes and interfaces used in a java program for database operations.Java.sql packages provide the necessary library support.   * **Java Database Connectivity (JDBC)**: * The Java Database Connectivity (JDBC) abstracts Database Connectivity from the Java Application. * It allows Java programmers to write a database application that can run on different platforms, i.e. database can be transferred from one vendor to another and the same java program can be used without alterations. * It also extends JDBC technology beyond the client to the server with connection pooling and distributed transactions. * **Java Database Connectivity API:** * The Java Database Connectivity (JDBC) API provides universal data access from the Java programming language. * The JDBC API defines interfaces, classes and Exceptions for writing database applications in Java by making database connections. * Using these interfaces and classes, Java code can connect to the database, send SQL statements, query the tables in the database and retrieve the results. * The value of the JDBC API is that an application can access virtually any data source and run on any platform with a Java Virtual Machine. * JDBC API includes: * **Interfaces:**   + **Driver**: The interface that every driver class must implement.   + **Connection**: It is a connection/communication channel with the database.   + **DatabaseMetaData**: This interface is implemented by driver vendors to let users know the capabilities of a Database Management System (DBMS).   + **Statement**: The object used for executing a static SQL statement and returning the results it produces.   + **ResultSet**: It is used to store a database result generated by using a SELECT query.   + **ResultSetMetaData**: An object that can be used to get information about the types and properties of the columns in a ResultSet object   + **PreparedStatement**: An object that represents a precompiled SQL statement.   + **CallableStatement**: The interface used to execute SQL stored procedures.     - **Classes:**       * **Types**: This class defines the constants that are used to identify generic SQL Types.       * **DriverManager**: Class which is responsible for selecting the database and creating the database connection.       * **DriverPropertyInfo**: Describes Driver properties for making a connection.       * **Date**: A thin wrapper around a millisecond value that allows JDBC to identify this as an SQL DATE value.       * **SQLPermssion**: The permission for which the SecurityManager checks.       * **Time**: A thin wrapper around the java.util.Date class that allows the JDBc API to identify this as an SQL TIME value.       * **TimeStamp**: A thin wrapper around the java.util.Date class that allows the JDBc API to identify this as an SQL TIMESTAMP value.         + **Exceptions:**       * SQLException: An exception that provides information on a database access error or other errors.       * SQLWarning: An exception that provides information on a database access warning.       * BatchUpdateException: Exception thrown when an error occurs during a batch update operation.       * DataTruncationException: An exception thrown as a DataTruncation exception when a data values is unexpectedly truncated. * **JDBC Architecture :**   Applications can access databases via the JDBC API using any of the JDBC drivers as shown in this:    http://www.jdbc-tutorial.com/images/jdbc.jpg   * The JDBC API supports both two-tier and three-tier processing models for database access.   **Two-tier Architecture for Data Access.**    The DBMS-proprietary protocol provides two-way communication between the client machine and the database server   * In the two-tier model, a Java application talks directly to the data source. This requires a JDBC driver that can communicate with the particular data source being accessed. * A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user.   **Three-tier Architecture for Data Access.**  The DBMS-proprietary protocol provides two-way communication between the database server and the server machine. HTTP, RMI, CORBA or other calls provide two way communication between the server machine and the client machine     * In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source. * The data source processes the commands and sends the results back to the middle tier, which then sends them to the user. * The advantage over 2 tier model is that the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. * Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages. * **JDBC Drivers:** * A JDBC driver translates standard JDBC calls into a network or database protocol or into a database library API call that facilitates communication with the database. * If the back-end database changes, only the JDBC driver need be replaced with few code modifications. * There are four distinct types of JDBC drivers:     **Type 1:** JDBC-ODBC Bridge driver (Bridge)  **Type 2:** Native-API/partly Java driver (Native)  **Type 3:** AllJava/Net-protocol driver (Middleware)  **Type 4:** All Java/Native-protocol driver (Pure)   * **Type 1 Driver :**   The type 1 driver, JDBC-ODBC Bridge, translates all JDBC calls into ODBC (Open DataBase Connectivity) calls and sends them to the ODBC driver. As such, the ODBC driver, as well as, in many cases, the client database code, must be present on the client machine. Figure shows a typical JDBC-ODBC environment  http://www.mscs.mu.edu/~schwerm/TechInfo/jdbc/images/type1.gif  **Type 1: JDBC-ODBC Birdge**  **Advantages :**   * The JDBC-ODBC Bridge allows access to almost any database, since the database's ODBC drivers are already available. * Type 1 drivers may be useful for those companies that have an ODBC driver already installed on client machines.   **Disadvavtages :**   * The performance is degraded since the JDBC call goes through the bridge to the ODBC driver, then to the native database connectivity interface. The result comes back through the reverse process. Considering the performance issue, type 1 drivers may not be suitable for large-scale applications. * The ODBC driver and native connectivity interface must already be installed on the client machine. Thus any advantage of using Java applets in an intranet environment is lost, since the deployment problems of traditional applications remain. * **Type 2 Driver :**   The distinctive characteristic of type 2 jdbc drivers are that Type 2 drivers convert JDBC calls into database-specific calls i.e. this driver is specific to a particular database. Some distinctive characteristic of type 2 jdbc drivers are shown below. Example: Oracle will have oracle native api.  http://www.jdbc-tutorial.com/images/jdbc-type2-driver.JPG  **Advantages**   * Type 2 drivers typically offer better performance than the JDBC-ODBC bridge as the layers of communication (tiers) are less than that of Type 1. * It uses Native API which is database specific.   Disadvantages   * Native API must be installed in the Client System and hence type 2 drivers cannot be used for the Internet. * Like Type 1 drivers, it’s not written in Java Language which forms a portability issue. * If we change the Database we have to change the native api as it is specific to a database. * Mostly obsolete now. * Usually not thread safe. * **Type 3 Driver:**   Type 3 database requests are passed through the network to the middle-tier server. The middle-tier then translates the request to the database. If the middle-tier server can in turn use Type1, Type 2 or Type 4 drivers.  http://www.jdbc-tutorial.com/images/jdbc-type3-driver.JPG   * The JDBC Type 3 driver, also known as the **Pure Java Driver for Database Middleware** is a database driver implementation which makes use of a [middle-tier](http://en.wikipedia.org/w/index.php?title=Middle-tier&action=edit) between the calling program and the database. * The middle-tier (Application Server) converts [JDBC](http://en.wikipedia.org/wiki/JDBC) calls directly or indirectly into the vendor-specific [Database](http://en.wikipedia.org/wiki/Database) protocol. * This differs from the Type 4 driver in that the protocol conversion logic resides not at the client, but in the middle-tier. * Type 3 drivers can be used for multiple databases; it depends on the number of databases the middleware has been configured to support. * The type 3 driver is [platform-independent](http://en.wikipedia.org/wiki/Platform-independent) as the platform-related differences are taken care by the middleware which provides additional advantages of security and firewall access. * Client -> JDBC Driver -> Middleware-Net Server -> Any Database   **Advantages:**   * Since the communication between client and the Middleware Server is database independent, there is no need for the vendor database library on the client machine. * The Middleware Server (can be a full fledged J2EE Application server) can provide typical middleware services like caching (connections, query results, and so on), load balancing, logging, auditing etc. * Can be used for internet application since there is no client side software needed. * At client side a single driver can handle any database(It works provided the middlware supports that database).   **Disadvantages:**   * Requires database-specific coding to be done in the middle tier. * An extra layer added may result in a time-bottleneck. * **Type 4 Driver:** * The JDBC Type 4 driver, also known as the Direct to Database, Pure Java Driver is a database driver implementation that converts [JDBC](http://en.wikipedia.org/wiki/JDBC) calls directly into the vendor-specific [database](http://en.wikipedia.org/wiki/Database) protocol. * The type 4 driver is written completely in [Java](http://en.wikipedia.org/wiki/Java_%28programming_language%29) and is hence [platform independent](http://en.wikipedia.org/wiki/Platform_independence). * It is installed inside the [Java Virtual Machine](http://en.wikipedia.org/wiki/Java_Virtual_Machine) of the client. * It provides better performance over the Type 1 and Type 2 drivers as it does not have the overhead of conversion of calls into ODBC or database API calls. * As the database protocol is vendor-specific, separate drivers, usually vendor-supplied, need to be used to connect to the database.   http://www.jdbc-tutorial.com/images/jdbc-type4-driver.JPG  Advantage   * The major benefit of using a type 4 jdbc drivers are that they are completely written in Java to achieve platform independence and eliminate deployment administration issues. It is most suitable for the web. * Number of translation layers is very less i.e. type 4 JDBC drivers don't have to translate database requests to ODBC or a native connectivity interface or to pass the request on to another server, performance is typically quite good. * You don’t need to install special software on the client or server. Further, these drivers can be downloaded dynamically.   Disadvantage   * With type 4 drivers, the user needs a different driver for each database.  * **Now we will learn programming part of the JDBC** * **JDBC Usage Sequence**   JDBC usage sequence   * establish a connection with a database * send SQL statements * process the result * close Statement * close Connection      * **Main Coding Sequence**  1. Load Driver 2. Create Connection 3. Create Statement 4. Execute Statement and get Results 5. Close all the objects created   **1** **LoadDriver :**  Two ways to load and register the Driver;  try     { **Class.forName**(“oracle.jdbc.driver.OracleDriver");} // for Oracle  {**Class.forName**(“oracle.gjt.mm.mysql.Driver”); } // for MySQL catch(SQLException e)    {....}  *JRE loads the Driver and the static constructor in the Driver is executed and it registers itself with the DriverManager*        ~ ~ ~ OR ~ ~ ~  DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());  *We are asking the JRE to first create an instance of the Driver, which in effect is loading it; then the registerDriver() registers the Driver with the DriverManager.*  **2 Create Connection :**  Connection conn;  conn = DriverManager.getConnection("protocol:subprotocol:url")  **Type1**:JDBC-ODBCBridge    Connection conn=DriverManager.getConnection("jdbc:odbc:ORCL");    Where:  jdbc= protocol  odbc=subprotocol ORCL= data source name (defined in ODBC)  **Type 2**: Partial Java and Native DBMS API    Connection conn=DriverManager.getConnection("jdbc.oracle.oci8:SCOTT/TIGER@webtech");    Where:  jdbc= protocol  oracle=subprotocol oci8=DBMS API (oracle call interface) SCOTT/TIGER@webtech= URL [username/password @ database name]  **Type 3**: Generic Network API  **Type 4**: Pure Java and DBMS network protocol    Connection conn=DriverManager. getConnection("jdbc:oracle:thin:@nowmana:1521:webtech","scott","tiger");    Where:  jdbc= protocol  oracle=subprotocol thin= network protocol @nowmana=host name 1521= port number of listener webtech=database name scott=username tiger=password  **3 Create Statement**  Statement stmt;  stmt = conn.createStatement();        // *this is an object that is the container to hold an sql request*  **4 Execute Statement and get Results**  Resultset rset;  rset = stmt.executeQuery("select col\_name from tbl\_name where condition");        // *the sql can be coded within the method or outside and referenced*  **5 Close all the objects created**  rset.close();  stmt.close();  conn.close();        *The Result Set, Statement, and Connection objects are closed.*   * **Examples** :   1 : Basic example of creating table with jdbc    public class jdbc{  public static void main(String[] args) throws SQLException {  try{  *// Load the Driver*  Class.forName("oracle.jdbc.driver.OracleDriver");  System.out.println("Driver loaded");    *// Establish a connection*  Connection conn=DriverManager.  getConnection("jdbc:oracle:thin:@hstslc007:1521:eltp","scott","tiger"); //Type 4  System.out.println("Database Connected");  *// Create Statement*  Statement st=conn.createStatement();  *// Execute Statement*  st.execute("create table prasoon\_singh(no number(10), name varchar2(20), sal  number(6,2*))");// Any SQL query can be written here.*  System.out.println("table created ");  st.close();  conn.close();  //Connection Closed  }  catch(Exception e)  {  System.out.println("error"+e);  }  }  }  **Exercise :**   * 1. : Following ex 1 , In that ex database manager has created the table and to help himself he has hired you to insert the following data in the same table -:  |  |  |  | | --- | --- | --- | | No | Name | Sal | | 1 | John | 20,000 | | 2 | Sam | 15,000 |   Can you please help him ?  2 : Now since Mr Sam has left the job please delete his record from the table and Replace him with the new joinee having name Cally and salary whatever you  want to give her .  **Examples :**    1. //Creating connection using Type 1 driver  import static java.lang.System.\*;  import java.sql.DriverManager;  import java.sql.Connection;  import java.sql.SQLException;  import java.util.\*;  public class TypeOneConnection {  public static void main(String args[]){  try{  //registering Type 1 driver  DriverManager.registerDriver(new sun.jdbc.odbc.JdbcOdbcDriver());  Enumeration DriverName=(Enumeration)DriverManager.getDrivers();  out.println("Driver Name : "+DriverName.toString());  //connecting to database  Connection conn = DriverManager.getConnection("jdbc:odbc:msadsn","satyam","satyam");  if(conn!=null)  out.println("Connection created successfully");  out.println("Connection Obejct: "+conn);  out.println("is connection closed : "+conn.isClosed());  out.println("closing the connection");  conn.close(); //closing the connection  out.println("is connection closed : "+conn.isClosed());  } catch(SQLException e) {  e.printStackTrace();  } catch(Exception ee) {  ee.printStackTrace();  }  }  }  Output:  Driver Name : java.util.Vector$1@1a758cb  Connection created successfully  Connection Obejct: sun.jdbc.odbc.JdbcOdbcConnection@e09713  is connection closed : false  closing the connection  is connection closed : true   1. //creating connection using Type 2 driver   import static java.lang.System.\*;  import java.sql.DriverManager;  import java.sql.SQLException;  import java.sql.Connection;  import java.util.\*;  public class TypeTwoConnection {  public static void main(String args[]){  try{  Class.forName("oracle.jdbc.driver.OracleDriver");  //if the driver is registered getDrivers() will return the Driver name  Enumeration DriverName=(Enumeration)DriverManager.getDrivers();  //printing the driver name  out.println("Driver Name: "+DriverName.toString());  Connection conn = DriverManager.getConnection("jdbc:oracle:oci8:@eltp","scott","tiger");  if(conn!=null)  out.println("Connection created successfully");  conn.setAutoCommit(false);  out.println("Auto commit :"+conn.getAutoCommit());  out.println("Database Metadata : "+conn.getMetaData());  out.println("is connection closed : "+conn.isClosed());  out.println("closing the connection");  conn.close(); //closing the connection  out.println("is connection closed : "+conn.isClosed());  } catch(ClassNotFoundException ce) {  ce.printStackTrace();  } catch(Exception ee) {  ee.printStackTrace();  }  }  }  Output:  Driver Name: java.util.Vector$1@e09713  Connection created successfully  Auto commit :false  Database Metadata : oracle.jdbc.driver.OracleDatabaseMetaData@1621e42  is connection closed : false  closing the connection  is connection closed : true  3. //creating connection using Type 4 driver  import static java.lang.System.\*;  import java.sql.DriverManager;  import java.sql.SQLException;  import java.sql.Connection;  import java.util.\*;  public class TypeTwoConnection {  public static void main(String args[]){  try{  DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());  Enumeration DriverName=(Enumeration)DriverManager.getDrivers();  out.println("Driver Name: "+DriverName.toString());  Connection conn = DriverManager.getConnection("jdbc:oracle:thin:@172.16.155.6:1521:eltp","scott","tiger");  if(conn!=null)  out.println("Connection created successfully");  out.println("Connection Obejct: "+conn);  out.println("is connection closed : "+conn.isClosed());  out.println("closing the connection");  conn.close(); //closing the connection  out.println("is connection closed : "+conn.isClosed());  } catch(ClassNotFoundException ce) {  ce.printStackTrace();  }  catch(Exception ee) {  ee.printStackTrace();  }  }  }  Output:  Driver Name: java.util.Vector$1@e09713  Connection created successfully  is connection closed : false  closing the connection  is connection closed : true   * **ResultSet :**   We have learned how to create the table and insert or update the data in it with the help of jdbc . But retrieving data from the table is also one of the critical issue in JDBC programming.We can use **ResulSet** to get this objective.   * The ResultSet interface provides methods for retrieving and manipulating the results of executed queries. * The rows that satisfy a particular query are called the result set. * The number of rows returned in a result set can be zero or more. * A user can access the data in a result set using a cursor one row at a time from top to bottom. * A cursor can be thought of as a pointer to the rows of the result set that has the ability to keep track of which row is currently being accessed. The JDBC API supports a cursor to move both forward and backward and also allowing it to move to a specified row or to a row whose position is relative to another row. * Initially the cursor is positioned before the first row. * The next method moves the cursor to the next row.   We can retrieve the data in two ways -:  Schema given to us is i.e Index 1 = id  Index 2 = Name  Index 3 = Address   |  |  |  | | --- | --- | --- | | id | Name | Address | |  |  |  |   **Method 1: Retrieving the Value of the Address Column Using the Index Number**  The following code shows how to retrieve the value of the address column using the index number:  ResultSet rs = null;  Statement stmt = null;  Connection conn = null;  try {  conn = getConnection(); // get a Connection object  // create a result set containing all data  // from your desired table  stmt = conn.createStatement();  String query = "SELECT id, name, address FROM employees";  rs = stmt.executeQuery(query);  // Fetch each row from the result set  while (rs.next()) {  // Get the data from the row using the column name  // note that using a column index is better than  // using the column name: using column name might  // add overhead: there is a need to  // get column metadata info.  String employeeAddress = rs.getString(3); // Address  ...  }  }  catch (SQLException e) {  // handle the exception  ...  }  **Method 2: Retrieving the Value of the Address Column Using the Column Name**  The following code shows how to retrieve the value of the address column using the column name:  ResultSet rs = null;  Statement stmt = null;  Connection conn = null;  try {  conn = getConnection(); // get a Connection object  // create a result set containing all data  // from your desired table  stmt = conn.createStatement();  String query = "SELECT id, name, address FROM employees";  rs = stmt.executeQuery(query);  // Fetch each row from the result set  while (rs.next()) {  // Get the data from the row using the column name  // note that using a column name might add  // overhead: there is a need to get column metadata info.  String employeeAddress = rs.getString("address");//index 3  ...  }  }  catch (SQLException e) {  // handle the exception  ...  }  finally {  // close ResultSet, Statement, Connection  }  For both methods (using the index of the column and using the column name), you invoke an  appropriate method to extract the data value of the given column.   * **Example :**   **Setting Up the Oracle Database**  The following code shows how to set up the Oracle database:  SQL> create table employees (  id varchar(10) not null primary key,  name varchar(20) not null,  age int  );  //*Table created*.  SQL> desc employees;  Name Null? Type  ---------------- -------- ------------  ID NOT NULL VARCHAR2(10)  NAME NOT NULL VARCHAR2(20)  AGE NUMBER(38)  *// insertion*  SQL> insert into employees(id, name, age) values('11', 'Alex Smith', 25);  SQL> insert into employees(id, name, age) values('22', 'Don Knuth', 65);  SQL> insert into employees(id, name, age) values('33', 'Mary Kent', 35);  SQL> insert into employees(id, name, age) values('44', 'Monica Seles', 30);  SQL> insert into employees(id, name) values('99', 'Alex Edison');  SQL> commit;  Commit complete.  SQL> select id, name, age from employees;  ID NAME AGE  ---------- -------------------- ----------  11 Alex Smith 25  22 Don Knuth 65  33 Mary Kent 35  44 Monica Seles 30  99 Alex Edison  We have created the table now we wil provide a simple Java class to demonstrate how to use ResultSet by querying the employees table and retrieving employee information from a database. Note that the age column can accept null values as well.  import java.sql.DriverManager;  import java.sql.Statement;  import java.sql.ResultSet;  import java.sql.ResultSetMetaData;  import java.sql.Driver;  import java.sql.Types;  import java.sql.SQLException;  import static java.lang.System.\*;   * public class employee{ * public static void main(String[] args) throws SQLException { * try{ * Connection conn = null; * Statement stmt = null; * ResultSet rs = null; * // Load the Driver * Class.forName("oracle.jdbc.driver.OracleDriver"); * System.out.println("Driver loaded"); * // Establish a connection * conn = DriverManager.getConnection("jdbc:oracle:thin:@hstslc011:1521:eltp" , "scott", "tiger"); * //conn=DriverManager.getConnection("jdbc:oracle:thin:@hstslc007:1521:eltp","scott","tiger"); //Type 4 * System.out.println("Database Connected"); * String query = "select \* from employees"; * // // create a statement * System.out.println(query); * stmt = conn.createStatement(); * // * // // execute query and return result as a ResultSet * // * rs = stmt.executeQuery(query); * // * // // extract data from the ResultSet * // * while (rs.next()) { * String id = rs.getString(1); * System.out.println("id="+id); * String name = rs.getString(2); * System.out.println("name="+name); * // age might be null (according to schema) * int age = rs.getInt(3); * if (rs.wasNull()) { * System.out.println("age=null"); * } else { * System.out.println("age="+age); * } * System.out.println("---------------"); * } * System.out.println("--DemoResultSet end--"); * } catch(Exception e){ * e.printStackTrace(); * System.exit(1); * } * } * }   **The important methods and concepts of this Examples are as follows:**  • **The getConnection**() method gets a database connection for the sample Oracle database.  • **createStatement()** creates a Statement object for sending SQL statements to the database.  • **executeQuery()** is used for Statement objects that return a ResultSet, which is basically  a SELECT statement.  • **next()** moves the cursor down one row from its current position. The first next() sets the  cursor on the first row; the next() method enables you to iterate through all the records  retrieved.  **• getString(int columnIndex)** retrieves the value of the designated column in the current row  of this ResultSet object as a String in the Java programming language. You use getString(1)  and getString(2) to get the id and name, respectively. (In the query, id is defined in the first  column, and name is defined in the second column.) Note that when using column index, the  first column is 1, the second is 2, and so on.  • **getInt(int columnIndex)** retrieves the value of the designated column in the current row of  this ResultSet object as an “int” in the Java programming language. You use getInt(3) to get  the age of an employee (note that age is defined in the third column of the query).  **• wasNull()** reports whether the last column read had a value of SQL NULL.  **• The ResultSet object’s getXXX() methods (such as getString() and getInt())** retrieve column  data. JDBC defines types to match the SQL data types, and there is a getXXX() method for  each. You can use the getXXX() method in two ways with the same semantics. (You can retrieve  the value of the designated column in the current row of this ResultSet object as an XXX type.)  • **getXXX(int columnIndex)** is the preferred way of getting data since there is no need to get the  column’s metadata information.  **• getXXX(String columnName)** might be a little bit slow because of getting the column’s metadata information.   * **Exercise :**   The database manager is finding some difficulty in retrieving the data for employee  Whose id is 22.Can you write the code to retrieve the name of that person.If you find any  Problem , can take help of your friend or facilitator.   * **ResultSetMetaData ;**   public interface **ResultSetMetaData** : An object that can be used to get information about the types and properties of the columns in a ResultSet object. The following code fragment creates the ResultSet object rs, creates the ResultSetMetaData object rsmd, and uses rsmd to find out how many columns rs has and whether the first column in rs can be used in a WHERE clause.  ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM TABLE2");  ResultSetMetaData rsmd = rs.getMetaData();  int numberOfColumns = rsmd.getColumnCount();  boolean b = rsmd.isSearchable(1);  **Examples:**   1. //   import java.sql.DriverManager;  import java.sql.Connection;  import java.sql.Statement;  import java.sql.ResultSet;  import java.sql.ResultSetMetaData;  import java.sql.Driver;  import java.sql.Types;  import java.sql.SQLException;  import static java.lang.System.\*;  class DBConnection {  public static Connection getDBConnection(Driver driver, String url,String user, String passwd)throws SQLException{  return DriverManager.getConnection(url, user, passwd);  }  }  class GenericQueryExecutor {  public void executeAndPrintQueryResults(String query) throws SQLException{  Connection conn = DBConnection.getDBConnection(new oracle.jdbc.driver.OracleDriver(), "jdbc:oracle:thin:@hstslc:eltp", "scott", "tiger");  Statement stmt = conn.createStatement();  //System.out.println(stmt);  stmt.executeQuery(query);    ResultSet rs = stmt.getResultSet();  if(rs==null) {  out.println("query executed... no ResultSet found...");  stmt.close();  conn.close();  return;  }    ResultSetMetaData rsMetaData = rs.getMetaData();  printColumnNames(rsMetaData);  printQueryResults(rs, rsMetaData, rsMetaData.getColumnCount());  rs.close();  stmt.close();  conn.close();  }    private void printColumnNames(ResultSetMetaData rsMetaData) throws SQLException {  int columnCount = rsMetaData.getColumnCount();  for(int i = 1; i<=columnCount; i++)  out.print("\t\t\t"+rsMetaData.getColumnName(i));  out.println();  }    private void printQueryResults  (ResultSet rs,ResultSetMetaData rsMetaData,int columnCount) throws SQLException {  while(rs.next()) {  for(int colIndex = 1; colIndex <= columnCount; colIndex++) {  printColumnValue(rs, rsMetaData.getColumnType(colIndex), colIndex);  }  out.println();  }  }    private void printColumnValue(ResultSet rs, int columnType, int columnIndex) throws SQLException {  switch(columnType) {  case Types.BIGINT:  case Types.INTEGER:  out.print("\t\t\t"+rs.getLong(columnIndex));  break;    case Types.DECIMAL:  case Types.DOUBLE:  case Types.FLOAT:  out.print("\t\t\t"+rs.getDouble(columnIndex));  break;    case Types.CHAR:  case Types.VARCHAR:  out.print("\t\t\t"+rs.getString(columnIndex));  break;    default:  out.print("\t\t\t"+rs.getString(columnIndex));  break;  }  }  }  public class ResultSetMetaDataDemo {  public static void main(String... args) {  try {  GenericQueryExecutor GQExecutor = new GenericQueryExecutor();  //String query = "select \* from dept";  String query = "select " +  "empno, ename, deptno " +  "from emp " +  "where " +  "deptno in " +  "(10,30,40,60)";    GQExecutor.executeAndPrintQueryResults(query);  }catch(SQLException se) {  se.printStackTrace();  }  }  }  **Output:**  ENO ENAME DEPTNO  1001 king 10  1002 peter 30  1003 parker 60  **Exercise**  Objectives:   1. Please go through code listings. Identify the probable errors, either compile time or run time. 2. Document the Error, Explanation of the error and the probable solution/fix for the errors identified   Pre-requisites:   1. You must have a good conceptual understanding of Session 2 – “JDBC- Introduction” of the Intermediate Java course  Code Listing 1 **public static void main(String args[]) {**    **Connection con = null;**    **try {**  **// Driver name type 1 <Bridge driver>**  **Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");**    **} catch(java.lang.ClassNotFoundException e) {**  **System.out.print("ClassNotFoundException: ");**    **}**    **try {**    **con = DriverManager.getConnection("jdbc:odbc:jdbcdsn","scott","tiger");**  **if(!con.isClosed()) {**  **System.out.println("Connection Created Successful!!!!");**  **}**    **con.close();**    **} catch (SQLException e) {**  **e.printStackTrace();**  **}**    **}** Code Listing 2 **public static void main(String args[]) {**    **Connection con = null;**    **// Driver name type 1 <Bridge driver>**  **Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");**    **con = DriverManager.getConnection("jdbc:odbc:jdbcdsn","scott","tiger");**  **if(!con.isClosed())**  **{**  **System.out.println("Connection Created Successful!!!!");**  **}**    **con.close();**      **}** Code Listing 3 **Hint-Check the table for column definition.**  **public static void main(String args[]){**  **int empID=146;**  **String password="pwd";**  **String status="update";**  **int bankAcc=879;**  **String dept="warehouse";**  **int rows=0;**  **boolean result=false;**    **try{**  **Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");**  **Connection conn = DriverManager.getConnection("jdbc:odbc:mydsn","scott", "tiger");**  **Statement stmt = conn.createStatement();**  **try{**  **String query = "INSERT INTO project7181 VALUES(" +empID+ ", ' " +password+ "', " +bankAcc+ ", '" +dept+ "',)";**  **rows=stmt.executeUpdate(query);**  **if (rows>0){**  **result = true;**  **System.out.println("The number of rows inserted are " +rows);**  **}**  **else {**  **System.out.println("No rows inserted ");**  **}**  **stmt.close();**  **conn.close();**  **}**  **catch(Exception e){**  **System.out.println(e);**  **}**    **}**  **catch(Exception e){**  **e.printStackTrace();**  **}**    **}** Code Listing 4 **public static void main(String args[]){**    **try{**  **Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");**  **Connection conn = DriverManager.getConnection("jdbc:odbc:mydsn","scott", "tiger");**  **Statement stmt = conn.createStatement();**  **ResultSet rs=null;**  **try{**  **rs=stmt.executeQuery("select \* from project777;");**  **if(rs.next()){**  **System.out.println("Table exists");**  **}**    **stmt.close();**  **conn.close();**  **}**  **catch(Exception e){**  **System.out.println(e);**  **}**    **}**  **catch(Exception e){**  **e.printStackTrace();**  **}**    **}**   Code Listing 5 **public static void main(String args[]){**  **int empID=146;**  **String password="pwd";**  **String status="update";**  **int bankAcc=879;**  **String dept="warehouse";**  **String ques="Your mother's maiden name";**  **String ans="Rose";**  **int rows=0;**  **boolean result=false;**  **try{**  **Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");**  **Connection conn = DriverManager.getConnection("jdbc:odbc:mydsn","scott", "tiger");**  **Statement stmt = conn.createStatement();**  **try{**  **String query = "INSERT INTO project7181 VALUES(" +empID+ ", '" +password+ "', " +bankAcc+ ", " +dept+ ",'"+ques+"',"+ans")";**  **rows=stmt.executeUpdate(query);**  **if (rows>0){**  **result = true;**  **System.out.println("The number of rows inserted are " +rows);**  **}**  **else {**  **System.out.println("No rows inserted ");**  **}**  **}**  **catch(Exception e){**  **System.out.println(e);**  **}**  **stmt.close();**  **conn.close();**  **}**  **catch(Exception e){**  **e.printStackTrace();**  **}**  **}** Code Listing 6   **public static void main(String args[]) {**    **Connection connection = null;**  **Statement statement = null;**  **try {**  **// Driver name type 1 <Bridge driver>**  **Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");**  **}**  **catch(java.lang.ClassNotFoundException e) {**  **System.out.print("ClassNotFoundException: ");**  **}**  **try {**  **connection = DriverManager.getConnection("jdbc:odbc:jdbcdsn","scott","tiger");**  **if(!connection.isClosed()) {**  **System.out.println("Connection Created Successful!!!!");**  **}**    **statement = connection.createStatement();**  **connection.setAutoCommit(false);**  **ResultSet resultset = statement.executeQuery("select \* from project7181;");**  **while(resultset.next())**  **{**  **System.out.println(resultset.getString(1));**  **System.out.println(resultset.getString(2));**  **System.out.println(resultset.getString(3));**  **}**  **connection.close();**    **} catch (SQLException e) {**  **e.printStackTrace();**  **}**  **}** 3. JDBC Prepared Statement & Callable Statement **OBJECTIVE:**   * Understand need of database update * Use of batch update * Use of Connection Pooling * Understand the use of Statement, PreparedStatement and CallableStatement * Difference between Statement, PreparedStatement and CallableStatement * Understand database transaction roles   **L3.1 Need for Database Updates**   * If you're not familiar with the term Database as it relates to the web, a database is a structured collection of data. * It can be anything from a list of email addresses to a set of names, addresses and phone numbers. * In other words DBMS is computer software designed for the purpose of managing databases based on a variety of data models. * Adding, deleting or modifying data on timely basis with new information is ‘updating database’.   **L3.2 Statement Object:**   * There are three basic types of SQL statements used in the JDBC API:   + CallabelStatement,   + Statement and   + PreparedStatement * The object used for executing a static SQL statement and returning the results it produces. * One ResultSet object per Statement object can be open at a time. * Reading of one ResultSet object is interleaved with the reading of another by using different Statement objects. * The Statement interface lets you execute a simple SQL statement with no parameters. * The SQL instructions are inserted into the Statement object when the Statement.executeXXX method is called. * All executeXXX () methods in the Statement interface implicitly close a statement's current ResultSet object if an open one exists. * **L3.2.1 Statement Methods:**  |  |  |  | | --- | --- | --- | | Return Type | Methods | Description | | boolean | execute(String sql) | Executes the given SQL statement, which may return multiple results. | | ResultSet | executeQuery(String sql) | Executes the given SQL statement, which returns a single ResultSet object. | | int | executeUpdate(String sql) | Executes the given SQL statement, which may be an INSERT, UPDATE, or DELETE statement or an SQL statement that returns nothing, such as an SQL DDL statement. | | Connection | getConnection() | Retrieves the Connection object that produced this Statement object. | | void | cancel() | Cancels this Statement object if both the DBMS and driver support aborting an SQL statement. | | int | getFetchSize() | Retrieves the number of result set rows that is the default fetch size for ResultSet objects generated from this Statement object. | | int | getMaxFieldSize() | Retrieves the maximum number of bytes that can be returned for character and binary column values in a ResultSet object produced by this Statement object. | | int | getMaxRows() | Retrieves the maximum number of rows that a ResultSet object produced by this Statement object can contain. | | ResultSet | getResultSet() | Retrieves the current result as a ResultSet object. | | int | getResultSetConcurrency() | Retrieves the result set concurrency for ResultSet objects generated by this Statement object. | | int | getResultSetHoldability() | Retrieves the result set holdability for ResultSet objects generated by this Statement object. | | int | getResultSetType() | Retrieves the result set type for ResultSet objects generated by this Statement object. | | boolean | isClosed() | Retrieves whether this Statement object has been closed. | | void | setFetchSize(int rows) | Gives the JDBC driver a hint as to the number of rows that should be fetched from the database when more rows are needed for ResultSet objects genrated by this Statement. | | void | setMaxFieldSize(int max) | Sets the limit for the maximum number of bytes that can be returned for character and binary column values in a ResultSet object produced by this Statement object. | | void | setMaxRows(int max) | Sets the limit for the maximum number of rows that any ResultSet object generated by this Statement object can contain to the given number. |  * **L3.2.2 Statement Demo:**   import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.SQLException;  import java.sql.Statement;  public class StatementDemo  {  public static void main(String str[]) throws SQLException  {  try{  Class.forName("oracle.jdbc.driver.OracleDriver");  Connection conn = DriverManager.getConnection("jdbc:oracle:thin:@hstslc007:1521:eltp","scott","tiger");  Statement stmt= conn.createStatement();  stmt.execute("create table employee" +  "(emp\_no number not null," +  "dept\_no number not null," +  "emp\_name varchar2(16) not null," +  "salary number not null)");  System.out.println("table created..");  stmt.close();  conn.close();  } // End of try block  catch(Exception e){  e.printStackTrace();  } // End of catch block  } //End of main()  } //End of class  **L3.3 Batch Updates:**  **(//** [**http://www.jguru.com/faq/view.jsp?EID=5079**](http://www.jguru.com/faq/view.jsp?EID=5079)**)**   * **L3.3.1 Basics** * One of the more advanced features of JDBC 2.0 is the ability to submit multiple update statements to the database for processing as a single unit. * This batch updating can be significantly more efficient compared to JDBC 1.0, where each update statement has to be executed separately. * **L3.3.2 Successful Execution**   Consider the following code segment demonstrating a batch update:  try{  Class.forName("oracle.jdbc.driver.OracleDriver");  Connection conn = DriverManager.getConnection("jdbc:oracle:thin:@hstslc007:1521:eltp","scott","tiger");  Statement stmt= conn.createStatement();    conn.setAutoCommit(false);  stmt.addBatch("insert into employee values(12,23,'johny',12000)");  stmt.addBatch("insert into employee values(13,23,'jesika',15000)");  stmt.addBatch("insert into employee values(14,26,'zafar',16000)");  stmt.addBatch("insert into employee values(15,26,'mohan',19000)");    int[] count = stmt.executeBatch();    conn.commit();  stmt.close();  conn.close();  } catch (BatchUpdateException be) {  // handle Exception  }     |  |  |  | | --- | --- | --- | | **Return Type** | **Method** | **Description** | | void | **addBatch**() | Adds a set of parameters to this PreparedStatement object's batch of commands. | | void | **clearParameters**() | Clears the current parameter values immediately | | boolean | **execute**() | Executes the SQL statement in this preparedStatement object, which may be any kind of SQL statement. | | ResultSet | [**executeQuery**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#executeQuery())() | Executes the SQL query in this PreparedStatement object and returns the ResultSet object generated by the query. | | **int** | [**executeUpdate**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#executeUpdate())() | Executes the SQL statement in this PreparedStatement object, which must be an SQL INSERT, UPDATE or DELETE statement; or an SQL statement that returns nothing, such as a DDL statement. | | [ResultSetMetaData](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/ResultSetMetaData.html) | [**getMetaData**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#getMetaData())() | Retrieves a ResultSetMetaData object that contains information about the columns of the ResultSet object that will be returned when this PreparedStatement object is executed. | | [ParameterMetaData](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/ParameterMetaData.html) | [**getParameterMetaData**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#getParameterMetaData())() | Retrieves the number, types and properties of this PreparedStatement object's parameters. | | void | setArray(int x, Array y) | Sets the designated parameter to the given Array object. | | void | [**setAsciiStream**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setAsciiStream(int, java.io.InputStream, int))(int parameterIndex, [InputStream](http://java.sun.com/j2se/1.5.0/docs/api/java/io/InputStream.html) x, int length) | Sets the designated parameter to the given input stream, which will have the specified number of bytes. | | Void | [**setBigDecimal**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setBigDecimal(int, java.math.BigDecimal))(int parameterIndex, [BigDecimal](http://java.sun.com/j2se/1.5.0/docs/api/java/math/BigDecimal.html) x) | Sets the designated parameter to the given java.math.BigDecimal value. | | void | [**setBinaryStream**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setBinaryStream(int, java.io.InputStream, int))(int parameterIndex, [InputStream](http://java.sun.com/j2se/1.5.0/docs/api/java/io/InputStream.html) x, int length) | Sets the designated parameter to the given input stream, which will have the specified number of bytes. | | void | [**setBlob**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setBlob(int, java.sql.Blob))(int i, [Blob](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/Blob.html) x) | Sets the designated parameter to the given Blob object. | | void | [**setBoolean**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setBoolean(int, boolean))(int parameterIndex, boolean x) | Sets the designated parameter to the given Java boolean value. | | void | [**setByte**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setByte(int, byte))(int parameterIndex, byte x) | Sets the designated parameter to the given Java byte value. | | void | [**setBytes**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setBytes(int, byte[]))(int parameterIndex, byte[] x) | Sets the designated parameter to the given Java array of bytes. | | void | [**setCharacterStream**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setCharacterStream(int, java.io.Reader, int))(int parameterIndex, [Reader](http://java.sun.com/j2se/1.5.0/docs/api/java/io/Reader.html) reader, int length) | Sets the designated parameter to the given Reader object, which is the given number of characters long. | | void | [**setClob**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setClob(int, java.sql.Clob))(int i, [Clob](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/Clob.html) x) | Sets the designated parameter to the given Clob object. | | **void** | [**setDate**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setDate(int, java.sql.Date))(int parameterIndex, [Date](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/Date.html) x) | Sets the designated parameter to the given java.sql.Date value. | | void | [**setDate**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setDate(int, java.sql.Date, java.util.Calendar))(int parameterIndex, [Date](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/Date.html) x, [Calendar](http://java.sun.com/j2se/1.5.0/docs/api/java/util/Calendar.html) cal) | Sets the designated parameter to the given java.sql.Date value, using the given Calendar object. | | void | [**setDouble**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setDouble(int, double))(int parameterIndex, double x) | Sets the designated parameter to the given Java double value. | | void | [**setFloat**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setFloat(int, float))(int parameterIndex, float x) | Sets the designated parameter to the given Java float value. | | void | [**setInt**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setInt(int, int))(int parameterIndex, int x) | Sets the designated parameter to the given Java int value. | | void | [**setLong**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setLong(int, long))(int parameterIndex, long x) | Sets the designated parameter to the given Java long value. | | void | [**setNull**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setNull(int, int))(int parameterIndex, int sqlType) | Sets the designated parameter to SQL NULL. | | void | [**setNull**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setNull(int, int, java.lang.String))(int paramIndex, int sqlType, [String](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html) typeName) | Sets the designated parameter to SQL NULL. | | void | [**setObject**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setObject(int, java.lang.Object))(int parameterIndex, [Object](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Object.html) x) | Sets the value of the designated parameter using the given object. | | void | [**setObject**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setObject(int, java.lang.Object, int))(int parameterIndex, [Object](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Object.html) x, int targetSqlType) | Sets the value of the designated parameter with the given object. | | void | [**setObject**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setObject(int, java.lang.Object, int, int))(int parameterIndex, [Object](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Object.html) x, int targetSqlType, int scale) | Sets the value of the designated parameter with the given object. | | void | [**setRef**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setRef(int, java.sql.Ref))(int i, [Ref](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/Ref.html) x) | Sets the designated parameter to the given REF(<structured-type>) value. | | void | [**setShort**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setShort(int, short))(int parameterIndex, short x) | Sets the designated parameter to the given Java short value. | | void | [**setString**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setString(int, java.lang.String))(int parameterIndex, [String](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html) x) | Sets the designated parameter to the given Java String value. | | void | [**setTime**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setTime(int, java.sql.Time))(int parameterIndex, [Time](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/Time.html) x) | Sets the designated parameter to the given java.sql.Time value. | | void | [**setTime**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setTime(int, java.sql.Time, java.util.Calendar))(int parameterIndex, [Time](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/Time.html) x, [Calendar](http://java.sun.com/j2se/1.5.0/docs/api/java/util/Calendar.html) cal) | Sets the designated parameter to the given java.sql.Time value, using the given Calendar object. | | void | [**setTimestamp**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setTimestamp(int, java.sql.Timestamp))(int parameterIndex, [Timestamp](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/Timestamp.html) x) | Sets the designated parameter to the given java.sql.Timestamp value. | | void | [**setTimestamp**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setTimestamp(int, java.sql.Timestamp, java.util.Calendar))(int parameterIndex, [Timestamp](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/Timestamp.html) x, [Calendar](http://java.sun.com/j2se/1.5.0/docs/api/java/util/Calendar.html) cal) | Sets the designated parameter to the given java.sql.Timestamp value, using the given Calendar object. | | void | [**setURL**](http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html#setURL(int, java.net.URL))(int parameterIndex, [URL](http://java.sun.com/j2se/1.5.0/docs/api/java/net/URL.html) x) | Sets the designated parameter to the given java.net.URL value. |   catch (SQLException e) {  //handle SQL exception  dbCon.rollback();  }   * Before carrying out a batch update, it is important to disable the auto-commit mode by calling setAutoCommit(false). This way, you will be able to rollback the batch transaction in case one of the updates fail for any reason. * When the Statement object is created, it is automatically associated a "command list", which is initially empty. We then add our SQL update statements to this command list, by making successive calls to the addBatch() method. * On calling executeBatch(), the entire command list is sent over to the database, and are then executed in the order they were added to the list. * If all the commands in the list are executed successfully, their corresponding update counts are returned as an array of integers. * **L3.3.2 Handling Failures in Execution** * Please note that you always have to clear the existing batch by calling clearBatch() before creating a new one. * If any of the updates fail to execute within the database, a BatchUpdateException is thrown in response to it. In case there is a problem in returning the update counts of each SQL statement, a SQLException will be thrown to indicate the error.   **L3.4 Prepared Statement Object:**   * The PreparedStatement interface descends from the Statement interface and uses a template to create a SQL request. * Use a PreparedStatement to send precompiled SQL statements with one or more parameters. * **L3.4.1 Methods of PreparedStatement:** * **L3.4.2 Demo of PreparedStatement:**   In the following example of setting a parameter, con represents an active connection:  try{    Class.forName("oracle.jdbc.driver.OracleDriver");  Connection conn =  DriverManager.getConnection("jdbc:oracle:thin:@hstslc007:1521:eltp",  "scott","tiger");    PreparedStatement pstmt = conn.prepareStatement("insert into  userlog values(?,?,?)");    pstmt.setString(1,"Manjiri");  pstmt.setString(2,"man123");  pstmt.setString(3,"M");  int i= pstmt.executeUpdate();    if(i!= 0)  {  System.out.println("row inserted...");  }    pstmt.close();  conn.close();    }   * **L3.4.3 JDBC Connection Pool:**   **//** **http://www.datadirect.com/developer/jdbc/topics/connpooling/index.ssp#anchor1**  **L3.4.3.1 Basics**   * **WHAT??**   + *Connection pooling* means that connections are reused rather than created each time a connection is requested.   + Establishing JDBC connections is resource-expensive, especially when the JDBC API is used in a middle-tier server environment   + To facilitate connection reuse, a memory cache of database connections, called a *connection pool*, is maintained by a connection pooling module as a layer on top of any standard JDBC driver product.      * **Features**   + Connection pooling is performed in the background and does not affect how an application is coded.   + However, the application must use a DataSource object (an object implementing the DataSource interface) to obtain a connection instead of using the DriverManager class.   + A class implementing the DataSource interface may or may not provide connection pooling. A DataSource object registers with a JNDI naming service.   + Once a DataSource object is registered, the application retrieves it from the JNDI naming service in the standard way. * **Comparison of various JDBC versions**   + Since JDBC 1.0, a lot of functionality has been added to the API. JDBC 2.0 introduced scrollable result sets, updatable result sets, batch update capability and support for new SQL3 data types.   + The new JDBC 3.0 specification, in its proposed final draft form at this point, is promising to add more robust transactions by way of savepoints, a number of resource pooling enhancements, retrieval of auto-generated keys and a lot more. * **L3.4.3.2 Connection Pooling Framework**   + The JDBC 3.0 API provides a general framework with "hooks" to support connection pooling rather than specifying a particular connection pooling implementation.   + In this way, third-party vendors or users can implement the specific caching or pooling algorithms that best fit their needs.   + The JDBC 3.0 API specifies a ConnectionEvent class and the following interfaces as the hooks for any connection pooling implementation: * ConnectionPoolDataSource * PooledConnection * ConnectionEventListener * Figure shows this general framework.      * **JDBCDriverVendorDataSource**   + A JDBC driver vendor must provide a class that implements the standard ConnectionPoolDataSource interface.   + This interface provides hooks that third-party vendors can use to implement pooling as a layer on top of their JDBC drivers.   + The ConnectionPoolDataSource interface acts as a "factory" that creates PooledConnection objects. * **JDBCDriverVendorPooledConnection**   + A JDBC driver vendor must provide a class that implements the standard PooledConnection interface.   + This interface allows third-party vendors to implement pooling on top of their JDBC drivers.   + A PooledConnection object acts as a "factory" that creates Connection objects.   + A PooledConnection object maintains the physical connection to the database; the Connection object created by the PooledConnection object is simply a handle to the physical connection. * **PoolingVendorDataSource** * A third-party vendor must provide a class that implements the DataSource interface. * This interface is the entry point that allows interaction with their pooling module. * The pooling vendor's class uses the JDBC driver's PooledConnectionDataSource object to create the PooledConnections that the pool manages. * **PoolingVendorConnectionCache**   + The JDBC 3.0 API does not specify the interfaces to be used between the DataSource object and the connection cache. The pooling vendor determines how these components interact.   + In Figure, the PoolingVendorConnectionCache class is used as a simple way to convey this concept.   + The connection cache module should have a class that implements the standard ConnectionEventListener interface.   + The ConnectionEventListener interface receives ConnectionEvent objects from PooledConnection objects when a connection closes or a connection error occurs.   + When a connection, created by a PooledConnection, closes, the connection cache module returns the PooledConnection object to the cache. * When an application makes a connection by calling DataSource.getConnection() on a PoolingVendorDataSource object, the PoolingVendorDataSource object performs a lookup in the connection cache to determine if a PooledConnection object is available. * If one is available, it is used. If a PooledConnection object is not available, the JDBC driver vendor’s ConnectionPoolDataSource creates a new PooledConnection object. * In either case, a PooledConnection object is made available. * The PoolingVendorDataSource object then invokes the PooledConnection.getConnection() method to obtain a Connection object, which it returns to the application to use as a normal connection. Because the JDBC driver vendor implements the PooledConnection interface, the JDBC driver creates the Connection object; * When the application closes the connection by calling the Connection.close() method, a ConnectionEvent is generated and is passed to the cache module. * The cache module returns the PooledConnection object that created the connection to the cache to be reused. * The application does not have access to the PooledConnection.close() method. Only the connection pooling module, as part of its clean-up activity, issues the PooledConnection.close() method to actually close the physical connection.  L3.4.3.3 Creating a Data Source This section provides examples on how to create pooled and non-pooled DataSource objects and register them to a JNDI naming service.  **//http://www.onjava.com/pub/a/onjava/synd/2001/08/21/jdbc.html?page=3** /\*DataDirect Connect *for* JDBC - Supports Oracle, SQL Server, DB2, Sybase, \*Informix, and MySQL \*DataDirect Connect *for* JDBC is the fastest and most comprehensive suite of Type 4 JDBC drivers \*for all major databases. Connect *for* JDBC is the SPECjAppServer/ECPerf performance and \*scalability leader, and supports advanced functionality such as distributed transactions, \*connection pooling, updating BLOB/CLOB data types, and Operating System (OS) \*authentication for Microsoft SQL Server, Oracle, DB2 and Sybase. Connect *for* JDBC consistently \*supports the latest database features and is fully compliant with Java SE 6.0, Java EE 5, JDBC \*3.0, and select JDBC 4.0 functionality. Using DataDirect Connect *for* JDBC, developers can easily \*build database-independent applications—a common architecture across all major databases \*makes it easy to switch databases or upgrade to new versions of an existing database.  \*/   * **DataDirect Connect *for* JDBC**   This example shows how to create a DataDirect Connect *for* JDBC DataSource object and register it to a JNDI naming service. The DataSource class provided by the DataDirect Connect *for* JDBC drivers is database-dependent. In the following example we use Oracle, so the DataSource class is com.ddtek.jdbcx.oracle.OracleDataSource.  If you want the client application to use:   * A *non-pooled* data source, the application can specify the JNDI name of this data source object as registered in the following code example ("jdbc/ConnectSparkyOracle"). * A *pooled* data source, the application must specify the JNDI name ("jdbc/SparkyOracle") as registered in the code example in the section "Creating a Data Source Using the DataDirect Connection Pool Manager."   //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //  // This code creates a DataDirect Connect *for* JDBC data source and  // registers it to a JNDI naming service. This DataDirect Connect for  // JDBC data source uses the DataSource implementation provided by  // DataDirect Connect *for* JDBC Drivers.  //  // This data source registers its JNDI name as <jdbc/ConnectSparkyOracle>.  // Client applications using non-pooled connections must perform a lookup  // for this name.  //  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // From DataDirect Connect *for* JDBC:  import com.ddtek.jdbcx.oracle.OracleDataSource;  import javax.sql.\*;  import java.sql.\*;  import javax.naming.\*;  import javax.naming.directory.\*;  import java.util.Hashtable;  public class OracleDataSourceRegisterJNDI  {    public static void main(String argv[])  {  try {  // Set up data source reference data for naming context:  // -----------------------------------------------------  // Create a class instance that implements the interface  // ConnectionPoolDataSource  OracleDataSource ds = new OracleDataSource();  ds.setDescription(  "Oracle on Sparky - Oracle Data Source");  ds.setServerName("sparky");  ds.setPortNumber(1521);  ds.setUser("scott");  ds.setPassword("test");  // Set up environment for creating initial context  Hashtable env = new Hashtable();  env.put(Context.INITIAL\_CONTEXT\_FACTORY,  "com.sun.jndi.fscontext.RefFSContextFactory");  env.put(Context.PROVIDER\_URL, "file:c:\\JDBCDataSource");  Context ctx = new InitialContext(env);  // Register the data source to JNDI naming service  ctx.bind("jdbc/ConnectSparkyOracle", ds);  } catch (Exception e) {  System.out.println(e);  return;  }  } // Main  } // class OracleDataSourceRegisterJNDI   * **Creating a Data Source Using the DataDirect Connection Pool Manager**   **DataDirect Connect *for* JDBC**  The following Java code example creates a data source for DataDirect Connect *for* JDBC and registers it to a JNDI naming service. The PooledConnectionDataSource class is provided by the DataDirect com.ddtek.pool package. In the following code example, the PooledConnectionDataSource object references a pooled DataDirect Connect *for* JDBC data source object. Therefore, the example performs a lookup by setting the DataSourceName attribute to the JNDI name of a registered pooled data source (in this example, jdbc/ConnectSparkyOracle, which is the DataDirect Connect *for* JDBC DataSource object created in section "Creating a DataDirect Data Source Object." L3.4.3.4 Connecting to a Data Source  * + Whether connection pooling is used does not affect application code.   + It does not require any code changes to the application because the application performs a lookup on a JNDI name of a previously registered data source.   + If the data source specifies a connection pooling implementation during JNDI registration (as described in section "Creating a Data Source Using the DataDirect Connection Pool Manager"), the client application benefits from faster connections through connection pooling. * **L3.4.3.5 Closing Connection Pool** * To ensure that the connection pool is closed correctly when an application stops running, the application must notify the DataDirect Connection Pool Manager when it stops. * If an application runs on JRE 1.3 or higher, notification occurs automatically when the application stops running. If an application runs on JRE 1.2, the application must explicitly notify the pool manager when it stops using the PooledConnectionDataSource.close method as shown in the following code:   if (ds instanceof com.ddtek.pool.PooledConnectionDataSource){  com.ddtek.pool.PooledConnectionDataSource pcds = (com.ddtek.pool.PooledConnectionDataSource) ds;  pcds.close();  }   * The PooledConnectionDataSource.close method also can be used to explicitly close the connection pool while the application is running.   **L3.5 Difference between Statement and PreparedStatement object**   * A Statement will always proceed through the following four steps for each SQL query sent to the database:   + Parse the incoming SQL query   + Compile the SQL query   + Plan/optimize the data acquisition path   + Execute the optimized query / acquire and return data * A PreparedStatement pre-executes steps (1) - (3) in the execution process above. Thus, when creating a PreparedStatement some pre-optimization is performed immediately. The effect is to lessen the load on the database engine at execution time.   **CODE Sample:**   * Statement example   // Assume a database connection, conn.  Statement stmnt = null;  ResultSet rs = null;  try  {  // Create the Statement  stmnt = conn.createStatement();  // Execute the query to obtain the ResultSet  rs = stmnt.executeQuery("select \* from aTable");  }  catch(Exception ex)  {  System.err.println("Database exception: " + ex);  }   * PreparedStatement example   // Assume a database connection, conn.  PreparedStatement stmnt = null;  ResultSet rs = null;  try  {  // Create the PreparedStatement  stmnt = conn.prepareStatement("select \* from aTable");  // Execute the query to obtain the ResultSet  rs = stmnt.executeQuery();  }  catch(Exception ex)  {  System.err.println("Database exception: " + ex);  }   * Another advantage of the PreparedStatement class is the ability to create an incomplete query and supply parameter values at execution time. * This type of query is well suited for filtering queries which may differ in parameter value only   SELECT firstName FROM employees WHERE salary > 50 SELECT firstName FROM employees WHERE salary > 200  **Code Example**  // Assume a database connection, conn.  PreparedStatement stmnt = null;  ResultSet rs = null;  try  {  // Create the PreparedStatement, leaving a '?' to indicate placement of a parameter.  stmnt = conn.prepareStatement( "SELECT firstName FROM employees WHERE salary > ?");  // Complete the statement  stmnt.setInt(1, 200);  // Execute the query to obtain the ResultSet  rs = stmnt.executeQuery();  }  catch(Exception ex)  {  System.err.println("Database exception: " + ex);  }  **L3.6 CallableStatement**  // http://java.sun.com/j2se/1.5.0/docs/guide/jdbc/getstart/callablestatement.html   * **L3.6.1 Basics** * A CallableStatement object provides a way to call stored procedures in a standard way for all DBMSs. * A stored procedure is stored in a database; the call to the stored procedure is what a CallableStatement object contains. * This call is written in an escape syntax that may take one of two forms: one form with a result parameter, and the other without one. * A result parameter, a kind of OUT parameter, is the return value for the stored procedure. Both forms may have a variable number of parameters used for input (IN parameters), output (OUT parameters), or both (INOUT parameters). A question mark serves as a placeholder for a parameter. * The syntax for invoking a stored procedure using the JDBC API is shown here. Note that the square brackets indicate that what is between them is optional; they are not themselves part of the syntax.   {call procedure\_name[(?, ?, ...)]}  The syntax for a procedure that returns a result parameter is:  {? = call procedure\_name[(?, ?, ...)]}  The syntax for a stored procedure with no parameters would looks this:  {call procedure\_name} L3.6.2 Using CallableStatement methods to call stored procedures//http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.doc/ad/tjvcscsp.htm To call stored procedures, you invoke methods in the CallableStatement class. The basic steps are:   1. Invoke the Connection.prepareCall method to create a CallableStatement object. 2. Invoke the CallableStatement.setXXX methods to pass values to the input (IN) parameters. 3. Invoke the CallableStatement.registerOutParameter method to indicate which parameters are output-only (OUT) parameters, or input and output (INOUT) parameters. 4. Invoke one of the following methods to call the stored procedure:   **CallableStatement.executeUpdate**  Invoke this method if the stored procedure does not return result sets.  **CallableStatement.executeQuery**  Invoke this method if the stored procedure returns one result set.  **CallableStatement.execute**  Invoke this method if the stored procedure returns multiple result sets.   1. If the stored procedure returns result sets, retrieve the result sets. See [Retrieve multiple result sets from a stored procedure in a JDBC application](http://publib.boulder.ibm.com/infocenter/db2luw/v8/topic/com.ibm.db2.udb.doc/ad/tjvjdmlt.htm). 2. Invoke the CallableStatement.getXXX methods to retrieve values from the OUT parameters or INOUT parameters. 3. Invoke the CallableStatement.close method to close the CallableStatement object when you have finished using that object.   The following code illustrates calling a stored procedure that has one input parameter, four output parameters, and no returned ResultSets. The numbers to the right of selected statements correspond to the previously-described steps.   * **L3.6.3 Code Example**   using CallableStatement methods for a stored procedure call with parameter markers    // PROCEDURE:  /\*  \* create or replace procedure getEmpCount( count OUT NUMBER )  \* is  \* begin  \* select count(emp\_name) into count from employee;  \* end;  \*/  try{  Class.forName("oracle.jdbc.driver.OracleDriver");  Connection conn = DriverManager.getConnection("jdbc:oracle:thin:@hstslc007:1521:eltp","scott","tiger");  // Statement stmt= conn.createStatement();  CallableStatement cs = conn.prepareCall("{call getEmpCount(?)}");      int i=0;  cs.setInt(1,i);  cs.registerOutParameter(1,Types.INTEGER);    boolean b = cs.execute();  System.out.println("count is : "+cs.getInt(1));  cs.close();  conn.close();    }  catch(Exception e){  e.printStackTrace();  System.out.println("..."+e);    }    int ifcareas;  int xsbytes;  String errbuff;  Connection con;  CallableStatement cstmt;  ResultSet rs;  ...  cstmt = con.prepareCall("CALL DSN8.DSN8ED2(?,?,?,?,?)");  // Create a CallableStatement object  cstmt.setString (1, "DISPLAY THREAD(\*)");  // Set input parameter (DB2 command)  cstmt.registerOutParameter (2, Types.INTEGER);  // Register output parameters  cstmt.registerOutParameter (3, Types.INTEGER);  cstmt.registerOutParameter (4, Types.INTEGER);  cstmt.registerOutParameter (5, Types.VARCHAR);  cstmt.executeUpdate(); // Call the stored procedure  ifcaret = cstmt.getInt(2); // Get the output parameter values  ifcareas = cstmt.getInt(3);  xsbytes = cstmt.getInt(4);  errbuff = cstmt.getString(5);  cstmt.close();  **L3.7 Database Transaction**  There are times when you do not want one statement to take effect unless another one completes. For example, when the proprietor of The Coffee Break updates the amount of coffee sold each week, he will also want to update the total amount sold to date. However, he will not want to update one without updating the other; otherwise, the data will be inconsistent. The way to be sure that either both actions occur or neither action occurs is to use a transaction. “*A transaction is a set of one or more statements that are executed together as a unit, so either all of the statements are executed, or none of the statements is executed. “*   * **L3.7.1 Commit**    + When a connection is created, it is in auto-commit mode. This means that each individual SQL statement is treated as a transaction and is automatically committed right after it is executed.   + The way to allow two or more statements to be grouped into a transaction is to disable auto-commit mode. This is demonstrated in the following line of code, where con is an active connection:   con.setAutoCommit(false);   * + Once auto-commit mode is disabled, no SQL statements are committed until you call the method commit explicitly. All statements executed after the previous call to the method commit are included in the current transaction and committed together as a unit.   + The following code, in which con is an active connection, illustrates a transaction:   con.setAutoCommit(false);  PreparedStatement updateSales = con.prepareStatement( "UPDATE COFFEES SET SALES = ? WHERE COF\_NAME LIKE ?");  updateSales.setInt(1, 50);  updateSales.setString(2, "Colombian");  updateSales.executeUpdate();  PreparedStatement updateTotal = con.prepareStatement(  "UPDATE COFFEES SET TOTAL = TOTAL + ? WHERE COF\_NAME LIKE ?");  updateTotal.setInt(1, 50);  updateTotal.setString(2, "Colombian");  updateTotal.executeUpdate();  con.commit();  con.setAutoCommit(true);  NOTE: It is advisable to disable auto-commit mode only while you want to be in transaction mode. This way, you avoid holding database locks for multiple statements, which increases the likelihood of conflicts with other users. L3.7.2 Setting and Rolling Back to a Savepoint  * The JDBC 3.0 API adds the method Connection.setSavepoint, which sets a savepoint within the current transaction. The Connection.rollback method has been overloaded to take a savepoint argument. * The example below inserts a row into a table, sets the savepoint svpt1, and then inserts a second row. When the transaction is later rolled back to svpt1, the second insertion is undone, but the first insertion remains intact. In other words, when the transaction is committed, only the row containing ?FIRST? will be added to TAB1:   Statement stmt = conn.createStatement();  int rows = stmt.executeUpdate("INSERT INTO TAB1 (COL1) VALUES " +  "(?FIRST?)");  // set savepoint  Savepoint svpt1 = conn.setSavepoint("SAVEPOINT\_1");  rows = stmt.executeUpdate("INSERT INTO TAB1 (COL1) " +  "VALUES (?SECOND?)");  ...  conn.rollback(svpt1);  ...  conn.commit(); L3.7.3 Releasing a SavepointThe method Connection.releaseSavepoint takes a Savepoint object as a parameter and removes it from the current transaction.Once a savepoint has been released, attempting to reference it in a rollback operation causes an SQLException to be thrown.Any savepoints that have been created in a transaction are automatically released and become invalid when the transaction is committed, or when the entire transaction is rolled back.Rolling a transaction back to a savepoint automatically releases and makes invalid any other savepoints that were created after the savepoint in question. **Summary**   * There are three basic types of SQL statements used in the JDBC API:   + CallabelStatement,   + Statement and   + PreparedStatement * The Statement interface lets you execute a simple SQL statement with no parameters. * One ResultSet object per Statement object can be open at a time. * Batch updateis the ability to submit multiple update statements to the database for processing as a single unit. * The PreparedStatement interface descends from the Statement interface and sends precompiled SQL statements with one or more parameters. * *Connection pooling* means that connections are reused rather than created each time a connection is requested. * Connection pooling is performed in the background and does not affect how an application is coded. * A stored procedure is stored in a database; the call to the stored procedure is what a CallableStatement object contains. * A database transaction is a set of one or more statements that are executed together as a unit, so either all of the statements are executed, or none of the statements is executed. * A database transaction includes Commit, Savepoint and rollback.  JDBC 3.0 The current draft proposes a number of enhancements including transaction savepoints and retrieval of auto-generated keys.  **Transaction Savepoints** To use transactions, a JDBC Connection object must not be in auto-commit mode. The default behavior is for a connection to be in auto-commit mode. To use transactions, a call must be made to setAutoCommit() with an argument of false.  // retrieve a connection to the database Connection conn = DriverManager.getConnection(myDBUrl);  // turn off auto-commit mode conn.setAutoCommit(false);  Once a connection is out of auto-commit mode, transactions must be explicitly committed to the database, or rolled back by making calls to commit() or rollback() on the connection. Calling the commit() method commits the changes to the database and implicitly begins a new transaction. Calling the rollback() method rolls the changes back as if they had never happened, and also implicitly begins a new transaction.  conn.setAutoCommit(false); Statement stmt = conn.createStatement();  // send 4 separate updates to the database stmt.executeUpdate(updateString1); stmt.executeUpdate(updateString2); stmt.executeUpdate(updateString3); stmt.executeUpdate(updateString4);  // commit the updates conn.commit();  Before JDBC 3.0, all four of these updates would have to be committed as a group or rolled back as a group. There would be no way of commiting a subset of the updates once they were added to the transaction. With JDBC 3.0, savepoints will offer finer-grained control of this behavior. During a transaction a named savepoint may be inserted between operations. This named savepoint acts as a marker in the transaction and the transaction may be rolled back to that marker, effectively removing all of the operations after the marker but leaving all of the operations before the marker in place.  conn.setAutoCommit(false);  Statement stmt = conn.createStatement();  // send 2 updates to the database  stmt.executeUpdate(updateString1);  stmt.executeUpdate(updateString2);  // create a savepoint  Savepoint sp1 = conn.setSavepoint("SP1");  // send 2 more updates to the database  stmt.executeUpdate(updateString3);  stmt.executeUpdate(updateString4);  // rollback to sp1, effectively removing the  // last 2 updates from the transaction  conn.rollback(sp1);  // commit the updates  conn.commit(); Auto-Generated Keys Some databases allow for certain columns to be given automatically generated key values. In this case, an insert statement would not be responsible for supplying a value for the column. The database would generate a unique value for the column and insert the value. This is often used for generating unique primary keys. A problem with this approach is that it may be difficult to get the value after the insert is executed. The JDBC 3.0 specification proposes a more functional Statement interface that provides access to these values after an insert.  Assume a table called USERS with 3 columns. The FIRST\_NAME column and LAST\_NAME column are varchars. The USER\_ID column is an auto generated column and should contain a unique identifier for each user in the table.  Statement stmt = conn.createStatement();  // insert a new user into the database // notice that the USER\_ID is not accounted for here stmt.executeUpdate("insert into users (first\_name, last\_name) values('Jeff', 'Brown')");  // Retrieve a result set containing all of the auto-generated keys from // the last update issued on this statement // the specific details of the format of this ResultSet are not clearly specified yet ResultSet rs = stmt.getGeneratedKeys(); Conclusion The JDBC API has matured a lot over the last few years. The API continues to get more flexible and powerful while remaining very simple and straightforward. The Java 2 Standard Edition (J2SE) version 1.4 will contain version 3.0 of JDBC. It may take some time before most driver vendors support all of the new functionality.  A related technology is Java Data Objects (JDO). JDO promises to hide more of the database specific code from the developer. Some benefits of JDO will be the addition of more compile time checking and more manipulation of data through the API instead of through SQL statements. JDO is intended to complement JDBC.  For the latest information on JDBC and JDO, visit <http://java.sun.com/products/jdbc/>.   * Scrolling through Resultset 2006-05-17 10:24:21  pandu345 [[Reply](http://www.oreillynet.com/cs/user/create/cs_msg?x-lr=cs_msg/82675&x-lr2=a/1058) | [View](http://www.oreillynet.com/cs/user/view/cs_msg/82675)]   Hi,  Here is my requrement,I have a table(username,requestdate,createdate).I want to make a report where I give user his department and some other details from different places and reauest dates and create dates.Now I am running a query for each user once to get his dates,I wanted to use a scrollable resultset,where I query database only once and use this result set each time instead of running a query every time.But my users are not ordered,initially I want dates of user 'arun' , then I need dates of user 'manick' etc.For this purpose how can I scroll the result.  ------------------------------------------------------------------  **Exercise**  Objectives:   1. Please go through code listings. Identify the probable errors, either compile time or run time. 2. Document the Error, Explanation of the error and the probable solution/fix for the errors identified   Pre-requisites:   1. You must have a good conceptual understanding of Session 3 – “JDBC- Prepared and Callable statements” of the Intermediate Java course  Code Listing 1 **public static void main(String args[]){**  **int empID=156;**  **String password="pwd";**  **int bankAcc=879;**  **String dept="warehouse";**  **String ques="Your mother's maiden name";**  **String ans="Rose";**  **String status="update";**  **int days=10;**  **int rows=0;**  **boolean result=false;**  **try{**  **Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");**  **Connection conn = DriverManager.getConnection("jdbc:odbc:mydsn","scott", "tiger");**  **try{**  **PreparedStatement psmt = conn.prepareStatement("INSERT INTO project7181 VALUES( ?,?,?,?,?,?,?)");**  **psmt.setInt(1,empID);**  **psmt.setString(2,password);**  **//psmt.setString(3,status);**  **psmt.setInt(3,bankAcc);**  **psmt.setString(4,dept);**  **psmt.setString(5,ques);**  **psmt.setString(6,ans);**  **psmt.setString(7,status);**  **rows=psmt.executeUpdate();**  **if (rows>0){**  **result = true;**  **System.out.println("The number of rows inserted are " +rows);**  **}**  **else {**  **System.out.println("No rows inserted ");**  **}**  **psmt.close();**  **}**  **catch(Exception e){**  **System.out.println(e);**  **}**  **conn.close();**  **}**  **catch(Exception e){**  **e.printStackTrace();**  **}**  **}**  **}** Code Listing 2 **public static void main(String args[]){**  **int empID=156;**  **String password="pwd";**  **int bankAcc=879;**  **String dept="warehouse";**  **String ques="Your mother's maiden name";**  **String ans="Rose";**  **String status="update";**  **int days=10;**  **int rows=0;**  **boolean result=false;**  **try{**  **Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");**  **Connection conn = DriverManager.getConnection("jdbc:odbc:mydsn","scott", "tiger");**  **try{**  **PreparedStatement psmt = conn.prepareStatement("INSERT INTO project7181 VALUES( ?,?,?,?,?,?)");**  **psmt.setInt(1,empID);**  **psmt.setString(2,password);**  **psmt.setString(3,status);**  **psmt.setString(4,dept);**  **psmt.setString(5,ques);**  **psmt.setString(6,ans);**  **rows=psmt.executeUpdate();**  **if (rows>0){**  **result = true;**  **System.out.println("The number of rows inserted are " +rows);**  **}**  **else {**  **System.out.println("No rows inserted ");**  **}**  **psmt.close();**  **}**  **catch(Exception e){**  **System.out.println(e);**  **}**  **conn.close();**  **}**    **catch(Exception e){**  **e.printStackTrace();**  **}**  **}**  **}** Code Listing 3 **public static void main(String args[]) {**  **Connection con = null;**    **try {**  **// Driver name type 1 <Bridge driver>**  **Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");**    **} catch(java.lang.ClassNotFoundException e) {**  **System.out.print("ClassNotFoundException: ");**    **}**  **try {**    **con = DriverManager.getConnection("jdbc:odbc:jdbcdsn","scott","tiger");**  **if(!con.isClosed()) {**  **System.out.println("Connection Created Successful!!!!");**  **}**    **con.close();**    **Statement smt = con.createStatement();**  **ResultSet rs = smt.executeQuery("select \* from project7181;");**  **while(rs.next())**  **{**  **System.out.println(rs.getString(1));**  **}**  **} catch (SQLException e) {**  **e.printStackTrace();**  **}**    **}** Code Listing 4 **public static void main(String args[]) {**  **Connection con = null;**  **try {**  **// Driver name type 1 <Bridge driver>**  **Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");**  **} catch(java.lang.ClassNotFoundException e) {**  **System.out.print("ClassNotFoundException: ");**  **}**  **try {**  **con = DriverManager.getConnection("jdbc:odbc:jdbcdsn","scott","tiger");**  **if(!con.isClosed()) {**  **System.out.println("Connection Created Successful!!!!");**  **}**  **con.close();**    **Statement smt = con.createStatement();**  **ResultSet rs = smt.executeQuery("select \* from project7181;");**  **while(rs.next())**  **{**  **System.out.println(rs.getString(7));**  **}**  **} catch (SQLException e) {**  **e.printStackTrace();**  **}**  **}**   Code Listing 5 **public static void main(String args[]) {**  **Connection connection = null;**  **Statement statement = null;**  **/\* Connection creation should always be inside a try-catch block**  **But the question arises, WHY ?? \*/**  **try {**  **/\* Driver name type 1 <Bridge driver>**  **Why is this Driver called Bridge Driver \*/**  **Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");**    **} catch(java.lang.ClassNotFoundException e) {**  **System.out.print("ClassNotFoundException: " + e);**  **}**  **try {**  **connection = DriverManager.getConnection("jdbc:odbc:jdbcdsn","scott","tiger");**  **/\* if(!connection.isClosed()) {**  **System.out.println("Connection Created Successful!!!!");**  **}**  **\*/**  **statement = connection.createStatement();**  **/\* Auto Commit is set FALSE/OFF**  **why is the Autocommit set OFF \*/**  **connection.setAutoCommit(false);**  **// Batch Insertion of values**  **statement.addBatch("insert into project7181 (emp\_id , password) values (101,'xyz')");**  **statement.addBatch("insert into project7181 (emp\_id , password) values (101,'xyz')");**  **int [] count = statement.executeBatch();**  **// Printing the values after Insertion**  **ResultSet resultset = statement.executeQuery("select \* from project7181;");**  **while(resultset.next())**  **{**  **System.out.println(resultset.getString(1));**  **System.out.println(resultset.getString(2));**  **System.out.println(resultset.getString(3));**  **}**  **connection.commit();**  **connection.close();**    **} catch (SQLException e) {**  **e.printStackTrace();**  **}** Code Listing 6 **public static void main(String args[]) {**  **Connection connection = null;**  **Statement statement = null;**  **try {**  **// Driver name type 1 <Bridge driver>**  **Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");**  **} catch(java.lang.ClassNotFoundException e) {**  **System.out.print("ClassNotFoundException: ");**  **}**  **try {**  **connection = DriverManager.getConnection("jdbc:odbc:jdbcdsn","scott","tiger");**  **if(!connection.isClosed()) {**  **System.out.println("Connection Created Successful!!!!");**  **}**  **statement = connection.createStatement();**  **//connection.setAutoCommit(false);**  **String procedure;**  **CallableStatement callablestatement;**  **String outParam;**  **// procedure with IN parameter**  **procedure = "CREATE OR REPLACE PROCEDURE myprocin(X IN VARCHAR) IS " +**  **"BEGIN "+**  **"INSERT INTO oracle\_table VALUES('string 2');"+**  **"x := 'invalue'; " // Assign value to x**  **+ "END;";**  **CallableStatement callablestatement = connection.prepareCall("{call myprocin(?)}");**  **callablestatement.setString(1,"string 1");**  **callablestatement.executeUpdate();**  **//procedure with OUT parameter**  **procedure = "CREATE OR REPLACE PROCEDURE myprocout(X OUT VARCHAR) IS " +**  **"BEGIN "+**  **"INSERT INTO oracle\_table VALUES('string 2');"+**  **"x := 'outvalue'; " // Assign value to x**  **+ "END;";**  **statement.executeUpdate(procedure);**  **callablestatement = connection.prepareCall("{call myprocout(?)}");**  **callablestatement.registerOutParameter(1,Types.VARCHAR);**  **callablestatement.execute();**  **String outParam = callablestatement.getString(1);**  **System.out.println("OUT parameter : "+outParam);**  **//procedure with IN/OUT parameter**  **procedure = "CREATE OR REPLACE PROCEDURE myprocinout(X IN OUT VARCHAR) IS " +**  **"BEGIN "+**  **"INSERT INTO oracle\_table VALUES(x);"+**  **"x := 'outvalue'; " // Assign value to x**  **+ "END;";**  **statement.executeUpdate(procedure);**  **callablestatement = connection.prepareCall("{call myprocinout(?)}");**  **callablestatement.registerOutParameter(1,Types.VARCHAR);**  **callablestatement.setString(1,"string 3");**  **callablestatement.execute();**  **outParam = callablestatement.getString(1);**  **System.out.println("OUT parameter : "+outParam);**  **//connection.commit();**  **connection.close();**  **} catch (SQLException e) {**  **e.printStackTrace();**  **}**  **}** 4.JDBC-Advanced concepts  * **Course Objectives :**   The objective of this chapter is to provide Java snippets, reusable code samples, classes, and methods that deal with the “scrollable” and “updatable” ResultSet objects. When writing this chapter, we relied on JDK 1.4 and the final release of the JDBC 3.0 specification.     * **Prerequisites :**   Experience in Java Programming is essential. The student must be comfortable with object-oriented Java, interfaces and abstract classes. Understanding of SQL usage and basics of ResulSet object with relational databases will be helpful.     * **Do you remember what is resulset and the cursor :** * A complete set of rows returned by a SQL statement is known as a *result set.*      * The cursor is a *pointer*, which points to one of the records in the result set. * **What Is a Scrollable ResultSet ?**      * A ResultSet is scrollable if you have the ability to move its cursor backward as well as forward. * TheJDBC 2.0 API introduced the scrollable ResultSet concept. The ResultSet object has methods that let you move the cursor to a particular row and check the position of the cursor. * Scrollable ResultSet objects make it possible to create a GUI tool for browsing ResultSet objects; this is probably one of the main uses of this feature. * Another use is moving to a particular row in order to update it.   Before you can use a scrollable ResultSet, you have to create one. The  following snippet shows one way to create a scrollable ResultSet object:  // assume getConnection() returns a Connection object  Connection conn = getConnection();  ...  Statement stmt = conn.createStatement(  ResultSet.TYPE\_SCROLL\_SENSITIVE,  ResultSet.CONCUR\_READ\_ONLY);  String query = "SELECT id, name FROM employees";  ResultSet rs = stmt.executeQuery(query);      When creating a Statement object, you need to specify two arguments to the method createStatement(). The first argument indicates the type of a ResultSet object and can be one of three constants:   * ResultSet.TYPE\_FORWARD\_ONLY: A constant indicating the type for a ResultSet object whose cursor may move only forward (creates a nonscrollable ResultSet object). * ResultSet.TYPE\_SCROLL\_INSENSITIVE: A constant indicating the type for a ResultSet object that is scrollable but generally not sensitive to changes made by others. * ResultSet.TYPE\_SCROLL\_SENSITIVE: A constant indicating the type for a ResultSet object that is scrollable and generally sensitive to changes made by others.   The second argument is one of two ResultSet constants for specifying  whether a result set is read-only or writable/updatable.   * **Dealing with Scrollable ResultSet :**   A scrollable ResultSet object allows the cursor to be moved to any row in the ResultSet. If a JDBC driver supports ResultSet scrollability, then you can use the JDBC API to create a scrollable ResultSet object.    **Creating an Insensitive Scrollable ResultSet :**  The following shows how to create an insensitive scrollable ResultSet:    ResultSet rs = null;  Statement stmt = null;  Connection conn = null;  try {  // get a Connection object  conn = getConnection();    // Create an insensitive scrollable result set  stmt = connection.createStatement(  ResultSet.TYPE\_SCROLL\_INSENSITIVE,  ResultSet.CONCUR\_READ\_ONLY);  // Create the desired scrollable ResultSet object  String query = "select id, name from employees";  rs = stmt.executeQuery(query);  }  catch (SQLException e) {  // handle the exception  }  finally {  // close resources: ResultSet, Statement, and Connection objects  }      **Creating a Sensitive Scrollable ResultSet :**  The following shows how to create a sensitive scrollable ResultSet:  ResultSet rs = null;  Statement stmt = null;  Connection conn = null;  try {  conn = getConnection(); // get a Connection object  // Create a sensitive scrollable result set  stmt = connection.createStatement(  ResultSet.TYPE\_SCROLL\_SENSITIVE,  ResultSet.CONCUR\_READ\_ONLY);  // Create the desired scrollable ResultSet object  String query = "select id, name from employees";  rs = stmt.executeQuery(query);  }  catch (SQLException e) {  // handle the exception  }  finally {  // close resources: ResultSet, Statement, and Connection objects  }  **How Do we Determine If a ResultSet Is Scrollable ?**    Given a ResultSet object, how do you determine whether that ResultSet is scrollable? By using the ResultSet object’s getType() method, you can answer this question:  public static boolean isScrollable(ResultSet rs) {  if (rs == null) {  return false;  }  try {  // get type of the result set  int type = rs.getType();    if ((type == ResultSet.TYPE\_SCROLL\_INSENSITIVE) ||  (type == ResultSet.TYPE\_SCROLL\_SENSITIVE)) {  // Result set is scrollable  return true;  }  else {  // Result set is not scrollable  return false;  }  }  catch (SQLException e) {  return false;  }    **How Do we Move the Cursor in a Scrollable ResultSet?**    A scrollable ResultSet object has a set of specific methods for moving cursors.  ResultSet Object’s ScrollingMethods     |  |  | | --- | --- | | **Method** | **Semantics** | | first() | Moves to the first record | | last() | Moves to the last record | | next() | Moves to the next record | | previous() | Moves to the previous record | | beforeFirst() | Moves to immediately before the first record | | afterLast() | Moves to immediately after the last record | | absolute(int) | Moves to an absolute row number, and takes a positive or negative argument | | relative(int) | Moves backward or forward a specified number of rows, and takes a positive or negative argument |     The following example demonstrates various methods for moving the cursor in a scrollable  ResultSet object:    ResultSet rs = null;  Statement stmt = null;  Connection conn = null;  try {  // get a Connection object  conn = getConnection();    // Create a scrollable result set  stmt = connection.createStatement(  ResultSet.TYPE\_SCROLL\_INSENSITIVE,  ResultSet.CONCUR\_READ\_ONLY);  // create your desired SQL query  String query = ""SELECT id, name FROM employees";    // create scrollable ResultSet object  rs = stmt.executeQuery(query);  // Move cursor forward  while (rs.next()) {  // Get data at cursor  String id = rs.getString(1);  String name = rs.getString(2);  }    // Move cursor backward  while (rs.previous()) {  // Get data at cursor  String id = rs.getString(1);  String name = rs.getString(2);  }  // Move cursor to the first row  rs.first();    // Move cursor to the last row  rs.last();    // Move cursor to the end, after the last row  rs.afterLast();  // Move cursor to the beginning, before the first row.  // cursor position is 0.  rs.beforeFirst();  // Move cursor to the second row  rs.absolute(2);  // Move cursor to the last row  rs.absolute(-1);  // Move cursor to the second-to-last row  rs.absolute(-2);  // Move cursor down 5 rows from the current row. If this moves  // cursor beyond the last row, cursor is put after the last row  rs.relative(5);  // Move cursor up 3 rows from the current row. If this moves  // cursor beyond the first row, cursor is put before the first row  rs.relative(-3);  }  catch (SQLException e) {  // handle the exception  }  **Exercise** : Can you make a program to count total number of rows wth the help of  Scrollable ResultSet.     * **Dealing with Updatable ResultSets :** * An updatable ResultSet object allows you to modify data in a table by using the ResultSet methods rather than by sending SQL queries to the database server. * To get updatable results, the Statement object used to create the result sets must have the concurrency type ResultSet.CONCUR\_UPDATABLE. * The query of an updatable result set must specify the primary key as one of the selected columns and select from only one table. * For some JDBC drivers, the SQL query SELECT \* FROM my\_table will return a read-only result set, so make sure you specify the column names.   **Creating an Updatable ResultSet**  **…………**  Connection conn = null;  Statement stmt = null;  ResultSet rs = null;  try {  conn = getConnection();  // create a statement that will return  // updatable result sets  stmt = conn.createStatement(  ResultSet.TYPE\_SCROLL\_SENSITIVE,  ResultSet.CONCUR\_UPDATABLE);  // Primary key pk\_column must be specified  // so that the result set is updatable  rs = stmt.executeQuery("SELECT pk\_column FROM my\_table");  }  catch (SQLException e) {  // handle exception  }    **How to Update a Row in a Database Table Using an Updatable Result Set ?**    Updating the current row of an updatable result set involves  calling the ResultSet.updateXXX() methods followed by a call to updateRow():  Connection conn = null;  Statement stmt = null;  ResultSet rs = null;  try {  // get a Connection object  conn = getConnection();  // Create an updatable result set  stmt = conn.createStatement(  ResultSet.TYPE\_SCROLL\_SENSITIVE,  ResultSet.CONCUR\_UPDATABLE);  rs = stmt.executeQuery("SELECT \* FROM my\_table");  // Move cursor to the row to update  rs.first();  // Update the value of column column\_1 on that row  rs.updateString("column\_1", "new data");  // Update the row; if autocommit is enabled,  // update is committed  rs.updateRow();  }  catch (SQLException e) {  // handle exception  }  **How Do we Insert a Row into a Database Table Using an Updatable ResultSet?**   * An updatable ResultSet object supports a specific row called the *insert row*. * It is a buffer for holding the values of a new row. * After you have filled the fields in the insert row, you can insert the new row   into the database using the Result.insertRow() method.    Connection conn = null;  Statement stmt = null;  ResultSet rs = null;  try {  // get a Connection object  conn = getConnection();    // Create an updatable result set  stmt = conn.createStatement(  ResultSet.TYPE\_SCROLL\_SENSITIVE,  ResultSet.CONCUR\_UPDATABLE);    String query = "select id, name from employees";  // Create the desired ResultSet object  rs = stmt.executeQuery(query);  // Move cursor to the "insert row"  rs.moveToInsertRow();  // Set values for the new row.  rs.updateString("id", "66");  rs.updateString("name", "Harrison Ford");  // Insert the row  rs.insertRow();  }  catch (SQLException e) {  // handle exception  }   * **Exercise :** * Can you make a program to delete the row from the database using Updatable ResulSet ?   Hint : To delete your desired row from a ResultSet object, create a  scrollable and updatable ResultSet object, then point to your  desired row, and finally  invoke the deleteRow() method.   * Can you write a program to update 3rd row of a Database   Hint : if (rs.absolute(3))  {  rs.updateString(1, " xyz");  }   * **Row Set Objects :** * Some DBMSs do not support result sets that are scrollable, and some do not support result sets that are updatable. If a driver for that DBMS does not add scrollability or updatability, you can use a RowSet object to do it. * All RowSet objects are derived from the ResultSet interface and therefore share its capabilities.   **Kinds of RowSet Objects:**  A RowSet object is considered either connected or disconnected.   * **A connected RowSet object** uses a driver based on JDBC technology (“JDBC driver”) to make a connection to a relational database and maintains that connection throughout its life span. * **A *disconnected* RowSet object** makes a connection to a data source   only to read in data from a ResultSet object or to write data back to the data  source. After reading data from or writing data to its data source, the RowSet  object disconnects from it, thus becoming “disconnected.”  During much of its life span, a disconnected RowSet object has no connection to its data source and operates independently. The next two sections tell you what being connected or disconnected means in terms of what a RowSet object can do.   * **Summary :**   + A default ResultSet object is not updatable and has a cursor that moves forward only.   **Exercise**  **Directions:-**   1. **Please go through code listings. Identify the probable errors, either compile time or run time.** 2. **Document the Error, Explanation of the error and the probable solution/fix for the errors identified**   **Pre-requisites:**   1. **You must have a good conceptual understanding of Session 5– “Servlet Basics” of the Intermediate Java course**  Code Listing 1 **protected void service(HttpServletRequest request, HttpServletResponse response)**  **throws ServletException, IOException {**  **response.setContentType("text/html;charset=UTF-8");**  **PrintWriter out = response.getWriter();**      **out.println("Hello Tom Hanks! Cruise into Servlets Intro ");**  **out.close();**  **}**  **{**  **out.println(e);**  **System.out.println(e);**  **}**  **out.close();**  **}**  **}** Code Listing 2 **public class Login extends HttpServlet {**  **public void service(HttpServletRequest request, HttpServletResponse response)**  **throws ServletException{**  **response.setContentType("text/html;charset=UTF-8");**  **PrintWriter out = response.getWriter();**    **try**  **{**  **Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");**  **Connection con=DriverManager.getConnection("jdbc:odbc:mydsn","scott", "tiger");**  **PreparedStatement pst = con.prepareStatement("select emp\_id,password,bank\_acc from project7181 where emp\_id=101 ");**  **ResultSet rs = pst.executeQuery();**  **while(rs.next())**  **{**  **out.println("<html><body bgcolor='wheat'><center><h3>The details of the employee are<br><br>");**  **out.println("The employee id is " +rs.getInt(1));**  **out.println("<br><br>");**  **out.println("The password is " +rs.getString(2));**  **out.println("<br><br>");**  **out.println("The Bank account is "+rs.getInt(3));**  **out.println("<h3><center><body><html>");**    **}**    **}**    **catch(Exception e)** Code Listing 3 **--------------------------------->//START OF HTML CODE//**  **<html>**  **<head>**  **<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">**  **<title>Login Page</title>**  **</head>**  **<body bgcolor="#dddeedd">**  **<center>**  **<h1>Login Page</h1>**  **Enter User Information**  **<form method="get" action="http://localhost:8084/Servlet/Login" >**  **UserName :<input type="text" name="username" maxlength="10" size="10" align="center"><br>**  **Password :<input type="password" name="password" maxlength="10" size="10" align="center"><br>**  **<input type="submit" value="Login">**  **</form>**    **</center>**  **</body>**  **</html>**  **--------------------------------->//END OF HTML CODE//**  **--------------------------------->//START OF Servlet CODE//**  **protected void doPost(HttpServletRequest request, HttpServletResponse response)**  **throws ServletException, IOException {**  **response.setContentType("text/html;charset=UTF-8");**  **PrintWriter out = response.getWriter();**  **int employeeid=101;**  **String password="101";**  **Connection connection = null;**  **Statement statement = null;**    **try {**  **// Driver name type 1 <Bridge driver>**  **Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");**    **} catch(java.lang.ClassNotFoundException e) {**  **System.out.print("ClassNotFoundException: ");**    **}**  **try {**  **connection = DriverManager.getConnection("jdbc:odbc:jdbcdsn","scott","tiger");**  **if(!connection.isClosed()) {**  **System.out.println("Connection Created Successful!!!!");**  **}**    **if( (employeeid==101) && (password=="101") ) {**  **out.println("<html><body> <h3>Login Successful</h3> </form> </body></html> ") ;**  **} else {**  **out.println("<html><body> <h3>Login Failure</h3> </form> </body></html> ") ;**  **}**      **}catch(SQLException e) {**  **e.printStackTrace();**  **}**  **out.close();**  **}**  **--------------------------------->//END OF Servlet CODE//** Code Listing 4 **--------------------------------->//START OF HTML CODE//**  **<html>**  **<head>**  **<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">**  **<title>Login Page</title>**  **</head>**  **<body bgcolor="#dddeedd">**  **<center>**  **<h1>Login</h1>**  **Enter DB User**  **<form method="post" action="http://localhost:8084/Servlet/Login" >**  **UserName :<input type="text" name="employeeid" maxlength="10" size="10" align="center"><br>**  **Password :<input type="password" name="password" maxlength="10" size="10" align="center"><br>**  **<input type="submit" value="Login">**  **</form>**    **</center>**  **</body>**  **</html>**  **</html>**  **--------------------------------->//END OF HTML CODE//**  **----------------------------------------------------->//START OF Servlet CODE//**  **protected void service(HttpServletRequest request, HttpServletResponse response)**  **throws ServletException, IOException {**  **response.setContentType("text/html;charset=UTF-8");**  **PrintWriter out = response.getWriter();**  **int employeeid= request.getParameter("employeeid");**  **String password = request.getParameter("password");**  **Connection connection = null;**  **Statement statement = null;**  **try {**  **// Driver name type 1 <Bridge driver>**  **Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");**  **} catch(java.lang.ClassNotFoundException e) {**  **System.out.print("ClassNotFoundException: ");**  **}**  **try {**  **connection = DriverManager.getConnection("jdbc:odbc:jdbcdsn","scott","tiger");**  **if(!connection.isClosed()) {**  **System.out.println("Connection Created Successful!!!!");**  **}**  **if( (employeeid==101) && (password=="101") ) {**  **out.println("<html><body> <h3>Login Successful</h3> </form> </body></html> ") ;**  **} else {**  **out.println("<html><body> <h3>Login Failure</h3> </form> </body></html> ") ;**  **}**  **}catch(SQLException e) {**  **e.printStackTrace();**  **}**  **---------------------------------------------------------------------->//END OF Servlet CODE//** 5. Servlets Basics **OBJECTIVE:**   * Understand basic structure of Servlets * Understand Servlet Life Cycle * Handling client requests with Form Data   **L5.1 Introduction to Servlets**   * Servlets Basics * Servlets are Java technology’s answer to Common Gateway Interface(CGI) programming. * They are programs that run on a Web server, acting as a middle layer between a requests coming from a Web browser or other HTTP client and databases or applications on the HTTP server. Their job is to:   + Read any data sent by the user   + Look up any other information about the request that is embedded in the HTTP request.   + Generate the results.   + Format the results inside a document.   + Format the results inside a document.   + Set the appropriate HTTP response parameters.   + Send the document back to the client. * **The Advantages of Servlets Over “Traditional” CGI** * **Efficient**   + With CGI, a new process is started for each HTTP request.   + With servlets, the Java Virtual Machine stays running and handles each request using a lightweight Java thread, not a heavyweight operating system process.   + In CGI, if there are *N* simultaneous requests to the same CGI program, the code for the CGI program is loaded into memory *N* times.   + With servlets, however, there would be *N* threads but only a single copy of the servlet class.   + Finally, when a CGI program finishes handling a request, the program terminates. This makes it difficult to cache computations, keep database connections open, and perform other optimizations that rely on persistent data.   + Servlets, however, remain in memory even after they complete a response, so it is straightforward to store arbitrarily complex data between requests. * **Convenient**   + Servlets have an extensive infrastructure for automatically parsing and decoding HTML form data, reading and setting HTTP headers, handling cookies, tracking sessions, and many other such high-level utilities. * **Powerful**   + Servlets support several capabilities that are difficult or impossible to accomplish with regular CGI.   + Servlets can talk directly to the Web server, whereas regular CGI programs cannot, at least not without using a server-specific API.   + Multiple servlets can also share data, making it easy to implement database connection pooling and similar resource-sharing optimizations.   + Servlets can also maintain information from request to request, simplifying techniques like session tracking and caching of previous computations. * **Portable**   + Servlets are written in the Java language and are thus portable * **Secure** * **Inexpensive**   + There are a number of free or very inexpensive Web servers available that are good for “personal” use or low-volume Web sites.   + This is in contrast to many of the other CGI alternatives, which require a significant initial investment to purchase a proprietary package. * **L5.1.1 Basic Structure of Servlet** * Servlets are basic java applications, only the difference is that they extend new classes and implement some unfamiliar methods. * All Servlets have two common things, * First, they extend one of two Servlet classes – GenericServlet or HttpServlet. Extending these classes provides a framework for creating a Servlet as well as significant default functionality. * Second, all Servlets override at least one method wherein custom functionality is implemented. * The method that is automatically called by the server in response to a client request is service(). * There are two other methods that are implemented by most Servlets – init() and destroy(). * The init() method is called a single time when the Servlet is first loaded. It is similar to a class constructor in that it provides a method where initialization code is guaranteed to be run. * The destroy() method is executed when the Servlet is unloaded. It is used to free any resources held by the Servlet. * From these details we can construct the skeleton of Servlet :   public class SkeletonServlet extends HttpServlet  {  public void init()  {  // initialization code goes here  }  public void service()  {  // meaningful work happens here  }  public void destroy()  {  // free resources here  }  }   * The actual implementation of different Servlets varies widely. Servlets may or may not implement the init() or destroy() methods. * **L5.1.2 Servlet that generates plain text**   import java.io.IOException;  import java.io.PrintWriter;  import javax.servlet.ServletException;  import javax.servlet.http.\*;  public class SampleServlet extends HttpServlet {    public void service(HttpServletRequest request,HttpServletResponse response) throws ServletException, IOException  {  //Set MIME type for http header  response.setContentType("text/plain");    //Get a handle to output stream  PrintWriter out = response.getWriter();    //Displaying plain text  out.println("First Sample Servlet...");  }    }   * **L5.1.3 Servlet that generates HTML**   import java.io.IOException;  import java.io.PrintWriter;  import javax.servlet.ServletException;  import javax.servlet.http.\*;  public class SampleServlet extends HttpServlet {    public void service(HttpServletRequest request,HttpServletResponse response) throws ServletException, IOException  {  //Set MIME type for http header  response.setContentType("text/html");    //Get a handle to output stream  PrintWriter out = response.getWriter();    //Displaying plain text  out.println("First Sample Servlet...");    out.print("<html>");  out.print("<body>");  out.print("<h1>");  out.print("Sample HTML Servlet");  out.print("</html>");  out.print("</body>");  out.print("</h1>");  }    }   * **L5.1.4 Servlets API** * A Servlet in its most general form, is an instance of a class, which implements the javax.servlet.Servlet interface. * Most servlets, however, extend one of the standard implementations of that interface, namely **javax.servlet.**GenericServlet and **javax.servlet.http.**HttpServlet. * Following are the most commonly used classes and interfaces that manages the Servlet and its communications with its client   + GenericServlet class   + HttpServlet class   + ServletRequest interface   + HttpServletRequest interface   + ServletResponse interface   + Http ServletResponse interface   These classes and interfaces are essential components of the **Servlet API.**  **Servlet**  **Packages**  **javax.Servlet javax.Servlet.http**  `  Servlet HttpServletRequest  ServletRequest HttpServletResponse  ServletResponse HttpSession  ServletConfig HttpSessionBindingListener  ServletContext HttpSessionContext  SingleThreadModel  `  **Interfaces**  GenericServlet Cookie  ServletInputStream HttpServlet  ServletOutputStream HttpSessionBindingEvent  HttpUtils  **Classes**  **Diagram: Servlet Interfaces and classes**   * **Class GenericServlet**   java.lang.Object  |  +--**javax.servlet.GenericServlet**  **Direct Known Subclasses:**  HttpServlet  public abstract class **GenericServlet**  extends java.lang.Object  implements Servlet, ServletConfig, java.io.Serializable   * GenericServlet provides simple versions of the life-cycle methods init and destroy, and of the methods in the ServletConfig interface.  |  |  |  | | --- | --- | --- | | **Return Type** | **Method** | **Description** | | void | destroy() | Destroys the servlet, cleaning up whatever resources are being held, and logs the  destruction in the servlet log file. | | java.lang.String | getInitParameter(java.lang.String name) | Returns a string containing the value of the named initialization parameter, or null if the  requested parameter does not exist. | | java.util.Enumeration | getInitParameterNames() | Returns the names of the initialization parameters for this servlet as an enumeration of  Strings, or an empty enumeration if there are no initialization parameters. | | ServletConfig | getServletConfig() | Returns a servletConfig object containing any startup configuration information for this servlet. | | ServletContext | getServletContext() | Returns a ServletContext object, which contains information about the network service in which the servlet is running. | | java.lang.String | getServletInfo() | Returns a string that contains information about the servlet, such as its author, version, and copyright. | | void | init() | This method is provided as a convenience so that servlet writers do not have to worry about storing the ServletConfig object. | | void | log(java.lang.String msg) | Writes the class name of the servlet and the given message to the servlet log file. | | void | log(java.lang.String msg,java.lang.Throwable t) | Logs the message with the root cause | | abstract void | service(ServletRequest req,ServletResponse res) | Carries out a single request from the client. |  * **Class HttpServlet**   java.lang.Object  |  +--javax.servlet.GenericServlet  |  +--**javax.servlet.http.HttpServlet**  public abstract class **HttpServlet**  extends GenericServlet  implements java.io.Serializable   * It extends the GenericServlet base class and provides an framework for handling the HTTP protocol. * Because it is an abstract class, servlet writers must subclass it and override at least one method. The methods normally overridden are:   + doGet, if HTTP GET requests are supported. Overriding the doGet method automatically also provides support for the HEAD and conditional GET operations.   + doPost, if HTTP POST requests are supported.   + doPut, if HTTP PUT requests are supported.   + doDelete, if HTTP DELETE requests are supported.   + The lifecycle methods init and destroy, when Servlet need to manage resources that are held for the lifetime of the servlet otherwise there is no need to specialize these methods.   + Notice that the service method is not typically overridden. The service method, as provided, supports standard HTTP requests by dispatching them to appropriate methods, such as the methods listed above that have the prefix "do".      |  |  |  | | --- | --- | --- | | **Return Type** | **Method** | **Description** | | protected void | doDelete(HttpServletRequest req, HttpServletResponse resp) | Performs the HTTP DELETE operation; the default implementation reports an HTTP  BAD\_REQUEST error. | | protected void | doGet (HttpServletRequest req, HttpServletResponse resp) | Performs the HTTP GET operation; the default implementation reports an HTTP BAD\_REQUEST error. | | protected void | doOptions (HttpServletRequest req, HttpServletResponse resp) | Performs the HTTP OPTIONS operation; the default implementation of this method  automatically determines what HTTP Options are supported. | | protected void | doPost (HttpServletRequest req, HttpServletResponse resp) | Performs the HTTP POST operation; the default implementation reports an HTTP  BAD\_REQUEST error. | | protected void | doPut (HttpServletRequest req, HttpServletResponse resp) | Performs the HTTP PUT operation; the default implementation reports an HTTP BAD\_REQUEST error. | | protected void | doTrace (HttpServletRequest req, HttpServletResponse resp) | Performs the HTTP TRACE operation; the default implementation of this method causes  a response with a message containing all of the headers sent in the trace request. | | protected long | getLastModified(HttpServletRequest req) | Gets the time the requested entity was last modified; the default implementation returns a negative number, indicating that the modification time is unknown and hence should not be used for conditional GET operations or for other cache control operations as this implementation will always return the contents. | | protected void | service (HttpServletRequest req, HttpServletResponse resp) | This is an HTTP-specific version of the Servlet.service method, which accepts HTTP specific parameters. | | void | service (ServletRequest req, ServletResponse res) | Implements the high level Serlet.service method by delegating to the HTTP-specific service method. |  * **Interface ServletRequest**   All known subinterfaces:  HttpServletRequest  public abstract interface **ServletRequest**   * Defines a servlet engine generated object that enables a servlet to get information about a client request. * The ServletRequest object includes parameter names and values, attributes, and an input stream. * HttpServletRequest, which extends ServletRequest can provide additional protocol-specific data  |  |  |  | | --- | --- | --- | | **Return Type** | **Method** | **Description** | | java.lang.Object | getAttribute(java.lang.String name) | Returns the value of the named attribute of this request. | | java.util.Enumeration | getAttributeNames() | Returns an enumeration of attribute names contained in this request. | | java.lang.String | getCharacterEncoding() | Returns the character set encoding for the input of this request. | | java.lang.String | getCharacterEncoding() | Returns the character set encoding for the input of this request. | | int | getContentLength() | Returns the size of the request entity data, or -1 if not known. | | java.lang.String void | getContentType() | Returns the Internet Media (MIME) Type of the request entity data, or null if not known. | | ServletInputStream | getInputStream() | Returns an input stream for reading binary data in the request body. | | java.lang.String | getParameter(java.lang.String name) | Returns a string containing the lone value of the specified parameter, or null if the parameter does not exist. | | java.util.Enumeration | getParameterNames() | Returns the parameter names for this request as an enumeration of strings, or an empty enumeration if there are no parameters or the input stream is empty. | | java.lang.String[] | getParameterValues(java.lang.String name) | Returns the values of the specified parameter for the request as an array of strings, or null if the named parameter does not exist. | | java.lang.String | getProtocol() | Returns the protocol and version of the request as a string of the form <protocol>/<major version>.<minor version>. | | java.io.BufferedReader | getReader() | Returns a buffered reader for reading text in the request body. | | java.lang.String | getRemoteAddr() | Returns the IP address of the agent that sent the request. | | java.lang.String | getRemoteHost() | Returns the fully qualified host name of the agent that sent the request. | | java.lang.String | getServerName() | Returns the host name of the server that received the request. | | int | getServerPort() | Returns the port number on which this request was received. | | void | setAttribute(java.lang.String key, java.lang.Object o) | This method stores an attribute in the request context; these attributes will be reset between requests. |  * **Interface ServletResponse**   All known subinterfaces:  HttpServletResponse  public abstract interface **ServletResponse**   * Interface for sending MIME data from the servlet's service method to the client. * Network service developers implement this interface; its methods are then used by servlets when the service method is run, to return data to clients. * The ServletResponse object is passed as an argument to the service method. * To write MIME bodies which consist of binary data, use the output stream returned by getOutputStream. * To write MIME bodies consisting of text data, use the writer returned by getWriter.      |  |  |  | | --- | --- | --- | | **Return Type** | **Method** | **Description** | | java.lang.String | getCharacterEncoding() | Returns the character set encoding used for this MIME body. | | ServletOutputStream | getOutputStream() | Returns an output stream for writing binary response data. | | java.io.PrintWriter | getWriter() | Returns a print writer for writing formatted text responses. | | void | setContentLength(int len) | Sets the content length for this response. | | void | setContentType(java.lang.String type) | Sets the content type for this response. |  * **Sample Code:**   import java.io.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  /\* This Servlet returns Sample Servlet!!, the number of times this Servlet was requested and  the Servlet information. \*/  public class ServletTest1 extends HttpServlet  {  /\* Static makes variable visible to all instances \*/  /\* Number of times Servlet requested \*/  static int numRequests = 0;    public void init(ServletConfig config) throws ServletException  {  /\* Passing ServletConfig to parent \*/  super.init(config);  /\* Initializing resources here \*/  }  public void service(HttpServletRequest request, HttpServletResponse response) throws  ServletException, IOException  {    /\* Setting MIME type for HTTP header \*/  response.setContentType("text/html");  /\* Getting handle to the output stream \*/  PrintWriter out = response.getWriter();  /\* Streaming HTML to the browser \*/  out.println("<HTML>");  out.println("<HEAD><TITLE>Sample Servlet</TITLE></HEAD>");  out.println("<BODY>");  out.println("<H1>Sample Servlet !!</H1>");  out.println("<BR>");    /\* Incrementing hit counter \*/  numRequests++;  out.println("<P>This servlet has been requested " + numRequests + " time(s)");  out.println("<BR>");    /\* printing Servlet Information \*/  String info = getServletInfo();  out.println("Servlet information :"+info);    /\* Closing the output stream \*/  out.close();    }  /\* Identifying the Servlet to the Web Server when required \*/  public String getServletInfo()  {  return "Sample Servlet Example By Komal Agrawal.Version 1.0";  }    public void destroy()  {  /\* The Servlet is being unloaded. Free resources used by the Servlet here. \*/  System.out.println("Servlet unloaded...");  }  }   * **Interface HttpServletRequest**   Public abstract interface **HttpServletRequest**  extends ServletRequest   * This interface gets data from the client to the servlet for use in the HttpServlet.service method.  |  |  |  | | --- | --- | --- | | **Return Type** | **Method** | **Description** | | java.lang.String | getAuthType() | Gets the authentication scheme of this request. | | Cookie[] | getCookies() | Gets the array of cookies found in this request. | | Long | getDateHeader(java.lang.String name) | Gets the value of the requested date header field of this request. | | java.lang.String | getHeader(java.lang.String name) | Gets the value of the requested header field of this request. | | java.util.Enumeration | getHeaderNames() | Gets the header names for this request. | | int | getIntHeader(java.lang.String name) | Gets the value of the specified integer header field of this request. | | java.lang.String | getMethod() | Gets the HTTP method (for example, GET, POST, PUT) with which this request was  made. | | java.lang.String | getPathInfo() | Gets any optional extra path information following the servlet path of this request's URI,but immediately preceding its query string. | | java.lang.String | getPathTranslated() | Gets any optional extra path information following the servlet path of this request's URI, but immediately preceding its query string, and translates it to a real path. | | java.lang.String | getRemoteUser() | Gets the name of the user making this request. | | java.lang.String | getRequestedSessionId() | Gets the session id specified with this request. | | java.lang.String | getRequestURI() | Gets, from the first line of the HTTP request, the part of this request's URI that is to the left of any query string. | | java.lang.String | getServletPath() | Gets the part of this request's URI that refers to the servlet being invoked. | | HttpSession | getSession() | Gets the current valid session associated with this request, if create is false or, if necessary, creates a new session for the request. | | HttpSession | getSession(boolean create) | Gets the current valid session associated with this request, if create is false or, if necessary, creates a new session for the request, if create is true. | | boolean | isRequestedSessionIdFromCookie() | Checks whether the session id specified by this request came in as a cookie. | | boolean | isRequestedSessionIdValid() | Checks whether this request is associated with a session that is valid in the current session context. |  * **Interface HttpServletResponse**   public abstract interface **HttpServletResponse**  extends ServletResponse   * This interface allows a servlet's service method to manipulate HTTP-protocol specified header information and return data to its client.  |  |  |  | | --- | --- | --- | | **Return Type** | **Method** | **Description** | | void | addCookie(Cookie cookie) | Adds the specified cookie to the response. | | boolean | containsHeader(java.lang.String name) | Checks whether the response message header has a field with the specified name. | | java.lang.  String | encodeRedirectURL(java.lang.String url) | Encodes the specified URL for use in the sendRedirect method or, if encoding is not  needed, returns the URL unchanged. | | java.lang.String | encodeURL(java.lang.String url) | Encodes the specified URL by including the session ID in it, or, if encoding is not  needed, returns the URL unchanged. | | void | sendError(int sc) | Sends an error response to the client using the specified status code and a default  message. | | void | sendRedirect(java.lang.String location) | Sends a temporary redirect response to the client using the specified redirect location  URL. | | void | setDateHeader(java.lang.String name, long date) | Adds a field to the response header with the given name and date-valued field. | | void | setHeader(java.lang.String name, java.lang.String value) | Adds a field to the response header with the given name and value. | | void | setIntHeader(java.lang.String name, int value) | Adds a field to the response header with the given name and integer value. | | void | setStatus(int sc) | Sets the status code for this response. |   **Sample Code:**  public void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException  {  long requestDate;    /\* import java.util.Date \*/  Date date= null;    response.setContentType("text/html");  PrintWriter out = response.getWriter();    requestDate = request.getDateHeader("Date");  if(requestDate == -1)  {  out.println("Date header not found !!");  }  else  {  try  {  date = new Date();  out.println(" Date :"+ date);  }  catch(IllegalArgumentException ilge)  {  out.println("Invalid Request date !!");  }  }    /\* Closing the output stream \*/  out.close();  }   * Following sample code demonstrates how the names of the headers are extracted with getHeaderNames() and the header values are returned using the getHeader() method (requires java.util.Enumaration to be imported)   public void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException  {  String headerName=” “;  response.setContentType("text/plain");  PrintWriter out = response.getWriter();    Enumeration enum = request.getHeaderNames();  while(enum.hasMoreElements())  {  headerName = (String)enum.nextElement();  /\* Get the value of the header \*/    out.println(headerName+”:”+request.getHeader(headerName));  }    out.close(); // Always close the output stream  }   * **L5.1.5 Servlet Lifecycle**  1. The server loads the servlet when it is first requested by a client or, if configured to do so, at server start-up. The servlet may be loaded from either a local or remote location using the standard java class loading facility. This step is equivalent to:   Class c = Class.forName(“com.sourcestream.MyServlet”);  The term *load* often refers to both loading and instantiating the servlet.   1. The server creates one or more instances of the servlet class depending on the implementation.   The server may create a single instance that services all requests through multiple threads or create a pool of instances from which one is chosen to service each new request. This step is equivalent to:  Servlet s =(Servlet)c.newInstance();   1. The server constructs the ServletConfig object that provides initialization information to the servlet. 2. Server calls the servlet’s init() method, passing ServletConfig object. init() method finishes execution prior to servlet processing the first request.   If Server has multiple instances, the init() method is called one time for each instance.   1. The server constructs a request and response object that provides methods for customizing servlet’s request - response. The type of request - response object passed in this method depends on whether the servlet extends the GenericServlet class or HttpServlet class. 2. The server calls Servlet’s service() passing the request - response objects when concurrent request arrives. Multiple service() method can run in separate threads. 3. The service() processes the client request by evaluating the ServletRequest or HttpServletRequest object & response using ServletResponse or HttpServletResponse. 4. If the server receives another request for this servlet, the process begins again at step 5. 5. When instructed to unload the servlet, perhaps by the server administrator or programmatically by the servlet itself, the server call the servlet’s destroy(). The servlet is then eligible for garbage collection.  * **L5.1.6 Servlet Debugging Strategies**   Debugging servlets can be tricky because they don’t execute directly. They are triggered by means of an HTTP request, and they are executed by the Web server. This remote execution makes it difficult to insert break points or to read debugging messages and stack traces.  So, approaches to servlet debugging differ somewhat from those used in general development.  Here are seven general strategies to debug the servlets that can make your life easier.   1. **Look at the HTML source.**   Go through HTML source to avoid mistakes like <TABLE> instead of </TABLE>. Better is touse a formal HTML validator on the servlet’s output.   1. **Return error pages to the client**.   Certain classes of errors are anticipated by the servlet. In such case, the servlet should build descriptive information about the problem and return it to the client in a regular page or by means of the sendError method of HttpServletResponse.   1. **Start the server from the command line.**   Most Web servers execute from a background process, and this process is often automatically started when the system is booted. If you are having trouble with your servlet, you should consider shutting down the server and restarting it from the command line. After this, System.out.println or System.err.println calls can be easily read from the window in which the server was started.  When something goes wrong with servlet got before it failed and to gather some information about the key data structures during the time period just before it failed. Simple println statements are surprisingly effective for this purpose. If you are running your servlets on a server that you cannot easily halt and restart, then do your debugging with the JSWDK, Tomcat, or the Java Web Server on your personal machine, and save deployment to the real server for later.   1. **Use the log file.**   The HttpServlet class has a method called log that lets you write information into a logging file on the server. Reading debugging messages from the log file is a bit less convenient than watching them directly from a window as with the previous approach, but using the log file does not require stopping and restarting the server.  There are two variations of this method: one that takes a String, and the other that takes a String and a Throwable (an ancestor class of Exception).   1. **Look at the request data separately.**   Servlets read data from the HTTP request, construct a response, and send it back to the client. If something in the process goes wrong, you want to discover if it is because the client is sending the wrong data or because the servlet is processing it incorrectly.   1. **Look at the response data separately.**   Once you look at the request data separately, you’ll want to do the same for the response data.   1. **Stop and restart the server.**   Most full-blown Web servers that support servlets have a designated location for servlets that are under development. Servlets in this location (e.g., the servlets directory for the Java Web Server) are supposed to be automatically reloaded when their associated class file changes. At times, however, some servers can get confused, especially when you only changes is to a lower-level class, not to the top-level servlet class. So, if it appears that changes you make to your servlets are not reflected in the servlet’s behavior, try restarting the server.   * **L5.2 Handling Client Request : Form Data**   **// Reference:** [**http://pdf.coreservlets.com/Form-Data.pdf**](http://pdf.coreservlets.com/Form-Data.pdf)   * **L5.2.1 Role of Form Data**   + Whenever you used Search Engine, visited any online booking site or asked something on Google, you’ve probably seen funny-looking URLs like:   **http://www.google.co.in/search?hl=en&q=Form+data&meta=.**   * + The part after the question mark (i.e., hl=en&q=Form+data&meta= ) is known as *form data* (or *query data*) and is the most common way to get information from a Web page to a server-side program. * **L5.2.2 Creating and Submitting HTML Forms**   + Form data can be attached to the end of the URL after a question mark (as above), for GET requests, or sent to the server on a separate line, for POST requests.   + Before going to handle Servlet request response objects you need to familiar with HTML forms.   <HTML>  <HEAD>  <TITLE>A Sample FORM using POST</TITLE>  </HEAD>  <BODY BGCOLOR="#FDF5E6">  <H1 ALIGN="CENTER">A Sample FORM using POST</H1>  <FORM ACTION="/servlet/pckgname.ShowParameters" METHOD="POST">    First Name: <INPUT TYPE="TEXT" NAME="firstName"><BR>  Last Name: <INPUT TYPE="TEXT" NAME="lastName"><BR>  Middle Initial: <INPUT TYPE="TEXT" NAME="initial"><BR>  Credit Card : <BR>    <INPUT TYPE="checkbox" NAME="cardType" VALUE="Visa">Visa<BR>  <INPUT TYPE="checkbox" NAME="cardType" VALUE="Master Card">Master Card<BR>  <INPUT TYPE="checkbox" NAME="cardType" VALUE="Amex">American Express<BR>  <INPUT TYPE="checkbox" NAME="cardType" VALUE="Discover">Discover<BR>  <INPUT TYPE="checkbox" NAME="cardType" VALUE="Java SmartCard">Java SmartCard<BR>  Credit Card Number:  <INPUT TYPE="PASSWORD" NAME="cardNum"><BR>  Repeat Credit Card Number:  <INPUT TYPE="PASSWORD" NAME="cardNum"><BR><BR>  <CENTER>  <INPUT TYPE="SUBMIT" VALUE="Submit Order">  </CENTER>  </FORM>  </BODY>  </HTML>   * **L5.2.3 Reading request parameters** * Each access to a servlet can have any number of request parameters associated with it. * getParameter()method of the HttpServletRequest, returns the value of the named parameter as a String or null if the parameter is not specified. * Call the getParameter method of the HttpServletRequest, supplying the case-sensitive parameter name as given in HTML tag as an argument to method.   gw  public void doPost(HttpServletRequest request, HttpServletResponse response)throws ServletException, IOException  {  response.setContentType("text/html");  /\* Getting handle to the output stream \*/  PrintWriter out = response.getWriter();  out.println("<html>");  out.println("<body>");  out.println("Reading single parameter...");  out.print;n(“<br>”);  out.println("First Name :"+request.getParameter("firstName"));  out.print;n(“<br>”);  out.println("Last Name :"+request.getParameter("lastName"));  out.println("</body>");  out.println("</html>");  out.close();  }   * Sometimes you may have multiple values for single parameter. * This time you need to use getParameterValues() instead of getParameter() * This method returns all the values of the named parameter as an array of String objects or null if the parameter was not specified. * A single value is returned in an array of length 1.     public void doPost(HttpServletRequest request, HttpServletResponse response)throws ServletException, IOException  {  response.setContentType("text/html");  /\* Getting handle to the output stream \*/  PrintWriter out = response.getWriter();  out.println("<html>");  out.println("<body>");    out.println("Reading multiple parameters...");    /\* this method returns String array \*/  String params[] = request.getParameterValues("cardType");  for(int i=0;i<params.length;i++)  {  out.println("parameter "+ i+": "+params[i]);  }  out.println("</body>");  out.println("</html>");  out.close();  }   * In addition to getting parameter values, a servlet can access parameter names using getParameterNames():   public Enumeration ServletRequest.getParameterNames()   * This method returns all the parameter names as an Enumeration of String object or an empty Enumeration if the servlet has no parameters. * The method is most often used for debugging. * **L5.2.4 Handling Missing and Malformed data** * On online web services sometimes it may happen that user fails to supply the necessary information. Here is the solution how to handle the situation; it has two solutions: use default values or redisplay the form (prompting the user for missing values). * While dealing with missing and malformed data; check for three conditions: * **The value is null.** A call to request.getParameter returns null if the form contains no textfield or other element of the expected name so that the parameter name does not appear in the request at all. * **The value is an empty string.** A call to request.getParameter returns an empty string (i.e., "") if the associated textfield is empty when the form is submitted. To check for an empty string, compare the string to "" by using equals or compare the length of the string to 0.   String param = request.getParameter("someName");  if ((param == null) || (param.trim().equals("")))  {  doSomethingForMissingValues(...);  }  Else  {  doSomethingWithParameter(param);  }   * **The value is a nonempty string of the wrong format.** What defines the wrong format is application specific: you might expect certain textfields to contain only numeric values, others to have exactly seven characters, and others to only contain single letters.   **Sample Code: MissingTest.jsp**  <FORM ACTION="./ServletMissingTest" METHOD="POST">    First Name: <INPUT TYPE="TEXT" NAME="firstName"><BR>  Last Name: <INPUT TYPE="TEXT" NAME="lastName"><BR>  Middle Initial: <INPUT TYPE="TEXT" NAME="initial"><BR>    Mother-tounge: <INPUT TYPE="TEXT" NAME="motherTongue"><BR>  Credit Card:  <BR>    <INPUT TYPE="radio" NAME="cardType"  VALUE="Visa">Visa<BR>  <INPUT TYPE="radio" NAME="cardType"  VALUE="Master Card">Master Card<BR>  <INPUT TYPE="radio" NAME="cardType"  VALUE="Amex">American Express<BR>  <INPUT TYPE="radio" NAME="cardType"  VALUE="Discover">Discover<BR>  <INPUT TYPE="radio" NAME="cardType"  VALUE="Java SmartCard">Java SmartCard<BR>    Credit Card Number: <INPUT TYPE="PASSWORD" NAME="cardNum"><BR>  Repeat Credit Card Number: <INPUT TYPE="PASSWORD" NAME="cardNum"><BR><BR>    <CENTER>  <INPUT TYPE="SUBMIT" VALUE="Submit Order">  </CENTER>    </FORM>  **Sample Code: ServletMissingTest.java**  public void doPost(HttpServletRequest request, HttpServletResponse response)  throws ServletException, IOException  {  response.setContentType("text/html");    PrintWriter out = response.getWriter();  out.println("<html>");  out.println("<body>");    String motherTongue = request.getParameter("motherTongue");  motherTongue = replaceIfMissing(motherTongue, "English");  out.println("Mother tongue : "+motherTongue);  out.println("<br>");    String cardType = request.getParameter("cardType");  cardType = replaceIfMissingOrDefault(cardType, "Visa");  out.println(" Credit card type : "+cardType);  out.println("<br>");    String fName = request.getParameter("firstName");  out.println("First Name :"+fName);  out.println("<br>");    out.println("Last Name :"+request.getParameter("lastName"));  out.println("<br>");      out.println("</body>");  out.println("</html>");  out.close();    }    /\* Replaces null strings, empty strings, with the replacement and returns ,  \* otherwise original string is returned.  \*/  private String replaceIfMissing(String orig,String replacement)  {  if ((orig == null) || (orig.trim().equals("")))  {  return(replacement);  } else {  return(orig);  }  }    /\* Replaces null strings, empty strings, or the string  \* "default" with the replacement.  \* Returns the original string otherwise.  \*/  private String replaceIfMissingOrDefault(String orig,String replacement)  {  if ((orig == null) ||(orig.trim().equals("")) ||(orig.equals("default")))  {  return(replacement);  }  else  {  return(orig + ", ");  }  }   * This is sample code which shows how to deal with missing fields in the form. * Malformed Value is a nonempty string in the wrong format * These fields are handled by displaying warning message.   Refer book for Example: Core Servlets and Java Server Programming by  Note: [**http://courses.coreservlets.com/**](http://courses.coreservlets.com/)  <http://java.sun.com/javaee/5/docs/api/>  **Summary:**   * Servlets are programs that run on a Web server, acting as a middle layer between a requests coming from a Web browser or other HTTP client and databases or applications on the HTTP server. * They extend one of two Servlet classes – GenericServlet or HttpServlet. * The init() method is called only once in Servlet’s life, and always before the servlet can service the client’s requests. * Most of the Servlet’s life is spent running a service() method for client request. * Container end’s servlet’s life by calling destroy() method. * The most commonly used classes and interfaces that manages the Servlet and its communications with its client are included in javax.servlet javax.servlet.http packages * Every request to a servlet runs in a seperate thread! There is only one instance of any perticular servlet class. * The HttpServlet's doGet() and doPost() methods take an HttpServletRequest and an HttpServletResponse objects. * The service() method determines whether doGet() or doPost() runs based on thye HTTP metod(GET, POST, etc.) of the HTTP request. * Form data is the most common way to get information from a Web page to a server-side program. * Form data can be attached to the end of the URL after a question mark, for GET requests, or sent to the server on a separate line, for POST requests. * You can get parameters from the request with getParameter("paramName") method. the return value is always String object. * If you have multiple parameter values for a given parameter name, use the getParameterValues("paramName") method that returns a String array. * While dealing with missing and malformed data, three conditions are checked: * The value is null * The value is an empty string * The value is a nonempty string of the wrong format   **Exercise**  **Objectives:**   1. **Please go through code listings. Identify the probable errors, either compile time or run time.** 2. **Document the Error, Explanation of the error and the probable solution/fix for the errors identified**   **Pre-requisites:**   1. **You must have a good conceptual understanding of Session 2 – “JDBC- Introduction” of the Intermediate Java course**  Code Listing 1 **1**  --------------------------------->//START OF HTML//  <html>  <head>  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  <title>Login Page</title>  </head>  <body bgcolor="#dddeedd">  <center>  <h1>Login</h1>  Enter DB User  <form method="post" action="http://localhost:8084/Case/Login">  UserName :<input type="text" name="username" align="center"><br>  Password :<input type="password" name="password" align="center"><br>  <input type="submit" value="Login">  </form>    </center>  </body>  </html>  --------------------------------->//END OF HTML//  --------------------------------->//START OF DEPLOYMENT DESCRIPTOR//  <?xml version="1.0" encoding="UTF-8"?><web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app\_2\_4.xsd">  <context-param>  <param-name>class</param-name>  <param-value>sun.jdbc.odbc.JdbcOdbcDriver</param-value>  </context-param>  <context-param>  <param-name>connect</param-name>  <param-value>jdbc:odbc:jdbcdsn</param-value>  </context-param>  <context-param>  <param-name>username</param-name>  <param-value>scott</param-value>  </context-param>  <context-param>  <param-name>password</param-name>  <param-value>tiger</param-value>  </context-param>  <servlet>  <servlet-name>Login</servlet-name>  <servlet-class>Login</servlet-class>  </servlet>  --------------------------------->//END OF DEPLOYMENT DESCRIPTOR//  --------------------------------->//START OF SERVLET FILE //  static int flag;  protected void doPost(HttpServletRequest request, HttpServletResponse response)  throws ServletException, IOException {  response.setContentType("text/html;charset=UTF-8");  PrintWriter out = response.getWriter();  ServletContext context=getServletContext();  String class1 = context.getInitParameter("class");  String connect=context.getInitParameter("connect");  String uname=context.getInitParameter("username");  String pass=context.getInitParameter("password");  String nam = request.getParameter("username");  String pas = request.getParameter("password");  try {  Class.forName(class1);  } catch (ClassNotFoundException ex) {  ex.printStackTrace();  }  try{  //Connection Created  Connection con1=DriverManager.getConnection(connect,uname,pass);  // values are fetched from the table using the oracle SELECT query command  Statement smt=con1.createStatement();  ResultSet rs=smt.executeQuery("select \* from project7181;");    // the USERNAME and PASSWORD entered by the user is validated here.  while(rs.next()) {  if(nam.equals(rs.getString(1))&& pas.equals(rs.getString(2)) ) {    flag=1;  break;  } else {  flag=0;  }  }  if(flag==1) {    // SUCESSFULLOGIN.JSP page is called if Validation successful  response.sendRedirect("SucessfulLogin.jsp");  //out.println("<html><body> <a href=http://localhost:8084/Case/SucessfulLogin.jsp><h3>Click Here to Proceed</h3> </form> </body></html> ");  } else {    // UNSUCESSFUL LOGIN lead to re call of the page  response.sendRedirect("UnSucessfulLogin.jsp");  }  }catch(SQLException e) {  e.printStackTrace();  }      out.close();  }  --------------------------------->//END OF SERVLET FILE // Code Listing 2 --------------------------------->//START OF HTML//  <html>  <head>  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  <title>Login Page</title>  </head>  <body bgcolor="#dddeedd">  <center>  <h1>Login</h1>  Enter DB User  <form method="post" action="http://localhost:8084/Case/Login">  UserName :<input type="text" name="username" align="center"><br>  Password :<input type="password" name="password" align="center"><br>  <input type="submit" value="Login">  </form>    </center>  </body>  </html>  --------------------------------->//END OF HTML//  --------------------------------->//START OF DEPLOYMENT DESCRIPTOR//  <?xml version="1.0" encoding="UTF-8"?><web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app\_2\_4.xsd">  <context-param>  <param-name>class</param-name>  <param-value>sun.jdbc.odbc.JdbcOdbcDriver</param-value>  </context-param>  <context-param>  <param-name>connect</param-name>  <param-value>jdbc:odbc:jdbcdsn</param-value>  </context-param>  <context-param>  <param-name>username</param-name>  <param-value>scott</param-value>  </context-param>  <context-param>  <param-name>password</param-name>  <param-value>tiger</param-value>  </context-param>  <servlet>  <servlet-name>Login</servlet-name>  <servlet-class>Login</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>Login</servlet-name>  <url-pattern>/Login</url-pattern>  </servlet-mapping>  --------------------------------->//END OF DEPLOYMENT DESCRIPTOR//  --------------------------------->//START OF SERVLET FILE //  static int flag;  protected void doPost(HttpServletRequest request, HttpServletResponse response)  throws ServletException, IOException {  response.setContentType("text/html;charset=UTF-8");  PrintWriter out = response.getWriter();  ServletContext context=getServletContext();  String class1 = context.getInitParameter("class");  String connect=context.getInitParameter("connect");  String uname=context.getInitParameter("username");  String pass=context.getInitParameter("password");  String nam = request.getParameter("username");  String pas = request.getParameter("password");  try {  Class.forName(class1);  } catch (ClassNotFoundException ex) {  ex.printStackTrace();  }  try{  //Connection Created  Connection con1=DriverManager.getConnection(connect,uname,pass);  // values are fetched from the table using the oracle SELECT query command  Statement smt=con1.createStatement();  ResultSet rs=smt.executeQuery("select \* from project7181;");    // the USERNAME and PASSWORD entered by the user is validated here.  while(rs.next()) {  if(nam.equals(rs.getString(1))&& pas.equals(rs.getString(2)) ) {    flag=1;  break;  } else {  flag=0;  }  }  if(flag==1) {    // SUCESSFULLOGIN.JSP page is called if Validation successful  response.sendRedirect("SucessfulLogin.jsp");  //out.println("<html><body> <a href=http://localhost:8084/Case/SucessfulLogin.jsp><h3>Click Here to Proceed</h3> </form> </body></html> ");  } else {    // UNSUCESSFUL LOGIN lead to re call of the page  response.sendRedirect("UnSucessfulLogin.jsp");  }  }catch(SQLException e) {  e.printStackTrace();  }      out.close();  }  --------------------------------->//END OF SERVLET FILE //  6.Servlets – Request and Response Headers     * **Course Objectives :**  Upon conclusion participants will have acquired these skills:      * Handling the Client Request using HTTP Request Headers. * Building a table of all the request headers. * Understanding of the various request headers. * Reducing download times by compressing pages. * Differentiating among types of browsers. * **Introduction :**   Efficient handling of HTTP protocol increases the performance of  Servlets. In depth knowledge of HTTP is not required here but it will have an immediate  impact on the performance and usability of your Servlets.  Here we will discuss the HTTP information that is sent from the browser to the server in the form of Request Headers. It explains the most important HTTP 1.1 request headers, summarizing how and why they would be used in a Servlet. As we see later, request headers are read and applied the same way in JSP pages as they are in Servlets. Note that   * HTTP request headers are distinct from the form (query) data discussed in the previous chapter. * Form data results directly from user input and is sent as part of the URL for GET requests and on a separate line for POST requests. * Request Headerson the other hand, are indirectly set by the browser and are sent immediately following the initial GET or POST request line.   For instance, the following example shows an HTTP request that might result from a user submitting a particular software search request to a Servlet at  **http://www.some\_software\_store.com/winzip/Search**. The request includes the  headers -:   * Accept * Accept-Encoding * Connection, Cookie * Host * Referer * User-Agent   all of which might be important to the operation of the Servlet, but  none of which can be derived from the form data or deduced automatically: the Servlet needs to explicitly  read the request headers to make use of this information.  GET/winzip/Search?keywords=winzip+rar HTTP/1.1  **Accept**: image/gif, image/jpg, \*/\*  **Accept-Encoding**: gzip  **Connection**: Keep-Alive  **Cookie**: userID=id68916  **Host**: www. some\_software\_store.com  **Referer**: http://www.some\_software\_store /findsoftware.html  **User-Agent**: InternetExplorer/7.0 (compatible; MSIE 6.0; Windows NT 5.0)   * **Reading HTTP request headers :**   By calling the **getHeader** method of HttpServletRequest with the name of header we can read the Header. For  ex: request.getHeader("Connection").Here request.getHeader is a method and  “Connection” is a Header. This call returns a String if the specified header was  supplied in the current request, null otherwise.    Header names are not case sensitive. So, for example,  request.getHeader(“Connection”) is interchangeable with the method given as  request.getHeader(“connection”).    Though getHeader is the one of the most common way  to read incoming headers, still a few headers are there which are used so commonly  that they have special access methods in HttpServletRequest. Following is a summary     * **getCookies :** The getCookies method returns the contents of the Cookieheader,   parsed and stored in an array of Cookie objects.   * **getAuthType and getRemoteUser :**The getAuthType and getRemoteUser methods   break the Authorization header into its component pieces.   * **getContentLength:** The getContentLength method returns the value of the   Content-Length header (as an int).   * **getContentType :**The getContentType method returns the value of the   Content-Type header (as a String).   * **getDateHeader and getIntHeader :** The getDateHeader and getIntHeader methods read the specified headers and then convert them to Date and int values,respectively. * **getHeaderNames :** Rather than looking up one particular header, you can use the   getHeaderNames method to get an Enumeration of all header names received on this particular request.   * **getHeaders :**In most cases, each header name appears only once in the request.   Occasionally, however, a header can appear multiple times, with each occurrence  listing a separate value. Accept-Language is one such example. You can use  getHeaders to obtain an Enumeration of the values of all occurrences of the header.  In addition to looking up the request headers, you can get information on  the main request line itself or by means of methods in HttpServletRequest. Here is a  summary of the four main methods -:     * **getMethod** : The getMethod method returns the main request method (normally,   GET or POST, but methods like HEAD, PUT, and DELETE are possible).   * **getRequestURI :** The getRequestURI method returns the part of the URL that comes after the host and port but before the form data. For example, for a URL of http://some\_host.com/servlet/search.SoftwareSearch?subject=winzip,   getRequestURI would return "/servlet/search.SoftwareSearch".   * **getQueryString :**The getQueryString method returns the form data. For example,   with http://some\_host.com/servlet/search.SoftwareSearch?subject=winzip,  getQueryString would return "subject=winzip".   * **getProtocol** The getProtocol method returns the third part of the request line,   which is generally HTTP/1.0 or HTTP/1.1.   * **Building a table of all the request headers :**   The following example shows a Servlet that simply creates a table of all the headers it receives,along with their associated values. It accomplishes this task by calling request.getHeaderNames to obtain an Enumeration of headers in the current request. It then loops down the Enumeration, puts the header name in the left table cell, and puts the result of getHeader in the right table cell. Recall that Enumeration is a standard interface in Java; it is in the java.util package and contains just two methods: hasMoreElements and nextElement. The servlet also prints three components of the main request line (method, URI, and protocol).  import java.io.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  import java.util.\*;  /\*\* Shows all the request headers sent on the current request. \*/  public class ShowRequestHeaders extends HttpServlet {  public void doGet(HttpServletRequest request,  HttpServletResponse response)  throws ServletException, IOException {  response.setContentType("text/html");  PrintWriter out = response.getWriter();  String title = "Servlet Example: Showing Request Headers";  String docType =  "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +  "Transitional//EN\">\n";  out.println(docType +  "<HTML>\n" +  "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +  "<BODY BGCOLOR=\"#FDF5E6\">\n" +  "<H1 ALIGN=\"CENTER\">" + title + "</H1>\n" +  "<B>Request Method: </B>" +  **request.getMethod()** + "<BR>\n" +  "<B>Request URI: </B>" +  **request.getRequestURI()** + "<BR>\n" +  "<B>Request Protocol: </B>" +  **request.getProtocol()** + "<BR><BR>\n" +  "<TABLE BORDER=1 ALIGN=\"CENTER\">\n" +  "<TR BGCOLOR=\"#FFAD00\">\n" +  "<TH>Header Name<TH>Header Value");  Enumeration headerNames = **request.getHeaderNames();**  while(headerNames.hasMoreElements()) {  String headerName = (String)headerNames.nextElement();  out.println("<TR><TD>" + headerName);  out.println(" <TD>" + **request.getHeader(headerName)**);  }  out.println("</TABLE>\n</BODY></HTML>");  }  /\*\* Since this servlet is for debugging, have it  \* handle GET and POST identically.  \*/  public void doPost(HttpServletRequest request,  HttpServletResponse response)  throws ServletException, IOException {  doGet(request, response);  }  }       * **Understanding the various request headers** :   Accessing request headers permits Servlets to perform a number of optimizations. Some of the features are not otherwise possible. This section summarizes  the headers most often used by Servlets  **Accept :**   * This header specifies the MIME types that the browser or other clients can * handle. A servlet that can return a resource in more than one format can examine the Accept header to decide which format to use. * For example, images in PNG format have some compression advantages over those in GIF, but not all browsers support PNG. If you have images in both formats, your servlet can call request.getHeader("Accept"), check for image/png, and if it finds a match, use *blah*.png filenames in all the IMG elements it generates.   Otherwise, it would just use *blah*.gif.  **Accept-Language :**   * This header specifies the client’s preferred languages in case the servlet can   produce results in more than one language.  **Authorization :**   * This header is used by clients to identify themselves when accessing   password-protected Web pages.    **Connection :**   * This header indicates whether the client can handle persistent HTTP connections. * Persistent connections permit the client or other browser to retrieve   multiple files (e.g., an HTML file and several associated images) with a single  socket connection, thus saving the overhead of negotiating several independent  connections.   * In HTTP 1.0, a value of Keep-Alive means that persistent connections should be used. * Each HTTP request results in a new invocation of a servlet (i.e., a thread calling   the servlet’s service and do*Xxx* methods), regardless of whether the  request is a separate connection.   * That is, the server invokes the servlet only   after the server has already read the HTTP request. This means that servlets  need to cooperate with the server to handle persistent connections.   * Consequently, the servlet’s job is just to make it *possible* for the server to use persistent connections; the servlet does so by setting the Content-Length   response header.  **Content-Length :**   * This header is applicable only to POST requests and gives the size of the POST data   in bytes.   * Rather than calling request.getIntHeader("Content-Length"),you can simply use request.getContentLength().   **Cookie**   * This header returns cookies to servers that previously sent them to the browser.   **Host**   * In HTTP 1.1, browsers and other clients are *required* to specify this header,which indicates the host and port as given in the original URL. * Because of the widespread use of virtual hosting (one computer handling Web sites for multiple domain names), it is quite possible that the server could not otherwise   determine this information.   * This header is not new in HTTP 1.1, but in HTTP 1.0 it was optional, not required.   **User-Agent**   * This header identifies the browser or other client making the request and can be   used to return different content to different types of browsers.  **Reducing download time by compressing pages** :   * Gzip compression scheme can dramatically reduce the download time of long   text pages.   * Most recent browsers know how to handle gzipped content, so the server can compress the document and send the smaller document over the network, after which the browser will automatically reverse the compression (no user action required). * Sending such compressed content can be a real time saver because compressing the document on the server and then uncompress it on the client reduces the time taken in data transfer and also network load, especially when dialup connections are used. * Browsers that support this feature indicate that they do so by setting the Accept-Encoding request header.   Implementing compression is straightforward since support for the gzip format is built in to the Java programming language by classes in java.util.zip. The servlet first checks the Accept-Encoding header to see if it contains an entry for gzip. If so, it uses a PrintWriter wrapped around a GZIPOutputStream and specifies gzip  as the value of the Content-Encoding response header. If gzip is not supported, the  servlet uses the normal PrintWriter and omits the Content-Encoding header.  To make it easy to compare regular and compressed performance with the same  browser, we also added a feature whereby we can suppress compression by including  ?disableGzip at the end of the URL.  **Example: 1**  import java.io.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  /\*\* Servlet with <B>long</B> output. Used to test  \* the effect of the gzip compression.  \*/  public class LongServlet extends HttpServlet {  public void doGet(HttpServletRequest request,  HttpServletResponse response)  throws ServletException, IOException {  response.setContentType("text/html");  **// Change the definition of "out" depending on whether**  **// or not gzip is supported.**  **PrintWriter out;**  **if (GzipUtilities.isGzipSupported(request) &&**  **!GzipUtilities.isGzipDisabled(request)) {**  **out = GzipUtilities.getGzipWriter(response);**  **response.setHeader("Content-Encoding", "gzip");**  **} else {**  **out = response.getWriter();**  **}**  // Once "out" has been assigned appropriately, the  // rest of the page has no dependencies on the type  // of writer being used.  String docType =  "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +  "Transitional//EN\">\n";  String title = "Long Page";  out.println  (docType +  "<HTML>\n" +  "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +  "<BODY BGCOLOR=\"#FDF5E6\">\n" +  "<H1 ALIGN=\"CENTER\">" + title + "</H1>\n");  String line = "Blah, blah, blah, blah, blah. " +  "Yadda, yadda, yadda, yadda.";  for(int i=0; i<10000; i++) {  out.println(line);  }  out.println("</BODY></HTML>");  **out.close(); // Needed for gzip; optional otherwise.**  }  }  **Example: 2**  import java.io.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  import java.util.zip.\*;  /\*\* Three small static utilities to assist with gzip encoding.  \* <UL>  \* <LI>isGzipSupported: does the browser support gzip?  \* <LI>isGzipDisabled: has the user passed in a flag  \* saying that gzip encoding should be disabled for  \* this request? (Useful so that you can measure  \* results with and without gzip on the same browser).  \* <LI>getGzipWriter: return a gzipping PrintWriter.  \* </UL>  \*/  public class GzipUtilities {  /\*\* Does the client support gzip? \*/  public static boolean isGzipSupported  (HttpServletRequest request) {  **String encodings = request.getHeader("Accept-Encoding");**  **return((encodings != null) &&**  **(encodings.indexOf("gzip") != -1));**  }  /\*\* Has user disabled gzip (e.g., for benchmarking)? \*/  public static boolean isGzipDisabled  (HttpServletRequest request) {  String flag = request.getParameter("disableGzip");  return((flag != null) && (!flag.equalsIgnoreCase("false")));  }  /\*\* Return gzipping PrintWriter for response. \*/  public static PrintWriter getGzipWriter  (HttpServletResponse response) throws IOException {  return(new PrintWriter  (new GZIPOutputStream  (response.getOutputStream())));  }  }    Since the Windows version of Internet Explorer 6 and onwards supports gzip,  this page was sent gzipped over the network and automatically reconstituted by the browser,  resulting in a large saving in download time.  Till here we learned how toHandling the Client Request .Now we will learn how to Generate the Server Response.  **Generating the Server Response: HTTP Response Headers :**  A response from a Web server normally consists of   * status line * one or more response headers (one of which must be Content-Type) * a blank line and * document.   To get the most out of your Servlets, you need to know how to use the status  line and response headers effectively, not just how to generate the document.  Response headers can be used to specify   * Cookies * to supply the page modification date (for client-side caching) * to instruct the browser to reload the page after a designated interval to give the file size so that persistent HTTP connections can be used * to designate the type of document being generated and * to perform many other tasks.   This part of the chapter shows how to generate response headers, explains  what the various headers are used for, and gives several examples.   * **Setting response headers from Servlets :**     The most general way to specify headers is to use the setHeader  method of HttpServletResponse. This method takes two strings: the header name and the  header value. As with setting status codes, you must specify headers *before* returning the  actual document.   * **setHeader(String headerName, String headerValue):** This method sets the response header with the designated name to the given value.     In addition to the general-purpose setHeader method, HttpServletResponse  also has two specialized methods to set headers that contain dates and integers:   * **setDateHeader(String header, long milliseconds):** This method saves you the   trouble of translating a Java date in milliseconds since 1970 (as returned by System.currentTimeMillis, Date.getTime, or Calendar.getTimeInMillis) into a GMT time string.   * **setIntHeader(String header, int headerValue):** This method spares you the minor inconvenience of converting an int to a String before inserting it into a header.   Finally, HttpServletResponse also supplies a number of convenience methods  for specifying common headers. These methods are summarized as follows.   * **setContentType(String mimeType):** This method sets the Content-Type header and is used by the majority of servlets. * **setContentLength(int length):** This method sets the Content-Length header, which is useful if the browser supports persistent (keep-alive) HTTP connections. * **addCookie(Cookie c):**This method inserts a cookie into the Set-Cookie header. There is no corresponding setCookie method, since it is normal to have multiple Set-Cookie lines. (we will discuss much about cookies in next chapter). * **sendRedirect(String address):** In send Redirect whenever the client makes any request it goes to the container, there the container decides whether the concerned servlet can handle the request or not.     If not then the servlet decides that the request can be handle by other servlet or jsp. Then the servlet calls the **sendRedirect()** method of the response object and sends back the response to the browser along with the status code. Then the browser sees the status code and look for that servlet which can now handle the request.   Again the browser makes a new request, but with the name of that servlet which can now handle the request and the result will be displayed to you by the browser. The URL will have the address of the new servlet. In all this process the client is unaware of the processing. * **Understanding what response headers are good for** :   Following is a summary of the most useful HTTP 1.1 response headers. A good understanding of these headers can increase the effectiveness of your servlets, so you should skim the descriptions to see what options are at your disposal   * **Allow :**The Allow header specifies the request methods (GET, POST, etc.) that the   server supports. It is required for 405 (Method Not Allowed) responses.The default  service method of servlets automatically generates this header for OPTIONS requests.   * **Connection:** A value of close for this response header instructs the browser not to   use persistent HTTP connections. Technically, persistent connections are the default when the client supports HTTP 1.1 and does *not* specify a Connection: close request header (or when an HTTP 1.0 client specifies Connection: keep-alive). However, since persistent connections require a Content-Length response header, there is no reason for a servlet to explicitly use the Connection header. Just omit the Content-Length header if you aren’t using persistent connections.     * **Content-Type**   The Content-Type header gives the MIME (Multipurpose Internet Mail Extension) type of the response document. Setting this header is so common that there is a special method in HttpServletResponse for it:setContentType. Most servlets specify text/html; they can, however, specify other types instead. This is important partly because servlets directly generate other MIME types (as in the Excel and JPEG examples of this chapter).  Here is the lists of some of the most common MIME types used by Servlets-:     |  |  | | --- | --- | | Type | Meaning | | application/msword | Microsoft Word document | | application/pdf | Acrobat (.pdf) file | | application/vnd.ms-excel | Excel spreadsheet | | application/vnd.ms-powerpoint | PowerPoint presentation | | application/x-gzip | Gzip archive | | image/jpeg | JPEG image | | text/css | HTML cascading style sheet | | text/html | HTML document | | video/mpeg | MPEG video clip |   **Example :** Building Excel Spreadsheets -:    Although Servlets usually generate HTML output, but this is not the only one. HTTP is fundamental to Servlets; HTML is not. Now, it is sometimes useful to generate  Microsoft Excel content so that users can save the results in a report and so that you can make use of the built-in formula support in Excel. Excel accepts input in at least three distinct formats: tab-separated data, HTML tables, and a native binary format.    The key in this example is to use the **Content-Type response**  **header** to tell the client that you are sending a spreadsheet. You use the shorthand  **setContentType** method to set the Content-Type header, and the MIME type for Excel spreadsheets is **application/vnd.ms-excel**. So, to generate Excel spreadsheets, just do:  response.setContentType("application/vnd.ms-excel");  PrintWriter out = response.getWriter();  Then, simply print some entries with tabs (\t in Java strings) in between. That’s it: no  DOCTYPE, no HEAD, no BODY: those are all HTML-specific things.  import java.io.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  /\*\* Servlet that creates Excel spreadsheet comparing  \* apples and oranges.  \*/  public class ApplesAndOranges extends HttpServlet {  public void doGet(HttpServletRequest request,  HttpServletResponse response)  throws ServletException, IOException {  **response.setContentType("application/vnd.ms-excel");**  PrintWriter out = response.getWriter();  out.println("\tQ1\tQ2\tQ3\tQ4\tTotal");  out.println("Apples\t78\t87\t92\t29\t=SUM(B2:E2)");  out.println("Oranges\t77\t86\t93\t30\t=SUM(B3:E3)");  }  }  The above example presents a simple servlet that builds an Excel spreadsheet that compares  apples and oranges. Note that =SUM(*col*:*col*) sums a range of columns in Excel.         * **Understand Deployment Descriptor ( web.xml )**   A Web application is a collection of servlets, JSP pages, static pages, classes, and other resources that can be packaged in a standard way and run on multiple containers from multiple vendors.   |  |  | | --- | --- | | **Application structure** |  |   A Web application exists in a structured hierarchy of directories, which is defined by the Java Servlet Specification. The root directory of the Web application contains all the public resources, such as images, HTML pages, and so on, stored directly or within subfolders.  A special directory called WEB-INF exists, which contains any files that are not publicly accessible to clients. (check in netbeans)  The WEB-INF directory is organized as follows:   * The /WEB-INF/web.xml deployment descriptor. * The /WEB-INF/classes/ directory for servlet and utility classes. The container makes these classes available to the Web application class loader. * The /WEB-INF/lib/ directory for JAR files. These files contain servlets, beans, and other utility classes useful to the Web application. The container adds all the JAR files from this directory to the Web application class path.  |  |  | | --- | --- | | **Deployment descriptor** |  |   The deployment descriptor must be a valid XML file, named web.xml, and placed in the WEB-INF subdirectory of the Web application. This file stores the configuration information of the Web application. The order in which the configuration elements must appear is important and is specified by the deployment descriptor DTD.  The root element of the deployment descriptor is the <web-app> element; all other elements are contained within it.   |  |  | | --- | --- | | **Specifying the servlet details** |  |   Each servlet is defined using a <servlet> element; it contains child elements that provide details about the servlet.  **<servlet-name>** : The servlet's unique name within the Web application is specified by the <servlet-name> element. The clients can access the servlet by specifying this name in the URL. It is possible to configure the same servlet class under different names.  **<servlet-class>** : The fully qualified class name used by the servlet container to instantiate the servlet is specified by the <servlet-class> element.  **<init-param>** : Each initialization parameter for a servlet is specified using an <init-param> element. It has two child elements -- <param-name> and <param-value> -- which give the name and value of the parameter. The value of the initialization parameter can be retrieved in the servlet code using the getInitParameter() method of the ServletConfig interface.  The following code demonstrates the use of the <servlet> element within the deployment descriptor:  <servlet>  <servlet-name> TestServlet </servlet-name>   <servlet-class> com.whiz.TestServlet </servlet-class>   <init-param>   <param-name>State</param-name>   <param-value>Hyderabad</param-value>   </init-param>   </servlet>  This code causes the servlet container to instantiate a servlet class com.whiz.TestServlet and associates it with the name TestServlet . It has one initialization parameter named "country," which has the value "India."   |  |  | | --- | --- | | **Servlet mappings** | page 6 of 8 |   In some cases, it might be required to map different URL patterns to the same servlet. For this, we use the <servlet-mapping> element.  The <servlet-mapping> element has two sub-elements: <servlet-name> and <url-pattern> . The <servlet-name> sub-element must match with one of the servlet names declared in the deployment descriptor. The <url-pattern> sub-element is the URL string to be mapped with the servlet.  **Using URL paths**  When a client request arrives for a particular servlet, the Web application that has the longest context path matching with the start of the request URL is chosen first. Then the requested servlet is chosen by the container by comparing the remaining part of the request URI with the mapped URLs. The mapping rules are as follows (the first successful match is taken):   1. If there is an exact match of the path of the request to the path of a servlet, that servlet is chosen. 2. The container will recursively try to match the longest path-prefix. This is done by stepping down the path tree a directory at a time, using the "/" character as a path separator. The longest match determines the servlet selected. 3. An extension is defined as the part of the last segment after the last "." character. If the last segment in the URL path contains an extension (for instance, .jsp), the servlet container will try to match a servlet that handles requests for the extension. 4. If none of the previous three rules results in a servlet match, the container will attempt to serve content appropriate for the resource requested. If a "default" servlet is defined for the application, it will be used.   Consider the following sets of servlet mappings in the deployment descriptor:  <servlet-mapping>   <servlet-name>servlet1</servlet-name>   <url-pattern>/my/test/\*</url-pattern>  </servlet-mapping>     <servlet-mapping>   <servlet-name>servlet2</servlet-name>   <url-pattern>/another </url-pattern>   </servlet-mapping>    <servlet-mapping>   <servlet-name>servlet3</servlet-name>   <url-pattern>\*.tst </url-pattern>   </servlet-mapping>  A string beginning with a "/" character and ending with a "/\*" postfix is used for path mapping. If the request path is /my/test/index.html, then servlet1 is invoked to handle the request. Here the match occurs as was described in step 2 above.  If the request path is /another, then servlet2 services the request. Here the matching occurs as was describe in step 1 above. But when the path is /another/file1.tst, servlet3 is chosen. This is because the URL mapping for servlet2 requires an exact match, which is not available, so the extension mapping as described above in step 3 is chosen.  However, if the request path is /my/test/new.tst, the request would be handled by servlet1 and not by servlet3 because a match occurs in step 2 itself.  import java.io.\*;  import java.net.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  public class Main extends HttpServlet {    protected void processRequest(HttpServletRequest request, HttpServletResponse response)  throws ServletException, IOException {  response.setContentType("text/html;charset=UTF-8");  PrintWriter out = response.getWriter();  out.println("<html>");  out.println("<head>");  out.println("<title>Servlet Main</title>");  out.println("</head>");  out.println("<body>");  out.println("<h>Servlet Main at " + request.getContextPath () + "</h3>");  out.println( "<br><a href='ContextDemo'> Context </a>" ) ;  out.println( "<br><a href='ConfigDemo'> Config </a>" ) ;  out.println("</body>");  out.println("</html>");  out.close();  }    // <editor-fold defaultstate="collapsed" desc="HttpServlet methods. Click on the + sign on the left to edit the code.">  /\*\* Handles the HTTP <code>GET</code> method.  \* @param request servlet request  \* @param response servlet response  \*/  protected void doGet(HttpServletRequest request, HttpServletResponse response)  throws ServletException, IOException {  processRequest(request, response);  }    /\*\* Handles the HTTP <code>POST</code> method.  \* @param request servlet request  \* @param response servlet response  \*/  protected void doPost(HttpServletRequest request, HttpServletResponse response)  throws ServletException, IOException {  processRequest(request, response);  }    /\*\* Returns a short description of the servlet.  \*/  public String getServletInfo() {  return "Short description";  }  // </editor-fold>  }   7. Servlets – JDBC & Session Tracking **OBJECTIVE:**   * Using Servlets with JDBC * Understand different ways for Session Tracking   **L7.1 Servlets using JDBC**   * + **L7.1.1 Access a DB from Servlets using JDBC**   Example shows a very simple servlet that uses the Oracle JDBC driver to perform a simple query, printing names and phone numbers for all employees listed in a database table. We assume that the database contains a table named EMPLOYEES, with at least two fields, NAME and PHONE.  /\* Example shows JDBC-enabled servlet \*/  import java.io.IOException;  import java.io.PrintWriter;  import javax.servlet.ServletException;  import javax.servlet.http.Cookie;  import javax.servlet.http.HttpServlet;  import javax.servlet.http.HttpServletRequest;  import javax.servlet.http.HttpServletResponse;  public class DBPhoneLookup extends HttpServlet  {    public void service(HttpServletRequest req, HttpServletResponse res)throws ServletException, IOException {  Connection con = null;  Statement stmt = null;  ResultSet rs = null;    res.setContentType("text/html");  PrintWriter out = res.getWriter();  out.println("<HTML><HEAD><TITLE>Phonebook</TITLE></HEAD>");  out.println("<BODY>");  try  {    // Load (and therefore register) the Oracle Driver  Class.forName("oracle.jdbc.driver.OracleDriver");    // Get a Connection to the database  con = DriverManager.getConnection("jdbc:oracle:thin:@hstslc007:1521:eltp","scott","tiger");    // Create a Statement object  stmt = con.createStatement();    // Execute an SQL query, get a ResultSet  rs = stmt.executeQuery("SELECT NAME, PHONE FROM EMPLOYEES");    // Display the result set as a list  out.println("<UL>");  while(rs.next() ) {  out.println("<LI>" + rs.getString("name") + " " + rs.getString("phone") );    }  out.println("</UL>");    }  catch(ClassNotFoundException e) {  out.println("Couldn't load database driver : " + e.getMessage() );  }  catch(SQLException e) {  out.println("SQLException caught: " + e.getMessage() );  }  finally {  // Always close the database connection.    try {  if (con != null)  con.close();  }  catch (SQLException ignored) { }    }  out.println("</BODY></HTML>");  }  }  In this example all DBPhoneLookup does is connect to the database, run a query that retrieves the names and phone numbers of everyone in the employees table, and display the list to the user.  **http://tbn0.google.com/images?q=tbn:ofDq306X8Kj0EM:http://www.nalsofmichigan.org/images/Great%2520Idea.jpg For more details refer “O’Reilly Java** **Servlet Programming”** byJason Hunter with William Crawford   * **L7.1.2 Validating** **Login Credentials with Database records** * Here is the sample servlet code for user authentication. * Assume the LoginBean class, it defines userId and password (has corresponding setter and getter methods). * Also, assume DBConnection class from ‘util’ package which returns Connection object.   /\* Import all necessary classes and interfaces \*/  public class LoginServlet extends HttpServlet  {  protected void doPost(HttpServletRequest request, HttpServletResponse response)  throws ServletException, IOException  {  HttpSession session=request.getSession(true);    String userid=(String)request.getParameter("userid");  String password=(String)request.getParameter("password");    LoginBean logBean = new LoginBean();  LoginServlet loginServlet = new LoginServlet();  RequestDispatcher dispatch;    logBean.setUserid(userid);  logBean.setPassword(password);    int login=loginServlet.getLogin(logBean);  if (login == 1)  {  session.setAttribute("UserId",userid);  dispatch = request.getRequestDispatcher("success.jsp");  dispatch.forward(request,response);  }  else  {  dispatch = request.getRequestDispatcher("invalidUser.jsp");  }  }    public int getLogin(LoginBean logBean)  {  Connection connection=null;  PreparedStatement pstmnt=null;  ResultSet resultset=null;    int flag=0;    String query="select userid,password from UserLogin where userid=?";  String user=logBean.getUserid();  String pwd=logBean.getPassword();  try  {  connection=DBConnection.getConnection();  pstmnt=connection.prepareStatement(query);  pstmnt.setString(1,user);  resultset=pstmnt.executeQuery();    while(resultset.next())  {  if(user!=null&&pwd!=null)  {  if(user.equals(resultset.getString("userid"))&&  pwd.equals(resultset.getString("password"))){  flag=1;  }  else{  flag=0;  }  }  }  }  catch(SQLException e){  e.printStackTrace();  }  finally{  try{  connection.close();  if(pstmnt!=null){  pstmnt.close();  }  }catch(Exception e){  e.printStackTrace();  }  }  return flag;  } // End of getLogin  }// End of class   * This sample code allows only to authenticated user to forward to success page. * LoginBean is a model to deal with database properties. * DBConnection is a class in user defined util package, in which database connection is created and used dynamically whenever needed. * In this example session is created for the user only if user is authenticated. * **L7.1.3 Updating Database Records**   Updating database records includes adding to database, modifying and deleting from database.  Example shows a servlet that maintains inventory detail. It assumes table in an Oracle database—INVENTORY (containing the product ID and amount in stock).  import java.io.\*;  import java.sql.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  public class DatabaseTransaction extends HttpServlet  {  public void service(HttpServletRequest req, HttpServletResponse res) throws ServletException,  IOException  {    res.setContentType("text/html");  PrintWriter out = res.getWriter();  Connection con = null;    out.println("<html><head><title>Product Database Record</title></head>");  out.println("<body>");  try  {    Class.forName("oracle.jdbc.driver.OracleDriver");  con = DriverManager.getConnection("jdbc:oracle:thin:@hstslc007:1521:eltp","scott","tiger");  // Turn on transactions  con.setAutoCommit(false);  Statement stmt = con.createStatement();    //Insert into database record  String insert\_query ="insert into inventory values('prd4',67)";  stmt.executeUpdate(insert\_query);  out.println("Product successfully added to database record !!");    // //Update databse records  // String update\_query ="update inventory set stock\_amount=(stock\_amount + 45) where  // prod\_id = 'prd1'";  // stmt.executeUpdate(update\_query);  // out.println("Product successfully updated to database record !!");    // // Delete database record  // String delete\_query="delete from inventory where prod\_id='pd24'";  // stmt.executeUpdate(delete\_query);  // out.println("Record successfully deleted from database record");    con.commit();    }  catch(ClassNotFoundException cf){  out.println("ClassNotFoundException caught: " + cf.getMessage() );  }  catch (SQLException e) {  out.println("SQLException caught: " + e.getMessage() );  }  finally  {  // Clean up.  try {  if (con != null) con.close();  } catch (SQLException sq) {  out.println("SQLException caught while closing connection : " + sq.getMessage() );    }  }  out.println("</body>");  out.println("</html>");  }  }  Here are a few notes on this example:   * First, the database record maintenance logic is in doPost() since the client's action is definitely not safely repeatable. * In doPost() method, three database transactions are performed   + Adding new records,   + Updating existing records &   + Deleting existing records. * Secondly, as the servlet runs, any exception thrown during driver initialization, connecting to the database or executing SQL causes execution to jump to the catch() block.   **L7.2 Session Tracking**  This section shows you how to use the servlet session tracking API to keep track of visitors as they move around at your site.   * **7.2.1 Introduction to Session Tracking**    + HTTP is a “stateless” protocol: each time a client retrieves a Web page, it opens a separate connection to the Web server, and the server does not automatically maintain contextual information about a client.   + This lack of client context causes a number of difficulties, like when clients at an on-line store add an item to their shopping carts, how does the server know what’s already in them?   + When and which client decides to proceed to checkout?   + So here the session comes in picture. Sessions are shared among the servlets accessed by a client. This is convenient for applications made up of multiple servlets.   + There are three typical ways to track the session:     - cookies,     - URL-rewriting and     - Hidden form fields. * **L7.2.2 Demo on Session Tracking**   + **Using Cookies**     - Cookies are a way for a server (or a servlet, as part of a server) to send some information to a client to store, and for the server to later retrieve its data from that client.     - Clients automatically return cookies by adding fields to HTTP request headers.     - Sample code to add cookies and getting them is shown below:   /\*  \* This Example shows two classes SetCookies & ViewCookies  \* First is class SetCookies, it sets new cookies to response object.  \* and dispatches the request response objects to the next servlet  \* /  import java.io.IOException;  import java.io.PrintWriter;  import javax.servlet.ServletException;  import javax.servlet.http.Cookie;  import javax.servlet.http.HttpServlet;  import javax.servlet.http.HttpServletRequest;  import javax.servlet.http.HttpServletResponse;  public class SetCookies extends HttpServlet  {  public void doGet(HttpServletRequest request,HttpServletResponse response)  throws ServletException, IOException  {    Cookie cookie1 = new Cookie("bookInfo123", "RustinQty1");  System.out.println(cookie1);  response.addCookie(cookie1);    Cookie cookie2 = new Cookie("bookInfo123", "Pages246");  System.out.println(cookie2);  response.addCookie(cookie2);    response.setContentType("text/html");  PrintWriter out = response.getWriter();    out.println("<html>");  out.println("<body>");    // request.getRequestDispatcher( "ViewCookies" ).forward(request,response) ;    out.print("<form action='ViewCookies' >") ;  out.print("<input type='submit'>") ;  out.print("</form>") ;    out.println("</html>");  out.println("</body>");  }  }  /\*  \* Class ViewCookies which receives request response objects from  \* SetCookies and displays its name & value.  \*/  public class ViewCookies extends HttpServlet {  protected void doGet(HttpServletRequest request, HttpServletResponse response)  throws ServletException, IOException {  response.setContentType("text/html;charset=UTF-8");  PrintWriter out = response.getWriter();    out.print("View Cookies<br>") ;    Cookie cookies[] = request.getCookies();    System.out.println(cookies);  System.out.println("Cookies obtained ");    if( cookies != null ) {  for(int i=0; i<cookies.length; i++) {  out.println("<br>Cookies :"+ (i+1));  out.println("<br>Name :" + cookies[i].getName());  out.println(" Value :" + cookies[i].getValue());  }  }  out.close();  }  }  **http://tbn0.google.com/images?q=tbn:ofDq306X8Kj0EM:http://www.nalsofmichigan.org/images/Great%2520Idea.jpghttp://www.iam.ubc.ca/guides/javatut99/servlets/TOC.html#client-state**   * + - A Servlet can send a cookie to a client by passing a Cookie object to the addCookie() method of HttpServletResponse:   Cookie cookie1 = new Cookie("bookInfo123", "RustinQty1");  response.addCookie(cookie1);   * + - A servlet retrieves cookies by calling the getCookies() method of HttpServlet-Request which returns the array of Cookie objects:   Cookie cookies[] = request.getCookies();  if( cookies != null ) {  for(int i=0; i<cookies.length; i++) {  out.println("<br>Cookies :"+ (i+1));  out.println("<br>Name :" + cookies[i].getName());  out.println(" Value :" + cookies[i].getValue());  }   * + - You can set number of attributes for cookies in addition to its name & value.   Here are some attribute setting methods of Cookie.   * + - public void Cookie.setVersion(int v)   Sets the version of a cookie. Servlets can send and receive cookies formatted to match either Netscape persistent cookies (Version 0) or the newer, somewhat experimental, RFC 2109 cookies (Version 1). Newly constructed cookies default to Version 0 to maximize interoperability.   * + - public void Cookie.setDomain(String pattern)   Specifies a domain restriction pattern. A domain pattern specifies the servers that should see a cookie. By default, cookies are returned only to the host that saved them.   * + - public void Cookie.setMaxAge(int expiry)   Specifies the maximum age of the cookie in seconds before it expires. A negative value indicates the default, that the cookie should expire when the browser exits. A zero value tells the browser to delete the cookie immediately.   * + - public void Cookie.setPath(String uri)   Specifies a path for the cookie, which is the subset of URIs to which a cookie should be sent.   * + - public void Cookie.setSecure(boolean flag)   Indicates whether the cookie should be sent only over a secure channel, such as SSL. By default, its value is false.   * + - public void Cookie.setComment(String comment)   Sets the comment field of the cookie. A comment describes the intended purpose of a cookie.   * + - public void Cookie.setValue(String newValue)   Assigns new value to a cookie. With Version 0 cookies, values should not contain the following: whitespace, brackets and parentheses, equals signs, commas, double quotes, slashes, question marks, at signs, colons, and semicolons.   * **Using URL Rewriting**   + Concept of URL rewriting is based on the attaching a unique ID (session ID which is generated by the server).   + With this approach, the client appends some extra data on the end of each URL that identifies the session, and the server associates that identifier with data it has stored about that session.   For example, with http://host/path/file.html;jsessionid=1234, the session information is attached as jsessionid=1234.   * + This is also an excellent solution, and even has the advantage that it works when browsers don’t support cookies or when the user has disabled them. * **Hidden Form Fields**   + One way to support anonymous session tracking is to use hidden form fields.   + As the name implies, these are fields added to an HTML form that are not displayed in the client's browser.   + They are sent back to the server when the form that contains them is submitted. We can include hidden form fields like this:   <FORM ACTION="/servlet/SampleCart" METHOD="POST">  …  <INPUT TYPE=hidden NAME="pin" VALUE="70676501">  <INPUT TYPE=hidden NAME="level" VALUE="expert">  …  </FORM>  **http://tbn0.google.com/images?q=tbn:ofDq306X8Kj0EM:http://www.nalsofmichigan.org/images/Great%2520Idea.jpgFor more detail understandings refer O.Reilly Java Servlet Programming: Unit 7, Session Tracking**   * **L7.2.3 Understanding the session-tracking API**   The Session Tracking API, as we call the portion of the Servlet API devoted to session tracking.   * **Session-Tracking Basics**    + Every user of a site is associated with a javax.servlet.http.HttpSession object that servlets can use to store or retrieve information about that user.   + A servlet uses request object's getSession() method to retrieve the current HttpSession object:   HttpSession session = request.getSession (true);  This method returns the current session associated with request making user. If user do not have any valid session, this method creates one if *create* is *true* or returns *null* if *create* is *false*.  Here is a summary of other methods available in the HttpSession class.     * **public Object getAttribute(String name)**   These methods extract a previously stored value from a session object. They return null if there is no value associated with the given name.   * **public void setAttribute(String name, Object value)**   These methods associate a value with a name. If the object supplied to setAttribute implements the HttpSessionBinding- Listener interface, the object’s valueBound method is called after it is stored in the session. Similarly, if the previous value implements HttpSessionBindingListener, its valueUnbound method is called.   * **public void removeAttribute(String name)**   These methods remove any values associated with the designated name. If the value being removed implements HttpSessionBindingListener, its valueUnbound method is called.   * **public Enumeration getAttributeNames()**   These methods return the names of all attributes in the session.   * **public String getId()**   This method returns the unique identifier generated for each session. It is sometimes used as the key name when only a single value is associated with a session, or when information about sessions is being logged.   * **public boolean isNew()**   This method returns *true* if the client (browser) has never seen the session, usually because it was just created rather than being referenced by an incoming client request. It returns *false* for preexisting sessions.   * **public long getCreationTime()**   This method returns the time in milliseconds since midnight, January 1, 1970 (GMT) at which the session was first built. To get a value useful for printing out, pass the value to the Date constructor or the setTimeIn-Millis method of GregorianCalendar.   * **public long getLastAccessedTime()**   This method returns the time in milliseconds since midnight, January 1, 1970 (GMT) at which the session was last sent from the client.   * **public int getMaxInactiveInterval()**   **public void setMaxInactiveInterval(int seconds)**  These methods get or set the amount of time, in seconds, that a session should go without access before being automatically invalidated. A negative value indicates that the session should never time out. Note that the time out is maintained on the server and is *not* the same as the cookie expiration date, which is sent to the client.   * **public void invalidate()**   This method invalidates the session and unbinds all objects associated with it.   * **7.2.4 Examples**   **Tracking User Access Counts**  /\*  \* Servlet Example to track user access count  \*/  import java.io.IOException;  import javax.servlet.ServletException;  import javax.servlet.http.HttpServlet;  import javax.servlet.http.HttpServletRequest;  import javax.servlet.http.HttpServletResponse;  import javax.servlet.http.HttpSession;  public class TrackAccessCount extends HttpServlet  {  protected void processRequest(HttpServletRequest request, HttpServletResponse response)  throws ServletException, IOException  {  HttpSession session = request.getSession(true);    String heading;  Integer accessCount =(Integer)session.getAttribute("accessCount");  if (accessCount == null)  {  accessCount = new Integer(1);  heading = "Welcome, Newcomer";  }  else  {  heading = "Welcome Back";  accessCount = new Integer(accessCount.intValue() + 1);  }    session.setAttribute("accessCount", accessCount);  session.setAttribute("heading",heading);    request.getRequestDispatcher("TrackAccessCount.jsp").forward(request,response);  }  public void doGet(HttpServletRequest request,HttpServletResponse response)  throws ServletException, IOException  {  processRequest(request, response);  }  }  Get the values of accessCount and heading in .jsp page as shown below:  <html>  <head>  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  <title>User Access Count</title>  </head>  <body bgcolor="#FDF5E6">  <center>  <h1>Access User Count</h1>  <%= session.getAttribute("heading")%>  <br>  Access Count <%= session.getAttribute("accessCount") %>  </center>  </body>  </html>  Output for above sample code:    **Figure 1: First visit to page**    **Figure 2: 24th visit to same page**  **Exercise**  Develop a program for Shopping Cart in Online Book Store. This online book store has different categories of books. User can view book information after selecting it from list of specific category of books. Keep track of each user using Session Tracking. Maintain separate cart for each user and at the end display bill for the items selected in cart. User should be authenticated to the store. 8.JSP Introduction and Scripting Elements  * **Introduction :**     You can take any existing HTML page and change its extension to “.jsp” instead of “.html”.  In fact, this is the perfect exercise for your first JSP. For example see the code below:  <HTML>  <BODY>  Hello Mr. John, What is the time now?  </BODY>  </HTML>  Take the HTML file and run it.  You used “.html” extension for this file to run. Right **?** Now change its extension from . “.html” to “.jsp”. Now load the new file, with the “.jsp” extension, in your browser.  You will see the same output, but it will take longer! But only the first time.  If you . reload it again, it will load normally.   * **Understanding the needs for JSP :**   Of course, it is not very useful to just write HTML pages with a “.jsp” . extension!  We now proceed to see what makes JSP so useful.   * What makes JSP useful is the ability to embed Java. * It is possible to use JSP to generate dynamic HTML pages that change in response to user actions or vary from user to user.     Put the following text in a file with .jsp extension, place it in your JSP directory, and view it in a browser.  <HTML>  <BODY>  Hello! The time is <%= new java.util.Date() %>  </BODY>  </HTML>  Notice that -:   * Each time you reload the page in the browser, it comes up with the current time. . * The character sequences <%= and %> enclose Java expressions, which are evaluated at run time.   Thus we can say that JavaServerPages (JSP) lets you separate the dynamic part of your pages from the static HTML. You simply write the regular HTML in the normal manner, using whatever Web-page-building tools you normally use. You then enclose the code for the dynamic parts in special tags, most of which start with "<%" and end with "%>".   * **JSP Life Cycle** :     JavaServerPages are made operable by having their contents (HTML tags,  JSP tags and scripts) translated into a servlet by the application server. This process  is responsible for translating both the dynamic and static elements.  Because JSPs are server-side technology, the processing of both the static  and dynamic elements of the page occurs in the server. The architecture of a  JSP/servlet-enabled Web site is often referred to as *thin-client* because most of the  business logic is executed on the server.  The following process outlines the tasks performed on a JSP file on the *first*  *invocation* of the file or when the underlying JSP file is changed by the developer     * The Web browser makes a request to the JSP page. * The JSP engine parses the contents of the JSP file.      * The JSP engine creates temporary servlet source code based on the contents of the JSP. The generated servlet is responsible for rendering the static elements of the JSP specified at design time in addition to creating the dynamic elements of the page. * The servlet source code is compiled by the Java compiler into a servlet class file. * The servlet is instantiated. The *init* and *service* methods of the servlet are called, and the servlet logic is executed. * The combination of static HTML and graphics combined with the dynamic elements specified in the original JSP page definition are sent to the Web   browser through the output stream of the servlet’s response object.  **JSP Servlet**  **JSP Parser**  **Java Compiler**  **Java Source**  **Web Page (HTML)**  **JSP Source**  **Web**  **Browser**  **Request**  **Result**  **Web Server**    The JSP processing life-cycle on first-time invocation  Subsequent invocations of the JSP file will simply invoke the *service* method  of the servlet created by the above process to serve the content to the Web browser. The servlet produced as a result of the above process remains in service until the application server is stopped, the servlet is manually unloaded, or a change is made to the underlying file, causing recompilation.   * **Static vs. dynamic text** : There are two types of data in a JSP page: * **Template Data**: The static part, anything that will be copied directly to the response by the JSP server (can be a simple HTML). * **JSP Elements**: The dynamic part, anything that will be translated and executed by the JSP server. There are three types of JSP tags:  1. Scripting Elements 2. Directives 3. Actions   In this chapter we will discuss Scripting elements only.     1. **Scripting Elements:** The simplest method of making a JSP dynamic is by directly   embedding bits of Java code between blocks of template text by use of scripting elements. We can classify scripting elements as follows:   1. Scriptlet 2. Expression 3. Declaration     **Scriptlet:** Scriptlets are great for providing low level functionality such as iteration,  loops, and conditional statements, but they also provide a method for embedding  complex chunks of code within a JSP.   * Scriptlets provide a method for directly inserting bits of Java code between   chunks of template text.   * A scriptlet is defined with a start ,<%, an end, %>, with code between. * Using Java, the script is identical to normal Java code but without needing   a class declaration or any methods.  The example below provides a simple JSP that loops to produce multiple lines of text.  Looping is accomplished the same as in Java but by placing the equivalent Java cod  inside scriptlet elements.  **Example :** Loop.jsp  <html>  <head>  <title>Loop Example</title>  </head>  <body>  <% for (int i=0; i<5;i++) { %>  Repeated 5 Times.<br>  <% } %>  </body>  </html>      **II. Expression:** Expressions provide an easy method of sending out dynamic strings to a client.   * An expression must have a start, <%=, end, %>, and an expression between. * An expression element differs in syntax from a scriptlet by an equal sign that must appear immediately after the start. * Expressions always send a string of text to a client, but the object produced as a result of an expression does not have to always end up as an instance of a String object. * Any object left as the result of an expression automatically has its toString() method called to determine the value of the expression.   <html>  <head>  <title>Loop Example</title>  </head>    <body>  <%="Hello World!" %> //Sciptlet tag  </body>    </html    Thus we can see that Expression woks like “println” also.  **III. Declaration :** Declarations are the third and final scripting element available for use in JSP.   * A declaration is used like a scriptlet to embed code in a JSP, but code embedded by a declaration appears outside of the \_jspService() method. * For this reason code embedded in a declaration can be used to declare new methods and global class variables * But caution should be taken because code in a declaration is not thread-safe, unless made so by the writer of that code.   <%@page contentType="text/html" %>  <html>  <body>  <%!  int cnt=0;  private int getCount(){  //increment cnt and return the value  cnt++;  return cnt;  }  %>  <p>Values of Cnt are:</p>  <p><%=getCount()%></p>  <p><%=getCount()%></p>  <p><%=getCount()%></p>  <p><%=getCount()%></p>  <p><%=getCount()%></p>  <p><%=getCount()%></p>  </body>  </html     * **Predefined Variables** :     When you wrote a doGet method for a servlet, you probably wrote something  like this:  public void doGet(HttpServletRequest **request**,  HttpServletResponse **response**)  throws ServletException, IOException {  response.setContentType("text/html");  HttpSession **session** = request.getSession();  PrintWriter **out** = response.getWriter();  out.println(...);  ...  }  The servlet API told you the types of the arguments to doGet, the methods to call  to get the session and writer objects, and their types. JSP changes the method name  from doGet to \_jspService and uses a JspWriter instead of a PrintWriter. But the idea is the same.  For JSP expressions and scriptlets to be useful, you need to know what variable names the autogenerated servlet uses. You are supplied with eight automatically defined local variables in \_jspService, sometimes called “implicit objects.” Nothing is special about these; they are merely the names of the local variables.    The available variables are request, response, out, session, application,  config, pageContext and page. Details for each are given below. An additional variable  called exception is available, but only in error pages.     * **Request:** This variable is the HttpServletRequest associated with the request; it gives you access to the request parameters, the request type   (e.g., GET or POST), and the incoming HTTP headers (e.g., cookies). |  |  |  |  |

# 9.JSP DIRECTIVES AND JSP ACTIONS

OBJECTIVES

* Understanding the *page* directive
* Participating in sessions
* Designating pages to process JSP errors
* Understanding the JSP Actions
* Understanding the jsp:include, jsp:forward and jsp:param actions.

C9.1 **Understanding the *page* directive**

* A JSP *directive* affects the overall structure of the servlet that results

from the JSP page.In JSP,there are three types of directives: page, include, taglib.

* The page directive lets you control the structure of the servlet by importing classes, customizing the servlet superclass, setting the content type, and so on.
* The page directive applies to an entire JSP page and any of its *static include files*, which together are called a *translation unit*. A static include file is a file whose content becomes part of the calling JSP page. You can use the page directive more than once in a translation unit.

The page directive lets you define one or more of the following case-sensitive attributes: import, contentType, isThreadSafe, session, buffer,autoflush, extends, info, errorPage, isErrorPage, and language.

|  |  |
| --- | --- |
| Attributes | Description |
| Import | Results in a Java import statement being inserted into the resulting file.  Eg:- <%@ page import="java.util.\*" %> <%-- example import --%> |
| contentType | specifies the content that is generated. This should be used if HTML is not used or if the character set is not the default [character set](http://en.wikipedia.org/wiki/Character_set).  Eg:- <%@ page contentType="text/html" %> <%-- example contentType--%> |
| errorPage | Indicates the page that will be shown if an exception occurs while processing the HTTP request.  Eg:- <%@ page errorPage="*Relative URL*" %> |
| isErrorPage | If set to true, it indicates that this is the error page. Default value is *false*.  Eg:- <%@ page isErrorPage="false" %> <%!-- Default --%> |
| isThreadSafe | It controls whether or not the servlet that results from the JSP page will implement the SingleThreadModel interface.  Eg:- <%@ page isThreadSafe="true" %> <%-- example for a thread safe JSP --%> |
| Session | The session attribute controls whether or not the page participates in  HTTP sessions.  Eg:- <%@ page session="true" %> <%-- Default --%> |
| Buffer | The buffer attribute specifies the size of the buffer used by the out variable,  which is of type JspWriter (a subclass of PrintWriter).  Eg:- <%@ page buffer="20kb" %> <%-- example for setting Buffer Size --%> |
| autoFlush | The autoflush attribute controls whether the output buffer should be automatically flushed when it is full or whether an exception should be raised  when the buffer overflows.  Eg:- <%@ page autoFlush="true" %> <%-- example for setting autoFlush --%> |
| extends | It indicates the superclass of the servlet that will be generated for the JSP page  Eg:- <%@ page extends="*package.class*" %> |

C9.2 **Participating in sessions**

The session attribute controls whether or not the page participates in

HTTP sessions.

The attribute takes one of the following two forms:

1. <%@ page session="true" %> <%-- Default --%>

A value of true (the default) indicates that the predefined variable session

(of type HttpSession) should be bound to the existing session if one

exists; otherwise, a new session should be created and bound to session.

2) <%@ page session="false" %>

A value of false means that no sessions will be used automatically and

attempts to access the variable session will result in errors at the time the

JSP page is translated into a servlet.

**Example**

The example consists of four pages that have been provided for understanding the session attribute. These are

* sessionForm.jsp
* session.jsp
* sessionResult.jsp
* sessionFailure.jsp

The sessionForm.jsp includes two text fields UserName and Password.It gets the value of the two fields and sets the session objects (username and password) with the retrieved values and shows the session.jsp page with Welcome message with the username.

The session.jsp page requires the username and password fields to be entered.If the information is provided the page displays a Welcome message along with two links *Next Page with session true* and *Next Page with session false*.

|  |
| --- |
| **session.jsp**  <%@page language="java" %>  <%  String userName = request.getParameter("txtUserName");  String password = request.getParameter("txtPassword");  if(userName == null)  userName = "";  if(password == null)  password = "";  if(!userName.equals("") && !password.equals("")){  session.setAttribute("SessionUser", userName);  session.setAttribute("SessionPassword", password);  out.println("Welcome " + userName + "!");  out.println("<br/><a href=sessionresult.jsp>  Next Page with session true.</a>");  out.println("<br/><a href=ShowFalseSession.jsp>  Next Page with session false.</a>");  }  else if(userName.equals("")){  out.println("<font color=red><b>User name required.</b></font>");  out.println("<br/><a href=sessionForm.jsp>Go back!</a>");  }  else if(password.equals("")){  out.println("<font color=red><b>Password required.</b></font>");  out.println("<br/><a href=sessionForm.jsp>Go back!</a>");  }  %> |

The sessionResult.jsp page contains the page directive in which attribute **session** has been set to *true* value for the session operation. But when you click on the another link ("Next Page with session false.") made on the session.jsp page then the sessionFailure.jsp page will be seen with the message "The value of session object is false!" because when the value of the session attribute of the page directive is false then the page does not support any type of session operations in the page.

|  |
| --- |
| **sessionResult.jsp**  <%@page language="java" session="true" %>  <%  String username = (String)session.getAttribute("SessionUser");  String password = (String)session.getAttribute("SessionPassword");  out.println("<b>Welcome " + username + "!</b>");  %> |

|  |
| --- |
| **sessionFailure.jsp**  <%@page language="java" session="false" %>  <%  out.println("The value of session attribute is false!");  %> |

C9.4 **Designating pages to process JSP errors**

* Few things make an application look less refined and professional than a server's default exception page. Even the most well-designed page usually shows the stack trace and exception name, and makes your application look broken.
* The JSP error page lets you catch, present, and report exceptions gracefully, instead of presenting the user with a technical, and possibly confusing, stack trace.

**The isErrorPage Attribute**

The isErrorPage attribute indicates whether or not the current page can act as the error page for another JSP page. Use of isErrorPage takes one of the following two forms:

<%@ page isErrorPage="true" %>

<%@ page isErrorPage="false" %> <%!-- Default --%>

**Example**

The example contains two files ErrorPageExample.jsp and error.jsp. The ErrorPageExample.jsp page consists of a simple text box which accepts an integer value and a button is used to submit the form.The entered number is divided by zero causing an error and the page is redirected to the specified file which is mentioned as the value of the errorPage attribute.This page shows the error generated in the calling JSP page when the value of the **isErrorPage** attribute is set to *true* value.

|  |
| --- |
| **ErrorPageExample.jsp**  <%@page errorPage="error.jsp" %>  <html>  <head><title>Showing Error Page.</title></head>  <body>  <form method="post">  <table border="0" cellspacing="0" cellpadding="0">  <tr>  <td>Enter a number: </td>  <td><input type="text" name="number" />  </tr>  <tr>  <td>&nbsp;</td>  <td><input type="submit" name="B1" value="Divide by zero" />  </tr>  </table>  </form>  <%  if(request.getParameter("number") != null){  if(!request.getParameter("number").equals("")){  int div = Integer.parse Int(request.getParameter("number")) / 0;  out.println("Answer is: " + div);  }  else{  out.println("<html><font color=r  ed>Please enter a number.</font></html>");  }  }  %>  </body>  </html>  **Error.jsp** |
| <%@page isErrorPage="true" %>  <html>  <head><title>Error Page.</title></head>  <font size="14" color="white">Your page generated an error:"<br/>  Exception:<br/></font>  <%= exception.toString() %>  </body>  </html> |

**JSP Actions**

JSP *actions* use constructs in XML syntax to control the behavior of the servlet engine. You can dynamically insert a file, reuse JavaBeans components, forward the user to another page, or generate HTML for the Java plugin.

**The jsp:include**

The Java servlet temporarily hands the request and response off to the specified JavaServer Page. Control will then return to the current JSP, once the other JSP has finished. Using this, JSP code will be shared between multiple other JSPs, rather than duplicated.

|  |
| --- |
| <html>  <head></head>  <body>  <jsp:include page="mycommon.jsp" >  <jsp:param name="extraparam" value="myvalue" />  </jsp:include>  name:<%=request.getParameter("extraparam")%>  </body>  </html> |

**Understanding why jsp:include is usually better than the include directive:-**

jsp:include is simply a different directive from include. The advantage of jsp:include is that it will *always* check for changes in the included file. We'll take a look at the code for each of the two includes, so that you can see the similarities and differences between them.

**The JSP include directive:-**

<%@ page language="java" contentType="text/html" %>

<html>

<head>

<title>JSP include element test</title>

</head>

<body>

This content is statically in the main JSP file.<br />

<%@ include file="included.html" %>

</body>

</html>

The below code is the same page converted to use the jsp:include tag.

**Converting to jsp:include:-**

<%@ page language="java" contentType="text/html" %>

<html>

<head>

<title>JSP include element test</title>

</head>

<body>

This content is statically in the main JSP file.<br />

<jsp:include page="included.html" flush="true" />

</body>

</html>

There are two big differences between the two code types.

**Firstly** the jsp:include element doesn't use the %@ syntax that is part of the include directive. Instead, the jsp prefix lets the JSP compiler know that it should look for the element in the standard JSP set of tags.

**Secondly** the attribute that specifies the file to include has changed from file to page. If you like, you can test the results of the new tag for yourself. Simply change the content of your own included.html file from the last installment, reload your browser page, and you'll see the new content immediately.

**How jsp:include works:-**

* It includes the *response* from the included URI, rather than the URI itself. This means that the indicated URI is *interpreted* and the *resulting response* is included.
* If the page is HTML, you get the HTML in effect unchanged. But if it's a Perl script, a Java servlet, or a CGI program, you'll get the interpreted result from that program.
* And, as the interpretation happens at every page request, the results are never cached as they were with the include directive. It's such a minor change, but it makes all the difference in the behavior you see.

**Syntax:-**

<jsp:include page="{ relativeURL | <%= expression %> }" flush="true | false" />

Or:

<jsp:include page="{relativeURL | <%= expression %>}" flush="true| false" >

<jsp:param name="parameterName"

value="{parameterValue | <%= expression %>}" />

</jsp:include>

**Note:-**

**The flush attribute:-**

flush indicates whether any existing buffer should be flushed before reading in the included content. The flush attribute is required in JSP 1.1, so you'll get an error if you leave it out of your code. In JSP 1.2, however, the flush attribute defaults to false. Because flushing is rarely a big concern, my advice is to leave flush set to true for JSP 1.1 and leave it off for JSP 1.2 and above.

**/\* Example of an jsp include directive:-**

<html>

<head>

<title>newInstance.com</title>

<meta http-equiv="Content-Type"

content="text/html; charset=iso-8859-1" />

<link href="/styles/default.css"

rel="stylesheet" type="text/css" />

</head>

<body>

<%@ include file="header.jsp" %>

<%@ include file="navigation.jsp" %>

<%@ include file="bookshelf.jsp" %>

<%@ include file="/mt-blogs/index.jsp" %>

<%@ include file="footer.jsp" %>

</body>

</html> \*/

**The jsp:forward**

Used to hand off the request and response to another JSP or servlet. Control will never return to the current JSP.

**How to use jsp:forward tag :-**

* The <jsp:forward> element forwards the request object containing the client request information from one JSP file to another file.
* The target file can be an HTML file, another JSP file, or a servlet, as long as it is in the same application context as the forwarding JSP file. The lines in the source JSP file the <jsp:forward> element are not processed.

**Syntax:-**

<jsp:forward page="{relativeURL | <%= expression %>}" />

Or:

<jsp:forward page="{relativeURL | <%= expression %>}" >

<jsp:param name="parameterName"

value="{parameterValue | <%= expression %>}" /> +

</jsp:forward>

**Example:-**

This sample code shows the use of tag. This checks the percentage of free memory and based on that opens new page using this tag.

|  |  |  |  |
| --- | --- | --- | --- |
| **DemoForward.jsp**   |  | | --- | | <html> <%   double freeMemory = Runtime.getRuntime().freeMemory();   double totalMemory = Runtime.getRuntime().totalMemory();   double percent = freeMemory/totalMemory;   if(percent<0.5){ %>   <jsp:forward page="one.jsp"/> <%}else{%> <jsp:forward page="two.html"/> <%}%> </html> |   **one.jsp**   |  | | --- | | <html> <body> <font color=”red”> VM Memory usage<50% </html> |   **two.html**   |  | | --- | | <html> <body bgcolor= “white”> <font color=”red”> VM Memory usage>50% </body> <html> | |

**The jsp:param tag**

**How to use jsp:param Tag:-**

* The <jsp:param> tag is used to provide key/value information.
* The <jsp:param> tag contains two attributes: 1) name 2) value.The name attribute specifies the parameter name and takes a string literal as a value. These parameters are evaluated at request time.
* This tag is used in the jsp:include, jsp:forward and jsp:params tags. A translation error occurs if the element is used elsewhere.
* When executing jsp:include or jsp:forward, the included page or forwarded page sees the original request object, with the original parameters augmented with the new parameters, with new values taking precedence over existing values when applicable.
* The scope of the new parameters is the jsp:include or jsp:forward call; that is, in the case of an jsp:include the new parameters (and values) will not apply after the include.

**Syntax:-**

<jsp:param name="parameterName"

value="{parameterValue | <%= expression %>}" />

**Examples:-**

* In the following example the <jsp:forward> tag is enclosing this tag.
* We can use multiple <jsp: param> tag inside the <jsp: forward> tag if we have to pass more than one parameter to the target file.
* In this example we are passing the name and values to the targeted file. These parameters will be retrieved by the targeted file by using request.getParameter() method. In this way we can pass and retrieve the parameters.

|  |
| --- |
| **ParamExample.jsp**  <html>  <head>  <title></title>  </head>  <body>  <jsp: forward page="Parameters.jsp">  <jsp: param name="myParam" value="Amar Patel"/>  <jsp: param name="Age" value="15"/>  </jsp: forward>  </body>  </html> |

|  |
| --- |
| **Parameters.jsp**  This page had a parameter forwarded to it:<br>  <b>Name:</b> <%= request.getParameter("myParam") %><br>  <b>Age:</b> <%= request.getParameter("Age") %> |

# SOLUTIONS

**UNIT2:Exercise**

Code Listing 1

Error:

java.sql.SQLException: [Microsoft][ODBC Driver Manager] Data source name not found and no default driver specified

Explanation:

When we don’t have a dsn defined then we have this kind of error.

Fix:

Create the DSN (data source name)

Code Listing 2

Error:

unreported exception java.lang.ClassNotFoundException; must be caught or declared to be thrown

unreported exception java.sql.SQLException; must be caught or declared to be thrown

Explanation:

When we dont surround the "creation of connection" by a try-catch block, then we get the above error**.**

Fix:

Add throws clause for java sql exception or surround by try-catch block.

Code Listing 3

Error:

java.sql.SQLException: [Oracle][ODBC][Ora]ORA-00936: missing expression

Explanation:-

Such error can arise when we do not declare enough values to be inserted in the database table, i.e. some arguments are missing.

Fix:-

Define enough values in the query or specify the columns to be inserted.

Code Listing 4

Error:-

java.sql.SQLException: [Oracle][ODBC][Ora]ORA-00942: table or view does not exist.

Explanation:-

Such error occurs when the table does not exist in the database.

Fix:-

Proper table name should be mentioned

Code Listing 5

Error:-

## java.sql.SQLException: [Oracle][ODBC][Ora]ORA-01756: quoted string not properly terminated.

Explanation:-

Such error occurs when the string argument in a statement query is not enclosed in single quotes. Since the query is converted to SQL query it has to be in the same format.

Fix:-

All string argument in double quotes has to be enclosed in single quotes.

Code Listing 6

Error:-

## java.sql.SQLException: [Oracle][ODBC]Invalid transaction state.

Explanation:-

Closing the connection before commit and roleback is the only problem to cause this error.

Fix:-

Execute a connection.commit statement before closing.

**UNIT3**:**Exercise**

Code Listing 1

Error:-

## java.sql.SQLException: [Oracle][ODBC][Ora]ORA-00913: too many values.

Explanation:-

Such error occurs when the arguments given in the insert query are more than the number of columns in the table.

Fix:-

Make sure the number of arguments matches the number of columns.

Code Listing 2

Error:-

## java.sql.SQLException: [Oracle][ODBC][Ora]ORA-01722: invalid number

Explanation:-

Such error occurs when the type of arguments in the insert query does not match with the type of the column in the table.

Fix:-

Make sure the type of arguments matches the type of columns.

Code Listing 3

Error:-

## unexpected error EXCEPTION\_ACCESS\_VIOLATION

Explanation:-

## When we close the connection and then try using it or don’t create a connection then we may get this error

Fix:-

Do not close the connection until you have finished using it. After closing the connection it must not be used or called.

Code Listing 4

Error:-

## java.sql.SQLException: [Oracle][ODBC]Invalid column number <7>.

Explanation:-

## When we try to access an index of a table which does not exist then we get such a kind of error.

Fix:-

Whenever we chose a particular index it should not be invalid or out of bound.

Code Listing 5

Error:-

## sun.jdbc.odbc.JdbcOdbcBatchUpdateException: [Oracle][ODBC][Ora]ORA-00001: unique constraint (SCOTT.SYS\_C0060855) violated

Explanation:-

## When we try to update a particular value in a table which is already defined as primary key then we get such kind of exception

Fix:-

While inserting values into table care should be taken that none of the values inserted for the column defined as primary key are redundant.

Code Listing 6

Errors:-

## java.sql.SQLException: [Oracle][ODBC][Ora]ORA-06550: wrong number or types of arguments in call to 'MYPROCIN'

## java.sql.SQLException: [Oracle][ODBC][Ora]ORA-06550: object SCOTT.MYPROCOUT is invalid

## java.sql.SQLException: [Oracle][ODBC][Ora]ORA-06550: object SCOTT.MYPROCINOUT is invalid

## Variable outParam might not have been initialized

Explanation:-

## When we try to call a particular procedure in oracle with wrong type of arguments as parameters then we get such kind of exception

Fix:-

Call the procedure with the right type of parameters or arguments

**UNIT4**:**Exercise**

Code Listing 1

Error:

## java.sql.SQLException: [Microsoft][ODBC Driver Manager] Invalid cursor state

Explanation:

## Since the cursor is at the last position it can't go to the next position

Fix:

# Instead of using .next() method use .previous()

Code Listing 2

Error:

## java.sql.SQLException: Cursor position (0) is invalid

Explanation:

## As the exception says the cursor position is invalid.

Fix:

# use a valid cursor position

Code Listing 3

Error:

## java.sql.SQLException: Column not found

Explanation:-

## As the exception says the column is invalid.

Fix:-

Use a valid column name as per the table created

Code Listing 4

Error:-

## java.sql.SQLException: Invalid Cursor Type: 1003

Explanation:-

## As the exception says the column is invalid.

Fix:-

Use a valid column name as per the table created

Code Listing 5

Error:-

## Exception in thread "main" java.lang.NullPointerException

Explanation:-

## A NullPointerException (NPE) is thrown when your code tries to access a member (either method or variable) of an Object that has not been instantiated

Fix:-

choose the exact row where to update or insert the values

**UNIT5**:**Exercise**

Code Listing 1

Error:

## javax.servlet.ServletException: Class com.NewServlet is not a Servlet

Explanation:

## When we dont extend the created class with either HttpServlet or GenericServlet then we come across such a kind of error

Fix:

## Extend the servlet class with either GenericServlet or HttpServlet to make it a servlet.

Code Listing 2

Error:

## unreported exception java.io.IOException; must be caught or declared to be thrown

Explanation:

This error will occur when we do not include IO Exception in throws. The Printwriter throws IO exception so it must be caught.

Fix:

The method, be it service, get or post must use throws ServletException and IOException.

Code Listing 3

Error:

## HTTP Status 405 - HTTP method GET is not supported by this URL

Explanation:-

## The specified HTTP method is not allowed for the requested resource (HTTP method GET is not supported by this URL) as in the html file we have specified the GET method but we are using the POST method in the servlet

Fix:-

Change both of the methods used to the same type ie if you are using post method use it both the places or if using get method then both should use the same.

Code Listing 4

Error:-

## There is no error. But There is an BUG cause this application will always cause the Login failure to get executed

Explanation:-

## As we have used the SERVICE method in the Servlet, but the method used in the Html is POST so the values are not passed into the getparametter hence the login is always a failure.

Fix:-

Change the Servlet Method to doPost or remove the dopost method in html file.

**UNIT6**:**Exercise**

Code Listing 1

Error:

## HTTP Status 404 - Login

## The requested resource is not available.

Explanation:

## The 404 or Not Found error message is an HTTP standard response code indicating that the

## client was able to communicate with the server but either the server could 0not find what

## was requested, or it was configured not to fulfill the request and not reveal the reason

## why. 404 errors should not be confused with "server not found" or similar errors, in which

## a connection to the destination server cannot be made at all.

Fix:

# Check the deployment descriptor if the server mapping is missing then it may give rise

Code Listing 2

Error:

java.lang.ClassNotFoundException: Login

javax.servlet.ServletException: Wrapper cannot find servlet class Login or a class it depends on.

Explanation:

The server encountered an internal error () that prevented it from fulfilling this request.

The server is aware that it has erred or is incapable of performing the request.

Fix:

According to the exception found check the error. here we have a class not found exception that means the class is not present or the full path of the class is either not defined.

so it can be fixed by either of the ways.