1. **Java Collections:**

import java.util.ArrayList;

import java.util.Collections;

public class TestSorting {

public ArrayList<Integer> createArray(){

ArrayList<Integer> arr = new ArrayList<Integer>();

arr.add(8);

arr.add(4);

arr.add(3);

arr.add(2);

arr.add(5);

return arr;

}

public void displayArray(ArrayList<Integer> arr){

for(int i=0; i<arr.size(); i++){

System.out.print(arr.get(i)+",");

}

}

public static void main(String[] args) {

// TODO Auto-generated method stub

TestSorting obj1 = new TestSorting();

ArrayList<Integer> arr = obj1.createArray();

obj1.displayArray(arr);

Collections.sort(arr);

System.out.println("after sorting");

obj1.displayArray(arr);

}

}

The above code is of java collections, in which an array list of integers is created and a number of integers are added in the array list. The array list is then sorted and displayed

1. **Threads:**
2. **Using Thread class:**

public class TryThread extends Thread {

public TryThread(String firstName, String secondName, long delay) {

this.firstName = firstName; // Store the first name

this.secondName = secondName; // Store the second name

aWhile = delay; // Store the delay

setDaemon(true); // Thread is daemon

}

public static void main(String[] args) {

// Create three threads

Thread first = new TryThread("Hopalong ", "Cassidy ", 200L);

Thread second = new TryThread("Marilyn ", "Monroe ", 300L);

Thread third = new TryThread("Slim ", "Pickens ", 500L);

System.out.println("Press Enter when you have had enough...\n");

first.start(); // Start the first thread

second.start(); // Start the second thread

third.start(); // Start the third thread

try {

System.in.read(); // Wait until Enter key pressed

System.out.println("Enter pressed...\n");

} catch (IOException e) { // Handle IO exception

System.out.println(e); // Output the exception

}

System.out.println("Ending main()");

return;

}

// Method where thread execution will start

public void run() {

try {

while(true) { // Loop indefinitely...

System.out.print(firstName); // Output first name

sleep(aWhile); // Wait aWhile msec.

System.out.print(secondName + "\n"); // Output second name

}

} catch(InterruptedException e) { // Handle thread interruption

System.out.println(firstName + secondName + e); // Output the exception

}

}

private String firstName; // Store for first name

private String secondName; // Store for second name

private long aWhile; // Delay in milliseconds

}

In the above code the thread is created by inheriting the thread class of java, try thread is a user defined class containing first name, last name and delay. Three instances of this thread are created in the main.

1. **Using Runnable:**

import java.io.IOException;

public class JumbleNames implements Runnable {

// Constructor

public JumbleNames(String firstName, String secondName, long delay) {

this.firstName = firstName; // Store the first name

this.secondName = secondName; // Store the second name

aWhile = delay; // Store the delay

}

// Method where thread execution will start

public void run() {

try {

while(true) { // Loop indefinitely...

System.out.print(firstName); // Output first name

Thread.sleep(aWhile); // Wait aWhile msec.

System.out.print(secondName+"\n"); // Output second name

}

} catch(InterruptedException e) { // Handle thread interruption

System.out.println(firstName + secondName + e); // Output the exception

}

}

public static void main(String[] args) {

// Create three threads

Thread first = new Thread(new JumbleNames("Hopalong ", "Cassidy ", 200L));

Thread second = new Thread(new JumbleNames("Marilyn ", "Monroe ", 300L));

Thread third = new Thread(new JumbleNames("Slim ", "Pickens ", 500L));

// Set threads as daemon

first.setDaemon(true);

second.setDaemon(true);

third.setDaemon(true);

System.out.println("Press Enter when you have had enough...\n");

first.start(); // Start the first thread

second.start(); // Start the second thread

third.start(); // Start the third thread

try {

System.in.read(); // Wait until Enter key pressed

System.out.println("Enter pressed...\n");

} catch (IOException e) { // Handle IO exception

System.out.println(e); // Output the exception

}

System.out.println("Ending main()");

return;

}

private String firstName; // Store for first name

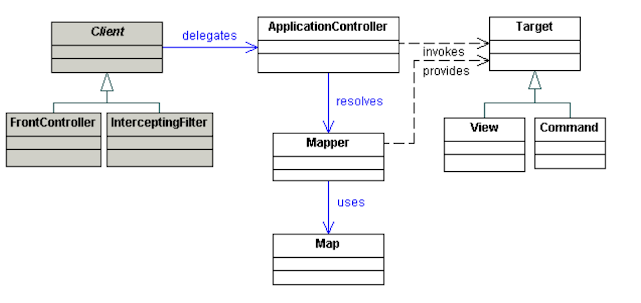
private String secondName; // Store for second name

private long aWhile; // Delay in milliseconds

}

In the above code the thread is created by implementing the runnable interface of java using the second method for thread creation, JumbleNames is a user defined class containing first name, last name and delay. Three instances of this thread are created in the main.

1. **Application Controller Design Pattern:**



The class diagram of Application controller pattern is shown above. The purpose of this pattern is to provide a centralized mechanism for the retrieval and invocation of the request processing components such as commands and views

1. **MVC:**

**//Model**

public class Student {

private String rollNo;

private String name;

public String getRollNo() {

return rollNo;

}

public void setRollNo(String rollNo) {

this.rollNo = rollNo;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

**//View**

public class StudentView {

public void printStudentDetails(String studentName, String studentRollNo){

System.out.println("Student: ");

System.out.println("Name: " + studentName);

System.out.println("Roll No: " + studentRollNo);

}

}

**//Controller**

public class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view){

this.model = model;

this.view = view;

}

public void setStudentName(String name){

model.setName(name);

}

public String getStudentName(){

return model.getName();

}

public void setStudentRollNo(String rollNo){

model.setRollNo(rollNo);

}

public String getStudentRollNo(){

return model.getRollNo();

}

public void updateView(){

view.printStudentDetails(model.getName(), model.getRollNo());

}

}

In the above code the MVC (Model View and Controller) framework of java has been shown. The code has three independent components, in which the view has the interface to communicate with the end user, the model communicates with the database and the controller gets the requests from views and pass them to controller and vice versa.

1. **Hibernate:**

**//Employee class**

public class Employee {

private int id;

private String firstName, lastName;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

}

**//StoreData class**

import java.util.List;

import org.hibernate.Query;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

public class StoreData {

public static void main(String[] args) {

// creating configuration object

Configuration cfg = new Configuration();

cfg.configure("hibernate.cfg.xml");// populates the data of the

// configuration file

// creating seession factory object

SessionFactory factory = cfg.buildSessionFactory();

// creating session object

Session session = factory.openSession();

// creating transaction object

Transaction t = session.beginTransaction();

Employee e1 = new Employee();

//e1.setId(1000);

e1.setFirstName("Irfan");

e1.setLastName("Younas");

session.persist(e1);// persisting the object

Employee emp;

Query q = session.createQuery("FROM com.nu.Employee");

List records = q.list();

emp = (Employee) session.get(Employee.class, new Integer(1));

System.out.println(emp.getFirstName());

t.commit();// transaction is committed

session.close();

System.out.println("successfully saved");

}

}

**//hibernate mapping**

<hibernate-mapping>

<class name="com.nu.Employee" table="employee8">

<id name="id">

<generator class="increment"></generator>

</id>

<property name="firstName"></property>

<property name="lastName"></property>

</class>

</hibernate-mapping>

In the above code the hibernate framework of java is implemented, in which an employee class has been created and store data class is created to store the data and the mapping is done using xml file as shown in hibernate mapping. The employee class contains the id, first and last name of employee.

1. **JSON:**

Owner = {

"name":"Ali",

"age":30,

"cars": {

"car1":"Honda",

"car2":"Corolla",

"car3":"cultus"

}

}

The above code shows a nested JSON object the cars object is encapsulated in the owner object.

1. **Android http url connection:**

URL url;

HttpURLConnection urlConnection = null;

try {

url = new URL("http://www.mysite.se/index.asp?data=99");

urlConnection = (HttpURLConnection) url

.openConnection();

InputStream in = urlConnection.getInputStream();

InputStreamReader isw = new InputStreamReader(in);

int data = isw.read();

while (data != -1) {

char current = (char) data;

data = isw.read();

System.out.print(current);

}

} catch (Exception e) {

e.printStackTrace();

} finally {

if (urlConnection != null) {

urlConnection.disconnect();

}

}

The code above shows the http url connection for android. In this the connection is opened using the url of a particular web page to get data, after the connection is established the data is read using the input reader object and used as desired. The connection is closed after getting the required data

1. **Servlet:**

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

/\*\*

\* Servlet implementation class MyServlet

\*/

@WebServlet("/MyServlet")

public class MyServlet extends HttpServlet {

private static final long serialVersionUID = 1L;

/\*\*

\* @see HttpServlet#HttpServlet()

\*/

public MyServlet() {

super();

// TODO Auto-generated constructor stub

}

/\*\*

\* @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)

\*/

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

// TODO Auto-generated method stub

// Set response content type

String message = "Welcome to the Servlet";

response.setContentType("text/html");

// Actual logic goes here.

PrintWriter out = response.getWriter();

out.println("<h1>" + message + "</h1>");

}

/\*\*

\* @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse response)

\*/

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

// TODO Auto-generated method stub

String message = "Servlet POST";

response.setContentType("text/html");

// Actual logic goes here.

PrintWriter out = response.getWriter();

out.println("<h1>" + message + "</h1>");

}

}

In the above code a java servlet is written named My Servlet. The doGet() and doPost() methods of the servlet are implemented, the message is printed as heading 1 in both of the methods

1. **Junit Testing:**

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

public class MyTests {

@Test

public void multiplicationOfZeroIntegersShouldReturnZero() {

MyClass tester = new MyClass(); // MyClass is tested

// assert statements

assertEquals(0, tester.multiply(10, 0), "10 x 0 must be 0");

assertEquals(0, tester.multiply(0, 10), "0 x 10 must be 0");

assertEquals(0, tester.multiply(0, 0), "0 x 0 must be 0");

}

}

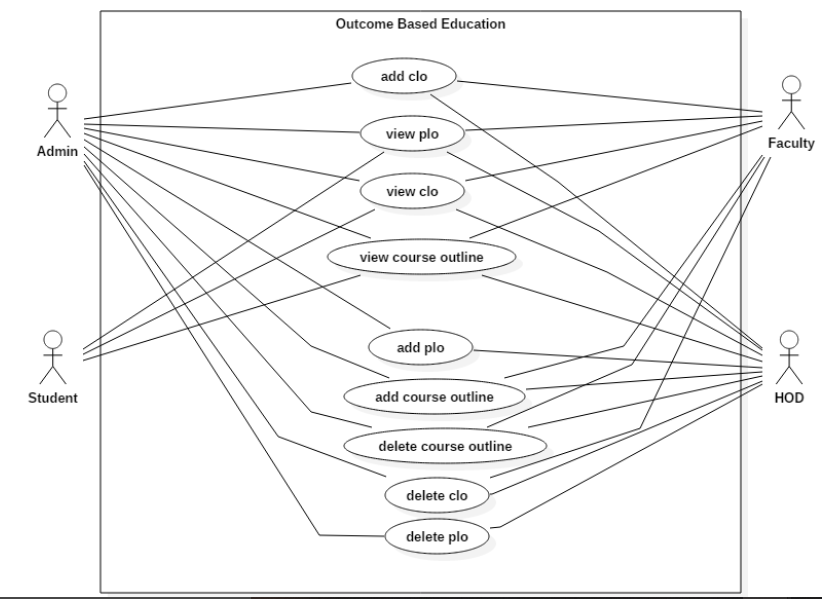
The above code demonstrates Junit testing in this code it tests MyClass for its multiply method that has two parameters of type integer.

1. **System Testing:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| # |  |  |  |  |  |
| 1 |  |  |  |  |  |
| 2 |  |  |  |  |  |
| 3 |  |  |  |  |  |
| 4 |  |  |  |  |  |
| 5 |  |  |  |  |  |
| 6 |  |  |  |  |  |
| 7 |  |  |  |  |  |
| 8 |  |  |  |  |  |
| 9 |  |  |  |  |  |
| 10 |  |  |  |  |  |

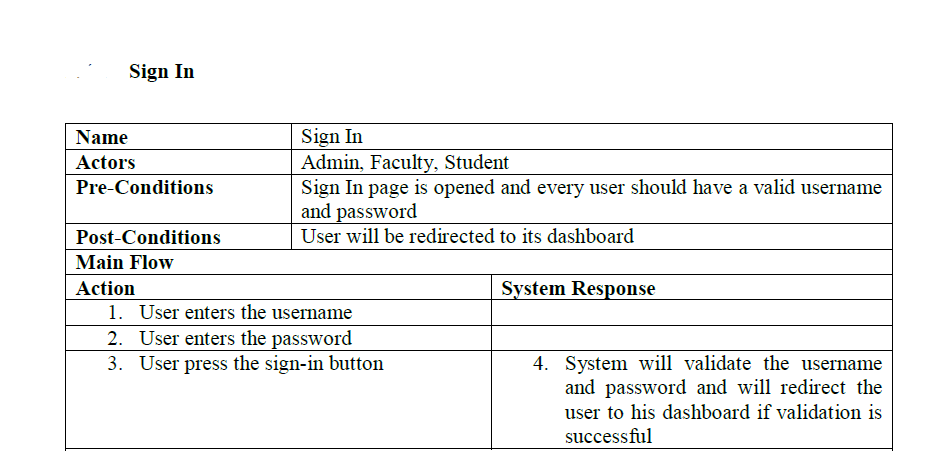
System testing is done to test the integrated system that whether it works as desired as an integrated system. This testing is done after the unit testing in which the individual units are tested for their functionality. System testing ensures that the individual units will not deviate from their functionality when they are integrated with other units and the system will perform in the right way after the overall integration.

1. **Use case Diagram:**

****

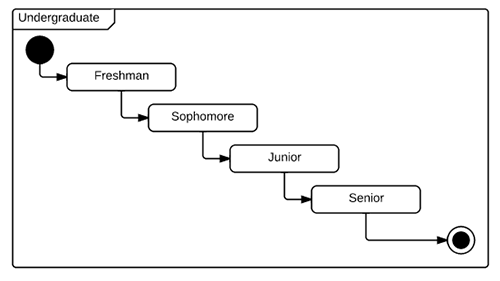
Use case diagram of an application to implement Outcome Based Education is shown above. In this diagram the use case of various actors of the system I.e. Student, faculty, admin and HOD are presented in the rectangle with each actor pointing to his use case.

1. **Use case Document:**

****

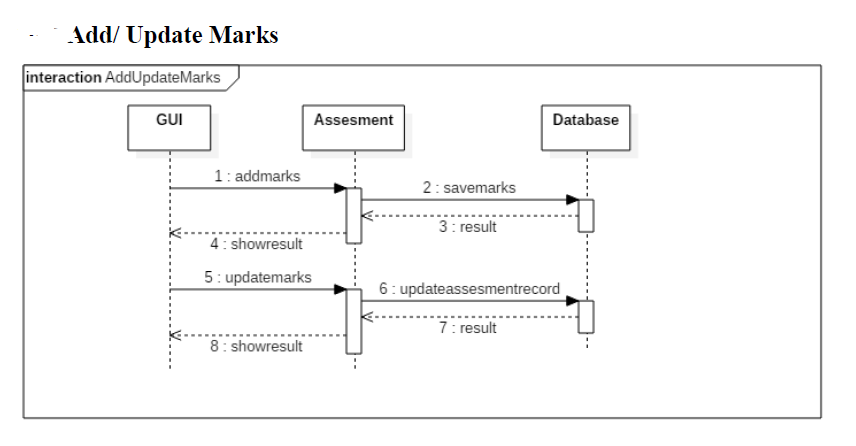
The sign in use case document has been shown above in which the user possible actions for the sign in has been listed and the corresponding system response against each action is also listed.

1. **State Diagram:**

****

The diagram above shows the different states of an undergraduate student. The filled black circle represents the start of state diagram and the black circle with the token marks the end of state diagram the intermediate states of the student are shown in the diagram

1. **Sequence Diagram:**

****

The sequence diagram of Add marks use case is shown above in which it depicts the entire process of adding marks. In this the user communicates with the GUI to add marks after he enters the marks, they are passed to the assessment class object and then recorded in the database for persistent storage.