Internal Assessment III – Nov 2019

Scheme and Solutions

Sub:	Advanced Java & J2EE					Sub Code:	17CS553	Branch:	ISE	
Date:	18-11-2019	Duration:	90 min's	Max Marks:	50	Sem / Sec:	V			OBE

Q.1 Explain the following legacy classes with an example i) Hashtable ii) Vector

Java.util.Vector Class in Java

The Vector class implements a growable array of objects. Vectors basically fall in legacy classes but now it is fully compatible with collections.

- Vector implements a dynamic array that means it can grow or shrink as required. Like an array, it contains components that can be accessed using an integer index
- They are very similar to ArrayList but Vector is synchronised and have some legacy method which collection framework does not contain.

Constructor:

- **Vector()**: Creates a default vector of initial capacity is 10.
- **Vector(int size):** Creates a vector whose initial capacity is specified by size.
- **Vector(int size, int incr):** Creates a vector whose initial capacity is specified by size and increment is specified by incr. It specifies the number of elements to allocate each time that a vector is resized upward.
- **Vector(Collection c):** Creates a vector that contains the elements of collection c.

```
// Demonstrate various Vector operations.
import java.util.*;

class VectorDemo {
  public static void main(String args[]) {

    // initial size is 3, increment is 2
```

-

```
    Vector<Integer> v = new Vector<Integer>(3, 2);

    System.out.println("Initial size: " + v.size());
    System.out.println("Initial capacity: " +
                       v.capacity());

    v.addElement(1);
    v.addElement(2);
    v.addElement(3);
    v.addElement(4);

    System.out.println("Capacity after four additions: " +
                       v.capacity());
    v.addElement(5);
    System.out.println("Current capacity: " +
                       v.capacity());

    Sys
  }
}
```

The output from this program is shown here:

```
Initial size: 0
Initial capacity: 3
Capacity after four additions: 5
Current capacity: 5
```

Hash Table

Hashtable

Hashtable was part of the original **java.util** and is a concrete implementation of a **Dictionary**. However, with the advent of collections, **Hashtable** was reengineered to also implement the **Map** interface. Thus, **Hashtable** is now integrated into the Collections Framework. It is similar to **HashMap**, but is synchronized.

The **Hashtable** constructors are shown here:

Hashtable( )
Hashtable(int *size*)
Hashtable(int *size*, float *fillRatio*)
Hashtable(Map<? extends K, ? extends V> *m*)

```
// Demonstrate a Hashtable.
import java.util.*;

class HTDemo {
  public static void main(String args[]) {
    Hashtable<String, Double> balance =
      new Hashtable<String, Double>();

    Enumeration<String> names;
    String str;
    double bal;

    balance.put("John Doe", 3434.34);
    balance.put("Tom Smith", 123.22);
    balance.put("Jane Baker", 1378.00);
    balance.put("Tod Hall", 99.22);
    balance.put("Ralph Smith", -19.08);

    // Show all balances in hashtable.
    names = balance.keys();


  while(names.hasMoreElements()) {
    str = names.nextElement();
    System.out.println(str + ": " +
                       balance.get(str));
  }


  }
}
```

Q. 2 Demonstrate the linked list for the collections with an example.

The LinkedList Class

The **LinkedList** class extends **AbstractSequentialList** and implements the **List**, **Deque**, and **Queue** interfaces. It provides a linked-list data structure. **LinkedList** is a generic class that has this declaration:

```
class LinkedList<E>
```

Here, E specifies the type of objects that the list will hold. **LinkedList** has the two constructors shown here:

```
LinkedList( )
LinkedList(Collection<? extends E> c)
```

```
// Demonstrate LinkedList.
import java.util.*;

class LinkedListDemo {
  public static void main(String args[]) {
    // Create a linked list.
    LinkedList<String> ll = new LinkedList<String>();

    // Add elements to the linked list.
    ll.add("F");
    ll.add("B");
    ll.add("D");
    ll.add("E");
    ll.add("C");
    ll.addLast("Z");
    ll.addFirst("A");

    ll.add(1, "A2");

    System.out.println("Original contents of ll: " + ll);

    // Remove elements from the linked list.
    ll.remove("F");
    ll.remove(2);

    System.out.println("Contents of ll after deletion: "
                       + ll);

    // Remove first and last elements.
    ll.removeFirst();
    ll.removeLast();

    System.out.println("ll after deleting first and last: "
                       + ll);

    // Get and set a value.




     String val = ll.get(2);
     ll.set(2, val + " Changed");

     System.out.println("ll after change: " + ll);
  }
}
```

The output from this program is shown here:

```
Original contents of ll: [A, A2, F, B, D, E, C, Z]
Contents of ll after deletion: [A, A2, D, E, C, Z]
ll after deleting first and last: [A2, D, E, C]
ll after change: [A2, D, E Changed, C]
```

Q. 3 Explain how collections can be accessed using an Iterator.

Accessing a Collection via an Iterator

Often, you will want to cycle through the elements in a collection. For example, you might want to display each element. One way to do this is to employ an *iterator*, which is an object that implements either the **Iterator** or the **ListIterator** interface. **Iterator** enables you to cycle through a collection, obtaining or removing elements. **ListIterator** extends **Iterator** to allow

```
// Demonstrate iterators.
import java.util.*;

class IteratorDemo {
  public static void main(String args[]) {
    // Create an array list.
    ArrayList<String> al = new ArrayList<String>();

    // Add elements to the array list.
    al.add("C");
    al.add("A");
    al.add("E");
    al.add("B");
    al.add("D");
    al.add("F");

    // Use iterator to display contents of al.
    System.out.print("Original contents of al: ");
    Iterator<String> itr = al.iterator();
    while(itr.hasNext()) {
      String element = itr.next();
      System.out.print(element + " ");
    }
    System.out.println();

    // Modify objects being iterated.
    ListIterator<String> litr = al.listIterator();
    while(litr.hasNext()) {
      String element = litr.next();
      litr.set(element + "+");
    }

    System.out.print("Modified contents of al: ");
    itr = al.iterator();
    while(itr.hasNext()) {
```

```
      String element = itr.next();
      System.out.print(element + " ");
    }
    System.out.println();

    // Now, display the list backwards.
    System.out.print("Modified list backwards: ");
    while(litr.hasPrevious()) {
      String element = litr.previous();
      System.out.print(element + " ");
    }
    System.out.println();
  }
}
```

The output is shown here:

```
Original contents of al: C A E B D F
Modified contents of al: C+ A+ E+ B+ D+ F+
Modified list backwards: F+ D+ B+ E+ A+ C+
```

Q. 4 Explain the Constructors of TreeSet class and write a Java program to create a TreeSet Collection and access it via Iterator.

The TreeSet Class

TreeSet extends **AbstractSet** and implements the **NavigableSet** interface. It creates a collection that uses a tree for storage. Objects are stored in sorted, ascending order. Access and retrieval times are quite fast, which makes **TreeSet** an excellent choice when storing large amounts of sorted information that must be found quickly.

TreeSet is a generic class that has this declaration:

class TreeSet<E>

Here, E specifies the type of objects that the set will hold.

TreeSet has the following constructors:

TreeSet( )
TreeSet(Collection<? extends E> *c*)
TreeSet(Comparator<? super E> *comp*)
TreeSet(SortedSet<E> *ss*)

The first form constructs an empty tree set that will be sorted in ascending order according to the natural order of its elements. The second form builds a tree set that contains the elements of *c*. The third form constructs an empty tree set that will be sorted according to the comparator specified by *comp*. (Comparators are described later in this chapter.) The fourth form builds a tree set that contains the elements of *ss*.

```
import java.util.*;
import java.util.TreeSet;
```

```
public class TreeSetDemo {
   public static void main(String args[])
   {
      // Creating an empty TreeSet
      TreeSet<String> set = new TreeSet<String>();

      // Use add() method to add elements into the Set
      set.add("Welcome");
      set.add("To");
      set.add("Geeks");
      set.add("4");
      set.add("Geeks");

      // Displaying the TreeSet
      System.out.println("TreeSet: " + set);

      // Creating an iterator
      Iterator value = set.iterator();

      // Displaying the values after iterating through the set
      System.out.println("The iterator values are: ");
      while (value.hasNext()) {
         System.out.println(value.next());
      }
   }
}
import java.util.*;
import java.util.TreeSet;

public class TreeSetDemo {
   public static void main(String args[])
   {
      // Creating an empty TreeSet
      TreeSet<String> set = new TreeSet<String>();

      // Use add() method to add elements into the Set
      set.add("Welcome");
      set.add("To");
      set.add("Geeks");
      set.add("4");
      set.add("Geeks");

      // Displaying the TreeSet
      System.out.println("TreeSet: " + set);
```

```
        // Creating an iterator
        Iterator value = set.iterator();

        // Displaying the values after iterating through the set
        System.out.println("The iterator values are: ");
        while (value.hasNext()) {
            System.out.println(value.next());
        }
    }
}
```

Output:
TreeSet: [4, Geeks, To, Welcome]
The iterator values are:
4
Geeks
To
Welcome

Q. 5 a) What is Servlet? Explain the lifecycle of a servlet

The Life Cycle of a Servlet

Three methods are central to the life cycle of a servlet. These are **init( )**, **service( )**, and **destroy( )**. They are implemented by every servlet and are invoked at specific times by the server. Let us consider a typical user scenario to understand when these methods are called.

First, assume that a user enters a Uniform Resource Locator (URL) to a web browser. The browser then generates an HTTP request for this URL. This request is then sent to the appropriate server.

Second, this HTTP request is received by the web server. The server maps this request to a particular servlet. The servlet is dynamically retrieved and loaded into the address space of the server.

Third, the server invokes the **init( )** method of the servlet. This method is invoked only when the servlet is first loaded into memory. It is possible to pass initialization parameters to the servlet so it may configure itself.

Fourth, the server invokes the **service( )** method of the servlet. This method is called to process the HTTP request. You will see that it is possible for the servlet to read data that has been provided in the HTTP request. It may also formulate an HTTP response for the client.

The servlet remains in the server's address space and is available to process any other HTTP requests received from clients. The **service( )** method is called for each HTTP request.

Finally, the server may decide to unload the servlet from its memory. The algorithms by which this determination is made are specific to each server. The server calls the **destroy( )** method to relinquish any resources such as file handles that are allocated for the servlet. Important data may be saved to a persistent store. The memory allocated for the servlet and its objects can then be garbage collected.

Q. 5 b) Write a short note on HTTP request and response.

Handling HTTP Requests and Responses

The **HttpServlet** class provides specialized methods that handle the various types of HTTP requests. A servlet developer typically overrides one of these methods. These methods are **doDelete( )**, **doGet( )**, **doHead( )**, **doOptions( )**, **doPost( )**, **doPut( )**, and **doTrace( )**. A complete description of the different types of HTTP requests is beyond the scope of this book. However, the GET and POST requests are commonly used when handling form input. Therefore, this section presents examples of these cases.

Handling HTTP GET Requests

Here we will develop a servlet that handles an HTTP GET request. The servlet is invoked when a form on a web page is submitted. The example contains two files. A web page is defined in **ColorGet.htm**, and a servlet is defined in **ColorGetServlet.java**. The HTML source code for **ColorGet.htm** is shown in the following listing. It defines a form that contains a select element and a submit button. Notice that the action parameter of the form tag specifies a URL. The URL identifies a servlet to process the HTTP GET request.

```
<html>
<body>
<center>
<form name="Form1"
  action="http://localhost:8080/servlets-examples/servlet/ColorGetServlet">
<B>Color:</B>
<select name="color" size="1">
```

```
<option value="Red">Red</option>
<option value="Green">Green</option>
<option value="Blue">Blue</option>
</select>
<br><br>
<input type=submit value="Submit">
</form>
</body>
</html>
```

The source code for **ColorGetServlet.java** is shown in the following listing. The **doGet( )** method is overridden to process any HTTP GET requests that are sent to this servlet. It uses the **getParameter( )** method of **HttpServletRequest** to obtain the selection that was made by the user. A response is then formulated.

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

public class ColorGetServlet extends HttpServlet {

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

    String color = request.getParameter("color");
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.println("<B>The selected color is:  ");
    pw.println(color);
    pw.close();
  }
}
```

Q. 6 What is Cookie? List out methods defined by cookie. Write a Java program to add a cookie

Cookies are small files which are stored on a user's computer. They are designed to hold a modest amount of data specific to a particular client and website, and can be accessed either by the web server or the client computer. This allows the server to deliver a page tailored to a particular user, or the page itself can contain some script which is aware of the data in the cookie and so is able to carry information from one visit to the website (or related site) to the next.

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

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

public String getValue()	Returns the value of the cookie.
public void setName(String name)	changes the name of the cookie.
public void setValue(String value)	changes the value of the cookie.

Other methods required for using Cookies

For adding cookie or getting the value from the cookie, we need some methods provided by other interfaces. They are:

1. **public void addCookie(Cookie ck):**method of HttpServletResponse interface is used to add cookie in response object.
2. **public Cookie[] getCookies():**method of HttpServletRequest interface is used to return all the cookies from the browser.

The HTML source code for **AddCookie.htm** is shown in the following listing. This page contains a text field in which a value can be entered. There is also a submit button on the page. When this button is pressed, the value in the text field is sent to **AddCookieServlet** via an HTTP POST request.

```
<html>
<body>
<center>
<form name="Form1"
  method="post"
  action="http://localhost:8080/servlets-examples/servlet/AddCookieServlet">
<B>Enter a value for MyCookie:</B>
<input type=textbox name="data" size=25 value="">
<input type=submit value="Submit">
</form>
</body>
</html>
```

The source code for **AddCookieServlet.java** is shown in the following listing. It gets the value of the parameter named "data". It then creates a **Cookie** object that has the name "MyCookie" and contains the value of the "data" parameter. The cookie is then added to the header of the HTTP response via the **addCookie( )** method. A feedback message is then written to the browser.

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

public class AddCookieServlet extends HttpServlet {

  public void doPost(HttpServletRequest request,
```

```
    HttpServletResponse response)
  throws ServletException, IOException {

    // Get parameter from HTTP request.
    String data = request.getParameter("data");

    // Create cookie.
    Cookie cookie = new Cookie("MyCookie", data);

    // Add cookie to HTTP response.
    response.addCookie(cookie);

    // Write output to browser.
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.println("<B>MyCookie has been set to");
    pw.println(data);
    pw.close();
  }
}
```

Q. 7 Demonstrate how servlet can accept parameters from HTML

Reading Servlet Parameters

The **ServletRequest** interface includes methods that allow you to read the names and values of parameters that are included in a client request. We will develop a servlet that illustrates their use. The example contains two files. A web page is defined in **PostParameters.htm**, and a servlet is defined in **PostParametersServlet.java**.

The HTML source code for **PostParameters.htm** is shown in the following listing. It defines a table that contains two labels and two text fields. One of the labels is Employee and the other is Phone. There is also a submit button. Notice that the action parameter of the form tag specifies a URL. The URL identifies the servlet to process the HTTP POST request.

```
<html>
<body>
<center>
<form name="Form1"
  method="post"
  action="http://localhost:8080/servlets-examples/
          servlet/PostParametersServlet">
<table>
<tr>
  <td><B>Employee</td>
  <td><input type=textbox name="e" size="25" value=""></td>
</tr>
<tr>
  <td><B>Phone</td>
  <td><input type=textbox name="p" size="25" value=""></td>
</tr>
</table>
```

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

The source code for **PostParametersServlet.java** is shown in the following listing. T **service( )** method is overridden to process client requests. The **getParameterNames( )** me returns an enumeration of the parameter names. These are processed in a loop. You can that the parameter name and value are output to the client. The parameter value is obta via the **getParameter( )** method.

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

public class PostParametersServlet
extends GenericServlet {

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

    // Get print writer.
    PrintWriter pw = response.getWriter();

    // Get enumeration of parameter names.
    Enumeration e = request.getParameterNames();

    // Display parameter names and values.
    while(e.hasMoreElements()) {
      String pname = (String)e.nextElement();
      pw.print(pname + " = ");
      String pvalue = request.getParameter(pname);
      pw.println(pvalue);
    }
    pw.close();
  }
}
```

Q. 8 Define JSP. Explain different types of JSP tags by taking suitable example

Java Server Pages (JSP) is a server-side programming technology that enables the creation of dynamic, platform-independent method for building Web-based applications. JSP have access to the entire family of Java APIs, including the JDBC API to access enterprise databases

Tags

1. 1. Directives

These types of tags are used primarily to import packages. Altenatively you can also use these tags to define error handling pages and for session information of JSP page.

1.Code:
<%@page language="java" %>
2.Code:
<%@page language="java" session="true" %>

2. Declarations

JSP declarations starts with '<%!' and ends with '%>'. In this you can make declarions such as int i = 1, double pi = 3.1415 etc. If needed, you can also write Java code inside declarations.

Example 1

Code:
<%!

int radius = 7;
double pi = 3.1415;

%>

3. Scriptlets

JSP Scriptlets starts with '<%' and ends with '%>'. This is where the important Java code for JSP page is written.

Example

Code:
<%
String id, name, dob, email, address;

id = request.getParameter("id");
name = request.getParameter("name");
dob = request.getParameter("dob");

```
    email = request.getParameter("email");
    address = request.getParameter("address");

            sessionEJB.addClient(id, name, dob, email, address);
 %>
```

request, response, session and out are the variables available in scriptlets.

4. Expressions

JSP expressions starts with '<%=' and ends with '%>'. If you want to show some value, you need to put it in between these tags.

Example:

Code:

```
<%!

double radius = 7;
double pi = 22/7;

double area()
{
   return pi*radius*radius;
}

%>

<html>
  <body>
            Area of circle is <%= area() %>
 </body>
</html>
```

5.JSP Comments

JSP comment marks text or statements that the JSP container should ignore. A JSP comment is useful when you want to hide or "comment out", a part of your JSP page.

Following is the syntax of the JSP comments −

```
<%-- This is JSP comment --%>
```

Following example shows the JSP Comments −

```
<html>
  <head><title>A Comment Test</title></head>

  <body>
    <h2>A Test of Comments</h2>
    <%-- This comment will not be visible in the page source --%>
  </body>
</html>
```