USN | | | | | | | | | |

Internal Assessment Test III – Nov. 2017

Sub:	Advanced Java and J2EE					Sub Code:	15CS553	Branch:	ISE
Date:	9/11/2017	Duration:	90 min's	Max Marks:	50	Sem / Sec:	V /A		

1. What is a servlet? Describe the life cycle of a servlet. (10 Marks)

The life cycle of a servlet involves three methods. These are **init( )**, **service( )**, and **destroy( )**. They are implemented by every servlet and are invoked at specific times by the server.
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.

2. What is the use of Get and Post? Explain with a program how do you handle Get requests using servlets. (10 Marks)

The GET and POST requests are commonly used when handling form input.

The GET method sends the encoded user information appended to the page request. The page and the encoded information are separated by the ?character.

The POST method transfers information via HTTP headers. The information is encoded as described in case of GET method and put into a header called QUERY_STRING.

```
<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>
```

```
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();
}
}
```

3. What is session tracking? How do you perform session tracking in JSP?

HTTP is a "stateless" protocol which means each time a client retrieves a Web page, the client opens a new connection to the Web server and the server does not keep any record of previous client request.Session tracking is a mechanism that is used to maintain state about a series of requests from the same user(requests originating from the same browser) across some period

of time. A session id is a unique token number assigned to a specific user for the duration of that user's session.

- **getAttribute** : it returns stored value from session object. It returns null if no value is associated with name.
- **setAttribute** : It associates a value with name.
- **removeAttribute** : It removes all the values associated with name.
- **getAttributeNames** : It returns all attributes names in the session.
- **getId** : it returns unique id in the session.
- **isNew** : It determine if session is new to client.
- **getcreationTime** : It returns time at which session was created.
- **getlastAccessedTime** : It returns time at which session was accessed last by client.
- **getMaxInactiveInterval** : It gets maximum amount of time session in seconds that access session before being invalidated.
- **setMaxInaxctiveInterval** : It sets maximum amount of time session in seconds between client requests before session being invalidated.

```
<%@ page import = "java.io.*,java.util.*" %>
<%
   // Get session creation time.
   Date createTime = new Date(session.getCreationTime());

   // Get last access time of this Webpage.
   Date lastAccessTime = new Date(session.getLastAccessedTime());

   String title = "Welcome Back to my website";
   Integer visitCount = new Integer(0);
   String visitCountKey = new String("visitCount");
   String userIDKey = new String("userID");
   String userID = new String("ABCD");

   // Check if this is new comer on your Webpage.
   if (session.isNew() ){
      title = "Welcome to my website";
      session.setAttribute(userIDKey, userID);
      session.setAttribute(visitCountKey,  visitCount);
   }
   visitCount = (Integer)session.getAttribute(visitCountKey);
   visitCount = visitCount + 1;
   userID = (String)session.getAttribute(userIDKey);
   session.setAttribute(visitCountKey,  visitCount);
%>

<html>
   <head>
```

```
•        <title>Session Tracking</title>
•     </head>
•
•     <body>
•        <center>
•           <h1>Session Tracking</h1>
•        </center>
•
•        <table border = "1" align = "center">
•           <tr bgcolor = "#949494">
•              <th>Session info</th>
•              <th>Value</th>
•           </tr>
•           <tr>
•              <td>id</td>
•              <td><% out.print( session.getId()); %></td>
•           </tr>
•           <tr>
•              <td>Creation Time</td>
•              <td><% out.print(createTime); %></td>
•           </tr>
•           <tr>
•              <td>Time of Last Access</td>
•              <td><% out.print(lastAccessTime); %></td>
•           </tr>
•           <tr>
•              <td>User ID</td>
•              <td><% out.print(userID); %></td>
•           </tr>
•           <tr>
•              <td>Number of visits</td>
•              <td><% out.print(visitCount); %></td>
•           </tr>
•        </table>
•
•     </body>
•  </html>
```

4. What is a cookie? How do you create and retrieve information from cookies using Servlets? Explain with an example. (10 Marks)

A *cookie* is stored on a client and contains state information. Cookies are valuable for tracking user activities. Acookie can save the user's name, address, and other information.
The user does not need to enter this data each time he or she visits the store.
Aservlet can write a cookie to a user's machine via the **addCookie( )** method of the **HttpServletResponse** interface. The data for that cookie is then included in the header of

the HTTP response that is sent to the browser.
The names and values of cookies are stored on the user’s machine. Some of the information that is saved for each cookie includes the following:

- The name of the cookie
- The value of the cookie
- The expiration date of the cookie
- The domain and path of the cookie

The expiration date determines when this cookie is deleted from the user’s machine. If an expiration date is not explicitly assigned to a cookie, it is deleted when the current browser session ends. Otherwise, the cookie is saved in a file on the user’s machine.
The domain and path of the cookie determine when it is included in the header of an HTTP request. If the user enters a URL whose domain and path match these values, the cookie is then supplied to the Web server. Otherwise, it is not.
There is one constructor for **Cookie**. It has the signature shown here:
Cookie(String *name*, String *value*)
Here, the name and value of the cookie are supplied as arguments to the constructor.

```
<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>
```

```
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();
}
}
```

The source code for **GetCookiesServlet.java** is shown in the following listing. It invokes the **getCookies( )** method to read any cookies that are included in the HTTP GET request. The names and values of these cookies are then written to the HTTP response. Observe that the **getName( )** and **getValue( )** methods are called to obtain this information.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class GetCookiesServlet extends HttpServlet {
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
// Get cookies from header of HTTP request.
Cookie[] cookies = request.getCookies();
// Display these cookies.
response.setContentType("text/html");
PrintWriter pw = response.getWriter();
pw.println("<B>");
for(int i = 0; i < cookies.length; i++) {
String name = cookies[i].getName();
String value = cookies[i].getValue();
pw.println("name = " + name +
"; value = " + value);
}
pw.close();
}
}
```

5. What is an iterator? How do you access the elements of a collection using iterator and ListIterator? Illustrate with an example.

When we 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 bidirectional traversal of a list, and the modification of elements. **Iterator** and **ListIterator** are generic interfaces which are declared as shown here:

```
interface Iterator<E>
interface ListIterator<E>
```

Here, **E** specifies the type of objects being iterated. The **Iterator** interface declares the methods shown in Table 17-8. The methods declared by **ListIterator** are shown in Table 17-9. In both cases, operations that modify the underlying collection are optional. For example, **remove( )** will throw **UnsupportedOperationException** when used with a read-only collection. Various other exceptions are possible.

Using an Iterator

Before you can access a collection through an iterator, you must obtain one. Each of the collection classes provides an **iterator( )** method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time. In general, to use an iterator to cycle through the contents of a collection, follow these steps:

1. Obtain an iterator to the start of the collection by calling the collection's **iterator( )** method.
2. Set up a loop that makes a call to **hasNext( )**. Have the loop iterate as long as **hasNext( )** returns **true**.
3. Within the loop, obtain each element by calling **next( )**.

For collections that implement **List**, you can also obtain an iterator by calling **listIterator( )**. As explained, a list iterator gives you the ability to access the collection in either the forward or backward direction and lets you modify an element. Otherwise, **ListIterator** is used just like **Iterator**.

The following example implements these steps, demonstrating both the **Iterator** and **ListIterator** interfaces. It uses an **ArrayList** object, but the general principles apply to any type of collection. Of course, **ListIterator** is available only to those collections that implement the **List** interface.

```
// 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();
}
}
```

6. What is a hashtable? Write the difference between hash table and hash map. Write an example using Hashtable to store the names of bank depositors and display their current balances

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.

Like **HashMap**, **Hashtable** stores key/value pairs in a hash table. However, neither keys nor values can be **null**. When using a **Hashtable**, you specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

Hashtable was made generic by JDK 5. It is declared like this:

class Hashtable<K, V>

Here, **K** specifies the type of keys, and **V** specifies the type of values.

Ahash table can only store objects that override the **hashCode( )** and **equals( )** methods that are defined by **Object**. The **hashCode( )** method must compute and return the hash code for the object. Of course, **equals( )** compares two objects. Fortunately, many of Java’s built-in classes already implement the **hashCode( )** method. For example, the most common type of **Hashtable** uses a **String** object as the key. **String** implements both **hashCode( )** and **equals**( ).

The **Hashtable** constructors are shown here:

Hashtable( )

Hashtable(int *size*)

Hashtable(int *size*, float *fillRatio*)

Hashtable(Map<? extends K, ? extends V> *m*)

HashMap	Hashtable
1) HashMap is **non synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code.	Hashtable is **synchronized**. It is thread-safe and can be shared with many threads.
2) HashMap **allows one null key and multiple null values**.	Hashtable **doesn't allow any null key or value**.
3) HashMap is a **new class introduced in JDK 1.2**.	Hashtable is a **legacy class**.
4) HashMap is **fast**.	Hashtable is **slow**.
5) We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap);	Hashtable is internally synchronized and can't be unsynchronized.
6) HashMap is **traversed by Iterator**.	Hashtable is **traversed by Enumerator and Iterator**.

Ex:

```
import java.util.*;
class HashMapDemo {
public static void main(String args[]) {
// Create a hash map.
HashMap<String, Double> hm = new HashMap<String, Double>();
// Put elements to the map
hm.put("John Doe", new Double(3434.34));
hm.put("Tom Smith", new Double(123.22));
hm.put("Jane Baker", new Double(1378.00));
hm.put("Tod Hall", new Double(99.22));
hm.put("Ralph Smith", new Double(-19.08));
// Get a set of the entries.
Set<Map.Entry<String, Double>> set = hm.entrySet();
// Display the set.
for(Map.Entry<String, Double> me : set) {
System.out.print(me.getKey() + ": ");
System.out.println(me.getValue());
}
System.out.println();
// Deposit 1000 into John Doe's account.
double balance = hm.get("John Doe");
hm.put("John Doe", balance + 1000);
```

```
System.out.println("John Doe's new balance: " +
hm.get("John Doe"));
}
}
```

7. What is a vector class? Illustrate its methods with an example.

Vector implements a dynamic array. It is similar to **ArrayList**, but with two differences: **Vector** is synchronized, and it contains many legacy methods that are not part of the Collections Framework.With the advent of collections, **Vector** was reengineered to extend **AbstractList** and to implement the **List** interface. With the release of JDK 5, it was retrofitted for generics and reengineered to implement **Iterable**. This means that **Vector** is fully compatible with collections, and a **Vector** can have its contents iterated by the enhanced **for** loop.

Vector is declared like this:

class Vector<E>

Here, **E** specifies the type of element that will be stored.

Here are the **Vector** constructors:

Vector( )
Vector(int *size*)
Vector(int *size*, int *incr*)
Vector(Collection<? extends E> *c*)

Method	Description
void addElement(E *element*)	The object specified by *element* is added to the vector.
int capacity( )	Returns the capacity of the vector.
Object clone( )	Returns a duplicate of the invoking vector.
boolean contains(Object *element*)	Returns **true** if *element* is contained by the vector, and returns **false** if it is not.
void copyInto(Object *array*[ ])	The elements contained in the invoking vector are copied into the array specified by *array*.
E elementAt(int *index*)	Returns the element at the location specified by *index*.
Enumeration<E> elements( )	Returns an enumeration of the elements in the vector.
void ensureCapacity(int *size*)	Sets the minimum capacity of the vector to *size*.
E firstElement( )	Returns the first element in the vector.
int indexOf(Object *element*)	Returns the index of the first occurrence of *element*. If the object is not in the vector, –1 is returned.
int indexOf(Object *element*, int *start*)	Returns the index of the first occurrence of *element* at or after *start*. If the object is not in that portion of the vector, –1 is returned.
void insertElementAt(E *element*, int *index*)	Adds *element* to the vector at the location specified by *index*.

```
// 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());
v.addElement(6);
v.addElement(7);
System.out.println("Current capacity: " +
v.capacity());
v.addElement(9);
v.addElement(10);
System.out.println("Current capacity: " +
v.capacity());
v.addElement(11);
v.addElement(12);
System.out.println("First element: " + v.firstElement());
System.out.println("Last element: " + v.lastElement());
if(v.contains(3))
System.out.println("Vector contains 3.");
// Enumerate the elements in the vector.
Enumeration vEnum = v.elements();
System.out.println("\nElements in vector:");
while(vEnum.hasMoreElements())
System.out.print(vEnum.nextElement() + " ");
System.out.println();
}
}
```