Question Bank with solutions

BIS402 Advanced JAVA

Module 1 –Collections Framework

1. Explain the core interfaces in the collection Framework.

Answer:

The interfaces that underpin collections are :

i)Collection -Enables you to work with groups of objects; it is at the top of the collections hierarchy.

ii)Deque - Extends Queue to handle a double-ended queue. (Added by Java SE 6.) iii)List -Extends Collection to handle sequences (lists of objects).

iv)NavigableSet- Extends SortedSet to handle retrieval of elements based on closest-match searches.

v)Queue -Extends Collection to handle special types of lists in which elements are removed only from the head.

vi)Set- Extends Collection to handle sets, which must contain unique elements.

vi)- SortedSet Extends Set to handle sorted sets.

Collection is a generic interface that has this declaration:

interface Collection<E>

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

Objects are added to a collection by calling add( ).You can add the entire contents of one collection to another by calling addAll( ).

You can remove an object by using remove( ). To remove a group of objects, call removeAll( ).

You can remove all elements except those of a specified group by calling retainAll( ).

To empty a collection, call clear( ).

You can determine whether a collection contains a specific object by calling contains( ).

To determine whether one collection contains all the members of another, call containsAll().

We determine when a collection is empty by calling isEmpty().The number of elements currently held in a collection can be determined by calling size( ).
The toArray() methods return an array that contains the elements stored in the invoking collection. The first returns an array of Object.

ii)List is a generic interface that has this declaration:
interface List<E>
Here, E specifies the type of objects that the list will hold.

iii) The Set interface defines a set.
It extends Collection and declares the behavior of a collection that does not allow duplicate elements.
So the add() method returns false if an attempt is made to add duplicate elements to a set. It does not define any additional methods of its own.
Set is a generic interface that has this declaration:
interface Set<E>
Here, E specifies the type of objects that the set will hold.

iv) The Queue Interface
The Queue interface extends Collection and declares the behavior of a queue, which is a first-in, first-out list. Queue is a generic interface that has this declaration:
interface Queue<E>
Some methods are:
E element( ) Returns the element at the head of the queue. The element is not removed.
boolean offer(E obj) Attempts to add obj to the queue. Returns true if obj was added and false otherwise.

E peek( ) Returns the element at the head of the queue. It returns null if the queue is empty. The element is not removed.

E poll( ) Returns the element at the head of the queue, removing the element in the process. It returns null if the queue is empty.

E remove( ) Removes the element at the head of the queue, returning the element in the process.

2. List any 6 methods with the method signature and its purpose from the collection interface and any exceptions thrown.

Answer:

1.boolean add(E obj)

Adds obj to the invoking collection. Returns true if obj was added to the collection. Returns false if obj is already a member of the collection and the collection does not allow duplicates.

2.boolean addAll(Collection<? extends E>

Adds all the elements of c to the invoking collection. Returns true if the operation succeeded (i.e.,
the elements were added). Otherwise, returns false.

3.void clear( )

Removes all elements from the invoking collection.

4.boolean contains(Object obj)

Returns true if obj is an element of the invoking collection. Otherwise, returns false.

5. boolean containsAll(Collection<?> c)

Returns true if the invoking collection contains all elements of c. Otherwise, returns false.

6.boolean equals(Object obj)

Returns true if the invoking collection and obj are equal. Otherwise, returns false.

The Exceptions thrown are:

- ✓ A ClassCastException is generated when one object is incompatible with another, such as when an attempt is made to add an incompatible object to a collection.

- ✓ A NullPointerException is thrown if an attempt is made to store a null object and null elements are not allowed in the collection.

- ✓ An IllegalArgumentException is thrown if an invalid argument is used.

An IllegalStateException is thrown if an attempt is made to add an element to a fixed-length collection that is full

3. Write short notes on ArrayList.

Answer:

The ArrayList class extends AbstractList and implements the List interface. ArrayList is a generic class that has this declaration:

```
class ArrayList<E>
```

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

ArrayList supports dynamic arrays that can grow as needed.

ArrayList has the constructors shown here:

```
ArrayList( )
ArrayList(Collection<? extends E> c)
ArrayList(int capacity)
```

Eg:

```
import java.util.*;
class ArrayListDemo {
public static void main(String args[]) { // Create an array list.
ArrayList<String> al = new ArrayList<String>();

System.out.println("Initial size of al: " + al.size());
  al.add("C"); al.add("A"); al.add("E"); al.add("B");
  al.add("D");  al.add(1, "A2");

System.out.println("Size of al after additions: " + al.size());
System.out.println("Contents of al: " + al);

// Remove elements from the array list.
al.remove("F");
```

```
al.remove(2);
System.out.println("Size of al after deletions: " + al.size());
System.out.println("Contents of al: " + al);
}
}
```

4.Write a program to initialize an ArrayList with 5 Integer objects. Calculate and display the sum and average of the items in the list.

Answer:

```
import java.util.*;
 public class Test1
{
    public static void main(String[] args) {
       ArrayList<Integer> list=new ArrayList<Integer>();
      list.add(10);list.add(20);list.add(30);
      list.add(40);
      list.add(50);
      double sum = 0;
       for (int i : list)
       {
          sum += i;
       }
       double average = sum / list.size();
       System.out.println("Average = " + average);
   }
}
```

5. Explain the constructors for TreeSet. Write a java program to create TreeSet collection and access via an Iterator.

Answer:

TreeSet extends AbstractSet and implements the NavigableSet interface.

TreeSet has the following constructors:

TreeSet( )

TreeSet(Collection<? extends E> c)

TreeSet(Comparator<? super E> comp)

TreeSet(SortedSet<E> ss)

Example that demonstrates a TreeSet:

```
import java.util.*;
class TreeSetDemo {
public static void main(String args[]) {
// Create a tree set.
TreeSet<String> ts = new TreeSet<String>();
// Add elements to the tree set.
ts.add("C"); ts.add("A");
ts.add("B");
ts.add("E");
 ts.add("F"); ts.add("D");
System.out.println(ts);
}
}
```

The output from this program is shown here:

```
[A, B, C, D, E, F]
```

As explained, because TreeSet stores its elements in a tree, they are automatically arranged in sorted order.

6. Explain any 2 legacy classes of Java's collection Framework.

Answer:

Early version of java did not include the Collections framework. It only defined several classes and interfaces that provide methods for storing objects. These classes are also known as Legacy classes.

The following legacy classes defined by java.util package

- Dictionary
- HashTable

Dictionary class:

Dictionary is an abstract class.It represents a key/value pair and operates much like Map.

Dictionary is classified as obsolete, because it is fully superseded by Map class.

With the advent of JDK 5, Dictionary was made generic. It is declared as shown here:

```
class Dictionary<K, V>
```

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

1. To add a key and a value, use the put() method.
2. Use get() to retrieve the value of a given key.

Hashtable class:

Hashtable stores key/value pair. However neither keys nor values can be null.

Hashtable is synchronized while HashMap is not.

Hashtable has following four constructors:

Hashtable() //This is the default constructor. The default size is 11.

Hashtable(int size) //This creates a hash table that has an initial size

Hashtable(int size, float fillratio)

Code snippet:

```
import java.util.*;
class HashTableDemo
{
  public static void main(String args[])
  {
    Hashtable<String,Integer> ht = new Hashtable<String,Integer>();
    ht.put("a",new Integer(100));
    ht.put("b",new Integer(200));
    ht.put("c",new Integer(300));
    ht.put("d",new Integer(400));
    Set st = ht.entrySet();     //entrySet  returns a set containing Map.Entry values
    Iterator itr=st.iterator();
    while(itr.hasNext())
    {
      Map.Entry m=(Map.Entry)itr.next();
      System.out.println(itr.getKey()+" "+itr.getValue());
    }
  }
}
```

7. Write short notes on all the methods defined by the SortedSet interface.

Answer:

The SortedSet interface extends Set and declares the behavior of a set sorted in ascending order.

SortedSet is a generic interface that has this declaration:

interface SortedSet<E>

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

Methods defined by the SortedSet interface are:

1.Comparator<? super E> comparator( )
Returns the invoking sorted set's comparator. If the natural ordering is used for this set, null is returned.
2.E first( )
Returns the first element in the invoking sorted set.
3.SortedSet<E> headSet(E end)
Returns a SortedSet containing those elements less than end that are contained in the invoking sorted set. Elements in the returned sorted set are also referenced by the invoking sorted set.

4.E last( )
Returns the last element in the invoking sorted set.
5.SortedSet<E> subSet(E start , E end)
Returns a SortedSet that includes those elements between start and end– 1. Elements in the returned collection are also referenced by the invoking object.
6.SortedSet<E> tailSet(E start)
Returns a SortedSet that contains those elements greater than or equal to start that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object.

8. Write a Java program to demonstrate ArrayDeque by using it to create a stack

Answer:
The following program demonstrates ArrayDeque by using it to create a stack:

```
// Demonstrate ArrayDeque.
import java.util.*;
class ArrayDequeDemo {
public static void main(String args[]) {
 // Create a tree set.
ArrayDeque<String> adq = new ArrayDeque<String>();
// Use an ArrayDeque like a stack.
adq.push("A");
adq.push("B");
 adq.push("D");
```

```
adq.push("E");
adq.push("F");
System.out.print("Popping the stack: ");
while(adq.peek() != null)
 System.out.print(adq.pop() + " ");
System.out.println();
}
}
```

The output is shown here:

```
Popping the stack: F E D B A
```

Module 2

1. Explain the need for strings. Explain different ways of creating Strings. (5 marks)

A string is needed to efficiently store a sequence of characters. One can perform all type of string operations without changing the original string.

The **String** class supports several constructors:

```
String s = new String();
```

will create an instance of **String** with no characters in it

```
String(char chars[ ])
```

Creates a **String** initialized by an array of characters

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
```

String(char *chars*[ ], int *startIndex*, int *numChars*)

specify a subrange of a character array as an initializer

startIndex specifies the index at which the subrange begins

numChars specifies the number of characters to use

```
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars, 2, 3); //cde
```

String(String *strObj*)

construct a **String** object that contains the same character sequence as another **String** object.

s1 and **s2** contain the same string

```
char c[] = {'J', 'a', 'v', 'a'};
```

String s1 = new String(c);String s2 = new String(s1);
System.out.println(s1);System.out.println(s2);

Java's **char** type uses 16 bits to represent the basic Unicode character set.
strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set.
8-bit ASCII strings are common

String(byte *chrs*[ ])
- *chrs* specifies the array of bytes

String(byte *chrs*[ ], int *startIndex*, int *numChars*)
- specify a subrange In each of these constructors
- byte-to-character conversion is done by using the defaultcharacter encoding

String(int *codePoints*[ ], int *startIndex*, int *numChars*)
- extended Unicode character set
- *codePoints* is an array that contains Unicode code points

- There are also constructors that let you specify **Charset**

2. Write a program to remove duplicate characters in a given string and display new string without duplicates. (5 marks)

```
public class StringDemo {

   public static void main(String[] args) throws IOException {

String s = "rabbit";
    String dupRem;

    for(int i=0; i<s.length(); i++){
       char ch = s.charAt(i);
       for(int j=i+1; j<s.length(); j++)
          if(ch==s.charAt(j)){
             s=s.replace(ch,' ');
          }
          }
    s=s.replace(" ","");
     System.out.println(s);
  }
}
```

Output : "rait"

3. Explain how strings can be modified in Java with different methods (6 marks)

1. substring() : can extract a substring

String substring(int *startIndex*): *startIndex* specifies the index at which the substring will begin and it returns a copy of the substring that begins at *startIndex* and runs to the end of the invoking string

String substring(int *startIndex*, int *endIndex*):*startIndex* specifies the beginning index, and *endIndex* specifies the stopping point. The string returned contains all the characters from the beginning index, up to, but not including, the ending index

2. String concat(String *str*): This method creates a new object that contains the invoking string with the contents of *str* appended to the end.

String s1 = "one";

String s2 = s1.concat("two");

String s1 = "one";

String s2 = s1 + "two";

3. String replace(char *original*, char *replacement*) :replaces all occurrences of one character in the invoking string with another character and returns a copy of the invoking string from which any leading and trailing whitespace has been removed.

String s = "Hello".replace('l', 'w');

"Hewwo" into **s**

4. String trim( ) :returns a copy of the invoking string from which any leading and trailing whitespace has been removed.
5. String toLowerCase( ) – changes all the characters to lower case, Nonalphabetical characters, such as digits, are unaffected
6. String toUpperCase( ) – changes all the characters to uppercase, Nonalphabetical characters, such as digits, are unaffected

4. Explain StringBuffer. How is it different from Strings? (6 marks)

supports a modifiable string

String represents fixed-length, immutable character sequences.

StringBuffer represents growable and writable character sequences. It may have characters and substrings inserted in the middle or appended to the end automatically grow has more characters preallocated than are actually needed.

It is different from Strings because Strings are immutable whereas StringBuffer is mutable. Hence there are methods in StringBuffer to insert, append, set a particular character, reverse that can't be done directly in Strings.

5. Explain the following StringBuffer methods i)insert ii)append iii)substring() iv) replace (6 marks)

insert() - inserts one string into another

StringBuffer insert(int *index*, String *str*)

StringBuffer insert(int *index*, char *ch*)

StringBuffer insert(int *index*, Object *obj*)

overloaded to accept values of all the primitive types, plus **String**s, **Object**s, and **CharSequence**s

index specifies the index at which point the string will be inserted into the invoking **StringBuffer**

```
StringBuffer str = new StringBuffer("Hello Jim. How are you?");
str.insert(6, "Cara and ");

System.out.println(str.toString());
```

append() : concatenates the string representation of any other type of data to the end of the invoking **StringBuffer** object. The result is appended to the current **StringBuffer** object

- StringBuffer append(String *str*)
- StringBuffer append(int *num*)
- StringBuffer append(Object *obj*)

```
StringBuffer str = new StringBuffer("Hello");
String s = str.append(" Jim").append(" How are you?").toString();
System.out.println(s);
```

substring()

- String substring(int *startIndex*) : returns the substring that starts at *startIndex* and runs to the end of the invoking **StringBuffer** object
- String substring(int *startIndex*, int *endIndex*) : returns the substring that starts at *startIndex* and runs through *endIndex*–1

replace()

- StringBuffer replace(int *startIndex*, int *endIndex*, String *str*)
- substring being replaced is specified by the indexes *startIndex* and *endIndex*
- substring at *startIndex* through *endIndex*–1 is replaced
- replacement string is passed in *str*

6. How to check occurrences of a substring of characters in a given string? (4 marks)

We can use a method called as regionMatches()

- compares a specific region inside a string with another specific region in another string.
- an overloaded form that allows you to ignore case in such comparisons
- boolean regionMatches(int *startIndex*, String *str2*, int *str2StartIndex*, int *numChars*)
- boolean regionMatches(boolean *ignoreCase*, int *startIndex*, String *str2*, int *str2StartIndex*, int *numChars*)
- *startIndex* specifies the index at which the region begins within the invoking **String** object.
- **String** being compared is specified by *str2*
- The index at which the comparison will start within *str2* is specified by *str2StartIndex.*
- length of the substring being compared is passed in *numChars.*
- if *ignoreCase* is **true**, the case of the characters is ignored

MODULE 4

1. **What is Servlet? Explain the lifecycle of servlet.**
   A Servlet is a small program that executes on server side of a web connection. A servlet extends functionality of a web server to generate dynamic web content.

The lifecycle of a servlet consists of 3 methods: **init(), service() and destroy()**
These are implemented with every servlet that is invoked at specific times by the server.

- user enters URL (Uniform Resource Locator) to a web browser.
  - browser generates HTTP request for this URL.
  - request is sent to server
- HTTP request is received by web server
  - server maps request to a particular server
  - servlet is dynamically retrieved and loaded into address space of the server
- server invokes **init()**

    - invoked only when servlet is first loaded into memory.
    - possible to pass initialization parameters to the servlet.
- **service()** method is invoked
    - called to process HTTPRequest.
    - it may also have to formulate a HTTP response.
    - called for each HTTPRequest
- sever calls **destroy()**
    - returns any resources allocated to servlets
    - memory allocated for servlet and objects are garbage collected.

2. Explain core classes and interfaces in javax.servlet package.

Core Interfaces	
Servlet	Defines lifecycle methods for a servlet
ServletConfig	Allows servlet to get initialization parameters
ServletContext	Defines a set of methods that a servlet uses to communicate with its servlet container, for example, to get the MIME type of a file, dispatch requests, or write to a log file.
ServletRequest	Used to read data from a client request
ServletResponse	Used to write data to a client response
Core Classes	
GenericServlet	Defines a generic servlet that implements Servlet and ServletConfig interfaces
ServletInputStream	Provides an input stream for reading binary data from a client request.
ServletOutputStream	Provides an output stream for sending binary data to the client.

3. Write a java program for reading client parameters using servlet and display them.

PostParam.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Post parameters Demo</title>
</head>
<body>
<h1>  Pass parameters using post </h1>
<div style= "width:350px; height:200px; padding: 10px; background-color:#ccccff; color: #333388">
    <form name = "form1" method="post" action="servlets/postParamServlet">
    <label>Parameter 1 : </label>
    <input type = text name = "e" size ="25" value = "" />
    <br><br>
    <label>Parameter 2 : </label>
    <input type = text name = "p" size = "25" value = "" />
    <br><br>
    <input type = submit value = "Submit" />
</form>
</div>
</body>
</html>
```

web.xml entry under tag <web-app>

```
<servlet>
   <description></description>
   <display-name>postParamServlet</display-name>
   <servlet-name>postParamServlet</servlet-name>
   <servlet-class>postParamServlet</servlet-class>
 </servlet>
 <servlet-mapping>
   <servlet-name>postParamServlet</servlet-name>
   <url-pattern>/servlets/postParamServlet</url-pattern>
 </servlet-mapping>
```

PostParamServlet.java

```
import java.io.*;
import javax.servlet.*;
import java.util.*;
public class postParamServlet extends GenericServlet {
    public void service(ServletRequest req, ServletResponse resp)
            throws IOException, ServletException  {

PrintWriter pw = resp.getWriter();

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

    pw.println("<h1> Parameters passed using get method are : </h1>");

//Display param name and values
    while(e.hasMoreElements() ) {
            String pname = (String) e.nextElement();
            pw.print("<h3>" +pname+" = ");
            String pvalue = req.getParameter(pname);
            pw.print(pvalue+"</h3>");
    }

    pw.close();
    }
}
```

4. Write short notes on HTTP request and response.

Method Summary	
String **getAuthType**()	Returns the name of the authentication scheme.
Cookie[]**gtCookies**()	Returns an array containing all of the Cookie objects the client sent with this request.

long **getDateHeader**(String name)	Returns the value of the specified request header as a long value that represents a Date object.
String **getHeader**(String name)	Returns the value of the specified request header as a String.
Enumeration **getHeaderNames**()	Returns an enumeration of all the header names this request contains.
int **getIntHeader**(String name)	Returns the value of the specified request header as an int.
String **getMethod**()	Returns the name of the HTTP method with which this request was made, for example, GET, POST
String **getPathInfo**()	Returns any extra path information associated with the URL the client sent before the query string.
String **getPathTranslated**()	Returns any extra path information after the servlet name but before the query string, and translates it to a real path.
String **getQueryString**()	Returns the query string that is contained in the request URL after the path.
String **getRemoteUser**()	Returns the login of the user making this request.
String **getRequestedSessionId**()	Returns the session ID specified by the client.
String **getRequestURI**()	Returns the part of this request's URL from the protocol name up to the query string in the first line of the HTTP request.
StringBuffer **getRequestURL**()	Reconstructs the URL the client used to make the request.
String **getServletPath**()	Returns the part of this request's URL that calls the servlet.
HttpSession **getSession**()	Returns the current session associated with this request, or if the request does not have a session, ***creates one***.

HttpSession **getSession**(boolean create)	Returns the current HttpSession associated with this request or, if there is no current session and create is true, returns a new session.
Boolean **isRequestedSessionIdFromCookie**()	Checks whether the requested session ID came in as a cookie.
Boolean **isRequestedSessionIdFromURL**()	Checks whether the requested session ID came in as part of the request URL.
Boolean **isRequestedSessionIdValid**()	Checks whether the requested session ID is still valid.

Enables a servlet to formulate an HTTP response to a client.

Several constants are defined. And they correspond to different status codes assigned to an HTTP response, eg.,

SC_OK, SC_NOT_FOUND

Method Summary	
void **addCookie**(Cookie cookie)	Adds the specified cookie to the response.
boolean **containsHeader**(String *name*)	Returns true if HTTP response header contains a field named *name*
String **encodeRedirectURL**(String url)	Encodes the specified URL for use in the sendRedirect method or, if encoding is not needed, returns the URL unchanged.
String **encodeURL**(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) throws IOException	Sends an error response to the client using the specified status code and clearing the buffer.
void **sendError**(int sc, String msg) throws IOException	Sends an error response to the client using the specified status.
void **sendRedirect**(String *location*) throws IOException	redirects to redirect *location* URL.

void **setDateHeader**(String name, long date)	Sets a response header with the given name and date-value (in milliseconds since Jan 1, 1970, GMT)
void **setHeader**(String name, String value)	Sets a response header with the given name and value.
void **setIntHeader**(String name, int value)	Sets a response header with the given name and integer value.
void **setStatus**(int sc)	Sets the status code for this response.

5.Define JSP. Explain different JSP tags by taking a suitable example.

JSP is a server side program that is similar to a Java Servlet.

- It is called by a client to provide a web service, the nature of which depends on J2EE.
- JSP differs from Servlet in that Servlet is written completely in Java, output String object has to be written in println() methods
- JSP is written in HTML, XML with JSP syntax.
- JSP offers same features as servlet
    - it is converted to a Java servlet the first time the client requests the JSP.

1. **Comment tag :** describes functionality of statements
   Syntax : <%-- comment --%>
2. **Declaration statement tags** : defines variables, objects, methods available to other components of the JSP program
   Syntax : <%! %>
3. **Directive Tags :** commands the JSP virtual engine to perform a specifiec task such as importing a Java package.
   Syntax: <%@ %>
    - 3 commonly used directives :
        - **import** : used to import Java packages
        - **include :** inserts a specified file into the JSP program
        - **taglib :** specifies a file that contains a tag library.

   <%@ page import = “java.sql.*” %>
   %@include file=”jim\books.html” %>
   <%taglib uri=”mytags.tld” %>
4. **Expression Tags** : used for an expression statement whose result replaces the expression tag when the JSP virtual engine resolves JSP tags

Syntax : <%= %>

5. **Scriptlet tags**: contains commonly used Java control statements and loops
   Syntax : <% %>

variableDemo.jsp

```
<html>
 <head><title>  Variable Demo </title>
 </head>
 <body>
<h2> Variable Demo </h2>
<%-- declare age --%>
      <%! int age=29; %>
<%-- expression statement --%>
<p><h3>  Your age is <%=age %> </h3></p>
  </body>
</html>
```

MultipleStmtDemo.jsp

```
<html>
 <head><title>  Multiple statements Demo </title>
 </head>
 <body>
  <h1>Multiple statements </h1>
      <%! int age=29;
        float salary= 5000;
        int empnumber=345; %>
<p><b>
Age is : <%= age%> </b></p>
<p><b>
Salary is :<%= salary%> </b></p>
<p><b>
Employee number is : <%= empnumber %> </b></p>

  </body>
</html>
```

6.Explain how sessions can be handled using JSP.

- a JSP database system is able to share information among JSP programs within a session by using session object

- each time session is created, a unique ID is assigned to the session and stored in a cookie
- This unique ID enables JSP programs to track multiple sessions simultaneously.
- In addition to **session ID**, session can also have other information, attributes.
- Attribute: can be login information, preferences, etc.
- ***session attributes*** can be retrieved and modified. To manage session attributes
    - **setAttribute()** – passes the name and value of the attribute to be set
    - **getAttributeNames()** – returns names of all the attributes in an Enumeration
    - **getAttribute()** – returns the value of the attribute's name passed in the method.

createSession.jsp

```
<html>
  <head><title> session Objects </title>
  </head>
    <body>

        <h1> Create Session</h1>
        <%! String AtName = "Product";
           String AtValue = "1234";
        %>
        <%
           session.setAttribute(AtName,AtValue);
        %>
        <p><a href="viewSession.jsp">View Session</a></p>
  </body>
</html>
```

viewSession.jsp

```
<html>
  <head><title> session Objects </title>
  </head>
    <body>
<%@ page import="java.util.*" %>
<%@ page import="javax.servlet.*" %>
        <h1> Get Attribute values</h1>
        <% Enumeration purchases = session.getAttributeNames();
         %>
        <%
          while(purchases.hasMoreElements()) {
```

```
            String AtName = (String)purchases.nextElement();
            String AtValue = (String) session.getAttribute(AtName);
         %>
        <p>  Attribute Name : <%= AtName %> </p>
        <p> Attribute Value: <%=AtValue %> </p>
        <% } %>

  </body>
</html>
```

MODULE 5

1.Explain 4 types of JDBC drivers

Type 1 JDBC-to-ODBC Driver

- ODBC (Open Database Connectivity)
    - Created b y Microsoft
    - The basis from which Sun Microsystem, Inc created JDBC
    - A DBMS independent database program. The JDBC-to-ODBC Driver
    - Also known as JDBC ODBC bridge
    - Used to translate DBMS calls between the JDBC specification and the ODBC specification
        - Receives message from J2EE component that conforms to the JDBC specification.
        - Then translates this into message format understood by the DBMS.
- Note : Avoid using in mission-critical systems because the extra translation might negatively impact performance.

Type 2 Java/Native Code Driver

- Uses Java classes to generate platform specific code, i.e. code understood by specific DBMS.
- The driver is compiled for used with particular operating system.
- Disadvantage
    - Loss of **portability**
    - **Platform dependent**

Type 3 JDBC Driver

- Also known as the Java Protocol
- Converts SQL queries into JDBC formatted statements.
- The JDBC formatted statements are translated into the format required by the DBMS.
- Platform independent.

Type 4 JDBC Driver

- Also known as the Type 4 database protocol.
- Similar to Type 3 except
  - SQL queries are translated into format required by DBMS
  - Does not need to be converted to JDBC format system.
- Fastest way to communicate SQL queries to DBMS as it does not have the overhead of conversion of calls to other API's.

1. **Explain the brief overview of the whole JDBC process**

1. **Loading** the JDBC driver
2. **Connecting** to the DBS
3. **Creating and executing** a statement
4. **Processing data** returned by the DBMS
5. **Terminating** the *connection* with the DBMS

- **Loading the JDBC driver**
  - Class.forName() is used to load J2EE component
  - We are going to develop an application that uses Microsoft access
  - Then we must write a routing that loads the JDBC/ODBC Bridge driver called sun.jdbc.odbc. JdbcOdbcDriver.

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver)

- **Connecting to the DBMS**
  - Connect using DriverManager.getConnection() method.
  - java.sql.DriverMagager class is the highes class in the java.sql hierarchy and is responsible for managing driver information.
  - Parameters passed
    - URL – a string object that contains the driver name and name of database being accessed.
    - Username (optional)
    - Password (optional)
  - Returns a Connection interface that is used to reference the database.

String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";

```
String password = "pswd";
private Connection Db;
try{
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver);
    Db = DriverManager.getConnection (url, userID, password);
}
```

- **<u>Create and Execute an SQL Statement.</u>**
  - After connection is made, SQL query can be sen to DBMS for processing
  - SQL query – consists of a series of SQL command that direct DBMS.
  - The createStatement () method of the Connection interface is used to create a Statement object.
  - The Statement object is then used to execute a query that returns a ResultSet object that containes response from the DBMS.

```
Statement stmt;
ResultSet rs;
try{
    String query = "SELECT * FROM Customers";
    stmt = Db.createStatement();
    rs = stmt.executeQuery(query);
    stmt.close();
}
…
```

- **Process Data Returned by the DBMS**
  - The java.sql.ResultSet object is assigned the results received form the DBMS after the query is processed.
  - next() method of the ResultSet class
    - The first time next() method is called, positions ResultSet pointer at the first row of the ResultSet.
    - Returns a Boolean value that if false indicates that no rows are present in the ResultSet.
    - True value means at least one row is present in ResultSet.
  - do…while() loop is used to loop through each record
  - getString() method – used to copy the column contents in the ResultSet
    - can pass the column name or the column number.

```
ResultSet rs;
String fname, lname;
boolean records = rs.next();
if(!records)
```

```
            System.out.println("No data returned");
        else {
          {
            do{
               fname = rs.getString(FirstName);
               lname = rs.getString(LastName);
               System.out.println(fname + " " + lname);
            }while(rs.next() );
         }
```

- **Terminate the connection to the database**
  - Use the close() method of the Connection object to terminate the connection to the DBMX.
  - Throws an exception if problem is encountered.

```
Db.close();
```

2.List and explain various statement objects in JDBC.

1. The Statement Object

- used whenever a J2EE component needs to immediately execute a query without having the query compiled.
  - executeQuery()
    - returns ResultSet object containing rows, columns and metadata
  - execute()
    - when multiple results are returned.
  - executeUpdate()
    - used to execute queries that contain UPDATE and DELETE SQL statements.
    - returns an integer indicating number of rows updated.
    - also used for INSERT statement

Code using executeQuery()

```
….
Connection db;
Statement stmt;
ResultSet rs;
try{
        String query = "SELECT * FROM customers";
        stmt = db.createStatement();
        rs = stmt.excecuteQuery(query);
        //…….
        stmt.close();
}catch(SQLException e) {
        System.err.println("SQL Error"  + err);
```

}

Code using executeUpdate() to update

```
….
Connection db;
Statement stmt;
int numRowsUpdated;
try{
        String query = "UPDATE customers SET PAID = 'Y' WHERE BALANCE = '0'";
        stmt = db.createStatement();
        numRowsUpdated= stmt.excecuteUpdate(query);
        //…….
        stmt.close();
}catch(SQLException e) {
        System.err.println("SQL Error"  + err);
}
```

2. PreparedStatement Object

- An SQL query must be compiled before the DBMS processes the query
- Compiling occurs after Statement objects's execution method.
- Compiling query is an acceptable overhead if the query is called once.
- compiling several times becomes an expensive overhead
- Solution : PreparedStatement Object
    - **precompiles** and executes an SQL statement, known as **late binding**
    - a ? placeholder is used in the query to later insert the value for the query.

Steps to use PreparedStatement Object

1. preparedStatement() method of the Connection object is called to return the PreparedStatement()
    - preparedStatement() method is passed the query.
2. setxxx() method is used to replace the ? placeholder.
    - xxx – refers to a data type, eg., setString()
        - 2 parameters are passed -
            - First parameter : identifies the position of the ? placeholder
            - Second parameter : the late binding variable.
3. the executeQuery() method of the PreparedStatement object is called.
    - it doesn't require a pramenter 'cos query is already associated with the PreparedStatement object.

Code using PreparedStatement

```
Connection db;
ResultSet rs;
```

….
try{
String query = “SELECT * FROM customers WHERE custNumber = ?”;
PreparedStatement pstmt = db.preparedStatement(query);
pstmt.setString(1, “124”);
rs = pstmt.executeQuery();
//…… other code
pstmt.close();
}catch(SQLException e) {
System.err.println(“SQL Error” + err);
}

3. CallableStatement Object

- used to call a stored procedure from within a J2EE object.
  - stored procedure : a block of code identified by a unique name, executed by invoking the name of the stored procedure.
- uses 3 types of parameters
  - IN :
    - contains data that needs to be passed to a stored procedure.
    - value is assigned using setxxx() method
  - OUT
    - contains value returned by stored procedure
    - must be registered using registerOutParameter() method.
    - later received using getxxx() method
  - INOUT
    - used to pass and retrieve information from a stored procedure.

Steps to use CallableStatement Object

1. **prepareCall()** of the Connection object is called and passed the query
   - returns a CallableStatement object.
2. **registerOutParameter()** has two arguments
   - 1st parameter: represents the number of the parameter in the stored procedure
   - 2nd parameter : data type of the value returnd by the stored procedure, eg. VARCHAR.
3. **execute()** method of CallableStatement object is used to execute the query
   - query need not be passed as it is already identified in the prepareCall() method.

Code to use CallableStatement Object.

Connection db;
ResultSet rs;
….
try{

```
        String query = "CALL LastOrderNumber(?)";
        CallableStatement cstmt = db.prepareCall(query);
        cstmt.registerOutParameter(1, Types.VARCHAR);
        cstmt.execute();
        int lastOrderNumber = cstmt.getString(1);

                cstmt.close();
}catch(SQLException e) {
        System.err.println("SQL Error"  + err);
}
```

4. **Write short notes on Scrollable result set**

- until release of JDBC 2.1 API, virtual cursor could only be moved down the ResultSet object.
- 6 methods used in JDBC 2.1 API to position the cursor
  - first()
  - last()
  - previous()
  - absolute() – positions cursor at row number specified
  - relative() – moves the virtual cursor by specified number of rows, positive or negative
  - getRow() - returns integer that contains the number of the current row in the ResultSet.
- 3 constants con be passed to the createStatement() method of the Connection object to handle ResultSet Scrollability,
  - TYPE_FORWARD_ONLY – restricts the virtual cursor to downward movement
  - TYPE_SCROLL_INSENSITIVE
    - permits cursor to move up and down
    - makes the ResultSet insensitive to changes made by another J2EE component to data in the table whose rows are reflected in the ResultSet.
  - TYPE_SCROLL_SENSITIVE
    - permits cursor to move up and down
    - makes the ResultSet sensitive to those changes.

<u>Code to use Scrollable ResultSet</u>

```
Connection db;
Statement stmt;
ResultSet rs;
String fname, lname;
….
try{
String query = "SELECT firstName, lastName FROM customers";
```

```
  stmt = db.createStatement(TYPE_SCROLL_INSENSITIVE);
        rs = stmt.executeQuery(query);
        Boolean rec = rs.next();
        if(! rec){
          System.out.println("No data returned");
          System.exit();
        }
        rs.first();
        rs.last();
        rs.previous();
        rs.absolute(5);
        rs.relative(-2);
        rs.relative(3);
        fname = rs.getString(2);
                lname = rs.getString (3);
                System.out.println(fname + " "+ lname);
stmt.close();
}catch(SQLException e) {
        System.err.println("SQL Error"  + err);
}
```

5. Write a java program to execute a database transaction.

A ***transaction*** : involves several tasks that are required to be completed.

- Eg. in a supermarket
  - item purchased must be registered
  - transaction must be totaled
  - customer must tender the amount of the purchase.
- If one task fails the entire transaction fails
- A database transaction consists of set of SQL statements
  - if one query fails, all the statements have to be rolled back.
  - is not completed until the J2EE component calls the commit() method
  - all statements prior to commit can be rolled back
  - once commit() is called none of the SQL statements can be rolled back.
- **commit()** method is automatically enable because DBMS has an **AutoCommit** feature that by default is set to true.
- If J2EE component is performing a transaction, **AutoCommit** feature must be deactivated.

- Some tasks in a transaction need not be rolled back if entire transaction should fail.
  - eg., there are 3 task, update customer table, insert order, send customer confirmation mail.
  - if sending confirmation mail fails, it need not be rolled back
- To control the number of tasks that can be rolled back, use **savepoint**.

- there can be many savepoints in a transaction
- first savepoint is created after the execution of the first query
- each savepoint is identified by a unique name

this name is passed to the **rollback()** method to specify point where the ***rollback is to stop***

```
Connection db;
Statement stmt1, stmt2;
ResultSet rs;
….
try{
        db.setAutoCommit(false);
        String q1 = “UPDATE customers SET street = ‘5 main’ WHERE firstName = ‘Bob’”;
String q1 = “UPDATE customers SET street = ‘6 main’ WHERE firstName = ‘Tim’”;

        stmt1 = db.createStatement();
        Savepoint s1 = db.setSavepoint(“sp1”);
        stmt2 = db.createStatement();
        rs = stmt1.executeUpdate(q1);
        rs = stmt2.executeUpdate(q2);
        db.commit();
        stmt1.close();
        stmt2.close();
        db.releaseSavepoint(“sp1”);
}catch(SQLException e) {
        try{
          db.rollback(sp1);
        }catch(SQLException e) {
                System.out.pritnln(“Roll back error” + e); System.exit();
        }
        System.err.println(“SQL Error”  + err);
}
```