CBCS SCHEME

USN | 1 | C | R | 2 | 2 | M | C | 0 | 9 | 0 |

22MCA341

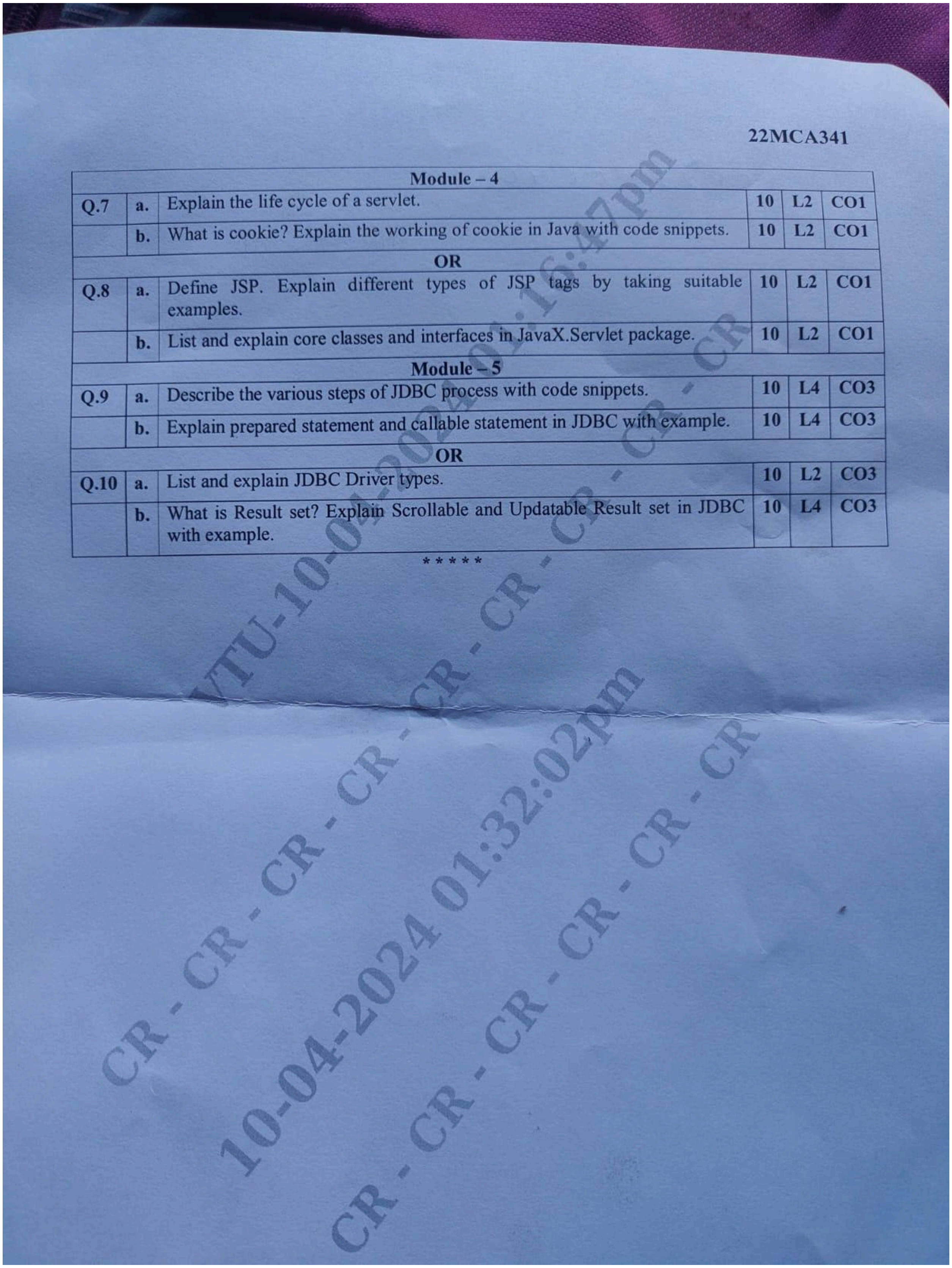
Third Semester MCA Degree Examination, Dec.2023/Jan.2024

Advanced Java & J2EE

Time: 3 hrs. Max. Marks: 100

Note: 1. Answer any FIVE full questions, choosing ONE full question from each module.
2. M : Marks , L: Bloom's level , C: Course outcomes.

		Module – 1	M	L	C
Q.1	a.	What is enum? Demonstrate the use of ordinal ( ), compareTo ( ), equals ( ) and values ( ) method with enumeration.	10	L2	CO1
	b.	Define with example for each of the following : (i) Auto boxing. (ii) Unboxing (iii) Type wrapper (iv) Marker Annotation.	10	L2	CO1
		OR			
Q.2	a.	Explain built in Annotations in detail with necessary example snippets.	10	L2	CO1
	b.	What is Annotation? Explain how do you obtain annotations at run time by use of reflection.	10	L2	CO1
		Module – 2			
Q.3	a.	Explain the following collection interfaces with example program: (i) Queue (ii) Sorted set.	10	L2	CO2
	b.	Describe ArrayList class and explain with its constructors. Demonstrate its usage with an example program.	10	L2	CO2
		OR			
Q.4	a.	Discuss the following map classes with example : (i) Hash map (ii) Tree map	10	L2	CO2
	b.	List and explain any five collection algorithms. Demonstrate various algorithms with an example program.	10	L3	CO2
		Module – 3			
Q.5	a.	Illustrate the use of following methods with an example: (i) insert (ii) append (iii) replace (iv) substring	10	L2	CO1
	b.	Differentiate string and string buffer classes. Write a program to demonstrate different constructors of string class.	10	L3	CO1
		OR			
Q.6	a.	Explain the following string comparison methods with an example : (i) equals ( ) (ii) compareTo ( ) (iii) = = (iv) equalsIgnoreCase	10	L2	CO1
	b.	Illustrate character extraction methods with examples.	10	L3	CO1



		Module – 4			
Q.7	a.	Explain the life cycle of a servlet.	10	L2	CO1
	b.	What is cookie? Explain the working of cookie in Java with code snippets.	10	L2	CO1
		OR			
Q.8	a.	Define JSP. Explain different types of JSP tags by taking suitable examples.	10	L2	CO1
	b.	List and explain core classes and interfaces in JavaX.Servlet package.	10	L2	CO1
		Module – 5			
Q.9	a.	Describe the various steps of JDBC process with code snippets.	10	L4	CO3
	b.	Explain prepared statement and callable statement in JDBC with example.	10	L4	CO3
		OR			
Q.10	a.	List and explain JDBC Driver types.	10	L2	CO3
	b.	What is Result set? Explain Scrollable and Updatable Result set in JDBC with example.	10	L4	CO3

* * * * *

1. What is enum? Demonstrate the use of ordinal(),equals(),compareTo() and values() method with enumeration.

Enumerations was added to Java language in JDK5. **Enumeration** means a list of named constant. In Java, enumeration defines a class type. An Enumeration can have constructors, methods and instance variables. It
is created using enumkeyword.Eachenumerationconstantis *public*,*static* and*final*by default.

Values( ) and ValueOf( ) method

All the enumerations has predefined methods **values()** and **valueOf()**. values() method returns an array of enum-type containing all the enumeration constants in it. Its general form is,

public **static** *enum-type[ ]* **values()**

valueOf() method is used to return the enumeration constant whose value is equal to the string passed in as argument while calling this method. It's general form is,

public **static** *enum-type* **valueOf** (String *str*)

Example of enumeration using values() and valueOf() methods:

```
enum Restaurants {

dominos, kfc, pizzahut, paninos, burgerking

}
class Test {
public static void main(String args[])
{
Restaurants r;
System.out.println("All constants of enum type Restaurants are:");
Restaurants rArray[] = Restaurants.values(); //returns an array of constants of type Restaurants
for(Restaurants a : rArray) //using foreach loop
System.out.println(a);
 r = Restaurants.valueOf("dominos"); System.out.println("I AM " + r);
}
}
```

All enumerations automatically inherited from **java.lang.Enum**. This class defines several methods that are available for use by all enumerations. We can obtain a value that indicates an enumeration constant's position in the list of constants. This is called its ordinal value, and it is retrieved by calling the **ordinal()** method, shown here:

final int ordinal( )

It returns the ordinal value of the invoking constant. **Ordinal values begin at zero.** We can compare the ordinal value of two constants of the same enumeration by using the **compareTo()** method. It has this general form:

final int compareTo(enum-type e)

The usage will be:

e1.compareTo(e2);Here, e1 and e2 should be the enumeration constants belonging to same enum type. If the ordinal value of e1 is less than that of e2, then compareTo() will return a negative value. If two ordinal values are equal, the method will return zero. Otherwise, it will return a positive number.

We can compare for equality an enumeration constant with any other object by using **equals( )**, which overrides the **equals( )** method defined by **Object**.

```
enum Person
{
        Married, Unmarried, Divorced, Widowed
}
enum MStatus
{
        Married, Divorced
}
class EnumDemo
{
        public static void main(String args[])
        {
                Person p1, p2, p3;
                MStatus           m=MStatus.Married;
                System.out.println("Ordinal       values
                are: "); for(Person p:Person.values())
                        System.out.println(p   +   "   has   a   value   "   +
                p.ordinal()); p1=Person.Married;
                p2=Person.Divorced
                ;
                p3=Person.Married;
                if(p1.compareTo(p2)
                <0)
                        System.out.println(p1   +   "   comes   before
                "+p2); else if(p1.compareTo(p2)==0)
                        System.out.println(p1 + " is same as "+p2);
                else
                        System.out.println(p1 + " comes after "+p2);
                if(p1.equals(p3))
                        System.out.println("p1    &    p3    are
                same"); if(p1==p3)
                        System.out.println("p1    &    p3    are
                same"); if(p1.equals(m))
                        System.out.println("p1 & m are same");
                else
                        System.out.println("p1 & m are not same");
                //if(p1==m) Generates error
                //System.out.println("p1 & m are same");
        }
}
```

1.b. Define with example for each of the following

a)Autoboxing

b)Unboxing

c)Type Wrapper

d) Marker Annotation

a) **Autoboxing:Autoboxing** is a process by which primitive type is automatically encapsulated(bo into its equivalent type wrapper

b) **Auto-Unboxing** is a process by which the value of an object is automatically extracted from a type Wrapper class.

```
class TypeWrap
{
        public static void main(String args[])
        {
                Character              ch=new              Character('#');
                System.out.println("Character  is  "  +  ch.charValue());
                Boolean               b=new               Boolean(true);
                System.out.println("Boolean is " + b.booleanValue());
                Boolean             b1=new             Boolean("false");
                System.out.println("Boolean          is          "          +
                b1.booleanValue());
                Integer iOb=new Integer(12); //boxing
                int       i=iOb.intValue();       //unboxing
                System.out.println(i + " is same as " +
                iOb); Integer a=new Integer("21");
                int            x=a.intValue();
                System.out.println("x is " +
                x);                 String
                s=Integer.toString(25);
                System.out.println("s   is   "
                +s);
        }
}
```

c) **Type Wrapper:** They convert primitive data types into objects. Objects are needed if we wish modify the arguments passed into a method (because primitive types are passed by value).

Character(char ch)

d) **Marker Annotations:**

The only purpose is to mark a declaration. These annotations contain no members and do not consist any data. Thus, its presence as an annotation is sufficient. Since, marker interface contains no members, simply determining whether it is present or absent is sufficient. **@Override** , @Deprecated is an example of Marker Annotation.

Example: - @TestAnnotation()

2.a. Explain built in annotations in detail with necessary examples.

Built-In Java Annotations

There are several built-in annotations in java. Some annotations are applied to java code and some to other annotations.

Built-In Java Annotations used in java code

@Override

@SuppressWarnings

@Deprecated

Built-In Java Annotations used in other annotations

@Target

@Retention

@Inherited

@Documented

@Override

@Override annotation assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.

Sometimes, we does the silly mistake such as spelling mistakes etc. So, it is better to mark @Override annotation that provides assurity that method is overridden.

```
class Animal{
void eatSomething(){System.out.println("eating something");}
 }
 class Dog extends Animal{
 @Override
void  eatsomething(){System.out.println("eating  foods");}//should  be
eatSometh ing
 }
 class TestAnnotation1{
public static void main(String args[]){
Animal a=new Dog();
a.eatSomething();
 }}
```

Output:

Comple Time Error

@SuppressWarnings

@SuppressWarnings annotation: is used to suppress warnings issued by the compiler.

```
import java.util.*;
class TestAnnotation2{
 @SuppressWarnings("unchecked")
public static void main(String args[]){
ArrayList list=new ArrayList();
list.add("sonoo");
list.add("vimal");
list.add("ratan");
 for(Object obj:list)
System.out.println(obj);
 }
 }
```

Now no warning at compile time.

If you remove the @SuppressWarnings("unchecked") annotation, it will show warning at compile time because we are using non-generic collection.

@Deprecated

@Deprecated annoation marks that this method is deprecated so compiler prints warning. It informs user that it may be removed in the future versions. So, it is better not to use such methods.

```
class A{
void m(){
System.out.println("hello m");}
 @Deprecated
void n(){System.out.println("hello n");}
 }
 class TestAnnotation3{
public static void main(String args[]){
 A a=new A();
a.n();
 }}
```

At Compile Time:

```
Note: Test.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

At Runtime:

```
hello n
```

@Target

@Target tag is used to specify at which type, the annotation is used.

The java.lang.annotation.ElementType enum declares many constants to specify the type of element where annotation is to be applied such as TYPE, METHOD, FIELD etc.

2.b. What is Annotation? Explain how you obtain annotations at runtime by reflection.

Java Annotations allow us to add metadata information into our source code, Annotations were added to the java from JDK 5.

Annotations, does not change the actions of a program.

Thus, an annotation leaves the semantics of a program unchanged.

However, this information can be used by various tools during both development and deployment.

□ Annotations start with ‘**@**’.

□ Annotations do not change action of a compiled program.

□ Annotations help to associate *metadata* (information) to the program elements i.e. instance variables, constructors, methods, classes, etc.

□ Annotations are not pure comments as they can change the way a program is treated by compiler.

Reflection is an API which is used to examine or modify the behavior of methods, classes, interfaces at runtime.

- The required classes for reflection are provided under java.lang.reflect package.

Reflection can be used to get information about –

- **Class** The getClass() method is used to get the name of the class to which an object belongs.
- **Constructors** The getConstructors() method is used to get the public constructors of the class to which an object belongs.
- **Methods** The getMethods() method is used to get the public methods of the class to which an objects belongs.

```
import java.lang.annotation.*;
import java.lang.reflect.*;
// An annotation type declaration.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
String
str(); int
val();
}
class Meta {
// Annotate a method.
@MyAnno(str = "Annotation Example", val = 100)
public static void myMeth() {
Meta ob = new Meta();
// Obtain the annotation for this method
// and display the values of the members.

try {

// First, get a Class object that represents

// this class.
Class c = ob.getClass();
// Now, get a Method object that represents
// this method.
Method m = c.getMethod("myMeth");
// Next, get the annotation for this class.
MyAnno anno = m.getAnnotation(MyAnno.class);
// Finally, display the values.
System.out.println(anno.str() + " " + anno.val());
} catch (NoSuchMethodException exc) {
```

```
System.out.println("Method Not Found.");
}
}
public static void main(String args[]) {
myMeth();
}
}
```

The output from the program is shown here:

```
Annotation Example 100
```

3.a. Explain the following collection interfaces with xample program.

i) Queue

ii)Sorted Set

ii)The SortedSet Interface

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:

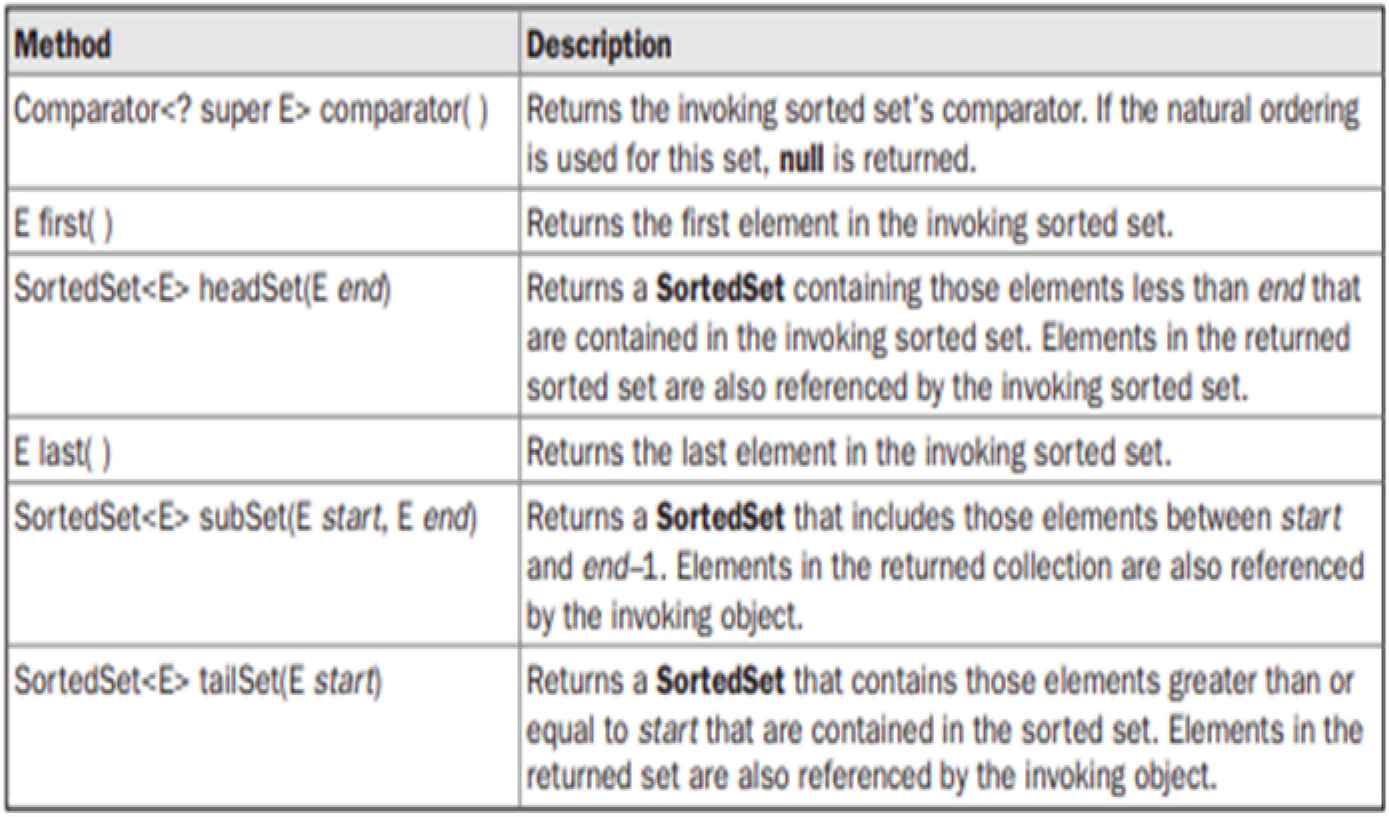
interface SortedSet<E>

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

In addition to those methods defined by Set, the SortedSet interface declares the methods summarized in Table 17-3. Several methods throw a NoSuchElementException when no items are contained in the invoking set. A ClassCastException is thrown when an object is incompatible with the elements in a set. A NullPointerException is thrown if an attempt is made to use a null object and null is not allowed in the set. An IllegalArgumentException

is thrown if an invalid argument is used.

SortedSet defines several methods that make set processing more convenient. To obtain the first object in the set, call first( ). To get the last element, use last( ). You can obtain a subset of a sorted set by calling subSet( ), specifying the first and last object in the set. If you need the subset that starts with the first element in the set, use headSet( ). If you want the subset that ends the set, use tailSet( ).

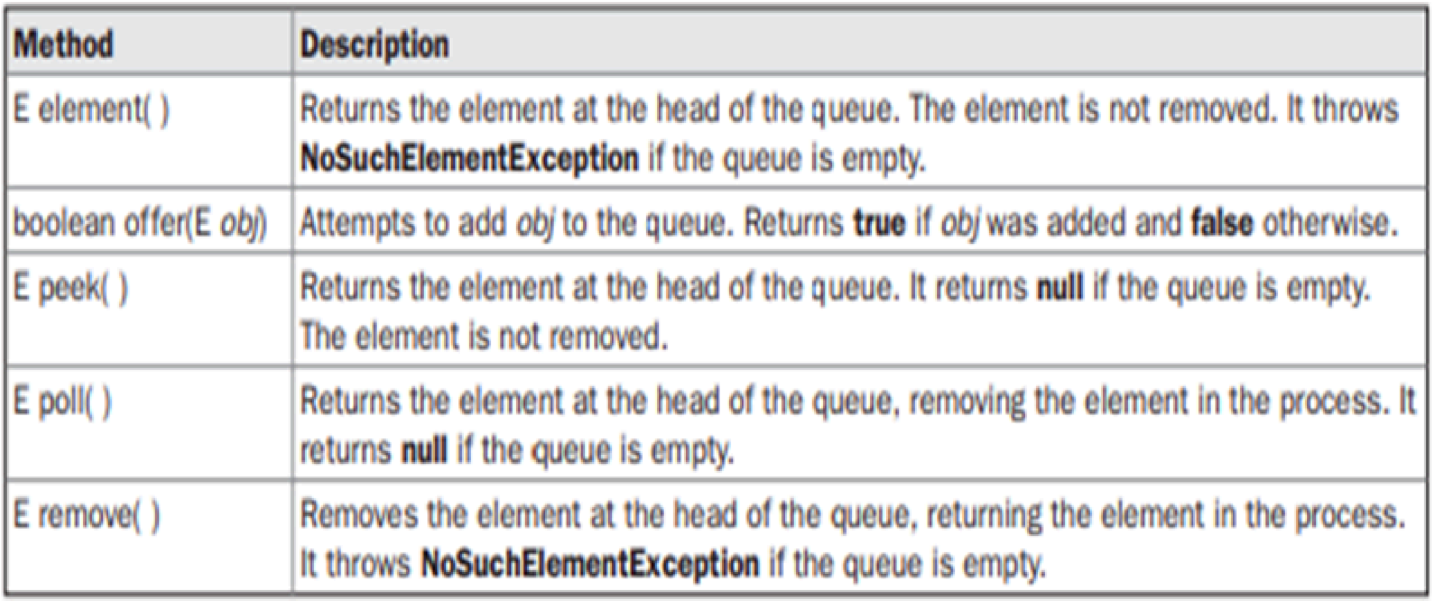


Method	Description
Comparator<? super E> comparator( )	Returns the invoking sorted set's comparator. If the natural ordering is used for this set, **null** is returned.
E first( )	Returns the first element in the invoking sorted set.
SortedSet<E> headSet(E *end*)	Returns a **SortedSet** containing those 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.
E last( )	Returns the last element in the invoking sorted set.
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.
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.

TABLE 17-3 The Methods Defined by **SortedSet**

i)The Queue Interface

The Queue interface extends Collection and declares the behavior of a queue, which is often a first-in, first-out list. However, there are types of queues in which the ordering is based upon other criteria. Queue is a generic interface that has this declaration: interface Queue<E>



Method	Description
E element( )	Returns the element at the head of the queue. The element is not removed. It throws **NoSuchElementException** if the queue is empty.
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. It throws **NoSuchElementException** if the queue is empty.

TABLE 17-5 The Methods Defined by **Queue**

Several methods throw a ClassCastException when an object is incompatible with the elements in the queue. A NullPointerException is thrown if an attempt is made to store a null object and null elements are not allowed in the queue. 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 queue that is full. A NoSuchElementException is thrown if an attempt is made to remove an element from an empty queue.

3.b. Describe ArrayList class and explain its constructors. Demonstrate its usage with an example program.

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. In Java, standard arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold. But, sometimes, we may not know until run time precisely how large an array we need. To handle this situation, the Collections Framework defines ArrayList. In essence, an ArrayList is a variable-length array of object references. That is, an ArrayList can dynamically increase or decrease in size. Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array can be shrunk.

ArrayList has the constructors shown here:

ArrayList( )

ArrayList(Collection<? extends E> c)

ArrayList(int capacity)

The first constructor builds an empty array list. The second constructor builds an array list that is initialized with the elements of the collection c. The third constructor builds an array list that has the specified initial capacity. The capacity is the size of the underlying array that is used to store the elements. The capacity grows automatically as elements are added to an array list.

```
// Demonstrate ArrayList.
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());
// Add elements to the array list.
al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D");
al.add("F");
al.add(1, "A2");
System.out.println("Size of al after additions: " +

al.size());

// Display the array list.

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. a. Discuss the following map classes with example.
i)Hash Map
ii)Tree Map

The HashMap class extends AbstractMap and implements the Map interface. It uses a hash table to store the map. This allows the execution time of get( ) and put( ) to remain constant even for large sets. HashMap is a generic class that has this declaration:

class HashMap<K, V>

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

The following constructors are defined:

HashMap( )

HashMap(Map<? extends K, ? extends V> m)

HashMap(int capacity)

HashMap(int capacity, float fillRatio)

The first form constructs a default hash map. The second form initializes the hash map by using the elements of m. The third form initializes the capacity of the hash map to capacity. The fourth form initializes both the capacity and fill ratio of the hash map by using its arguments.

The meaning of capacity and fill ratio is the same as for HashSet, described earlier. The default capacity is 16. The default fill ratio is 0.75.

HashMap implements Map and extends AbstractMap. It does not add any methods of its own.

The TreeMap Class

The TreeMap class extends AbstractMap and implements the NavigableMap interface. It creates maps stored in a tree structure. A TreeMap provides an efficient means of storing key/value pairs in sorted order and allows rapid retrieval. You should note that, unlike a hash map, a tree map guarantees that its elements will be sorted in ascending key order.

TreeMap is a generic class that has this declaration:

```
class TreeMap<K, V>
```

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

The following TreeMap constructors are defined:

```
TreeMap( )
TreeMap(Comparator<? super K> comp)
TreeMap(Map<? extends K, ? extends V> m)
TreeMap(SortedMap<K, ? extends V> sm)
```

The first form constructs an empty tree map that will be sorted by using the natural order of its keys. The second form constructs an empty tree-based map that will be sorted by using the Comparator comp. (Comparators are discussed later in this chapter.) The third form initializes a tree map with the entries from m, which will be sorted by using the natural order of the keys. The fourth form initializes a tree map with the entries from sm, which will be sorted in the same order as sm.

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

4.b. List and explain any five collection algorithms. Demonstrate various algorithms with an example program.

Method	Description
static <T> boolean addAll(Collection <? super T> *c*, T ... *elements*)	Inserts the elements specified by *elements* into the collection specified by *c*. Returns **true** if the elements were added and **false** otherwise.
static <T> Queue<T> asLifoQueue(Deque<T> *c*)	Returns a last-in, first-out view of *c*. (Added by Java SE 6.)
static <T> int binarySearch(List<? extends T> *list*, T *value*, Comparator<? super T> *c*)	Searches for *value* in *list* ordered according to *c*. Returns the position of *value* in *list*, or a negative value if *value* is not found.
static <T> int binarySearch(List<? extends Comparable<? super T>> *list*, T *value*)	Searches for *value* in *list*. The list must be sorted. Returns the position of *value* in *list*, or a negative value if *value* is not found.
static <E> Collection<E> checkedCollection(Collection<E> *c*, Class<E> *t*)	Returns a run-time type-safe view of a collection. An attempt to insert an incompatible element will cause a **ClassCastException**.

static <E> List<E> checkedList(List<E> *c*, Class<E> *t*)	Returns a run-time type-safe view of a **List**. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <K, V> Map<K, V> checkedMap(Map<K, V> *c*, Class<K> *keyT*, Class<V> *valueT*)	Returns a run-time type-safe view of a **Map**. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <E> List<E> checkedSet(Set<E> *c*, Class<E> *t*)	Returns a run-time type-safe view of a **Set**. An attempt to insert an incompatible element will cause a **ClassCastException**.

```
//       Demonstrate       various
algorithms. import java.util.*;

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

    //   Create   and   initialize   linked   list.
    LinkedList<Integer>       ll       =       new
    LinkedList<Integer>(); ll.add(-8);
    ll.add(20);
    ll.add(-20);
    ll.add(8);

    //   Create   a   reverse   order   comparator.
    Comparator<Integer>            r            =
    Collections.reverseOrder();

    //   Sort   list   by   using   the
    comparator.    Collections.sort(ll,
    r);

    System.out.print("List sorted in reverse:
    "); for(int i : ll)
      System.out.print(i+  "  ");

    System.out.println();
```

```
//      Shuffle      list.
Collections.shuffle(ll);

//    Display    randomized    list.
System.out.print("List    shuffled:
"); for(int i : ll)
 System.out.print(i  +  "  ");

System.out.println();
```

```
        System.out.println("Minimum:           "           +
        Collections.min(ll)); System.out.println("Maximum:
        " + Collections.max(ll));
      }
    }
```

5.a. Illustrate the use of the following methods with an example

i)insert

ii)append

iii)replace

iv)substring

i)append()

The append( ) method concatenates the string representation of any other type of data to the end of the invoking StringBuffer object. It has several overloaded versions. Here are a few of its forms:

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

String.valueOf( ) is called for each parameter to obtain its string representation. The result is appended to the current StringBuffer object. The buffer itself is returned by each version of append( ). This allows subsequent calls to be chained together, as shown in the following example:

```
// Demonstrate append().
class appendDemo {
public static void main(String args[]) {
String s;
int a = 42;
StringBuffer sb = new StringBuffer(40);
s = sb.append("a = ").append(a).append("!").toString();
System.out.println(s);
}
}
```

ii)insert()

The insert( ) method inserts one string into another. It is overloaded to accept values of all the simple types, plus Strings, Objects, and CharSequences. Like append( ), it calls String.valueOf( ) to obtain the string representation of the value it is called with. This string is then inserted into the invoking StringBuffer object. These are a few of its forms:

StringBuffer insert(int index, String str)
StringBuffer insert(int index, char ch)
StringBuffer insert(int index, Object obj)

Here, index specifies the index at which point the string will be inserted into the invoking StringBuffer object.

The following sample program inserts “like” between “I” and “Java”:

```
// Demonstrate insert().
class insertDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("I Java!");
sb.insert(2, "like ");
System.out.println(sb);
}
}
```

iii)reverse()

You can reverse the characters within a StringBuffer object using reverse( ), shown here:

StringBuffer reverse( )

This method returns the reversed object on which it was called.

The following program demonstrates reverse( ):

```
// Using reverse() to reverse a StringBuffer.
class ReverseDemo {
public static void main(String args[]) {
StringBuffer s = new StringBuffer("abcdef");
System.out.println(s);
s.reverse();
System.out.println(s);
}
}
```

iv)replace()

You can replace one set of characters with another set inside a StringBuffer object by calling replace( ). Its signature is shown here:

StringBuffer replace(int startIndex, int endIndex, String str)

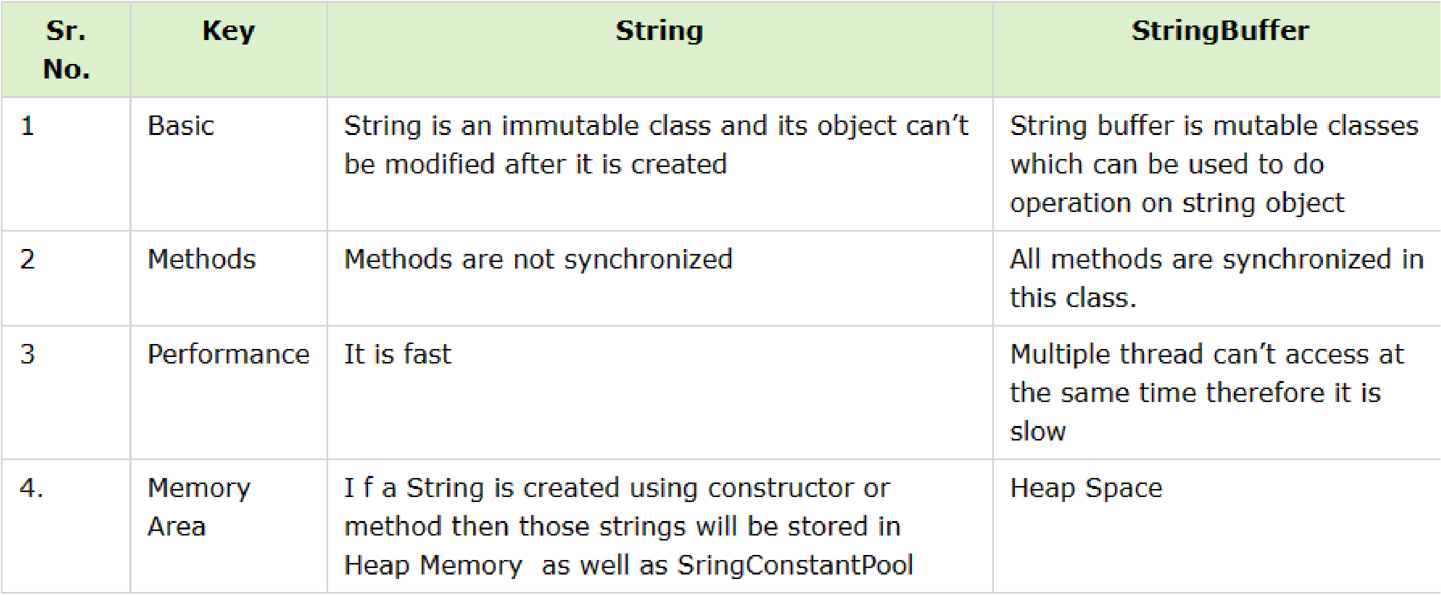
The substring being replaced is specified by the indexes startIndex and endIndex. Thus, the substring at startIndex through endIndex1 is replaced. The replacement string is passed in – str.

The resulting StringBuffer object is returned.

The following program demonstrates replace( ):

```
// Demonstrate replace()
class replaceDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("This is a test.");
sb.replace(5, 7, "was");
System.out.println("After replace: " + sb); } }
```

5.b. Differentiate between String and StringBuffer classes. Write a program to demonstrate different constructors of String class.



Sr. No.	Key	String	StringBuffer
1	Basic	String is an immutable class and its object can't be modified after it is created	String buffer is mutable classes which can be used to do operation on string object
2	Methods	Methods are not synchronized	All methods are synchronized in this class.
3	Performance	It is fast	Multiple thread can't access at the same time therefore it is slow
4.	Memory Area	I f a String is created using constructor or method then those strings will be stored in Heap Memory as well as SringConstantPool	Heap Space

```
public class StringConstructorDemo {
    public static void main(String[] args) {
        // Creating an empty string using the default constructor
        String emptyString = new String();
        System.out.println("Empty String: " + emptyString);

        // Creating a string from another string
        String originalString = "Hello, World!";
        String copiedString = new String(originalString);
        System.out.println("Copied String: " + copiedString);

        // Creating a string from a byte array
        byte[] byteArray = {72, 101, 108, 108, 111}; // ASCII values for "Hello"
        String fromByteArray = new String(byteArray);
        System.out.println("String from Byte Array: " + fromByteArray);

        // Creating a string from a character array
        char[] charArray = {'J', 'a', 'v', 'a'};
        String fromCharArray = new String(charArray);
        System.out.println("String from Character Array: " + fromCharArray);

        // Creating a string from Unicode code points
        int[] codePoints = {72, 101, 108, 108, 111}; // Unicode code points for "Hello"
        String fromCodePoints = new String(codePoints, 0, codePoints.length);
        System.out.println("String from Code Points: " + fromCodePoints);

        // Creating a string from a StringBuffer
        StringBuffer stringBuffer = new StringBuffer("DataFlair");
        String fromStringBuffer = new String(stringBuffer);
        System.out.println("String from StringBuffer: " + fromStringBuffer);

        // Creating a string from a StringBuilder
        StringBuilder stringBuilder = new StringBuilder("Java");
        String fromStringBuilder = new String(stringBuilder);
        System.out.println("String from StringBuilder: " + fromStringBuilder);
    }
}
```

6.a. Explain the following string comparison methods with an example:

i)equals()

ii)compareTo()

iii)==

iv) equalsIgnoreCase()

i)equals()

The equals( ) method compares the characters inside a String object. String s1 = "Hello";

String s2 = new String(s1); System.out.println(s1.equals(s2)); //true System.out.println((s1 == s2)); //false

ii)compareTo(): This method is used to check whether a string is less than, greater than or equal to the other string. The meaning of less than, greater than refers to the dictionary order (based on Unicode). It has this general form:

int compareTo(String str)

This method will return 0, if both the strings are same. Otherwise, it will return the difference between the ASCII values of first non-matching character. If you want to ignore case differences when comparing two strings, use compareToIgnoreCase(), as shown here:

int compareToIgnoreCase(String str)

iii)==:

The == operator compares two object references to see whether they refer to the same instance.

iv) equalsIgnoreCase()

To compare two strings for equality by ignoring the case

boolean equalsIgnoreCase(String str)

6.b. Illustrate character extraction methods with examples.

charAt() : This method is used to extract a single character from a String. It has this general form: char charAt(int where)

Here, where is the index of the character that you want to obtain. The value of where must be nonnegative and specify a location within the string. For example,

char ch;

ch= “Hello”.charAt(1); //ch now contains e

getChars() : If you need to extract more than one character at a time, you can use this method. It has the following general form:

void getChars(int sourceStart, int sourceEnd, char target[], int targetStart)

getBytes() : It is an alternative to getChars() that stores the characters in an array of bytes. It uses the default character-to-byte conversions provided by the platform. Here is its simplest form:

byte[ ] getBytes( )

Other forms of getBytes( ) are also available. getBytes( ) is most useful when you are exporting a String value

into an environment that does not support 16-bit Unicode characters. For example, most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

toCharArray() : If you want to convert all the characters in a String object into a character array, the easiest way is to call toCharArray( ). It returns an array of characters for the entire string. It has this general form:

char[ ] toCharArray( )

7.a. Explain the lifecycle of a servlet.

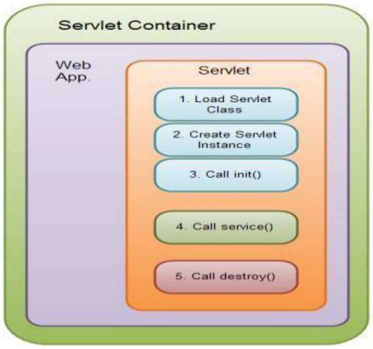
Java Servlets are programs that run on a Web or Application server

🎬 Act as a middle layer between a request coming from a Web browser or other HTTP client and databases or applications on the HTTP server.

🎬 Using Servlets, you can collect input from users through web page forms, present records from a database or another source, and create web pages dynamically.

🎬 Servlets are server side components that provide a powerful mechanism for developing web applications.

A servlet life cycle can be defined as the entire process from its creation till the destruction. The following are the paths followed by a servlet



- The servlet is initialized by calling the init () method.
- The servlet calls service() method to process a client's request.
- The servlet is terminated by calling the destroy() method.
- Finally, servlet is garbage collected by the garbage collector of the JVM.

Now let us discuss the life cycle methods in details.

The init() method :

- The init method is designed to be called only once.
- It is called when the servlet is first created, and not called again for each user request. So, it is used for one-time initializations, just as with the init method of applets.
- The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started.
- The init() method simply creates or loads some data that will be used throughout the life of the servlet.

The init method definition looks like this:

```
public void init() throws ServletException {
// Initialization code...
}
```

The service() method :

- The service() method is the main method to perform the actual task.
- The servlet container (i.e. web server) calls the service() method to handle requests coming from the client(browsers) and to write the formatted response back to the client.
- Each time the server receives a request for a servlet, the server spawns a new thread and calls service. The service() method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls doGet, doPost, doPut, doDelete, etc. methods as appropriate.

Signature of service method:

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

- The service () method is called by the container and service method invokes doGe, doPost, doPut, doDelete, etc.methods as appropriate.
- So you have nothing to do with service() method but you override either doGet() or doPost() depending on what type of request you receive from the client.
- The doGet() and doPost() are most frequently used methods with in each service request.

Here is the signature of these two methods.

The doGet() Method

A GET request results from a normal request for a URL or from an HTML form that has no METHOD specified and it should be handled by doGet() method.

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

```
// Servlet code
}
```

The doPost() Method

A POST request results from an HTML form that specifically lists POST as the METHOD and it should be handled by doPost() method.

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
// Servlet code
}
```

The destroy() method :

- The destroy() method is called only once at the end of the life cycle of a servlet.
- This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities.
- After the destroy() method is called, the servlet object is marked for garbage collection.

The destroy method definition looks like this:

```
public void destroy() {
// Finalization code...
}
```

7.b. What is a cookie? Explain the working of a cookie in java with code snippets.

Cookies are small bits of textual information that a web server sends to a browser and that the browser later returns unchanged when visiting the same web site or domain Sending cookies to the client:

1. Creating a cookie object

- Cookie():constructs a cookie.
- Cookie(String name, String value)constructs a cookie with a specified name and value. EX:

```
Cookie ck=new Cookie("user",”mca");
```

2. Setting the maximum age

setMaxAge() is used to specify how long (in seconds) the cookie should be valid. Ex:cookie.setMaxAge(60*60*24);

3. Placing the cookie into the HTTP response headers.

We use response.addCookie to add cookies in the HTTP response header as follows: response.addCookie(cookie);

Reading cookies from the client:

1. Call request.getCookies(). This yields an array of cookie objects.
2. Loop down the array, calling getName on each one until you find the cookie of interest.

Ex:

```
String cookieName=“userID”;
Cookie[]
cookies=request.getCookies();
If(cookies!=null)
{
for(int
i=0;i<cookies.length;i++){
Cookie cookie=cookies[i];
if(cookieName.equals(cookie.getName
())){
doSomethingwith(cookie.getValue());
}}}
```

8.a. Define JSP. Explain different types of JSP tags by taking suitable examples.

1. <u>**JSP scriptlet tag**</u> A scriptlet tag is used to execute java source code in JSP.

<% java source code %>

In this example, we are displaying a welcome message.

```
<html>
<body>
<% out.print("welcome to jsp"); %>
</body>
```

</html>

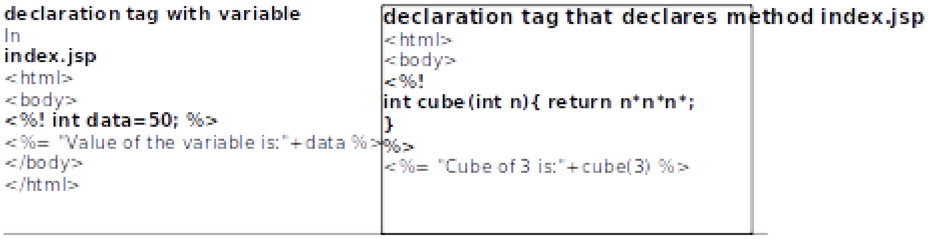
2. **JSP Declaration Tag**

The **JSP declaration tag** is used *to declare variables, objects and methods*.

The code written inside the jsp declaration tag is placed outside the service() method of auto generated servlet.

So it doesn't get memory at each request.

<%! field or method declaration %>



```
declaration tag with variable
In
index.jsp
<html>
<body>
<%! int data=50; %>
<%= "Value of the variable is:"+data %>
</body>
</html>
```

```
declaration tag that declares method index.jsp
<html>
<body>
<%!
int cube(int n){ return n*n*n*;
}
%>
<%= "Cube of 3 is:"+cube(3) %>
```

JSP Expression Tag

Expression Tag is used to print out java language expression that is put between the tags. An expression tag can hold any java language expression that can be used as an argument to the out.print() method.

Syntax of Expression Tag
<%= *JavaExpression* %>
<%= (2*5) %> //note no ; at end of statement.

1. **JSP directives**

The jsp directives are messages that tells the web container how to translate a JSP page into the corresponding servlet.

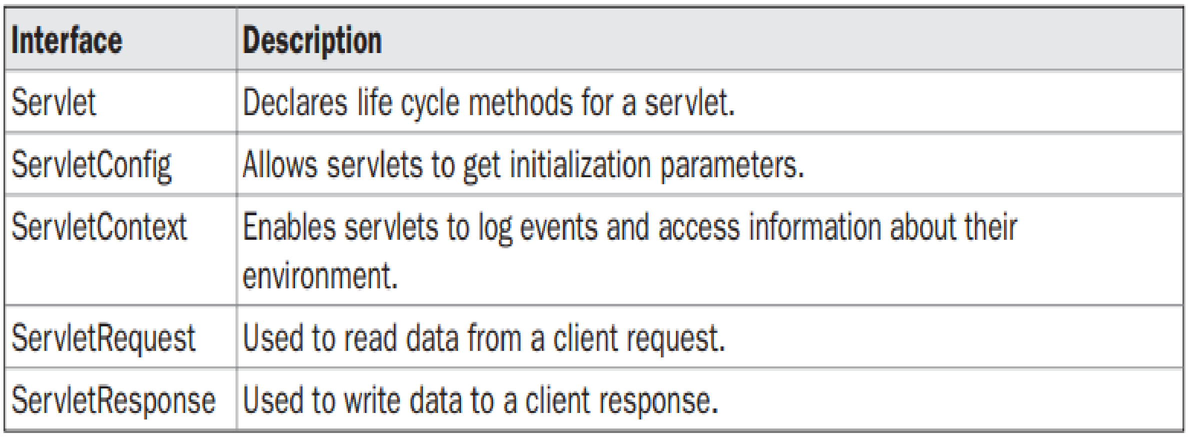
Syntax <%@ directive attribute="value" %> There are three types of directives:

1. **import directive**
2. **include directive**
3. **taglib directive**

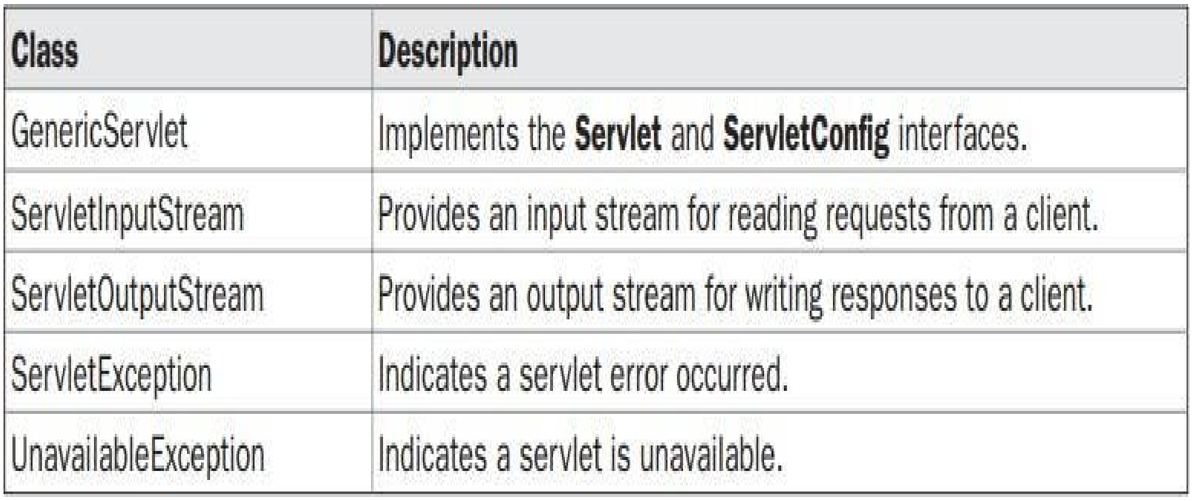
4. **JSP Comments**

JSP comment marks text or statements that the JSP container should ignore. syntax of the JSP comments <%- - This is JSP comment - -%>

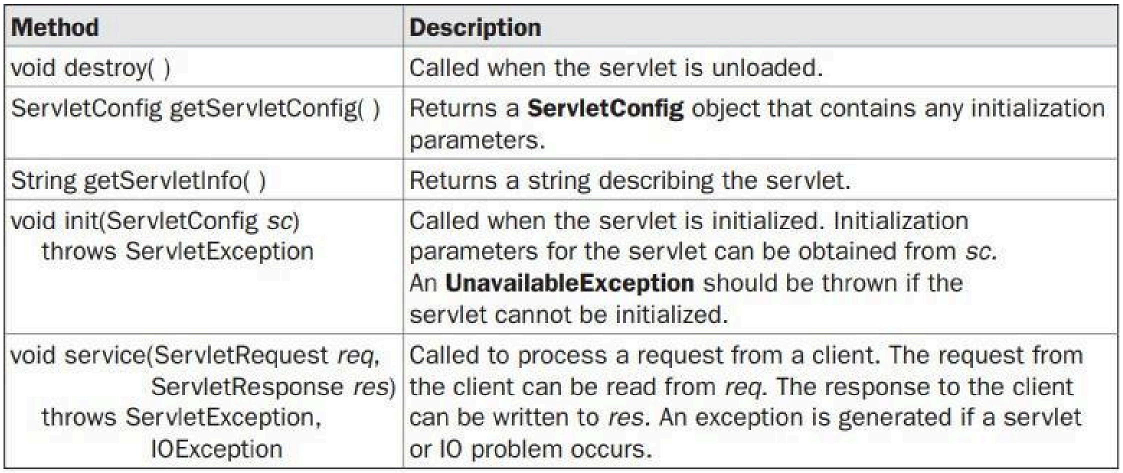
8.b. List and explain core classes and interfaces in javax.servlet package.



Interface	Description
Servlet	Declares life cycle methods for a servlet.
ServletConfig	Allows servlets to get initialization parameters.
ServletContext	Enables servlets to log events and access information about their environment.
ServletRequest	Used to read data from a client request.
ServletResponse	Used to write data to a client response.



Class	Description
GenericServlet	Implements the **Servlet** and **ServletConfig** interfaces.
ServletInputStream	Provides an input stream for reading requests from a client.
ServletOutputStream	Provides an output stream for writing responses to a client.
ServletException	Indicates a servlet error occurred.
UnavailableException	Indicates a servlet is unavailable.

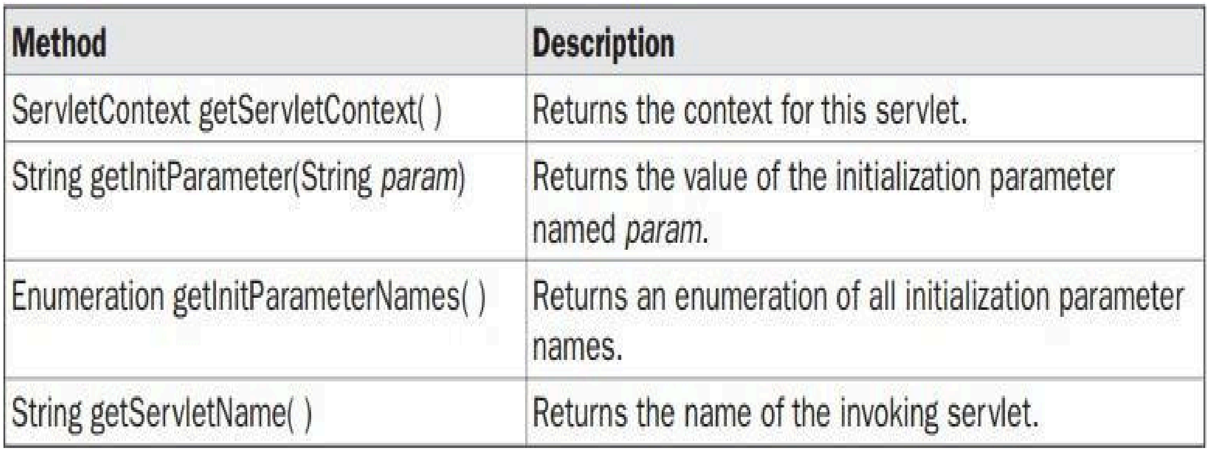


Method	Description
void destroy( )	Called when the servlet is unloaded.
ServletConfig getServletConfig( )	Returns a **ServletConfig** object that contains any initialization parameters.
String getServletInfo( )	Returns a string describing the servlet.
void init(ServletConfig *sc*) throws ServletException	Called when the servlet is initialized. Initialization parameters for the servlet can be obtained from *sc*. An **UnavailableException** should be thrown if the servlet cannot be initialized.
void service(ServletRequest *req*, ServletResponse *res*) throws ServletException, IOException	Called to process a request from a client. The request from the client can be read from *req*. The response to the client can be written to *res*. An exception is generated if a servlet or IO problem occurs.

TABLE 31-1 The Methods Defined by **Servlet**

The ServletConfig Interface

The **ServletConfig** interface allows a servlet to obtain configuration data when it is loaded. The methods declared by this interface are summarized here:



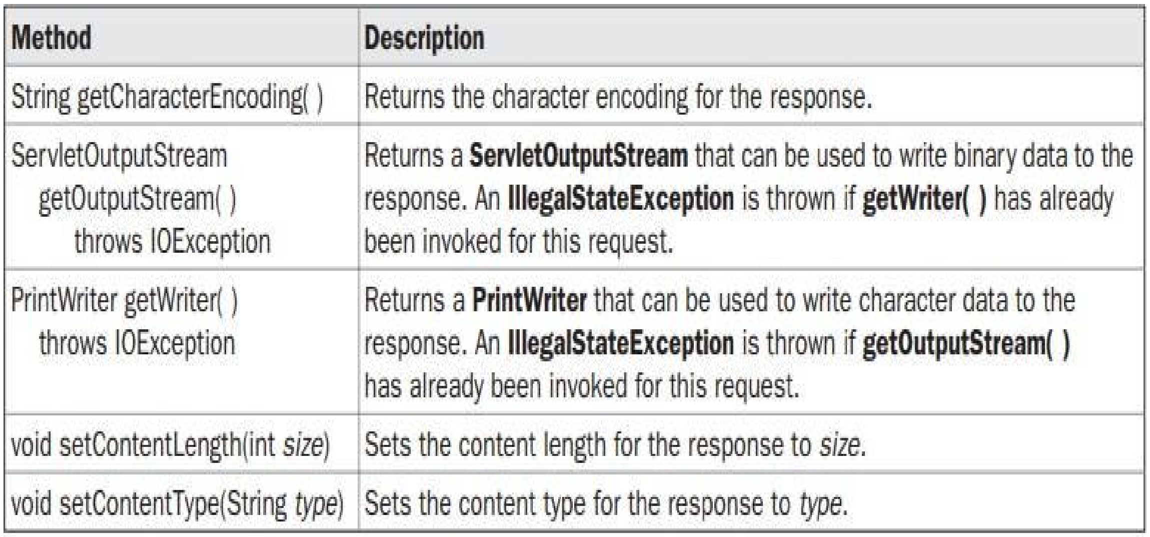
Method	Description
ServletContext getServletContext( )	Returns the context for this servlet.
String getInitParameter(String *param*)	Returns the value of the initialization parameter named *param*.
Enumeration getInitParameterNames( )	Returns an enumeration of all initialization parameter names.
String getServletName( )	Returns the name of the invoking servlet.

Method	Description
Object getAttribute(String *attr*)	Returns the value of the server attribute named *attr*.
String getMimeType(String *file*)	Returns the MIME type of *file*.
String getRealPath(String *vpath*)	Returns the real path that corresponds to the virtual path *vpath*.
String getServerInfo( )	Returns information about the server.
void log(String *s*)	Writes *s* to the servlet log.
void log(String *s*, Throwable *e*)	Writes *s* and the stack trace for *e* to the servlet log.
void setAttribute(String *attr*, Object *val*)	Sets the attribute specified by *attr* to the value passed in *val*.

TABLE 31-2 Various Methods Defined by **ServletContext**

Method	Description
String getCharacterEncoding( )	Returns the character encoding for the response.
ServletOutputStream getOutputStream( ) throws IOException	Returns a **ServletOutputStream** that can be used to write binary data to the response. An **IllegalStateException** is thrown if **getWriter( )** has already been invoked for this request.
PrintWriter getWriter( ) throws IOException	Returns a **PrintWriter** that can be used to write character data to the response. An **IllegalStateException** is thrown if **getOutputStream( )** has already been invoked for this request.
void setContentLength(int *size*)	Sets the content length for the response to *size*.
void setContentType(String *type*)	Sets the content type for the response to *type*.

TABLE 31-4 Various Methods Defined by **ServletResponse**



Method	Description
String getCharacterEncoding( )	Returns the character encoding for the response.
ServletOutputStream getOutputStream( ) throws IOException	Returns a **ServletOutputStream** that can be used to write binary data to the response. An **IllegalStateException** is thrown if **getWriter( )** has already been invoked for this request.
PrintWriter getWriter( ) throws IOException	Returns a **PrintWriter** that can be used to write character data to the response. An **IllegalStateException** is thrown if **getOutputStream( )** has already been invoked for this request.
void setContentLength(int *size*)	Sets the content length for the response to *size*.
void setContentType(String *type*)	Sets the content type for the response to *type*.

TABLE 31-4 Various Methods Defined by **ServletResponse**

Generic Servlet Class:

- The GenericServlet class provides implementations of the basic life cycle methods for a servlet.
- GenericServlet implements the Servlet and ServletConfig interfaces. In addition, a method to append a string to the server log file is available.
- The signatures of this method are
  shown here: void log(String s)
  void log(String s, Throwable e)

Here, s is the string to be appended to the log, and e is an exception that occurred.

Servlet Input Stream:

- The ServletInputStream class extends InputStream.
- It is implemented by the servlet container and provides an input stream that a servlet developer can use to read the data from a client request.
- It defines the default constructor.
- A method is provided to read bytes from the stream.
  int readLine(byte[ ] buffer, int offset, int size) throws IOException

ServletOutputStream:

- The ServletOutputStream class extends OutputStream.
- It is implemented by the servlet container and provides an output stream that a servlet developer can use to write data to a client response.
- A default constructor is defined.
- It also defines the print( ) and println( ) methods, which output data to the stream. javax.servlet defines two exceptions.
- The first is ServletException, which indicates that a servlet problem has occurred.

- The second is UnavailableException, which extends ServletException. It indicates that a servlet is unavailable.

9.a. Describe the various steps of the JDBC process with code snippets.

Seven Basic Steps in Using JDBC

1. Load the Driver
2. Define the Connection UR
3. Establish the Connection
4. Create a Statement Object
5. Execute a query
6. Process the results
7. Close the Connection

1. Load the JDBC driver

When a driver class is first loaded, it registers itself with the driver Manager Therefore, to register a driver, just load it!

Example:

String driver = “sun.jdbc.odbc.JdbcOdbcDriver”; Class.forName(driver); Or

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

2. Define the Connection URL

jdbc : subprotocol : source

each subprotocol has its own syntax for the source

jdbc : odbc : DataSource

Ex: jdbc : odbc : Employee

jdbc:msql://host[:port]/database

Ex: jdbc:msql://foo.nowhere.com:4333/accounting

3. Establish the Connection

rns a Connection object

to the database from the DBMS.

nection() if access is granted; else getConnection() throws a SQLException.

the database.

String url = jdbc : odbc : Employee;

Connection c = DriverManager.getConnection(url,userID,password);

access to the database.

Properties or Sometimes DBMS grants access to a database to anyone without using username or password.

Ex: Connection c = DriverManager.getConnection(url) ;

4. Create a Statement Object

A Statement object is used for executing a static SQL statement and obtaining the results produced by it.

Statement stmt = con.createStatement();

This statement creates a Statement object, stmt that can pass SQL statements to the DBMS using connection, con.

5. Execute a query

Execute a SQL query such as SELECT, INSERT, DELETE, UPDATE Example String SelectStudent= "select * from STUDENT";

6. Process the results

table rows are retrieved in sequence.

7. Close the Connection

connection.close();

stpone this step if additional database operations are expected

```
package j2ee.p9;
import java.sql.*;
import java.io.*;
public class Studentdata {
        public static void main(String[] args) {
                Connection con;
                PreparedStatement pstmt;
                Statement stmt;
                ResultSet rs;
                String uname, pword;
                Integer marks,count;
                try
                {
Class.forName("com.mysql.jdbc.Driver"); // type1 driver
                        try{

        con=DriverManager.getConnection("jdbc:mysql://127.0.0.1/mca","roo
t","system"); //  type1 access connection
BufferedReader br=new BufferedReader(new
InputStreamReader(System.in));
                        do
                        {


System.out.println("\n1. Insert.\n2. Select.\n3. Update.\n4.
Delete.\n5. Exit.\nEnter your choice:");
```

```
int choice=Integer.parseInt(br.readLine());
                                                  switch(choice)
                                                  {
case 1: System.out.print("Enter UserName :");
uname=br.readLine();
System.out.print("Enter Password :");
pword=br.readLine();
pstmt=con.prepareStatement("insert into student
                values(?,?)");
pstmt.setString(1,uname);
pstmt.setString(2,pword);
pstmt.execute();
System.out.println("\nRecord Inserted
                successfully.");
                                                          break;
                                                          case 2:
stmt=con.createStatement();
rs=stmt.executeQuery("select *from student");
if(rs.next())
{
System.out.println("User Name\tPassword\n-----
                ----------------------------");
do
{
uname=rs.getString(1);
pword=rs.getString(2);

System.out.println(uname+"\t"+pword);
}while(rs.next());
}
else
System.out.println("Record(s) are not
                available in database.");
break;
case 3:
System.out.println("Enter User Name to
                update :");
uname=br.readLine();
System.out.println("Enter new password
                :");
pword=br.readLine();
stmt=con.createStatement();
count=stmt.executeUpdate("update
                student set password='"+pword+"'where username='"+uname+"'");
System.out.println("\n"+count+" Record
                Updated.");
break;
case 4: System.out.println("Enter User Name to
                delete record:");
uname=br.readLine();
stmt=con.createStatement();
count=stmt.executeUpdate("delete from
                student where username='"+uname+"'");


if(count!=0)
System.out.println("\nRecord
                "+uname+" has deleted.");
else
System.out.println("\nInvalid
                USN, Try again.");
break;

case 5: con.close(); System.exit(0);
default: System.out.println("Invalid choice, Try
                again.");
}//close of switch
                                                        }while(true);
}//close of nested try
catch(SQLException e2)
                                                        {
System.out.println(e2);
                                                        }
catch(IOException e3)
                                                        {
System.out.println(e3);
                                                        }
                                        }//close of outer try
                                        catch(ClassNotFoundException e1)
                                        {
```

```
                                System.out.println(e1);
                        }
                }
        }
```

9.b Explain prepared statement and callable statement in JDBC with example.

The preparedStatement object allows you to execute parameterized queries. A SQL query can be precompiled and executed by using the PreparedStatement object. · Ex: Select * from publishers where pub_id=?

Here a query is created as usual, but a question mark is used as a placeholder for a value· that is inserted into the query after the query is compiled.

The preparedStatement() method of Connection object is called to return the· PreparedStatement object.

Ex: PreparedStatement stat; stat= con.prepareStatement(“select * from publisher where pub_id=?”)

```
import java.sql.*;
public class JdbcDemo {
public static void main(String args[]){
try{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con=DriverManager.getConnection("jdbc:odbc:MyDataSource","khutub","");
PreparedStatement pstmt;
pstmt= con.prepareStatement("select * from employee whereUserName=?");
pstmt.setString(1,"khutub");
ResultSet rs1=pstmt.executeQuery();
while(rs1.next()){
System.out.println(rs1.getString(2));
}
} // end of try
catch(Exception e){System.out.println("exception"); }
} //end of main
} // end of class
```

Callable Statement:

The CallableStatement object is used to call a stored procedure from within a J2EE object. A Stored procedure is a block of code and is identified by a unique name.

The type and style of code depends on the DBMS vendor and can be written in PL/SQL Transact-SQL, C, or other programming languages.

IN, OUT and INOUT are the three parameters used by the CallableStatement object to call a stored procedure.

The IN parameter contains any data that needs to be passed to the stored procedure and· whose value is assigned using the setxxx() method.

The OUT parameter contains the value returned by the stored procedures. The OUT parameters must be registered using the registerOutParameter() method, later retrieved by using the getxxx()

The INOUT parameter is a single parameter that is used to pass information to the stored procedure and retrieve information from the stored procedure.

```
Connection con;
try{
String query = "{CALL LastOrderNumber(?))}";
CallableStatement stat = con.prepareCall(query);
stat.registerOutParameter( 1 ,Types.VARCHAR);
stat.execute();
String lastOrderNumber = stat.getString(1);
stat.close();
}
catch (Exception e){}
```

10.a. List and explain JDBC Driver Types.

Type 1: JDBC-to-ODBC Driver

- Microsoft created ODBC (Open Database Connection), which is the basis from which Sun created JDBC. Both have similar driver specifications and an API.
- The JDBC-to-ODBC driver, also called the JDBC/ODBC Bridge, is used to translate DBMS calls between the JDBC specification and the ODBC specification.
- MS Access and SQL Server contains ODBC driver written in C language using pointers, but java does not support the mechanism to handle pointers.
- So JDBC-ODBC Driver is created as a bridge between the two so that JDBC-ODBC bridge driver translates the JDBC API to the ODBC API.

➔ **Type-1 ODBC Driver for MS Access and SQL Server**

Drawbacks of Type-I Driver:

- ODBC binary code must be loaded on each client.
- Transaction overhead between JDBC and ODBC.
- It doesn't support all features of Java.
- It works only under Microsoft, SUN operating systems.

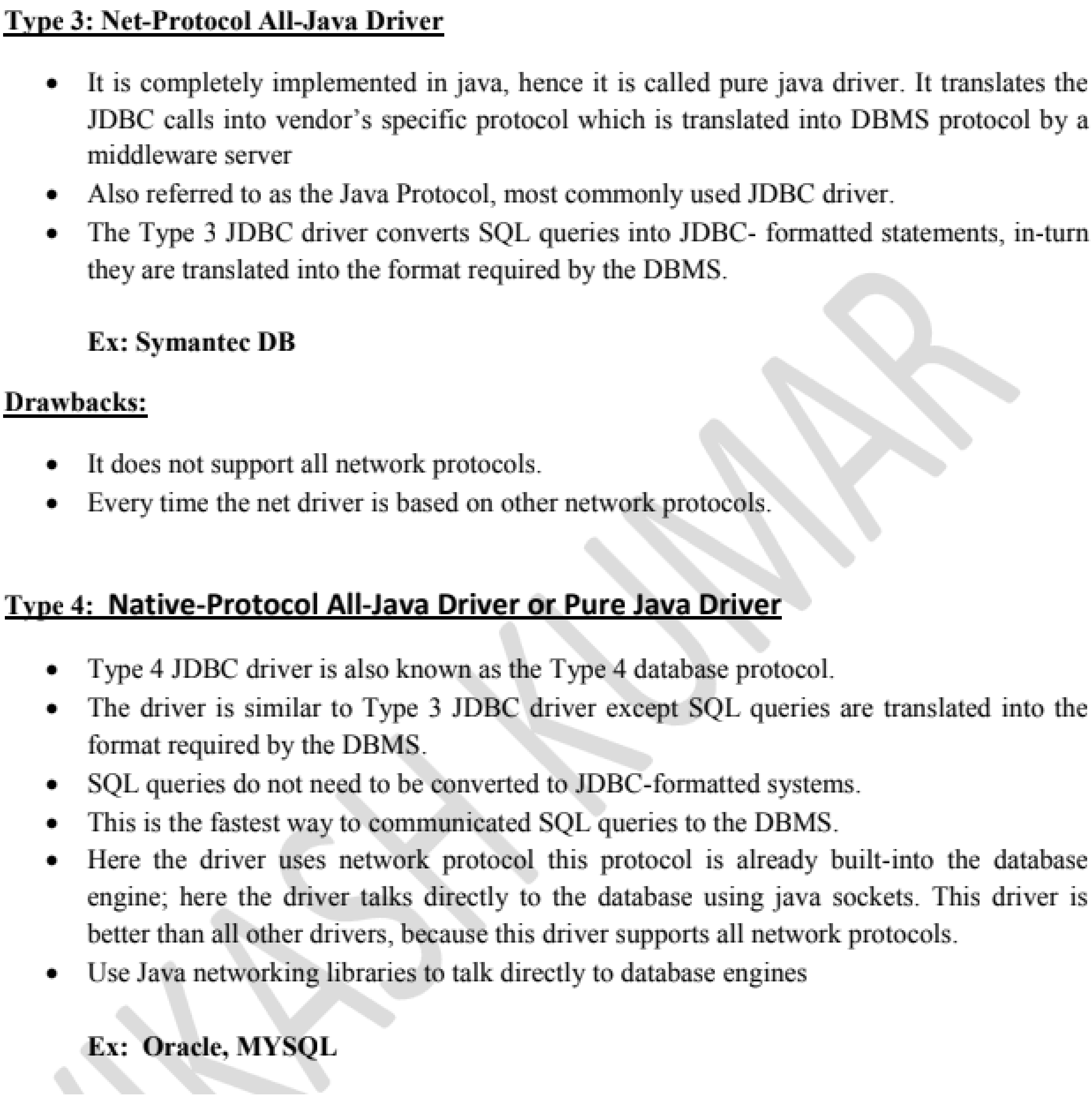
Type 2: Java/Native Code Driver or Native-API Partly Java Driver

- It converts JDBC calls into calls on client API for DBMS.
- The driver directly communicates with database servers and therefore some database client software must be loaded on each client machine and limiting its usefulness for internet
- The Java/Native Code driver uses Java classes to generate platform- specific code that is code only understood by a specific DBMS.

Ex: Driver for DB2, Informix, Intersoly, Oracle Driver, WebLogic drivers

Drawbacks of Type-I Driver:

- Some database client software must be loaded on each client machine
- Loss of some portability of code.
- Limited functionality
- The API classes for the Java/Native Code driver probably won't work with another manufacturer's DBMS.



Type 3: Net-Protocol All-Java Driver

- It is completely implemented in java, hence it is called pure java driver. It translates the JDBC calls into vendor's specific protocol which is translated into DBMS protocol by a middleware server
- Also referred to as the Java Protocol, most commonly used JDBC driver.
- The Type 3 JDBC driver converts SQL queries into JDBC- formatted statements, in-turn they are translated into the format required by the DBMS.

Ex: Symantec DB

Drawbacks:

- It does not support all network protocols.
- Every time the net driver is based on other network protocols.

Type 4: Native-Protocol All-Java Driver or Pure Java Driver

- Type 4 JDBC driver is also known as the Type 4 database protocol.
- The driver is similar to Type 3 JDBC driver except SQL queries are translated into the format required by the DBMS.
- SQL queries do not need to be converted to JDBC-formatted systems.
- This is the fastest way to communicated SQL queries to the DBMS.
- Here the driver uses network protocol this protocol is already built-into the database engine; here the driver talks directly to the database using java sockets. This driver is better than all other drivers, because this driver supports all network protocols.
- Use Java networking libraries to talk directly to database engines

Ex: Oracle, MYSQL

10.b. What is Resultset? Explain Scrollable and updatable Resultset in JDBC with example.

In JDBC 2.1 API the virtual cursor can be moved backwards or positioned at a specific

row.

Six methods are there for Resultset object.

They are first(), last(), previous(), absolute(), relative() and getrow().

first() Moves the virtual cursor to the first row in the Resultset.

last() Positions the virtual cursor at the last row in the Resultset

previous() Moves the virtual cursor to the previous row.

absolute() Positions the virtual cursor to a specified row by the an integer value passed to the method.

relative() Moves the virtual cursor the specified number of rows contained in the parameter. The parameter can be positive or negative integer.

getRow() Returns an integer that represents the number of the current row in the Resultset.

To handle the scrollable ResultSet , a constant value is passed to the Statement object that is created using the createStatement(). Three constants.

TYPE_FORWARD_ONLY restricts the virtual cursor to downward movement

TYPE_SCROLL_INSENSITIVE and TYPE_SCROLL_SENSITIVE (Permits the virtual cursor to Move in any direction)

```
try {
String query = "SELECT FirstName,LastName FROM Customers";
Statement stmt;
ResultSet rs;
stmt = con.createStatement();
rs = stmt.executeQuery (query);
while(rs.next()){
rs.first();
rs.previous();
rs.absolute(10);
rs.relative(-2);
rs.relative(2);
System.out.println(rs.getString(1) + rs. getString (2));
}
stmt.close();}catch ( Exception e ){}
```

Update ResultSet

Once the executeQuery() of the Statement object returns a ResultSet, the updatexxx() is used to change the value of column in the current row of the ResultSet.

The xxx in the updatexxx() is replaced with the data type of the column that is to be updated. Note: updatexxx() updateString(), updateInt()

The updatexxx() requires two parameters. The first is either the number or name of the column of the ResultSet that is being updated and the second is the value that will replace the value in the column of the ResultSet.

A value in a column of the ResultSet can be replaced with a NULL value by using the updateNull().

It requires one parameter, which is the number of column in the current row of the ResultSet. The updateNull() don‟t accept name of the column as a parameter.

Note The updateRow() is called after all the updatexxx() are called.

Delete Row in the ResultSet

The deleteRow() is used to remove a row from a ResultSet.

The deleteRow() is passed an integer that contains the number of the row to be deleted.

First use the absolute() method to move the virtual cursor to the row in the Resultset that should be deleted.

The value of that row should be examined by the program to assure it is the proper row before the deleteRow() is called.

The deleteRow() is then passed a zero integer indicating that the current row must be deleted.

```
rs.deleteRow(0);
```

```
try {
String query = "select * from customers where firstname = 'mary' and lastname = 'jones'";
stmt = con.createStatement(rs.CONCUR_UPDATABLE);
rs = stmt.executeQuery (query);
}
catch ( SQLException error ){System.out.println(error)}
try {
rs.updateString ("LastName", "Smith");
rs.updateRow();
con.close();
} catch ( SQLException e ){System.out.println(e)}
```