CBCS SCHEME

USN | 1 | C | P | 2 | 3 | M | C | 0 | 4 | 7 |

22MCA341

Third Semester MCA Degree Examination, Dec.2024/Jan.2025
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.	Illustrate with an example how enumerations are declared and used in Java Programming.	8	L2	CO1
	b.	Describe Auto boxing and Un-boxing and how it is different from boxing and unboxing. Illustrate with an example.	6	L2	CO1
	c.	Justify Java Enumeration is a class type with an example.	6	L2	CO1
		OR			
Q.2	a.	Explain the various types of wrappers used in Java.	8	L2	CO1
	b.	What are an annotation? Explain the following built-in annotations: (i) Override (ii) Inherited (iii) Retention	6	L2	CO1
	c.	Explain the following methods of Java-lang-Ennum with example : (i) Ordinal (ii) CompareTo (iii) Equals ( )	6	L2	CO1
		Module – 2			
Q.3	a.	Demonstrate linked lists for collections with example.	7	L3	CO1
	b.	Demonstrate ArrayList Class collection with example.	6	L3	CO1
	c.	Explain ArrayList Class and explain the following method : (i) insert (ii) append (iii) replace (iv) substring.	7	L3	CO1
		OR			
Q.4	a.	Explain the following map classes : (i) HashMap (ii) TreeMap	7	L2	CO1
	b.	Discuss the following collection integers set list.	6	L2	CO1
	c.	Explain any four legacy collection of framework.	7	L2	CO1
		Module – 3			
Q.5	a.	Provide an example demonstrating character extraction from a string using charAt( ).	8	L2	CO2
	b.	List and explain two constructors used for creating string in Java.	4	L2	CO2
	c.	Discuss the differences between StringBuffer and StringBuilder.	8	L2	CO2

		OR			
Q.6	a.	Provide an example illustrating the use of the toString () method.	6	L2	CO2
	b.	Provide an example illustrating the use of equalsIgnoreCase ().	8	L2	CO2
	c.	Discuss how string concatenation work with other data types in Java.	6	L2	CO2
		Module – 4			
Q.7	a.	Explain the life cycle of a Servlet.	4	L2	CO3
	b.	Discuss potential security considerations related to the use of cookies in JSP.	8	L2	CO3
	c.	Discuss different techniques for session tracking in Servlets.	8	L2	CO3
		OR			
Q.8	a.	Provide a code example demonstrating how to set and retrieve cookies in a Servlet.	8	L3	CO3
	b.	Define JavaServer Pages (JSP) and their role in web development.	6	L2	CO3
	c.	Discuss the integration of Tomcat with Java Server pages. What steps are involved in deploying JSP pages in Tomcat?	6	L2	CO3
		Module – 5			
Q.9	a.	Discuss the role of the JDBC / ODBC bridge in database connectivity.	8	L2	CO3
	b.	Compare and contrast statement, prepared statement and callable statement.	8	L2	CO3
	c.	Outline the basic steps involved in a typical JDBC process.	4	L2	CO3
		OR			
Q.10	a.	Discuss how a Java application interacts with database using JDBC.	6	L2	CO3
	b.	Explain how to setup and associate the JDBC / ODBC bridge with a database.	8	L2	CO3
	c.	Discuss the significance of the Java.sql.connection interface in database connectivity.	6	L2	CO3

*****

1.a. Illustrate with an example how enumerations are declared and used in java Programming.

Enumeration means a list of named constants. In Java, enumeration defines a class type. An Enumeration can have constructors, methods and instance variables. It is defined using an enum keyword.

How to Define and Use an Enumeration

An enumeration can be defined simply by creating a list of enum variable. Let us take an example for list of Subject variable, with different subjects in the list.

```
enum Subject
{
//Enumerationdefined

Java, Cpp, C, Dbms
}
```

Identifiers Java, Cpp, C and Dbms are called enumeration constants. These are public, static and final by default.
Variables of Enumeration can be defined directly without any new keyword.

```
Subject sub;
```

Variables of Enumeration type can have only enumeration constants as value. We define an enum variable as enum_variable = enum_type.enum_constant;

```
sub = Subject.Java;
```

Two enumeration constants can be compared for equality by using the = = relational operator.

Example:

```
if(sub == Subject.Java) {
...
}
```

Example of Enumeration

```
enum WeekDays
{ sun, mon, tues, wed, thurs, fri, sat }

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

```
WeekDays wk; //wk is an enumeration variable of type WeekDays
wk = WeekDays.sun; //wk can be assigned only the constants defined System.out.println("Today
is "+wk);
}
}
```

Output :

Today is sun

1.b. Describe auto boxing and un-boxing and how it is different from boxing and unboxing. Illustrate with an example.

Boxing : Process of converting primitive type to corresponding wrapper.

Ex: Integer i =new Integer(25);

UnBoxing : Process of extracting value for type wrapper.

int a = i.intValue(i);

Autoboxing and Unboxing

Autoboxing and Unboxing features was added in Java5.

- **Autoboxing** is a process by which primitive type is automatically encapsulated(boxed) into its equivalent type wrapper
- **Auto-Unboxing** is a process by which the value of an object is automatically extracted from a type Wrapper class.

Benefits of Autoboxing / Unboxing

1. Autoboxing / Unboxing lets us use primitive types and Wrapper class objects interchangeably.
2. We don't have to perform Explicit **type casting**.
3. It helps prevent errors, but may lead to unexpected results sometimes. Hence must be used with care.
4. Auto-unboxing also allows you to mix different types of numeric objects in an expression. When the values are unboxed, the standard type conversions can be applied.

```
// Demonstrate autoboxing/unboxing.
class AutoBox {
public static void main(String args[]) {
Integer iOb = 100; // autobox an int
int i = iOb; // auto-unbox
System.out.println(i + " " + iOb); // displays 100 100
}
}
```

1.c. Justify Java Enumeration is a class type with an example.

Java Enumeration is a class type. We can't instantiate an enum using new. Enum has constructors, add instance variables and methods, and even implement interfaces. Each enumeration constant is an object of its enumeration type. Thus, when we define a constructor for an enum, the constructor is called when each
enumeration constant is created.

Ex 1:

```
// Use an enum constructor, instance variable, and method.
enum Apple {
Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);
private int price; // price of each apple
// Constructor
Apple(int p) { price = p; }
int getPrice() { return price; }
}
class EnumDemo3 {
public static void main(String args[])
{
Apple ap;
// Display price of Winesap.
System.out.println("Winesap costs " +
Apple.Winesap.getPrice() +
" cents.\n");
// Display all apples and prices.
System.out.println("All apple prices:");
for(Apple a : Apple.values())
System.out.println(a + " costs " + a.getPrice() +" cents.");

}
}
```

Output:

```
Winesap costs 15 cents.
All apple prices:
Jonathan costs 10 cents.
GoldenDel costs 9 cents.
RedDel costs 12 cents.
Winesap costs 15 cents.
```

2.a. Explain the various types of wrappers used in java.

Java uses primitive types (also called simple types), such as **int or double**, to hold the basic data types supported by the language. Primitive types, rather than objects, are used for these quantities for the sake of performance. Using objects for these values would add an unacceptable overhead to even the simplest of calculations. Thus, the primitive types are not part of the object hierarchy, and they do not inherit **Object**.

Java uses primitive data types such as int, double, float etc. to hold the basic data types. Eg. Int a =10;

Despite the performance benefit offered by the primitive types, there are times when you will need an object representation. For example, **you can't pass a primitive type by reference to a method.** Also, many of the standard data structures implemented by Java operate on an object, which means that you can't use these data structures to store primitive types. To handle these (and other) situations, Java provides **type wrappers**, which are classes that encapsulate a primitive type within an object.

Primitive	Wrapper
boolean	java.lang.Boolean
byte	java.lang.Byte
char	java.lang.Character
double	java.lang.Double
float	java.lang.Float
int	java.lang.Integer
long	java.lang.Long
short	java.lang.Short
void	java.lang.Void

Need of Wrapper Classes

1. They convert primitive data types into objects. Objects are needed if we wish to modify arguments passed into a method (because primitive types are passed by value).
2. The classes in java.util package handles only objects and hence wrapper classes help in this case also.

3. Data structures in the Collection framework, such as **ArrayList** (reference types) and not primitive types.

2.b. What are an annotation? Explain the following built-in annotations:

1. **i)Override: @Override** It is a marker annotation that can be used only on methods. A method annotated with **@Override** must override a method from a superclass. If it

doesn't, a compile-time error will result . It is used to ensure that a superclass method is actually overridden, and not simply overloaded.

```
class Base
{
    public void Display()
    {
        System.out.println("Base display()");
    }
     public static void main(String args[])
    {
        Base    t1    =
        new
        Derived();
        t1.Display();
    }
}
class Derived extends Base
{
    @Override
    public void Display()
    {
        System.out.println("Derived display()");
    }
}
```

Output:
Derived display()

ii)inherited:

The @Inherited annotation signals that a custom annotation used in a class should be inherited by all of its sub classes. For example:

```
@Inherited
public @interface MyCustomAnnotation {

}
@MyCustom
Annotation
public    class
MyParentCla
ss {
  ...
}
public class MyChildClass extends MyParentClass {
   ...
```

}

Here the class MyParentClass is using annotation @MyCustomAnnotation which is marked with @inherited annotation. It means the sub class MyChildClass inherits the @MyCustomAnnotation.

iii)retention

It indicates how long annotations with the annotated type are to be retained.

import java.lang.annotation.Retention; import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME) @interface MyCustomAnnotation {

}

Here we have used RetentionPolicy.RUNTIME. There are two other options as well. Lets see what do they mean:

RetentionPolicy.RUNTIME: The annotation should be available at runtime, for inspection via java reflection.

RetentionPolicy.CLASS: The annotation would be in the .class file but it would not be available at runtime. RetentionPolicy.SOURCE: The annotation would be available in the source code of the program, it would neither be in the .class file nor be available at the runtime.

Complete in one example

```
import java.lang.annotation.Documented;

import java.lang.annotation.ElementType;

import java.lang.annotation.Inherited;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import
java.lang.annotation.Target;


@Documented
@Target(ElementType.ME
THOD)          @Inherited
@Retention(RetentionPolic
y.RUNTIME)          public
```

```
@interface
MyCustomAnnotation{
   int
   studentAge(
   ) default 18;
   String
   studentNam
   e();    String
   stuAddress(
   );
   String stuStream() default "CSE";
}
@MyCustomAnnotation(            studentName="umesh",
stuAddress="India") public class MyClass {
...
}
```

2.c. Explain the following methods of java-lang-enum with example:

i)Ordinal:

Ordinal Value indicates an enumeration constant's position in the list of constants. It is retrieved by calling the **ordinal( )** method. This method returns the ordinal value of the invoking constant. Ordinal values begin at **'0'**.

Syn: final int ordinal( )

ii) CompareTo()

The ordinal value of two constants of the same enumeration can be compared by using the compareTo( ) method. It has this general form:

Syn: final int compareTo(enum-type e)

iii)equals()

We can compare for equality an enumeration constant with any other object by using equals( ), which overrides the equals( ) method defined by Object. Although equals( ) can compare an enumeration constant to any other object, those two objects will only be equal if they both refer to the same constant, within the same enumeration.

Syn: final int equals(enum-type e)

```
// Demonstrate ordinal(), compareTo(), and equals().
// An enumeration of apple varieties.
enum Apple {
Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
class EnumDemo4 {
public static void main(String args[])
```

```
{
Apple ap, ap2, ap3;
// Obtain all ordinal values using ordinal().
System.out.println("Here are all apple constants" +" and their ordinal values: ");
for(Apple a : Apple.values())
System.out.println(a + " " + a.ordinal());
ap = Apple.RedDel;
ap2 = Apple.GoldenDel;
ap3 = Apple.RedDel;
System.out.println();
// Demonstrate compareTo() and equals()
if(ap.compareTo(ap2) < 0)
System.out.println(ap + " comes before " + ap2);
if(ap.compareTo(ap2) > 0)
System.out.println(ap2 + " comes before " + ap);
if(ap.compareTo(ap3) == 0)
System.out.println(ap + " equals " + ap3);
System.out.println();
if(ap.equals(ap2))
System.out.println("Error!");
if(ap.equals(ap3))
System.out.println(ap + " equals " + ap3);
if(ap == ap3)
System.out.println(ap + " == " + ap3);
}
}
```

3. a. Demonstrate linked lists for collections with example.

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

class LinkedList<E>

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

LinkedList( )

LinkedList(Collection<? extends E> c)

// Demonstrate LinkedList.

import java.util.*;

class LinkedListDemo {

```
public static void main(String args[]) {
// Create a linked list.
LinkedList<String> ll = new LinkedList<String>();
// Add elements to the linked list.
ll.add("F");
ll.add("B");
ll.add("D");
ll.add("E");
ll.add("C");
ll.addLast("Z");
ll.addFirst("A");
ll.add(1, "A2");
System.out.println("Original contents of ll: " + ll);
// Remove elements from the linked list.
ll.remove("F");
ll.remove(2);
System.out.println("Contents of ll after deletion: "+ ll);
// Remove first and last elements.
ll.removeFirst();
ll.removeLast();
System.out.println("ll after deleting first and last: "+ ll);
// Get and set a value.
String val = ll.get(2);
ll.set(2, val + " Changed");
System.out.println("ll after change: " + ll);
}
}
```

The output from this program is shown here:

Original contents of ll: [A, A2, F, B, D, E, C, Z]

Contents of ll after deletion: [A, A2, D, E, C, Z]

ll after deleting first and last: [A2, D, E, C]

ll after change: [A2, D, E Changed, C]

3.b. Demonstrate ArrayList Class collection with example.

When working with ArrayList, you will sometimes want to obtain an actual array that contains the contents of the list. You can do this by calling toArray( ), which is defined by Collection.

Several reasons exist why you might want to convert a collection into an array, such as:

• To obtain faster processing times for certain operations

• To pass an array to a method that is not overloaded to accept a collection

• To integrate collection-based code with legacy code that does not understand collections

```
// Convert an ArrayList into an array.
import java.util.*;
class ArrayListToArray {
public static void main(String args[]) {
// Create an array list.
ArrayList<Integer> al = new ArrayList<Integer>();
// Add elements to the array list.
al.add(1);
al.add(2);
al.add(3);
al.add(4);
System.out.println("Contents of al: " + al);
// Get the array.
Integer ia[] = new Integer[al.size()];
ia = al.toArray(ia);
int sum = 0;
// Sum the array.
for(int i : ia) sum += i;
System.out.println("Sum is: " + sum);
}
}
```

2. c. Explain ArrayList Class and explain the following method.

When working with ArrayList, you will sometimes want to obtain an actual array that contains the contents of the list. You can do this by calling toArray( ), which is defined by Collection.

Several reasons exist why you might want to convert a collection into an array, such as:

• To obtain faster processing times for certain operations

• To pass an array to a method that is not overloaded to accept a collection
• To integrate collection-based code with legacy code that does not understand collections
// Convert an ArrayList into an array.

```
i) insert – Add element at a specific index
list.add(index, element);
```

- Inserts the element at the specified index
- Shifts existing elements to the right

Example:

```
list.add(1, "B");  // Inserts "B" at index 1
```

---

i) append **– Add element at the end**

```
list.add(element);
```

- Adds an element to the end of the list

Example:

```
list.add("D");  // Appends "D" at the end
```

---

iii) replace **– Update element at a specific index**

```
list.set(index, element);
```

- Replaces the element at the given index

Example:

```
list.set(1, "Z");  // Replaces element at index 1 with "Z"
```

---

iv) substring **– Extract part of a string**

This is a **String** method, **not** an ArrayList method.

```
String sub = str.substring(startIndex, endIndex);
```

- Extracts part of a string from `startIndex` to `endIndex - 1`.

Example:

```
String word = "HelloWorld";
```

```
String part = word.substring(0, 5);  // "Hello"
```

Use with ArrayList:

```
String sub = list.get(0).substring(0, 3);  // On first element in list
```

4.a. Explain the following map classes.

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.

```
import java.util.*;

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

    // Create a hash map.
    HashMap<String, Double> hm = new HashMap<String, Double>();

    // Put elements to the map
    hm.put("John Doe", new Double(3434.34));
    hm.put("Tom Smith", new Double(123.22));
    hm.put("Jane Baker", new Double(1378.00));
    hm.put("Tod Hall", new Double(99.22));
    hm.put("Ralph Smith", new Double(-19.08));

    // Get a set of the entries.
    Set<Map.Entry<String, Double>> set = hm.entrySet();

    // Display the set.
    for(Map.Entry<String, Double> me : set) {
      System.out.print(me.getKey() + ": ");
      System.out.println(me.getValue());
    }

    System.out.println();

    // Deposit 1000 into John Doe's account.
    double balance = hm.get("John Doe");
    hm.put("John Doe", balance + 1000);

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

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. Discuss the following collection integers set list.

The Set interface defines a set. It extends Collection and declares the behavior of a collection that does not allow duplicate elements. Therefore, 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.

```
Set<Integer> set = new HashSet<>();

set.add(10);

set.add(20);

set.add(10);    // Duplicate, will be ignored

System.out.println(set);  // Output: [10, 20]
```

4.c Explain any four legacy collections framework.

Stack

Stack is a subclass of Vector that implements a standard last-in, first-out stack. Stack only defines the default constructor, which creates an empty stack. With the release of JDK 5, Stack was retrofitted for generics and is declared as shown here:
class Stack<E>
Here, E specifies the type of element stored in the stack.
Stack includes all the methods defined by Vector and adds several of its own
To put an object on the top of the stack, call push( ). To remove and return the top element, call pop( ). An EmptyStackException is thrown if you call pop( ) when the invoking stack is empty. You can use peek( ) to return, but not remove, the top object. The empty( ) method returns true if nothing is on the stack. The search( ) method determines whether an object exists on the stack and returns the number of pops that are required to bring it to the top of the stack

```
// Demonstrate the Stack class.
import java.util.*;

class StackDemo {
  static void showpush(Stack<Integer> st, int a) {
    st.push(a);
    System.out.println("push(" + a + ")");
    System.out.println("stack: " + st);
  }

  static void showpop(Stack<Integer> st) {
    System.out.print("pop -> ");
    Integer a = st.pop();
    System.out.println(a);
    System.out.println("stack: " + st);
  }

  public static void main(String args[]) {
    Stack<Integer> st = new Stack<Integer>();

    System.out.println("stack: " + st);
    showpush(st, 42);
    showpush(st, 66);
    showpush(st, 99);
    showpop(st);
    showpop(st);
    showpop(st);

    try {
      showpop(st);
    } catch (EmptyStackException e) {
      System.out.println("empty stack");
    }
  }
}
```

Dictionary: Dictionary is an abstract class that represents a key/value storage repository and operates much like Map. Given a key and value, you can store the value in a Dictionary object. Once the value is stored, you can retrieve it by using its key. Thus, like a map, a dictionary can be thought of as a list of key/value pairs. Although not currently deprecated, Dictionary is classified as obsolete, because it is fully superseded by Map.

Method	Purpose
Enumeration<V> elements( )	Returns an enumeration of the values contained in the dictionary.
V get(Object *key*)	Returns the object that contains the value associated with *key*. If *key* is not in the dictionary, a **null** object is returned.
boolean isEmpty( )	Returns **true** if the dictionary is empty, and returns **false** if it contains at least one key.
Enumeration<K> keys( )	Returns an enumeration of the keys contained in the dictionary.
V put(K *key*, V *value*)	Inserts a key and its value into the dictionary. Returns **null** if *key* is not already in the dictionary; returns the previous value associated with *key* if *key* is already in the dictionary.
V remove(Object *key*)	Removes *key* and its value. Returns the value associated with *key*. If *key* is not in the dictionary, a **null** is returned.
int size( )	Returns the number of entries in the dictionary.

TABLE 17-17 The Abstract Methods Defined by **Dictionary**

Properties

Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String. The Properties class is used by many other Java classes. For example, it is the type of object returned by System.getProperties( ) when obtaining environmental values. Although the Properties class, itself, is not generic, several of its methods are.

Properties defines the following instance variable:

Properties defaults;

This variable holds a default property list associated with a Properties object. Properties defines these constructors:

Properties( )

Properties(Properties propDefault)

Method	Description
String getProperty(String *key*)	Returns the value associated with *key*. A **null** object is returned if *key* is neither in the list nor in the default property list.
String getProperty(String *key*, String *defaultProperty*)	Returns the value associated with *key*. *defaultProperty* is returned if *key* is neither in the list nor in the default property list.
void list(PrintStream *streamOut*)	Sends the property list to the output stream linked to *streamOut*.
void list(PrintWriter *streamOut*)	Sends the property list to the output stream linked to *streamOut*.
void load(InputStream *streamIn*) throws IOException	Inputs a property list from the input stream linked to *streamIn*.
void load(Reader *streamIn*) throws IOException	Inputs a property list from the input stream linked to *streamIn*. (Added by Java SE 6.)
void loadFromXML(InputStream *streamIn*) throws IOException, InvalidPropertiesFormatException	Inputs a property list from an XML document linked to *streamIn*.
Enumeration<?> propertyNames( )	Returns an enumeration of the keys. This includes those keys found in the default property list, too.
Object setProperty(String *key*, String *value*)	Associates *value* with *key*. Returns the previous value associated with *key*, or returns null if no such association exists.
void store(OutputStream *streamOut*, String *description*) throws IOException	After writing the string specified by *description*, the property list is written to the output stream linked to *streamOut*.
void store(Writer *streamOut*, String *description*) throws IOException	After writing the string specified by *description*, the property list is written to the output stream linked to *streamOut*. (Added by Java SE 6.)
void storeToXML(OutputStream *streamOut*, String *description*) throws IOException	After writing the string specified by *description*, the property list is written to the XML document linked to *streamOut*.
void storeToXML(OutputStream *streamOut*, String *description*, String *enc*)	The property list and the string specified by *description* is written to the XML document linked to *streamOut* using the specified character encoding.
Set<String> stringPropertyNames( )	Returns a set of keys. (Added by Java SE 6.)

TABLE 17-19 The Methods Defined by **Properties**

```
// Demonstrate a Property list.
import java.util.*;
class PropDemo {
public static void main(String args[]) {
Properties capitals = new Properties();
capitals.put("Illinois", "Springfield");
capitals.put("Missouri", "Jefferson City");
capitals.put("Washington", "Olympia");
capitals.put("California", "Sacramento");
capitals.put("Indiana", "Indianapolis");
// Get a set-view of the keys.
Set states = capitals.keySet();
// Show all of the states and capitals.
for(Object name : states)
System.out.println("The capital of " +
name + " is " +
capitals.getProperty((String)name)
+ ".");
System.out.println();
// Look for state not in list -- specify default.
String str = capitals.getProperty("Florida", "Not Found");
System.out.println("The capital of Florida is "
+ str + ".");
}
```

}

The Enumeration Interface

The Enumeration interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. This legacy interface has been superseded by Iterator. Although not deprecated, Enumeration is considered obsolete for new code. However, it is used by several methods defined by the legacy classes (such as Vector and Properties), is used by several other API classes, and is currently in widespread use in application code. Because it is still in use, it was retrofitted for generics by JDK 5. It has this declaration:

interface Enumeration<E>

where E specifies the type of element being enumerated.

Enumeration specifies the following two methods:

boolean hasMoreElements( )

E nextElement( )

When implemented, hasMoreElements( ) must return true while there are still more elements to extract, and false when all the elements have been enumerated. nextElement( ) returns the next object in the enumeration. That is, each call to nextElement( ) obtains the next object

5.a. Provide an example demonstrating character extraction from a string using charAt()

CharAt() is used to extract a single character from a String. We can refer directly to an individual character via the charAt( ) method.

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. charAt( ) returns the character at the specified location.

Ex: char ch;

ch = "abc".charAt(1); assigns the value “b” to ch.

5.b. List and explain two constructors for creating string in java

Constructors

1. **String()**

Initializes a newly created String object so that it represents an empty character sequence.

```
Ex: String s=new String ("Good Morning");
```

2. **String(byte[] bytes):**
   Constructs a new String by decoding the specified array of bytes using the platform's default charset.

```
Ex: byte[] barr={65,89,78};

String s2=new String(barr);
```

3. **String(byte[] bytes, Charset charset):**

Constructs a new String by decoding the specified array of bytes using the specified charset.

```
Ex: byte[] barr={65,89,78};
Charset cs=Charset.defaultCharset();

String s2=new String(barr,cs);
```

4. **String(byte[] bytes, String charsetName)**

Constructs a new String by decoding the specified array of bytes using the specified charset.

```
Ex: byte[] b_arr = {71, 101, 95, 107, 120}; String s = new String(b_arr, "US-ASCII");
```

5. **String(char[] value)**

Allocates a new String so that it represents the sequence of characters currently contained in the character array argument.

```
Ex: char char_arr[] = {'C', 'M', 'R"}; String s = new String(char_arr);
```

6. **String(char[] char_array, int start_index, int count)** – Allocates a String from a given character
7. Example: array but choose count characters

```
Ex: char char_arr[] = {'C', 'M', 'R"}; String s = new String(char_arr , 1, 2);
```

8. **String(char[] value, int offset, int count)**

Allocates a new String that contains characters from a subarray of the character array argument.

```
Ex: int[] uni_code = {90, 80, 101, 67,65}; String s = new String(uni_code, 1, 3);
```

5.c. Discuss the differences between stringbuffer and stringbuilder.

Feature	StringBuffer	StringBuilder
🔒 **Thread Safety**	**Thread-safe** (synchronized)	**Not thread-safe** (not synchronized)
⚡ **Performance**	Slower (due to synchronization)	Faster (no overhead of synchronization)
📝 **Use Case**	In **multi-threaded** environments	In **single-threaded** environments
📦 **Package**	`java.lang`	`java.lang`
🔄 **Mutability**	Mutable	Mutable
📜 **Introduced In**	Java 1.0	Java 1.5

6.a. Provide an example illustrating the use of toString() method.

Every class implements toString( ) because it is defined by Object.

To implement toString( ), simply return a String object that contains the human-readable string that appropriately describes an object of your class.

Ex:

```
// Override toString() for Box class.
class Box {
double width;
double height;
double depth;
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
public String toString() {
return "Dimensions are " + width + " by " +
depth + " by " + height + ".";
}
```

```
}
class toStringDemo {
public static void main(String args[]) {
Box b = new Box(10, 12, 14);
String s = "Box b: " + b; // concatenate Box object
System.out.println(b); // convert Box to string
System.out.println(s);
}
}
```

6.b. Provide an example illustrating the use of equalsIgnoreCase()

equalsIgnoreCase( ).

Equals() is used to compare two strings for equality by ignoring case.

General form:

boolean equalsIgnoreCase(String str)

Here, str is the String object being compared with the invoking String object. It, too, returns true if the strings contain the same characters in the same order, and false otherwise.

Ex:

```
// Demonstrate equals() and equalsIgnoreCase().
class equalsDemo {
public static void main(String args[]) {
String s1 = "Hello";
String s2 = "Hello";
String s3 = "Good-bye";
String s4 = "HELLO";
System.out.println(s1 + " equals " + s2 + " -> " +
s1.equals(s2));
System.out.println(s1 + " equals " + s3 + " -> " +
s1.equals(s3));
```

```
System.out.println(s1 + " equals " + s4 + " -> " +
s1.equals(s4));
System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +
s1.equalsIgnoreCase(s4));
}
}
```

The output from the program is shown here:

```
Hello equals Hello -> true
Hello equals Good-bye -> false
Hello equals HELLO -> false
Hello equalsIgnoreCase HELLO -> true
```

6.c. Discuss how string concatenation work with other datatypes in java.

The + operator, which concatenates two strings, producing a String object as the result.

This allows you to chain together a series of + operations.

Ex:

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);
```

String Concatenation with Other Data Types

We can concatenate strings with other types of data using + operator

Ex: int age = 9;

```
String s = "He is " + age + " years old.";
System.out.println(s);
```

7.a. Explain the life cycle of 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. Discuss potential security considerations related to the use of cookies in JSP.

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.

- The problem is privacy, not security
- Servers can remember your previous actions
- If you give out personal information, servers can link that information to your previous actions
- Servers can share cookie information through use of a cooperating third party like doubleclick.net
- Poorly designed sites store sensitive information like credit card numbers directly in cookie

7.c. Discuss different techniques for session tracking in Servlets.

- **Session tracking is the capability of a server to maintain the current state of a single client's sequential requests.**

- **Session simply means a particular interval of time.**
- **Session Tracking is a way to maintain state of a user.**

There are four techniques used in Session tracking:

- Cookies
- Hidden Form Field
- URL Rewriting

Built-in session-tracking API

- HttpSession

Cookies: 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.

- associate cookie with data on server
- String sessionID = makeUniqueString();
- HashMap sessionInfo = new HashMap();
- HashMap globalTable = findTableStoringSessions();
- globalTable.put(sessionID, sessionInfo);
- Cookie sessionCookie =
- new Cookie("JSESSIONID", sessionID);
- sessionCookie.setPath("/");
- response.addCookie(sessionCookie);

URL Rewriting:

Client appends some extra data on the end of each URL that identifies the session

– Server associates that identifier with data it has stored about that session

E.g., http://host/path/file.html;jsessionid=1234

Advantage

Works even if cookies are disabled or unsupported

Disadvantages

– Must encode all URLs that refer to your own site

– All pages must be dynamically generated

– Fails for bookmarks and links from other sites

Hidden Form Field:

<INPUT TYPE="HIDDEN" NAME="session" VALUE="...">

Advantage

– Works even if cookies are disabled or unsupported

Disadvantages

– Lots of tedious processing

– All pages must be the result of form submissions

HTTPSession:

– Call request.getSession to get HttpSession object

This is a hashtable associated with the user

Look up information associated with a session.

– Call getAttribute on the HttpSession object, cast the return value to the appropriate type, and check whether the result is null.

Store information in a session.

– Use setAttribute with a key and a value.

Discard session data.

– Call removeAttribute discards a specific value.

– Call invalidate to discard an entire session.

8.a. Provide a code example demonstrating how to set and retrieve cookies in a servlet.

```
package j2ee.prg4;

import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
/**
 * Servlet implementation class store
 */
@WebServlet("/store")
public class store extends HttpServlet {
        private static final long serialVersionUID = 1L;

    /**
     * @see HttpServlet#HttpServlet()
     */
    public store() {
        super();
        // TODO Auto-generated constructor stub
    }

        /**
         * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse response)
         */
        protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
```

```
		// Setting the HTTP Content-Type response header to text/html
		response.setContentType("text/html;charset=UTF-8");
		// Returns a PrintWriter object that can send character text to the client.
		PrintWriter out=response.getWriter();
		try
		{
			//Requesting input color from html page and storing in String variable s1
			String s1=request.getParameter("color");
			//Checking the color either RED or Green or Blue
			if (s1.equals("RED")||s1.equals("BLUE")||s1.equals("GREEN"))
			{
				// Creating cookie object ck1 and storing the selected color
				Cookie ck1=new Cookie("color",s1);
				//adding the cookie to the response
				response.addCookie(ck1);
				//writing the output in the html format
				out.println("<html>");
				out.println("<body>");
				out.println("You selected: "+s1);
				out.println("<form action='retrieve' method='post'>");
				out.println("<input type='Submit' value='submit'/>");
				out.println("</form>");
				out.println("</body>");
				out.println("</html>");
}
		finally
		{
			//Closing the output object
		    out.close();
		}
	 }
}
```

retrieve.java

```
package j2ee.prg4;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Servlet implementation class retrieve
 */
@WebServlet("/retrieve")
public class retrieve extends HttpServlet {
	private static final long serialVersionUID = 1L;
```

```
    /**
     * @see HttpServlet#HttpServlet()
     */
    public retrieve() {
        super();
        // TODO Auto-generated constructor stub
    }
            /**
             * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse response)
             */
            protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {

                        // Setting the HTTP Content-Type response header to text/html
                        response.setContentType("text/html;charset=UTF-8");
                        // Returns a PrintWriter object that can send character text to the client.
                        PrintWriter out=response.getWriter();
                        try
                        {
                                    //Requesting all the cookies and stored in cookie array ck[]
                                                Cookie ck[]=request.getCookies();
                                                out.println("<html>");
                                                out.println("<head>");
                                                out.println("<title>servlet</title>");
                                                out.println("</head>");
                                                // Getting the value from cookie and setting the HTML form background color
out.println("<body bgcolor="+ck[0].getValue()+">");
                                                //Getting the value from cookie and displaying the color name in HTML form
                                                out.println("You selected color is: "+ck[0].getValue()+"</h1>");
                                                out.println("</body>");
                                                out.println("</html>");
                        }
                        finally
                        {
                                    //closing the printwriter object out
                                    out.close();
                        }
            }

}
```

Index.jsp

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<!-- send the form data to the url store and the post method is used  -->
<form action="store" method="post">
<!-- Display the Radio button with three option  -->
RED:<input type="radio" name="color" value="RED"/><br>
```

```
GREEN:<input type="radio" name="color" value="GREEN"/><br>
BLUE:<input type="radio" name="color" value="BLUE"/><br>
<input type="submit" value="submit"/>
</form>
</body>
</html>
```

8.b. Define Java Server Pages and their role in Web Development.

Java Server Page technology is used to create dynamic web applications.

JSP pages are easier to maintain then a Servlet.

JSP pages are opposite of Servlets as a servlet adds HTML code inside Java code, while JSP adds Java code inside HTML using JSP tags.

Everything a Servlet can do, a JSP page can also do it.

JSP enables us to write HTML pages containing tags, inside which we can include powerful Java programs

Why JSP is preffered over servlets?

 JSP provides an easier way to code dynamic web pages.

 JSP does not require additional files like, java class files, web.xml etc

 Any change in the JSP code is handled by Web Container(Application server like tomcat), and doesn't require re-compilation.

 JSP pages can be directly accessed, and web.xml mapping is not required like in servlets.

Advantage of JSP

 Easy to maintain and code.

 High Performance and Scalability.

 JSP is built on Java technology, so it is platform independent.

8.c. Discuss the integration of Tomcat with Java Server Pages. What steps are involved in deploying JSP pages in Tomcat?

To create servlets, We will need access to a servlet development environment. The one used by this chapter is Tomcat. Tomcat is an open-source product maintained by the Jakarta Project of the Apache Software Foundation. It contains the class libraries, documentation, and run-time support that you will need to create and test servlets. At the time of this writing, the current version is 5.5.17, which supports servlet specification 2.4. We can download Tomcat from jakarta.apache.org.

This is the location assumed by the examples in this book. If you load Tomcat in a different

location, you will need to make appropriate changes to the examples. You may need to set the environmental variable JAVA_HOME to the top-level directory in which the Java Development Kit is installed.

To start Tomcat, select Configure Tomcat in the Start | Programs menu, and then press Start in the Tomcat Properties dialog.

When you are done testing servlets, you can stop Tomcat by pressing Stop in the Tomcat Properties dialog.

The directory

C:\Program Files\Apache Software Foundation\Tomcat 5.5\common\lib\

contains servlet-api.jar. This JAR file contains the classes and interfaces that are needed to build servlets. To make this file accessible, update your CLASSPATH environment variable so that it includes

C:\Program Files\Apache Software Foundation\Tomcat 5.5\common\lib\servlet-api.jar

Alternatively, you can specify this file when you compile the servlets. For example, the following command compiles the first servlet example:

javac HelloServlet.java -classpath "C:\Program Files\Apache Software Foundation\
Tomcat 5.5\common\lib\servlet-api.jar"

Once you have compiled a servlet, you must enable Tomcat to find it. This means putting it into a directory under Tomcat's webapps directory and entering its name into a web.xml file. To keep things simple, the examples in this chapter use the directory and web.xml file that Tomcat supplies for its own example servlets. Here is the procedure that you will follow. First, copy the servlet's class file into the following directory:

C:\Program Files\Apache Software Foundation\Tomcat 5.5\webapps\servlets-examples\WEB-INF\classes

Next, add the servlet's name and mapping to the web.xml file in the following directory:

C:\Program Files\Apache Software Foundation\Tomcat 5.5\webapps\servlets-examples\WEB-INF

For instance, assuming the first example, called HelloServlet, you will add the following lines in the section that defines the servlets:

<servlet>

<servlet-name>HelloServlet</servlet-name>

<servlet-class>HelloServlet</servlet-class>

</servlet>

Next, you will add the following lines to the section that defines the servlet mappings.

<servlet-mapping>

<servlet-name>HelloServlet</servlet-name>

<url-pattern>/servlet/HelloServlet</url-pattern>

</servlet-mapping>

9.a. Discuss the role of the JDBC/ODBC bridge in database connectivity.

The JDBC type 1 driver which is also known as a JDBC-ODBC Bridge is a convert JDBC methods into ODBC function calls.

Sun provides a JDBC-ODBC Bridge driver by "**sun.jdbc.odbc.JdbcOdbcDriver**".

The driver is a platform dependent because it uses ODBC which is depends on native libraries of the operating system and also the driver needs other installation for example, ODBC must be installed on the computer and the database must support ODBC driver

Type 1 is the simplest compare to all other driver but it's a platform specific i.e. only on Microsoft platform.

The JDBC-ODBC Bridge is use only when there is no PURE-JAVA driver available for a particular database.

Advantage:

- Connect to almost any database on any system, for which ODBC driver is installed.
- It's an easy for installation as well as easy(simplest) to use as compare the all other

Disadvantages:

- The ODBC Driver needs to be installed on the client machine.
- It's a not a purely platform independent because its use ODBC which is depends on native libraries of the operating system on client machine.
- Not suitable for applets because the ODBC driver needs to be installed on the client machine.

9.b. Compare and Contrast statement, preparedstatement and callable statement.

Feature / Interface	`Statement`	`PreparedStatement`	`CallableStatement`
Use Case	For static SQL queries	For dynamic SQL with parameters	For calling stored procedures

Feature / Interface	`Statement`	`PreparedStatement`	`CallableStatement`
Introduced In	JDBC 1.0	JDBC 2.0	JDBC 3.0
SQL Injection Safe?	No – prone to SQL injection	Yes – parameters are precompiled	Yes – parameters are precompiled
Precompiled?	No – compiled every time	Yes – compiled once and reused	Yes
Performance	Slower (recompiled every time)	Faster (precompiled & cached)	Faster (precompiled stored procedures)
Parameter Support	No	Yes – `?` placeholders	Yes – `?` and OUT parameters supported
Used For	Simple, ad-hoc queries (e.g., SELECT)	Repeated queries with different parameters	Executing stored procedures with IN/OUT params
Supports Batching?	Yes	Yes	Yes
Syntax Example	`stmt.execute("SELECT * FROM users");`	`pstmt = con.prepareStatement("SELECT * FROM users WHERE id = ?");`	`cstmt = con.prepareCall("{call getUser(?)}");`

9.c. Outline the basic steps involved in a typical JDBC process.

Step 0: import the java.sql package

An application that uses the jdbc API must import the java.sql package

import java.sql.*;

Step 1: Load a JDBC Driver

Prior to JDBC 4.0 it is needed to seperately load the driver and register the driver but in jdbc 4.0 it is no longer needed to

register the driver

Class.forName(“sun.jdbc.odbc:jdbcodbcDriver”);

Step 2: Establishing a connection

Once a driver is loaded we can establish a connection to db

Connection con= DriverManager.getconnection(dburl,username,password)

DriverManager Connects to given JDBC URL with given user name and password

A Connection represents a session with a specific database.

The connection to the database is established by getConnection(), which requests access to the database from the DBMS.

A Connection object is returned by the getConnection() if access is granted; else getConnection() throws a

SQLException.

Sometimes a DBMS requires extra information besides userID & password to grant access to the database.

This additional information is referred as properties and must be associated with Properties or Sometimes DBMS grants access

to a database to anyone without using username or password.

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

Step 3: Create a statement

A statement object is needed to execute the query and obtain the results produced by it.

Statement st= con.createStatement();

Step 4: Execute the statement

The db statements can be executed by using methods like executeQuery().

executeQuery() takes querystring as an argument and returns the results as ResultSet object

ResultSet object contains the data returned by the query and the methods for retrieving the data

Ex: ResultSet rs=stmt.executeQuery("select * from employee");

Step 5: Process the result

The ResultSet consists of tuples and returns one tuple at a time when the next() is applied.

ResultSet acts as an iterator

```
While(rs.next())
{
System.out.println(rs.getString(1)+" "+rs.getInt("salary");
}
```

Getters can be used by referring position/name to retrieve the values

Step 6: Close the statement

stmt.close();

Step 7: Close the connection

Commit()

con.close()

10.a. Discuss how a Java application interacts with database using JDBC.

JDBC provides API or Protocol to interact with different databases.

With the help of **JDBC** driver we can connect with different types of databases.

Driver is must needed for connection establishment with any database.

A driver works as an interface between the client and a database server.

JDBC have so many classes and interfaces that allow a java application to send request made by user to any specific **DBMS**(Data Base Management System).

JDBC supports a wide level of portability.

JDBC provides interfaces that are compatible with java application

JDBC has four main components as under and with the help of these components java application can connect with database.

The **JDBC API** - it provides various methods and interfaces for easy communication with database.

The **JDBC** DriverManager - it loads database specific drivers in an application to establish connection with database.

The **JDBC** test suite - it will be used to test an operation being performed by **JDBC** drivers.

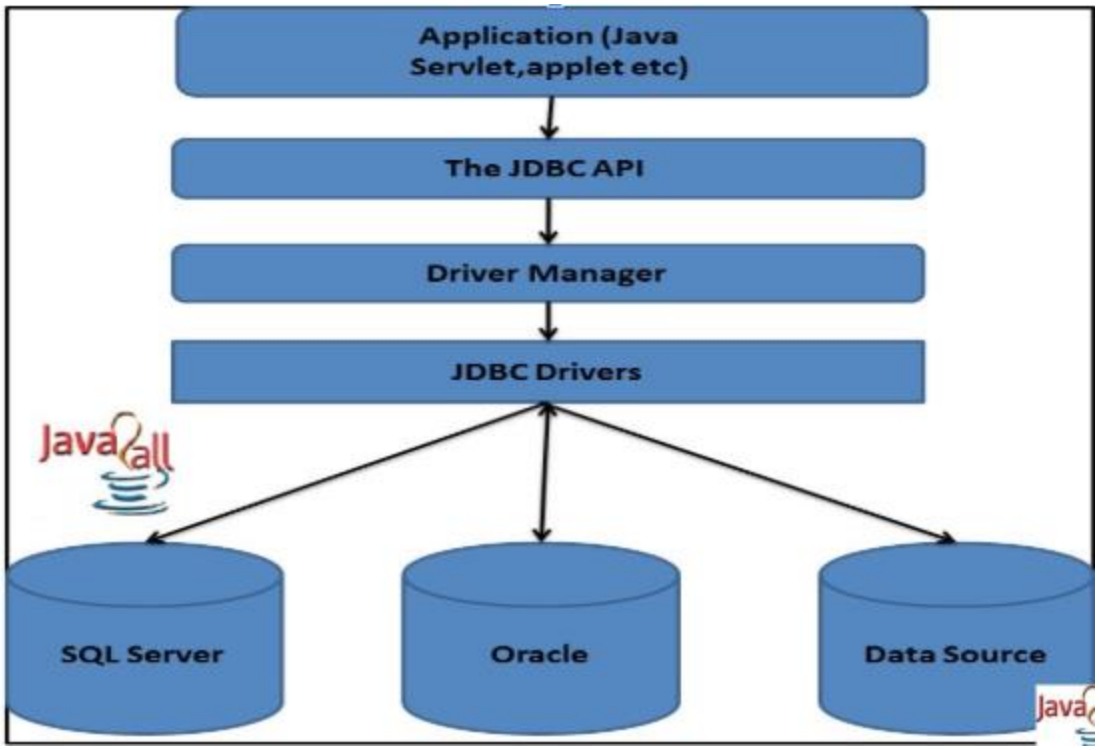
The **JDBC-ODBC** bridge - it connects database

10.b. Explain how to setup and associate the JDBC/ODBC bridge with a da

tabase.

The DriverManager plays an important role in **JDBC** architecture.

It uses some database specific drivers to communicate our **J2EE** application to database.



As per the diagram first of all we have to program our application with **JDBC API**.

With the help of **DriverManager** class than we connect to a specific database with the help of spcific database driver.

Java drivers require some library to communicate with the database.
Some drivers are pure java drivers and some are partial.

So with this kind of JDBC architecture we can communicate with specific database.

10.c. Discuss the significance of the Java.sql.connection interface in database connectivity.

The `java.sql.Connection` interface is a core component of JDBC that represents an active connection between a Java application and a relational database. It is responsible for managing the session with the database, allowing the application to execute SQL statements, manage transactions, and retrieve database metadata. Through this interface, developers can create `Statement`, `PreparedStatement`, and `CallableStatement` objects to interact with the database, perform commit and rollback operations for transaction control, and configure connection properties. Proper use of the `Connection` interface ensures secure, efficient, and consistent database operations in Java-based applications.