Internal Assessment 2 – October 2019

Scheme and Solutions

Sub:	Advanced Java and J2EE					Sub Code:	**17CS553**	Branch:	ISE
Date:	15.10.19	Duration:	90 min’s	Max Marks:	50	Sem / Sec:	5		OBE

Q. 1 a) What is Enumeration? Explain with an example

The **Enum in Java** is a data type which contains a fixed set of constants.

It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY) , directions (NORTH, SOUTH, EAST, and WEST), season (SPRING, SUMMER, WINTER, and AUTUMN or FALL), colors (RED, YELLOW, BLUE, GREEN, WHITE, and BLACK) etc. According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters.

Java Enums can be thought of as classes which have a fixed set of constants (a variable that does not change). The Java enum constants are static and final implicitly. It is available since JDK 1.5.

```
class EnumExample1{
//defining the enum inside the class
public enum Season { WINTER, SPRING, SUMMER, FALL }
//main method
public static void main(String[] args) {
//traversing the enum
for (Season s : Season.values())
System.out.println(s);
}}
```

Output:

```
WINTER
SPRING
SUMMER
FALL
```

Q. 1 b) Demonstrate how enumeration can be applied as a class types

Java Enumerations Are Class Types

As explained, a Java enumeration is a class type. Although you don't instantiate an **enum** using **new**, it otherwise has much the same capabilities as other classes. The fact that **enum** defines a class gives powers to the Java enumeration that enumerations in other languages simply do not have. For example, you can give them constructors, add instance variables and methods, and even implement interfaces.

It is important to understand that each enumeration constant is an object of its enumeration type. Thus, when you define a constructor for an **enum**, the constructor is called when each enumeration constant is created. Also, each enumeration constant has its own copy of any instance variables defined by the enumeration. For example, consider the following version of **Apple**:

```
// 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.");
  }
}
```

The output is shown here:

```
Winesap costs 15 cents.

All apple prices:
Jonathan costs 10 cents.
GoldenDel costs 9 cents.
RedDel costs 12 cents.
Winesap costs 15 cents.
Cortland costs 8 cents.
```

When the variable **ap** is declared in **main( )**, the constructor for **Apple** is called once for each constant that is specified. Notice how the arguments to the constructor are specified, by putting them inside parentheses after each constant, as shown here:

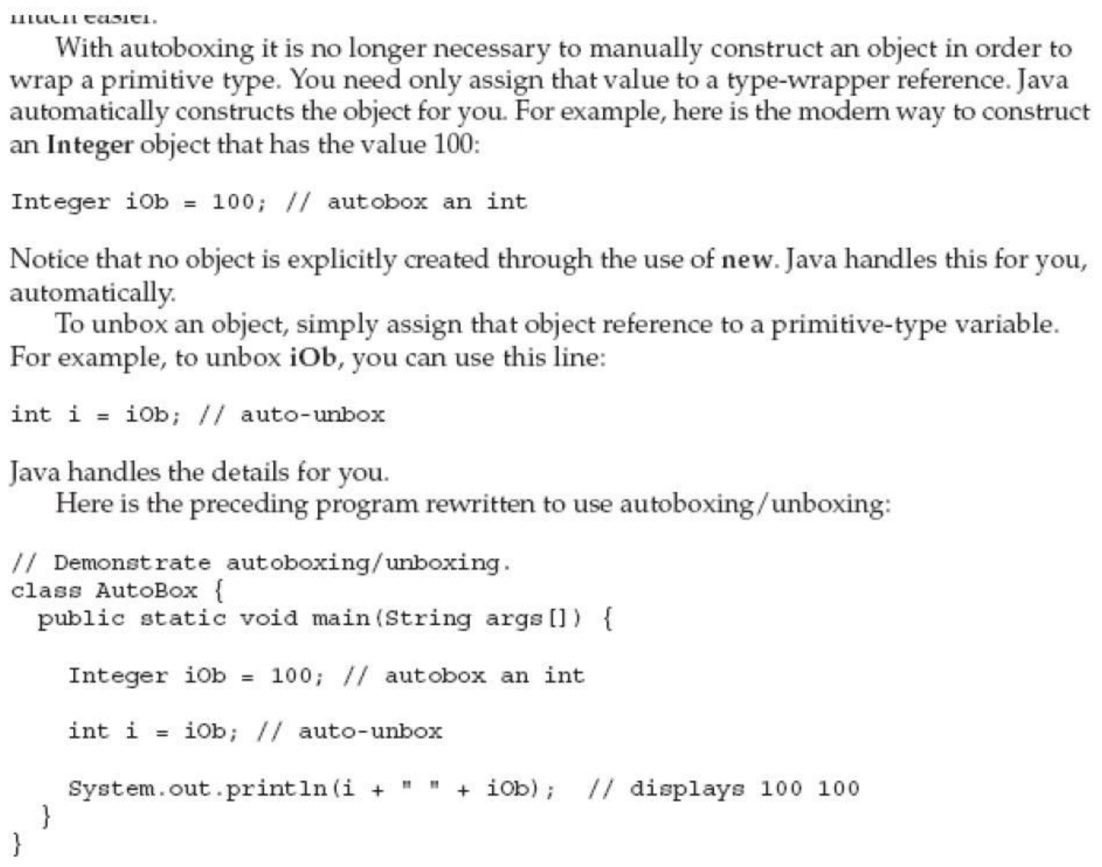
```
Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);
```

Q. 2 a) What is Autoboxing? Write a Java program that demonstrates autoboxing and unboxing

Autoboxing

Beginning with JDK 5, Java added two important features: *autoboxing* and *auto-unboxing*. Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed. There is no need to explicitly construct an object. Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed. There is no need to call a method such as **intValue( )** or **doubleValue( )**.

b)



With autoboxing it is no longer necessary to manually construct an object in order to wrap a primitive type. You need only assign that value to a type-wrapper reference. Java automatically constructs the object for you. For example, here is the modern way to construct an **Integer** object that has the value 100:

```
Integer iOb = 100; // autobox an int
```

Notice that no object is explicitly created through the use of **new**. Java handles this for you, automatically.

To unbox an object, simply assign that object reference to a primitive-type variable. For example, to unbox **iOb**, you can use this line:

```
int i = iOb; // auto-unbox
```

Java handles the details for you.

Here is the preceding program rewritten to use autoboxing/unboxing:

```
// 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
  }
}
```

Q. 2 b) Demonstrate single member annotation with an example

Single-Member Annotations

A *single-member* annotation contains only one member. It works like a normal annotation except that it allows a shorthand form of specifying the value of the member. When only one member is present, you can simply specify the value for that member when the annotation is applied—you don't need to specify the name of the member. However, in order to use this shorthand, the name of the member must be **value**.

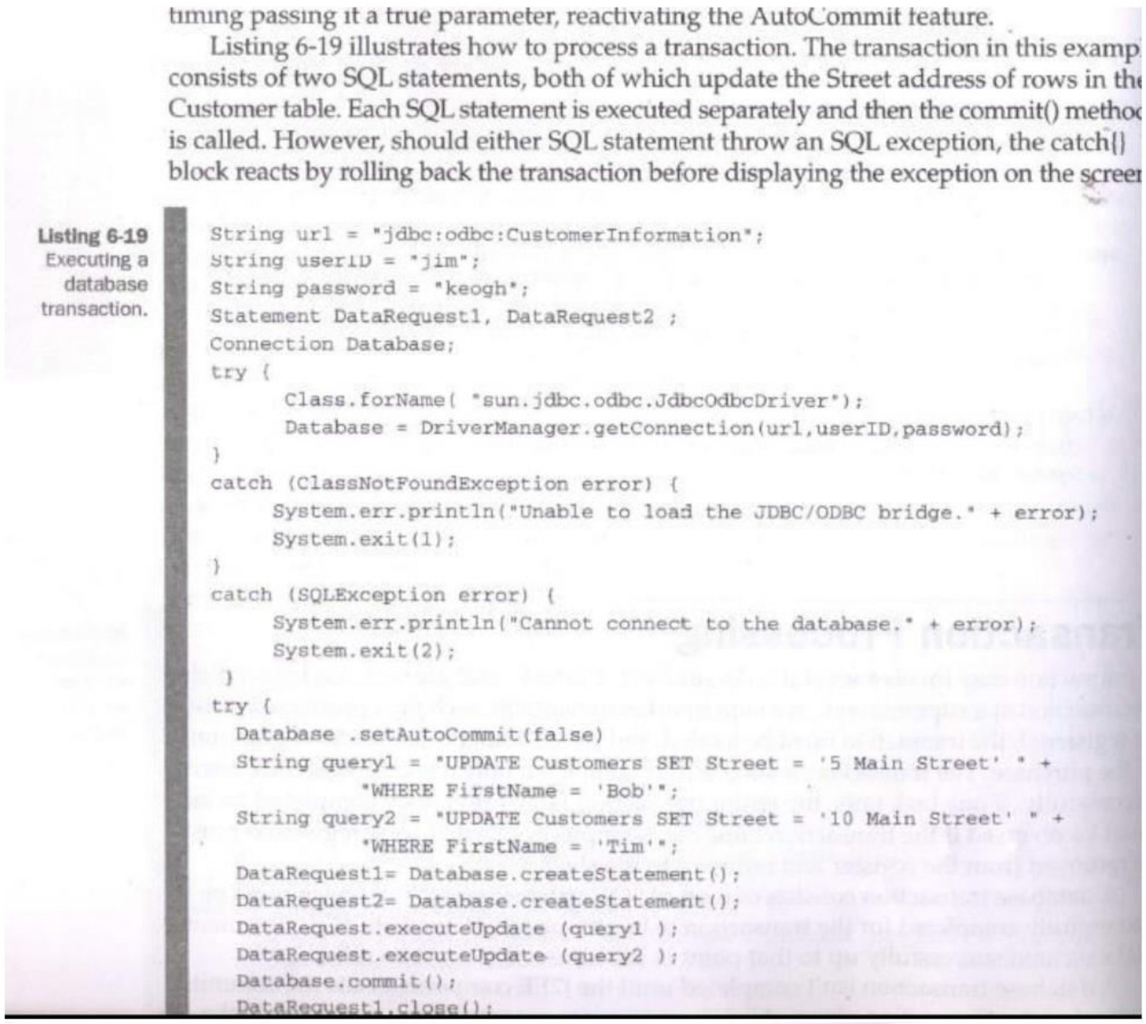
Here is an example that creates and uses a single-member annotation:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// A single-member annotation.
@Retention(RetentionPolicy.RUNTIME)
@interface MySingle {
  int value(); // this variable name must be value
}

class Single {

  // Annotate a method using a single-member annotation.
  @MySingle(100)
  public static void myMeth() {
    Single ob = new Single();

    try {
      Method m = ob.getClass().getMethod("myMeth");

      MySingle anno = m.getAnnotation(MySingle.class);

      System.out.println(anno.value()); // displays 100

    } catch (NoSuchMethodException exc) {
       System.out.println("Method Not Found.");
    }
  }




  public static void main(String args[]) {
    myMeth();
  }
}
```

As expected, this program displays the value 100. In the program, **@MySingle** is used to annotate **myMeth( )**, as shown here:

```
@MySingle(100)
```

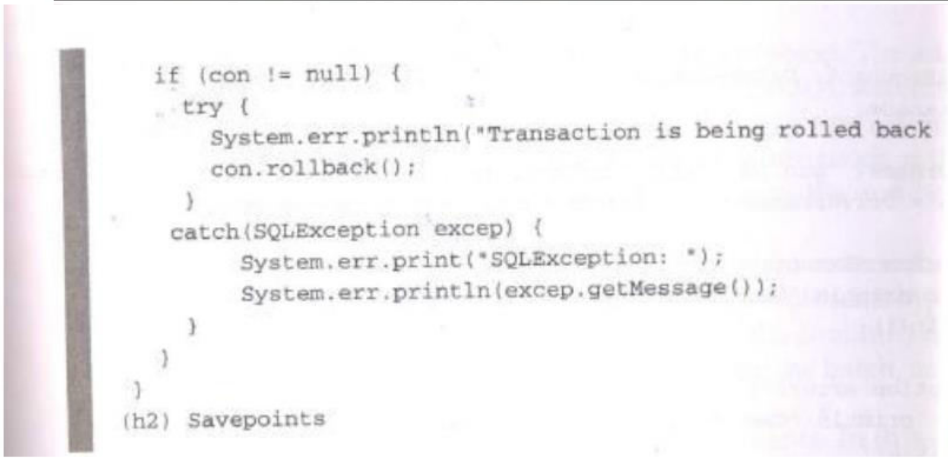
Q. 3 a) What is transaction? Write a Java program to execute a database transaction.


timing passing it a true parameter, reactivating the AutoCommit feature.

Listing 6-19 illustrates how to process a transaction. The transaction in this examp consists of two SQL statements, both of which update the Street address of rows in the Customer table. Each SQL statement is executed separately and then the commit() methoc is called. However, should either SQL statement throw an SQL exception, the catch{} block reacts by rolling back the transaction before displaying the exception on the screer

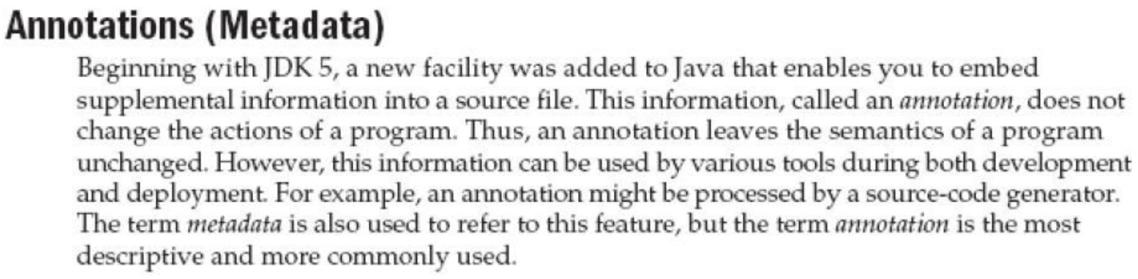
Listing 6-19 Executing a database transaction.

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
Statement DataRequest1, DataRequest2 ;
Connection Database;
try {
      Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
      Database = DriverManager.getConnection(url,userID,password);
}
catch (ClassNotFoundException error) {
     System.err.println("Unable to load the JDBC/ODBC bridge." + error);
     System.exit(1);
}
catch (SQLException error) {
     System.err.println("Cannot connect to the database." + error);
     System.exit(2);
 }
try {
  Database .setAutoCommit(false)
  String query1 = "UPDATE Customers SET Street = '5 Main Street' " +
           "WHERE FirstName = 'Bob'";
  String query2 = "UPDATE Customers SET Street = '10 Main Street' " +
           "WHERE FirstName = 'Tim'";
  DataRequest1= Database.createStatement();
  DataRequest2= Database.createStatement();
  DataRequest.executeUpdate (query1 );
  DataRequest.executeUpdate (query2 );
  Database.commit();
  DataRequest1.close();
```



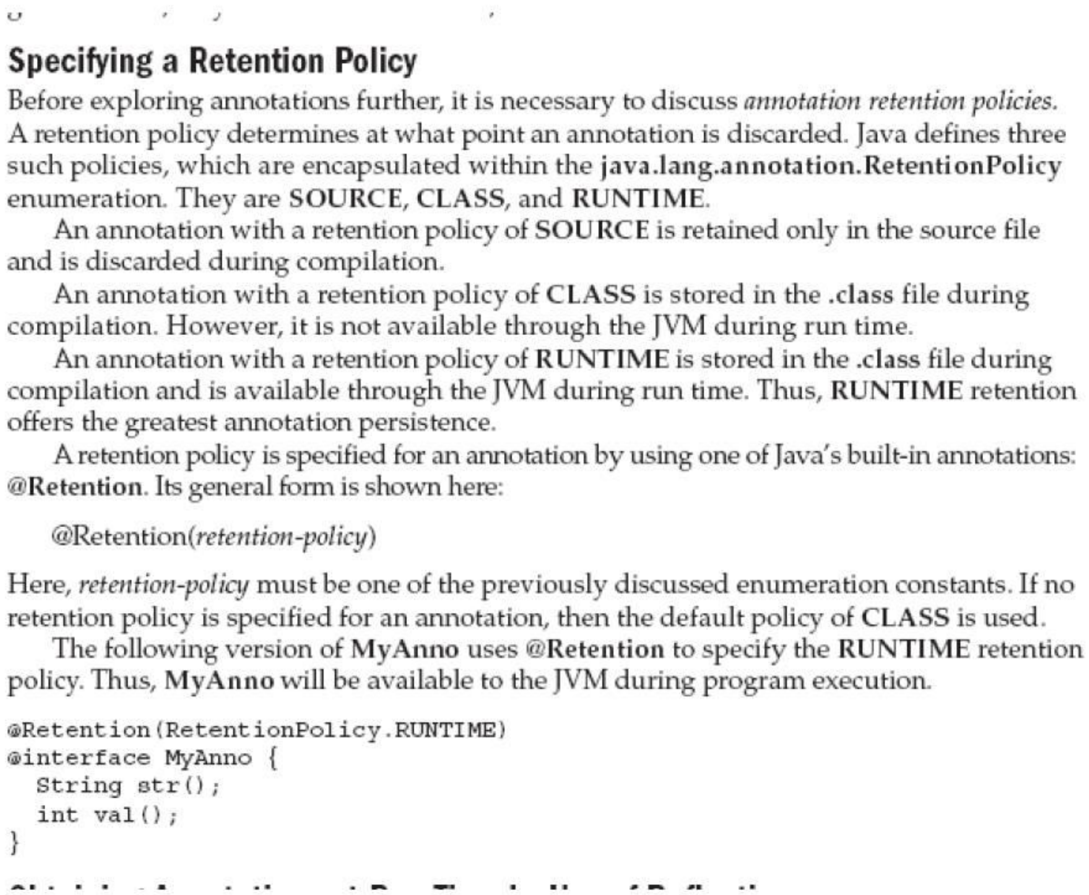
```
    if (con != null) {
      try {
        System.err.println("Transaction is being rolled back
        con.rollback();
      }
     catch(SQLException excep) {
          System.err.print("SQLException: ");
          System.err.println(excep.getMessage());
      }
    }
  }
(h2) Savepoints
```

Q. 4 a) What is Annotation? Explain various retention policies for annotations in Java



Annotations (Metadata)

Beginning with JDK 5, a new facility was added to Java that enables you to embed supplemental information into a source file. This information, called an *annotation*, 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. For example, an annotation might be processed by a source-code generator. The term *metadata* is also used to refer to this feature, but the term *annotation* is the most descriptive and more commonly used.



Specifying a Retention Policy

Before exploring annotations further, it is necessary to discuss *annotation retention policies.* A retention policy determines at what point an annotation is discarded. Java defines three such policies, which are encapsulated within the **java.lang.annotation.RetentionPolicy** enumeration. They are **SOURCE**, **CLASS**, and **RUNTIME**.

An annotation with a retention policy of **SOURCE** is retained only in the source file and is discarded during compilation.

An annotation with a retention policy of **CLASS** is stored in the **.class** file during compilation. However, it is not available through the JVM during run time.

An annotation with a retention policy of **RUNTIME** is stored in the **.class** file during compilation and is available through the JVM during run time. Thus, **RUNTIME** retention offers the greatest annotation persistence.

A retention policy is specified for an annotation by using one of Java's built-in annotations: **@Retention**. Its general form is shown here:

@Retention(*retention-policy*)

Here, *retention-policy* must be one of the previously discussed enumeration constants. If no retention policy is specified for an annotation, then the default policy of **CLASS** is used.

The following version of **MyAnno** uses **@Retention** to specify the **RUNTIME** retention policy. Thus, **MyAnno** will be available to the JVM during program execution.

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str();
  int val();
}
```

To do this, we need to configure *@Retention* with one of three retention policies:

1. *RetentionPolicy.SOURCE* – visible by neither the compiler nor the runtime
2. *RetentionPolicy.CLASS* – visible by the compiler
3. *RetentionPolicy.RUNTIME* – visible by the compiler and the runtime

Q 4 b) Discuss how reflections can be used at runtime with annotations

[illegible]

Here is a program that assembles all of the pieces shown earlier and uses reflection to display the annotation associated with a method.

```
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
```

This program uses reflection as described to obtain and display the values of **str** and **val** in the **MyAnno** annotation associated with **myMeth( )** in the **Meta** class. There are two things to pay special attention to. First, in this line

```
MyAnno anno = m.getAnnotation(MyAnno.class);
```

notice the expression **MyAnno.class**. This expression evaluates to a **Class** object of type **MyAnno**, the annotation. This construct is called a *class literal*. You can use this type of expression whenever a **Class** object of a known class is needed. For example, this statement could have been used to obtain the **Class** object for **Meta**:

```
Class c = Meta.class;
```

Q. 5 a) What is the Result Set? Explain the types of Result Sets in JDBC

1. ResultSet object contain the methods that are used to copy data from ResultSet into java collection object or variable for further processing.

2. Data in the ResultSet is logically organized into the virtual table for further processing. Result set along with row and column it also contains meta data.

3. ResultSet uses virtual cursor to point to a row of the table.

Scrollable Result Set

1. Until the release of JDBC 2.1 API , the virtual cursor can move only in forward directions. But today the virtual cursor can be positioned at a specific row.
2. There are six methods to position the cursor at specific location in addition to next() in scrollable result set. firs() ,last(), absolute(), relative(), previous(), and getRow().
3. first() ………… position at first row.
4. last()…………position at last row.
5. previous()……….position at previous row.
6. absolute()………. To the row specified in the absolute function
7. relative()………… move relative to current row. Positive and negative no can be given.
   Ex. relative(-4) … 4 position backward direction.
8. getRow() ……… returns the no of current row.
9. There are three constants can be passed to the createStatement()
10. Default is TYPE_FORWARD_ONLY. Otherwise three constant can be passed to the create statement 1.) TYPE_SCROLL_INSENSITIVE

    2.) TYPE_SCROLL_SENSITIVE

11. TYPE_SCROLL makes cursor to move both direction. INSENSITIVE makes changes made by J2EE component will not reflect. SENSITIVE means changes by J2EE will reflect in the result set.
    Example code.

    ```
    String sql=" select * from emp";
    DR=Db.createStatement(TYPE_SCROLL_INSENSITIVE);
    RS= DR.executeQuery(sql);
    ```
12. Now we can use all the methods of ResultSet.

updatable Result Set.

1. Rows contained in the result set is updatable similar to how rows in the table can be updated. This is possible by sending CONCUR_UPDATABLE.
2. There are three ways in which result set can be changed. These are updating row , deleting a row, inserting a new row.
3. **Update ResultSet**

A ResultSet can have one of two concurrency levels:

1. ResultSet.CONCUR_READ_ONLY
2. ResultSet.CONCUR_UPDATABLE

- Once the executeQuery() method of the statement object returns a result set. updatexxx() method is used to change the value of column in the current row of result set.
- It requires two parameters, position of the column in query. Second parameter is value
- updateRow() method is called after all the updatexxx() methods are called.

  Example:

```
try{
        String query= "select Fname, Lname from Customers
        where Fname= 'Mary' and Lanme='Smith';
        DataRequest= Db.
        createStatement(ResultSet.CONCUR_UPDATABLE);
        Rs= DataRequest.executeQuery(query);
        Rs.updateString("LastName","Smith");
        Rs.updateRow();
}
```

4. **Delete row in result set**
   - ❖ By using absolute method positioning the virtual cursor and calling deleteRow(int n) n is the number of rows to be deleted.
   - ❖ Rs.deleteRow(0) current row is deleted.
5. **Insert Row in result set**
   - ❖ Once the executeQuery() method of the statement object returns a result set. updatexxx() method is used to insert the new row of result set.
   - ❖ It requires two parameters, position of the column in query. Second parameter is value
   - ❖ insertRow() method is called after all the updatexxx() methods are called.

```
try{
        String query= "select Fname, Lname from Customers
        where Fname= 'Mary' and Lanme='Smith';
        DataRequest= Db.
        createStatement(ResultSet.CONCUR_UPDATABLE);
        Rs= DataRequest.executeQuery(query);
        Rs.updateString(1 ,"Jon");
        Rs.updateString(2 ,"Smith");
        Rs.insertRow();
}
```

ResultSet Holdability

The ResultSet holdability determines if a ResultSet is closed when the commit() method of the underlying connection is called.

There are two types of holdability:

1. ResultSet.CLOSE_CURSORS_OVER_COMMIT
2. ResultSet.HOLD_CURSORS_OVER_COMMIT

The CLOSE_CURSORS_OVER_COMMIT holdability means that all ResultSet instances are closed when connection.commit() method is called on the connection that created the ResultSet.

The HOLD_CURSORS_OVER_COMMIT holdability means that the ResultSet is kept open when the connection.commit() method is called on the connection that created the ResultSet.

Q. 6 a) What is Collection Framework? And explain methods defined by Collection interface

- The Java Collections Framework standardizes the way in which groups of objects are handled by your programs.
- The framework had to be high-performance.
- The implementations for the fundamental collections(dynamic arrays, linked lists, trees, and hash tables) are highly efficient.
- The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.

interface Collection<E>

E specifies the type of objects that the collection

Collection extends the Iterable interface.

Iterating through the list cane be done through the iteratable interface.

Methods in collection interface

add

boolean add(E obj)

adds obj to the invoking collection.

Returns true if obj was added to the collection.

Returns false if obj is already a member of the collection and the collection does not allow duplicates.

addAll

boolean addAll(Collection<? extends E> c)

Adds all the elements of c to the invoking collection.

Returns true if the operation succeeded

(i.e., the elements were added). Otherwise, returns false.

clear

void clear( )

Removes all elements from the invoking collection.

contains

boolean contains(Object obj)

Returns true if obj is an element of the invoking collection.

Otherwise, returns false.

containsAll

boolean containsAll(Collection<?> c)

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

equals

boolean equals(Object obj)

Returns true if the invoking collection and obj are equal.

Otherwise, returns false.

hashCode

int hashCode( ) Returns the hash code for the invoking collection.

isEmpty

boolean isEmpty( )

Returns true if the invoking collection is empty.

Otherwise, returns false.

iterator

Iterator<E> iterator( ) Returns an iterator for the invoking collection.

remove

boolean remove(Object obj)

Removes one instance of obj from the invoking collection.

Returns true if the element was removed. Otherwise, returns false.

removeAll

boolean removeAll(Collection<?> c)

clear

void clear( )

Removes all elements from the invoking collection.

contains

boolean contains(Object obj)

Returns true if obj is an element of the invoking collection.

Otherwise, returns false.

containsAll

boolean containsAll(Collection<?> c)

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

equals

boolean equals(Object obj)

Returns true if the invoking collection and obj are equal.

Otherwise, returns false.

hashCode

int hashCode( ) Returns the hash code for the invoking collection.

isEmpty

boolean isEmpty( )

Returns true if the invoking collection is empty.

Otherwise, returns false.

iterator

Iterator<E> iterator( ) Returns an iterator for the invoking collection.

remove

boolean remove(Object obj)

Removes one instance of obj from the invoking collection.

Returns true if the element was removed. Otherwise, returns false.

removeAll

boolean removeAll(Collection<?> c)

Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false.

retainAll

boolean retainAll(Collection<?> c)

Removes all elements from the invoking collection except those in c.

Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false.

size

int size( ) Returns the number of elements held in the invoking collection.

toArray

Object[ ] toArray( )

Returns an array that contains all the elements stored in the invoking collection.

The array elements are copies of the collection elements.

The array elements are copies of the collection elements.

If the size of array equals the number of elements, these are returned in array.

Q. 7 a) Explain the following built in annotations with program as an example @override @inherited @Retention

Annotations (Metadata)

Beginning with JDK 5, a new facility was added to Java that enables you to embed supplemental information into a source file. This information, called an *annotation*, 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. For example, an annotation might be processed by a source-code generator. The term *metadata* is also used to refer to this feature, but the term *annotation* is the most descriptive and more commonly used.

@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 eatSomething
}

class TestAnnotation1{
public static void main(String args[]){
Animal a=new Dog();
a.eatSomething();
}}
```

Output:Comple Time Error

@Inherited

By default, annotations are not inherited to subclasses. The @Inherited annotation marks the annotation to be inherited to subclasses.

```
@Inherited
```

```
@interface ForEveryone { }//Now it will be available to subclass also

@interface ForEveryone { }
class Superclass{}

class Subclass extends Superclass{}
```

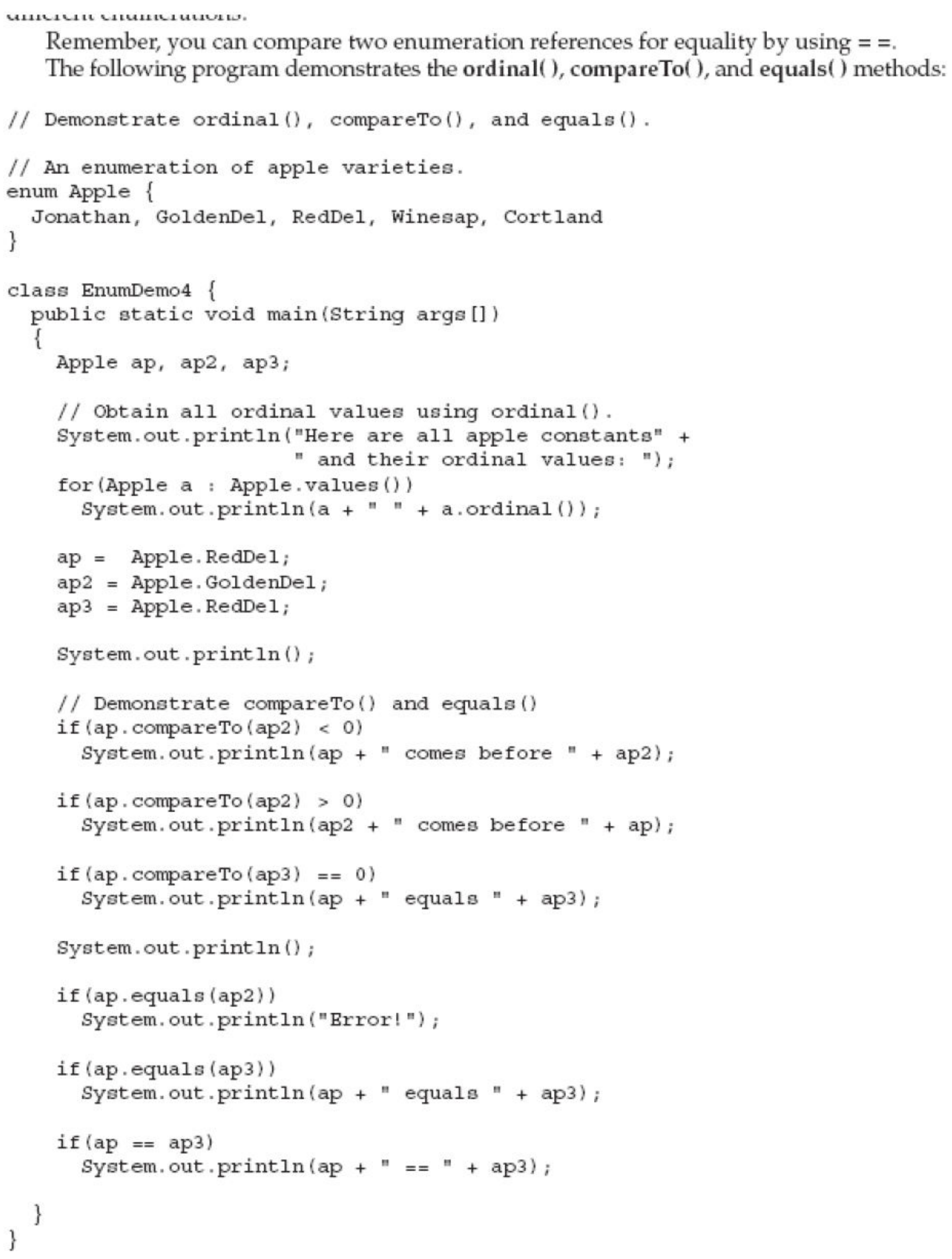
@Retention

@Retention annotation is used to specify to what level annotation will be available.

```
//Creating annotation
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface MyAnnotation{
int value();
}

//Applying annotation
class Hello{
@MyAnnotation(value=10)
public void sayHello(){System.out.println("hello annotation");}
}

//Accessing annotation
class TestCustomAnnotation1{
public static void main(String args[])throws Exception{

Hello h=new Hello();
Method m=h.getClass().getMethod("sayHello");

MyAnnotation manno=m.getAnnotation(MyAnnotation.class);
System.out.println("value is: "+manno.value());
}}
```

```
Output:value is: 10
```

Q. 7 b) Explain the following methods of java.lang.Enum with an example program i)ordinal( ) ii)compareTo( )



different enumerations.

Remember, you can compare two enumeration references for equality by using = =.

The following program demonstrates the **ordinal( )**, **compareTo( )**, and **equals( )** methods:

```
// 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);

  }
}
```

The output from the program is shown here:

```
Here are all apple constants and their ordinal values:
Jonathan 0
GoldenDel 1
RedDel 2
Winesap 3
Cortland 4

GoldenDel comes before RedDel
RedDel equals RedDel

RedDel equals RedDel
RedDel == RedDel
```