Module - 1: Enumerations, Autoboxing and Annotations(metadata):

SYLLABUS

Enumerations, Autoboxing and Annotations(metadata):

Enumerations, Enumeration fundamentals, the values() and valueOf() Methods, java enumerations are class types, enumerations Inherits Enum, example, type wrappers, Autoboxing, Autoboxing and Methods, Autoboxing/Unboxing occurs inExpressions, Autoboxing/Unboxing, Boolean and character values, Autoboxing/Unboxing helps prevent errors, A word of Warning. Annotations, Annotation basics, specifying retention policy, Obtaining Annotations at run time by use of reflection, Annotated element Interface, Using Default values, Marker Annotations, Single Member annotations, Built-In annotations.

ENUMERATIONS

- ✓ **"Enumeration** means a list of named constant**,** enum in java is a data typethat contains 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) etc.
- ✓ It is available from JDK 1.5.
- ✓ An Enumeration can have constructors, methods and instance variables. It is created using **enum** keyword.
- ✓ Each enumeration constant is *public*, *static* and *final* by default.
- ✓ Even though enumeration defines a class type and have constructors, you donot instantiate an **enum** using **new**.
- ✓ Enumeration variables are used and declared in much a same way as you doa primitive variable.

ENUMERATION FUNDAMENTALS

How to Define and Use an Enumeration

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

```
enum Subject          //Enumeration defined
{
      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.

Ex: 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;

  Ex: sub = Subject.Java;

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

  Example:

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

Program 1: 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 WeekDayswk =
                WeekDays.sun;
 //wk can be assigned only the constants defined under enum type Weekdays
                 System.out.println("Today is "+wk);
        }
}
```

Output : Today is sun

Program 2: Example of Enumeration using switch statement

```
enum Restaurants
{
	DOMINOS, KFC, PIZZAHUT, PANINOS, BURGERKING
}
class Test
{
	public static void main(String args[])
	{
		Restaurants r;
		r = Restaurants. PANINOS;
		switch(r)
		{

			case DOMINOS:
					System.out.println("I AM " + r.DOMINOS);
					break;
			case KFC:
					System.out.println("I AM " + r.KFC);
					break;
			case PIZZAHUT:
					System.out.println("I AM " + r.PIZZAHUT);
					break;
			case PANINOS:
					System.out.println("I AM " + r.PANINOS);
					break;
			case BURGERKING:
					System.out.println("I AM " + r.BURGERKING);
					break;
		}
	}
}
```

Output:

```
I AM PANINOS
```

values() and valueOf() Methods

- ✓ The java compiler internally adds the values() method when it creates an enum.
- ✓ The values() method returns an array containing all the values of the enum.
- ✓ Its general form is,

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

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

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

Program 3: 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 Restaurantsfor(Restaurants a :
                    rArray) //using foreach loop
                              System.out.println(a);
                    r = Restaurants.valueOf("dominos");
                    System.out.println("I AM " + r);
          }
}
```

Output:

```
All constants of enum type Restaurants are:
DOMINOS
KFC
PIZZAHUT
PANINOS
BURGERKIN
G
I AM DOMINOS
```

Points to remember about Enumerations

1. Enumerations are of class type, and have all the capabilities that a Java classhas.
2. Enumerations can have Constructors, instance Variables, methods and caneven implement Interfaces.
3. Enumerations are not instantiated using **new** keyword.
4. All Enumerations by default inherit **java.lang.Enum** class.
5. enum may implement many interfaces but cannot extend any class because it internally extends Enum class

---

JAVA ENUMERATIONS ARE CLASS TYPES

Enumeration with Constructor, instance variable and Method

- ✓ Java Enumerations Are Class Type.
- ✓ It is important to understand that each enum constant is an object of its enumeration type.
- ✓ When you define a constructor for an enum, the constructor is called wheneach enumeration constant is created.
- ✓ Also, each enumeration constant has its own copy of any instance variables defined by the enumeration

Program 4: Example of Enumeration with Constructor, instance variable andMethod

```
enum Apple2
{
	Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);

		// variable
		int price;

		// Constructor
		Apple2(int p)
		{
			price = p;
		}

		//method
		int getPrice()
		{
			return price;
		}
}

public class EnumConstructor
{

	public static void main(String[] args)
	{

		Apple2 ap;

		// Display price of Winesap.
System.out.println("Winesap costs " + Apple2.Winesap.getPrice() + " cents.\n");
System.out.println(Apple2.GoldenDel.price);

		// Display all apples and prices.
		System.out.println("All apple prices:");
		for(Apple2 a : Apple2.values())
			System.out.println(a + " costs " + a.getPrice() + " cents.");
	}

}
```

Output:

```
Winesap costs 15 cents.9
All apple prices:
```

```
Jonathan costs 10 cents.
GoldenDel costs 9 cents.
RedDel costs 12 cents.
Winesap costs 15 cents.
Cortland costs 8 cents.
```

- ✓ In this example as soon as we declare an enum variable(Apple2 ap) the constructor is called once, and it initializes value for every enumeration constant with values specified with them in parenthesis.

ENUMERATIONS INHERITS ENUM

- ✓ All enumerations automatically inherit one: **java.lang.Enum.** This class defines several methods that are available for use by all enumerations.
- ✓ You 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.
- ✓ It has this general form: final int ordinal( )
- ✓ It returns the ordinal value of the invoking constant. Ordinal values begin at zero.
- ✓ Thus, in the Apple enumeration, Jonathan has an ordinal value of zero, GoldenDel has an ordinal value of 1, RedDel has an ordinal value of 2, and so on.
- ✓ You can compare the ordinal value of two constants of the same enumerationby using the **compareTo( )** method.
- ✓ It has this general form: final int compareTo(enum-type e), Here, enum-type is the type of the enumeration, and e is the constant being compared to the invoking constant
- ✓ If the invoking constant has an ordinal value less than e's, then compareTo( ) returns a negative value.
- ✓ If the two ordinal values are the same, then zero is returned.

- ✓ If the invoking constant has an ordinal value greater than e’s, then a positive value is returned.

Program 5: Example with ordinal(), comapareTo and equals() methods

```
enum Apple5
{
        Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

public class EnumOrdinal {

        public static void main(String[] args) {
                // TODO Auto-generated method stubApple5
                ap, ap2, ap3;

                // Obtain all ordinal values using ordinal().
                System.out.println("Here are all apple constants and their ordinal values: ");

                for(Apple5 a : Apple5.values())
                        System.out.println(a + " " + a.ordinal());
                        //System.out.println(a + " " + a);

                ap =   Apple5.RedDel; ap2
                = Apple5.GoldenDel;ap3
                = Apple5.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);
                S               ystem.out.println();
```

```
                   .equals(ap3))
                         System.out.println(ap + " equals " + ap3);
if(ap.equals(ap2))
                   if(ap == ap3)
                         System.out.println(ap + " == " + ap3);
         }
```

Output:

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

GoldenDel
comes before
RedDelRedDel
equals RedDel

RedDel
equals
RedDel
RedDel
==
RedDel
```

```
System.out.println("Error!");
```

```
if(ap
```

TYPE WRAPPERS

- ✓ Java uses primitive data types such as int, double, float etc. to hold the basic data types for the sake of performance.
- ✓ Despite the performance benefits offered by the primitive data types, there are situations when you will need an object representation of the primitive data type.
- ✓ For example, many data structures in Java operate on objects. So you cannot use primitive data types with those data structures.
- ✓ To handle such type of situations, Java provides type Wrappers which provide classes that encapsulate a primitive type within an object.

- Character : It encapsulates primitive type char within object.

Character (char *ch*)

- Boolean : It encapsulates primitive type boolean within object.

Boolean (boolean *boolValue*)

- Numeric type wrappers : It is the most commonly used type wrapper.

Byte Short Integer Long Float Double

- Above mentioned Classes comes under Numeric type wrapper. These classes encapsulate byte, short, int, long, float, double primitive type.

BOXING AND UNBOXING

Example:

The following program demonstrates how to use a numeric type wrapper to encapsulate a value and then extract that value.

Program 7: Demonstrate a type wrapper.

```
 class Wrap
{
          public static void main(String args[])
          {
                    Integer iOb = new Integer(100);int
                    i = iOb.intValue();
                    System.out.println(i + " " + iOb); // displays 100 100
          }
}
```

- ✓ This program wraps the integer value 100 inside an Integer object called iOb. The program then obtains this value by calling int Value() and stores the result in i.
- ✓ The process of encapsulating a value with in an object is called **boning.**
- ✓ Thus, in the program, this line boxes the value 100 into an Integer:

Integer iOb = new Integer(100); // **boxing.**

- ✓ The process of extracting a value from a type wrapper is called **unboxing.** For example, the program unboxes the value in iOb with this statement:

int i = iOb.intValue(); // **unboxing**

The same general procedure used by the preceding program to box and unbox values has been employed since the original version of Java. However, with the release of JDK 5, Java fundamentally improved on this through the addition of autoboxing, described next.

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.

Program 8: Example of Autoboxing and Unboxing

```
class Test
{
        public static void main(String[] args)
        {
                Integer iob = 100;
//Auto-boxing of int i.e converting primitive data type int to a Wrapper class Integer

               int i = iob;
//Auto-unboxing of Integer i.e converting Wrapper class Integer to a primitve type int
               System.out.println(i+"  "+iob);

                Character  cob = 'a';
 //Auto-boxing of char i.e converting primitive data type char to a Wrapper class Character

              char  ch = cob;
//Auto-unboxing of Character i.e converting Wrapper class Character to a primitive type char
                System.out.println(cob+"  "+ch);
        }
}
```

Output :

```
100 100
a a
```

Autoboxing / Unboxing in Expressions

✓ Whenever we use object of Wrapper class in an expression, automatic unboxing and boxing is done by JVM.

```
Integer iOb;
iOb = 100;        //Autoboxing of int
```

```
    ++iOb;
```

- ✓ When we perform increment operation on Integer object, it is first unboxed,then incremented and then again reboxed into Integer type object.
- ✓ This will happen always, when we will use Wrapper class objects in expressions or conditions etc.

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 **typecasting**.
3. It helps prevent errors, but may lead to unexpected results sometimes. Hencemust 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 canbe applied.

Program 9: Example of Autoboxing / Unboxing with different types of numericobjects in an expression

```
class Test
{
        public static void main(String args[])
         {
             Integer i = 35;
             Double d = 33.3;
             d = d + i;
             System.out.println("Value of d is " + d);
        }
}
```

Ouput: Value of d is 68.3

Note: When the statement **d = d + i;** was executed, i was auto-unboxed into int, dwas auto-unboxed into double, addition was performed and then finally, auto- boxing of d was done into Double type Wrapper class.

Autoboxing/Unboxing Helps Prevent Errors

- ✓ In addition to the convenience that it offers, autoboxing/unboxing can alsohelp prevent errors. For example, consider the following program:

Program 10: Example an error produced by manual unboxing.

```
class UnboxingError
{
        public static void main(String args[])
        {
                Integer iOb = 1000; // autobox the value 1000
                int i = iOb.byteValue(); // manually unbox as byte !!!System.out.println(i); //
                does not display 1000 !
        }
}
```

- ✓ This program displays not the expected value of 1000, but –24!
- ✓ The reason is that the value inside iOb is manually unboxed by calling byteValue( ), which causes the truncation of the value stored in iOb, which is 1,000.
- ✓ This results in the garbage value of –24 being assigned to i.
- ✓ Auto-unboxing prevents this type of error because the value in iOb willalways autounbox into a value compatible with int.
- ✓ In general, because autoboxing always creates the proper object, and auto-unboxing always produces the proper value, there is no way for the process to produce the wrong type of object or value.

ANNOTATION / METADATA

- ✓ An **annotation is a kind of metadata** in java which can be applied at various elements of java source code so that later some tool, debugger or application program can take advantage of these annotations; and help analysing the program in positive and constructive way.
- ✓ We can annotate classes, methods, variables, parameters and packages in javaOR in one word almost everything.
- ✓ It is important to learn that the annotations applied on java source code is compiled into bytecode with other class members, and using ***reflection*** programmer can query this metadata information to decide the appropriate action to perform in any particular context.

What is this metadata in java language context? Why we even care about them?

Let’s understand the need to metadata with an example.

Below is a sourcecode of class which is declared as final:

```
public final class MyFinalClass
{
   //other class members
}
```

- ✓ Now we have ‘final’ keyword in class declaration. And the impact of this declaration is that you can’t extend this class or make a child class of it. How compiler will understand this? Simply because of ‘**final**‘ keyword. Right? Well, this is called metadata.
- ✓ ***A metadata is data about data.*** Metadata adds some additional flags/informations on your actual data (i.e. in above case the class MyFinalClass), and in runtime either you or JVM who understand these flags/information, can utilize this metadata information to make appropriate decisions based on context.

- ✓ In java, **we use the annotations to denote metadata**. We can annotate classes, interface, methods, parameters and even packages also. We have to utilize the metadata information represented by these annotations in runtime usually.

BUILT-IN ANNOTATIONS IN JAVA

- ✓ Java **Annotation** is a tag that represents the *metadata* i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.
- ✓ Obliviously you can define your own but java does provide some in-built annotations too for ready-made use.
- ✓ Java defines many built-in annotations. Most are specialized, but seven are general purpose.
- ✓ Some annotations are applied to java code and some to other annotations.

Built-In Java Annotations used in java code: Three are imported from java.lang.

- @Override
- @SuppressWarnings
- @Deprecated

Built-In Java Annotations used in other annotations: Four **are** imported from java.lang.annotation

- @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 isbetter to mark @Override annotation that provides assurity that method is overridden.

Program 11: Example of @Override annotation

```
class Animal
 {
   void eatSomething()
   {
         System.out.println("eating something");
         System.out.println("base class");
   }
 }

 class Dog extends Animal
 {

    @Override
   void eatsomething()                 //should be eatSomething
   {
         System.out.println("eating foods");
         System.out.println("derived class");
   }
 }

public class AnnotationDemo1 {

         public static void main(String[] args)
         {
                  // TODO Auto-generated method stubDog
                  a=new Dog();
            a.eatSomething();

         }

}
```

@SuppressWarnings

- ✓ @SuppressWarnings annotation: is used to suppress warnings issued by the compiler.
- ✓ This annotation ***instructs the compiler to suppress the compile time warnings*** specified in the annotation parameters.
- ✓ e.g. to ignore the warnings of unused class attributes and methods use @SuppressWarnings("unused") either for a given attribute or at class level forall the unused attributes and unused methods.

Program 12: Example of @SuppressWarnings annotation

```
/**
 * In the program if you remove the @SuppressWarnings("unchecked") annotation,
 *  it will show warning at compile time because we are using  non-generic collection.
 */


import java.util.*;
class AnnotationDemo2
{
        @SuppressWarnings("unchecked")
   public static void main(String args[])
   {

          ArrayList list=new ArrayList();
            list.add("a");
            list.add("b");
            list.add("c");

            for(Object obj:list)
            System.out.println(obj);

   }
 }
```

@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.

Program 13: Example of @@Deprecated annotation

```
/**
 *@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 betternot to use such
methods.
 */
class A
{
   void m()
   {
         System.out.println("hello m");
   }
  @Deprecate
   dvoid ~~n~~()
   {
         System.out.println("hello n");
   }
}
class  AnnotatonDemo3
{
   public static void main(String args[])
   {
    A a=new A();
   a.~~n~~();
   }
 }
```

Annotations Applied To Other Annotations

- ✓ Generally below discussed four annotations are used inside other annotationsto hint compiler that how new annotation should be treated by JVM.

@Retention

- ✓ This annotation ***specifies how the marked annotation is stored in javaruntime***.
- ✓ Whether it is limited to source code only, embedded into the generated classfile, or it will be available at runtime through reflection as well.

Program 14: Example of specifying Retention Policies(Refer Program 18)

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

    //@Retention(RetentionPolicy.CLASS)
    @Retention(RetentionPolicy.RUNTIME)
//@Retention(RetentionPolicy.SOURCE)
public @interface MyCustomAnnotation
{
   //some code
}
```

@Documented

- ✓ This annotation ***indicates that new annotation should be included into java documents*** generated by java document generator tools.

Program 15: Example of @Documented (Refer Program 18)

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

@Documented

public @interface MyCustomAnnotation

{

  //Some other code

}
```

@Target

- ✓ Use @Target annotation to ***restrict the usage of new annotation on certainjava elements*** such as class, interface or methods.
- ✓ After specifying the targets, you will be able to use the new annotation on given elements only.
- ✓ **@Documented**

Program 16: Example of @Target

```
import java.lang.annotation.ElementType;

import java.lang.annotation.Target;

@Target(value={ElementType.TYPE,ElementType.METHOD,ElementType.CONSTR
UCTOR,ElementType.ANNOTATION_TYPE,ElementType.FIELD,
ElementType.LOCAL_VARIABLE,ElementType.PACKAGE,ElementType.PARAM
ETER})
```

```
public @interface MyCustomAnnotation

{

  //Some other code

}
```

@Inherited

- ✓ When you apply this annotation to any other annotation i.e. @MyCustomAnnotation; and @MyCustomAnnotation is applied of any class MyParentClass then @MyCustomAnnotation will be available to all child classes of MyParentClass as well.
- ✓ It essentially means that when you try to lookup the annotation @MyCustomAnnotation on any class X, then ***all the parent classes of X unto n level are queried for @MyCustomAnnotation***; and if annotation is present at any level then result is true, else false.
- ✓ Please note that by default annotations applied on parent class are not available to child classes.

Program 17: Example of *@Inherited (Refer Program 18)*

```
import java.lang.annotation.Inherited;

    @Inherited

public @interface MyCustomAnnotation

{

  //Some other code

}
```

Custom /User defined Annotations in Java

- ✓ Java allows you to create your own metadata in form of custom annotations. You can create your own annotations for specific purposes and use them as well. Let's learn how to do create custom annotations.

Creating Custom Annotations

To create a custom annotation, you must use the keyword "**@interface**". Other important things to remember while creating custom annotations are listed below:

- Each method declaration defines an element of the annotation type.
- Method declarations must not have any parameters or a throws clause.
- Return types are restricted to primitives, String, Class, enums, annotations,and arrays of the preceding types.
- Methods can have default values.

Types of Annotation

There are three types of annotations.

1. Marker Annotation
2. Single-Value Annotation
3. Multi-Value Annotation

1) Marker Annotation

An annotation that has no method, is called marker annotation. For example:

```
@interface MyAnnotation
{
}
```

The @Override and @Deprecated are marker annotations.

2) Single-Value Annotation

An annotation that has one method, is called single-value annotation. For example:

```
@interface MyAnnotation
{
        int value();
}
```

We can provide the default value also. For example:

```
@interface MyAnnotation
{
        int value() default 0;
}
```

How to apply Single-Value Annotation

Let's see the code to apply the single value annotation.@MyAnnotation(value=10)

The value can be anything.

3) Multi-Value Annotation

An annotation that has more than one method, is called Multi-Value annotation. For example:

```
@interface MyAnnotation
{
        int value1(); String
        value2();
        String value3();
}
}
```

We can provide the default value also. For example:

```
@interface MyAnnotation
{
        int value1() default 1;
        String value2() default "";
        String value3() default "xyz";
}
```

How to apply Multi-Value Annotation

Let's see the code to apply the multi-value annotation.

```
@MyAnnotation(value1=10,value2="varun",value3="Bengaluru")
```

Program 18: Example of custom annotation using @Retention , @Target, @Documented

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Documented

@interface MyAnnotation2
{
        int value() default 2;
        String name() default "cse";
}

//Applying annotation
class Hello
{
        @MyAnnotation2(value=4,name="ise")

        public void sayHello()
        {
                //some code
```

```
          }

}


public class CustomAnnotation
{

          public static void main(String[] args)
          {
                    // TODO Auto-generated method stub

                    try
                    {
                              Hello h=new Hello();

                              // First, get a Class object that represents this class.
                              Class c = h.getClass();

                              System.out.println(c);

                              // Now, get a Method object that represents this method.
                              Method m = c.getMethod("sayHello");

                              System.out.println(m);

                              // Next, get the annotation for this class.

                    MyAnnotation2 anno= m.getAnnotation(MyAnnotation2.class);

                              System.out.println(anno);

                              // Finally, display the values.
                              System.out.println( "hi "+anno.value());
                              System.out.println("hello "+anno.name());


                    }
                    catch(Exception e)
                    {
                    System.out.println("no such method exception"+e.getMessage());
                    }

                    }

}
```

Some Restrictions on annotations

There are a number of restrictions that apply to annotation declarations.

- ✓ First, no annotation can inherit another.
- ✓ Second, all methods declared by an annotation must be without parameters.
- ✓ Furthermore, they must return one of the following:
    - Aprimitive type, such as int or double
    - An object of type String or Class
    - An enum type
    - Another annotation type
    - An array of one of the preceding types

QUESTIONS

1. What is enumeration in java? With an example syntax explain enumeration.
2. Implement Java enumeration by using default explicit, parametrised constructors and method? Explain values() and valueOf() method with suitable example.
3. Enumerations are class type. Justify with suitable example program.
4. With an example program describe a)ordinal() b)compareTo() c) equals() d) Enum class
5. Describe Boxing and Unboxing in java? Write java program for auto boxingand auto unboxing.
6. What is Annotation? List some of the advantages of Annotation in java
7. Apply the annotation for method overriding, method deprecation andwarning avoidance. Describe the above with suitable program.
8. What is Annotation? Explain the types on annotations.
9. Implement a java program
   a) By applying annotation at RUNTIME.
   b) By specify annotation target at method.
   c) Make annotation available for subclass
10. List some of the restrictions imposed on annotations.