CMR
INSTITUTE OF
TECHNOLOGY

USN | | | | | | | | | |

Internal Assessment Test 1– Feb. 2025

Sub:	**Advanced Java& J2EE**							**Sub Code:**	**22MCA341**
Date:	**6/2/2025**	**Duration:**	**90 min's**	**Max Marks:**	**50**	**Sem:**	**III**	**Branch:**	**MCA**

Note : Answer FIVE FULL Questions, choosing ONE full question from each Module

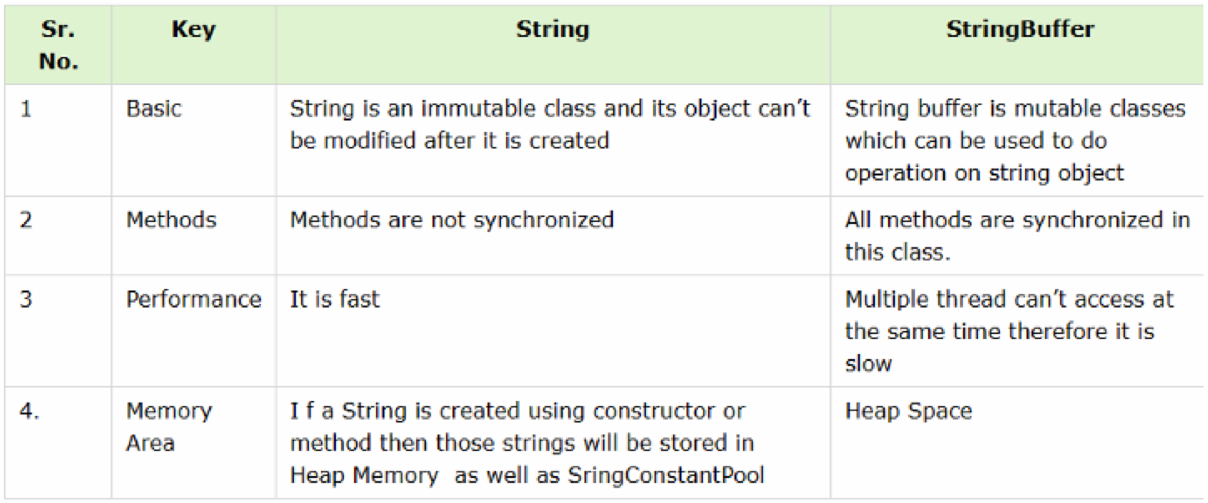
		MARKS	OBE	
			CO	RBT
	PART I			
1	What is enum? Demonstrate the use of ordinal(), compareTo() and equals() method with enumeration	10	CO1	L2
	OR			
2.	Differentiate String and StringBuffer classes. Write a program to demonstrate different constructors of String class.	10	CO3	L3
	PART II			
3	Write a program to create a enum with different months of a year Write a switch case that accepts enum value to perform the events based on the month which the user inputs.	10	CO1	L3
	OR			
4.	Illustrate the use of the following with an example i) equals() ii) compareTo() iii)= = iv) equalsIgnoreCase()	10	CO3	L2
	PART III			
5.	Discuss built-in annotations with example program	10	CO1	L2
	OR			
6.	What is annotation? Explain how do you obtain annotation at run time by using reflection?	10	CO1, CO3	L2
	PART IV			
7.	Explain the use of Values() and valueOf() methods with suitable examples?	10	CO1	L2
	OR			
8.	Illustrate character extraction methods with example.	10	CO3	L2
	PART V			
9.	Write a java program that accept a string, delete the vowels of a string and print .	10	CO3	L3
	OR			
10.	Define with example for each of the following: i)autoboxing ii)Auto unboxing iii) Type Wrapper iv) Marker Annotation	10	CO2	L3

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

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.
We can't inherit a superclass when declaring an enum, all enumerations automatically inherit one class i.e., java.lang.Enum. The **Enum** class defines several methods such as **ordinal( )**, **compareTo( )**, **equals( )** and so on, that are available for use by all enumerations. 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( )
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)
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);
}
}
```

2. Differentiate string and stringbuffer classes. Write a program to demonstrate different constructors of String class.



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

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

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

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

                // Creating a string from a character array
                char[] charArray = {'J', 'a', 'v', 'a'};
                String fromCharArray = new String(charArray);
                System.out.println("String from Character Array: " + fromCharArray);
                // Creating a string from Unicode code points
                int[] codePoints = {72, 101, 108, 108, 111}; // Unicode code points for
               "Hello"
               String fromCodePoints = new String(codePoints, 0, codePoints.length);
                System.out.println("String from Code Points: " + fromCodePoints);

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

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

3. **Write a program to create a enum with different months of a year. Write a switch case that accepts enum value to perform the events based on the month which the user inputs.**

```
import java.util.Scanner;
enum Month {
   JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST,
SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER;
}

class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Ask user to enter a month
        System.out.println("Enter a month (e.g., JANUARY, FEBRUARY, etc.):");
        String input = scanner.next().toUpperCase();

        try {
            Month month = Month.valueOf(input);

            // Switch case to handle different months
            switch (month) {
                case JANUARY:
                    System.out.println("New Year's Celebration!");
                    break;
                case FEBRUARY:
```

```
                System.out.println("Valentine's Day Event!");
                break;
            case MARCH:
                System.out.println("Spring Festival!");
                break;
            case APRIL:
                System.out.println("April Fool's Day Pranks!");
                break;
            case MAY:
                System.out.println("Mother's Day Celebration!");
                break;
            case JUNE:
                System.out.println("Summer Vacation Starts!");
                break;
            case JULY:
                System.out.println("Independence Celebrations!");
                break;
            case AUGUST:
                System.out.println("Monsoon Season Events!");
                break;
            case SEPTEMBER:
                System.out.println("Back to School!");
                break;
            case OCTOBER:
                System.out.println("Halloween Party!");
                break;
            case NOVEMBER:
                System.out.println("Thanksgiving Feast!");
                break;
            case DECEMBER:
                System.out.println("Christmas and New Year Festivities!");
                break;
            default:
                System.out.println("Invalid month!");
        }
    } catch (IllegalArgumentException e) {
        System.out.println("Invalid month entered. Please try again.");
    }

    scanner.close();
  }
}
```

4. **Illustrate the us e of the following with an example**

i)equals() ii)compareTo() iii)== iv)equalsIgnoreCase()

i)equals()

Equals() is used to compare two strings for equality. It return true if both the strings are same otherwise return false.

General form:

boolean equals(Object str)

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

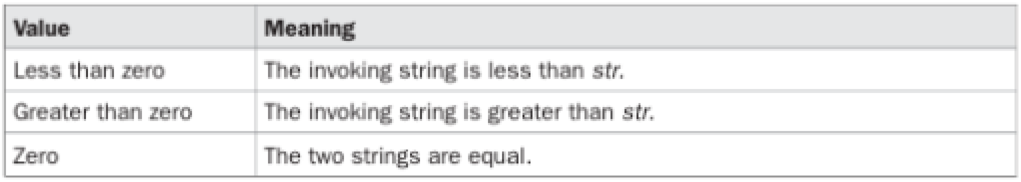
```
Ex: String str=”college”;
String str1=”college”
System.out.println(str.equals(str1);
```

ii)compareTo(): This method is used to compare the strings as per lexicographic order.

General form:

int compareTo(String str)

Here, str is the String being compared with the invoking String. The result of the comparison is returned and is interpreted, as shown here:



Value	Meaning
Less than zero	The invoking string is less than *str.*
Greater than zero	The invoking string is greater than *str.*
Zero	The two strings are equal.

Ex:

```
String s1=”Ram”;
String s2=”seethe”;
System.out.println(s1.compareTo(s2));
```

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

Ex:

```
class EqualsNotEqualTo {
public static void main(String args[]) {
String s1 = "Hello";
String s2 = new String(s1);
System.out.println(s1 + " equals " + s2 + " -> " +
s1.equals(s2));
System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
}
}
```

iv) equalsIgnoreCase() To compare two strings for equality by ignoring the case

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";
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));
}
}
```

5. **Discuss built-in annotations with example program**

Built-In Java Annotations
@Override
@SuppressWarnings
@Deprecated
@Target
@Retention
@Inherited
@Documented

@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()
```
@SuppressWarnings
It is used to inform the compiler to suppress specified compiler warnings. The warnings to suppress are specified by name, in string form. This type of annotation can be applied to any type of declaration.
Java groups warnings under two categories. They are : deprecation and unchecked.. Any unchecked warning is generated when a legacy code interfaces with a code that use generics.

```
class DeprecatedTest
{
@Deprecated
public void Display()
{
System.out.println("Deprecatedtest display()");
}
}

public class SuppressWarningTest
{
// If we comment below annotation, program generates
// warning
@SuppressWarnings({"checked", "deprecation"}) public static void main(String args[])
{
DeprecatedTest d1 = new DeprecatedTest(); d1.Display();
}
}
```
Output:
```
Deprecatedtest display()
```

@Deprecated It is a marker annotation. It indicates that a declaration is obsolete and has been replaced by a newer form.The Javadoc @deprecated tag should be used when an element has been deprecated.

```
public class DeprecatedTest
{
@Deprecated
public void Display()
{
System.out.println("Deprecatedtest display()");
}

public static void main(String args[])
{
DeprecatedTest d1 = new DeprecatedTest();
d1.Display();
}
}
```

Output:

```
Deprecatedtest display()
```

@Documented It is a marker interface that tells a tool that an annotation is to be documented.

Annotations are not included by Javadoc comments. Use of @Documented annotation in the code enables tools like Javadoc to process it and include the annotation type information in the generated document.

```
java.lang.annotation.Documented @Documented
public @interface MyCustomAnnotation {
//Annotation body
}
@MyCustomAnnotation public class MyClass {
//Class body
}
```

While generating the javadoc for class MyClass, the annotation @MyCustomAnnotation would be included in that.

@Inherited

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

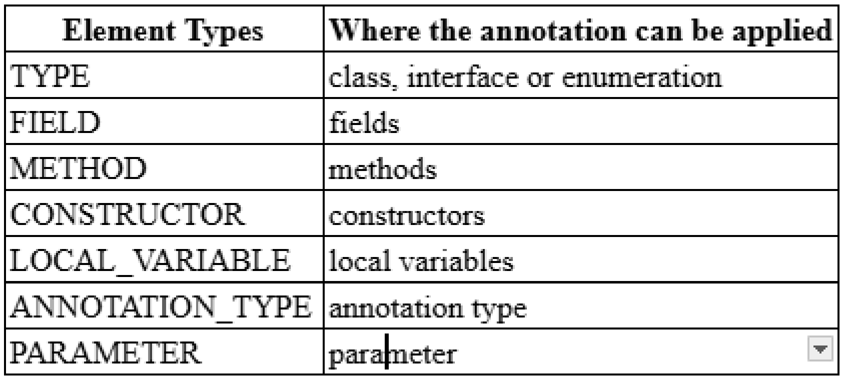
```
@Inherited
public @interface MyCustomAnnotation {

}
@MyCustomAnnotation public class MyParentClass {
...
}
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.

@Target

It specifies where we can use the annotation. For example: In the below code, we have defined the target type as METHOD which means the below annotation can only be used on methods.
The java.lang.annotation.ElementType enum declares many constants to specify the type of element where annotation is to be applied such as TYPE, METHOD, FIELD etc.
element TypesWhere the annotation can be applied



Element Types	Where the annotation can be applied
TYPE	class, interface or enumeration
FIELD	fields
METHOD	methods
CONSTRUCTOR	constructors
LOCAL_VARIABLE	local variables
ANNOTATION_TYPE	annotation type
PARAMETER	parameter

Ex:

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

@Target({ElementType.METHOD})
// u can also target multiple elements
//@Target({ ElementType.FIELD, ElementType.METHOD}) public @interface
MyCustomAnnotation {

}
public class MyClass { @MyCustomAnnotation

public void myMethod()
{
//Doing something
}
}
```

@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.METHOD)
@Inherited
@Retention(RetentionPolicy.RUNTIME)
 public @interface MyCustomAnnotation{
int studentAge() default 18;
 String studentName();
String stuAddress();
String stuStream() default "CSE";
}
@MyCustomAnnotation( studentName="umesh", stuAddress="India")
 public class MyClass {
...
}
```

6. **What is annotation? Explain how do you obtain annotation at run time by using reflection?**

Annotations were added to the java from JDK 5.
 Annotations, does not change the actions of a program.
 Thus, an annotation leaves the semantics of a program unchanged.
 However, this information can be used by various tools during both development and deployment.
  Annotations start with ‘@’.
  Annotations do not change action of a compiled program.
  Annotations help to associate metadata (information) to the program elements i.e. instance variables,
 constructors, methods, classes, etc.
  Annotations are not pure comments as they can change the way a program is treated by compiler.
 Reflection is an API which is used to examine or modify the behavior of methods, classes, interfaces at
 runtime.
The required classes for reflection are provided under java.lang.reflect package.
 Reflection can be used to get information about
● Class: The getClass() method is used to get the name of the class to which an object belongs.
 ● Constructors: The getConstructors() method is used to get the public constructors of the class
 to which an object belongs.
 ● Methods: The getMethods() method is used to get the public methods of the class to which
 an objects belongs.

```
 Import  java.lang.annotation.*;
 import java.lang.reflect.*;
 //
 An annotation type declaration.
 @Retention(RetentionPolicy.RUNTIME)
 @interface MyAnno {
 String str();
 Int  val();
 }
 class Meta {
 // Annotate a method.
 @MyAnno(str = "Annotation Example", val = 100)
 public static void myMeth() {
 Meta ob = new Meta();
 // Obtain the annotation for this method
 // and display the values of the members.
 try {
 // First, get a Class object that represents
 // this class.
 Class c = ob.getClass();
 // Now, get a Method object that represents
 // this method.
 Method m = c.getMethod("myMeth");
 // Next, get the annotation for this class.
 Finally,
 MyAnno anno = m.getAnnotation(MyAnno.class);
 //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
```

7. **Explain the use of Values() and valueOf() methods with suitable examples?**

Values( ) and ValueOf( ) method

All the enumerations has predefined methods values() and valueOf().

values() method returns an array of enum-type containing all the enumeration constants in it.

Its general form is,

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

valueOf() method is used to return the enumeration constant whose value is equal to the string
passed in as argument while calling this method.

It's general form is,

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

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

```
enum Restaurants {
dominos, kfc, pizzahut, paninos, burgerking
}
class Test {
public static void main(String args[])
{
Restaurants r;
System.out.println("All constants of enum type Restaurants are:");
```

```
 Restaurants  rArray[] = Restaurants.values(); //returns an array of constants of type
Restaurants
 for(Restaurants a : rArray) //using foreach loop
 System.out.println(a);
 r = Restaurants.valueOf("dominos");
System.out.println("I AM " + r);
 }
 }
```

8. **Illustrate character extraction methods with example.**

The String class provides a number of ways in which characters can be extracted from a String object.

a. 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.

b. getChars( ):

When we need to extract more than one character at a time, we can use the getChars( ) method

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

Here, sourceStart specifies the index of the beginning of the substring, and sourceEnd specifies an index that is one past the end of the desired substring. Thus, the substring contains the characters from sourceStart through sourceEnd–1. The array that will receive the characters is specified by target.

The index within target at which the substring will be copied is passed in targetStart.

Ex:

```
class getCharsDemo {
public static void main(String args[]) {
String s = "This is a demo of the getChars method.";
int start = 10;
int end = 14;
char buf[] = new char[end - start];
s.getChars(start, end, buf, 0);
System.out.println(buf);
}
}
```

output:

```
demo
```

4.c. getBytes( )

getBytes( ) stores the characters in an array of bytes and it uses the default character-to-byte conversions provided by the platform.

General form:

byte[ ] getBytes( )

getBytes( ) is most useful when you are exporting a String value into an environment that does not support 16-bit Unicode characters. For example, most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

Ex:

```
String s=new String(“Hello World Hello”);
byte[] b=getBytes();
for(byte b1:ba)
Syste.out.println(b1);
```

4.d. toCharArray( )

If we want to convert all the characters in a String object into a character array, we can use CharArray( ). It returns an array of characters for the entire string.

General form:

char[ ] toCharArray( )

Ex:

```
String sa=”Good Morning”;
Char[] ch=sa.toCharArray();
```

9. **Write a java program that accept a string, delete the vowels of a string and print .**

```
import java.util.Scanner;
class Main {
    public static void main(String[] args) {

      Scanner scanner = new Scanner(System.in);

      // Ask user to enter a string
      System.out.println("Enter a string:");
      String input = scanner.nextLine();

      // Remove vowels from the string
      String result = input.replaceAll("[AEIOUaeiou]", "");

      // Print the modified string
      System.out.println("String after removing vowels: " + result);

      scanner.close();
   }
}
```

10. Define with example for each of the following:
i)autoboxing ii)Auto unboxing iii) Type Wrapper iv) Marker Annotation

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

Ex: class AutoBox4 {

```
public static void main(String args[]) {
Integer iOb = 100;
Double dOb = 98.6;
dOb = dOb + iOb;
System.out.println("dOb after expression: " + dOb);
}
}
```

c) Type Wrapper: They convert primitive data types into objects. Objects are needed if we wish modify the arguments passed into a method (because primitive types are passed by value).
Ex: Integer i=900;
d) Marker Annotations:
The only purpose is to mark a declaration. These annotations contain no members and do not consist any data. Thus, its presence as an annotation is sufficient. Since, marker interface contains no members, simply determining whether it is present or absent is sufficient. @Override , @Deprecated is an example of Marker Annotation.

Example Program:

```
import java.lang.annotation.*;
import java.lang.reflect.*;
// A marker annotation.
@Retention(RetentionPolicy.RUNTIME)
@interface MyMarker { }
class Marker {
// Annotate a method using a marker.
// Notice that no ( ) is needed.
@MyMarker
public static void myMeth() {
Marker ob = new Marker();
try {
Method m = ob.getClass().getMethod("myMeth");
// Determine if the annotation is present.
if(m.isAnnotationPresent(MyMarker.class))
System.out.println("MyMarker is present.");
} catch (NoSuchMethodException exc) {
System.out.println("Method Not Found.");
}
}
public static void main(String args[]) {
myMeth();
}
}
```

The output, shown here, confirms that @MyMarker is present:

```
MyMarker is present.
```