USN | | | | | | | | | | |

Internal Assessment Test 1 – Sept. 2017

Sub:	Advanced Java and J2EE					Sub Code:	15CS553	Branch:	ISE
Date:	21/9/2017	Duration:	90 min's	Max Marks:	50	Sem / Sec:	V /A		

1 (a) **What is an annotation? Explain built in annotations in detail.**

Java that enables us 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.

Annotation Basics

An annotation is created through a mechanism based on the **interface**. Let's begin with an example. Here is the declaration for an annotation called **MyAnno**:

```
// A simple annotation type.
@interface MyAnno {
String str();
int val();
}
```

First, notice the **@** that precedes the keyword **interface**. This tells the compiler that an annotation type is being declared. Next, notice the two members **str( )** and **val( )**. All annotations consist solely of method declarations. However, we don't provide bodies for these methods. Instead, Java implements these methods. Moreover, the methods act much like fields.

An annotation cannot include an **extends** clause. However, all annotation types automatically extend the **Annotation** interface. Thus, **Annotation** is a super-interface of all annotations. It is declared within the **java.lang.annotation** package. It overrides **hashCode( ),equals( )**, and **toString( )**, which are defined by **Object**. It also specifies **annotationType( )**, which returns a **Class** object that represents the invoking annotation.

Once we have declared an annotation, we can use it to annotate a declaration. Any type of declaration can have an annotation associated with it. For example, classes, methods, fields, parameters, and **enum** constants can be annotated. Even an annotation can be annotated.

In all cases, the annotation precedes the rest of the declaration.

When you apply an annotation, we give values to its members. For example, here is an example of **MyAnno** being applied to a method:

```
// Annotate a method.
@MyAnno(str = "Annotation Example", val = 100)
public static void myMeth() { // ...
}
```

This annotation is linked with the method **myMeth().**The name of the annotation, preceded by an **@**, is followed by a parenthesized list of member initializations. To give a member a value, that member's name is assigned a value. Therefore, in the example, the string "Annotation Example" is assigned to the **str** member of **MyAnno**.

2 (a) What are marker annotations? Explain with an example

A marker annotation is a special kind of annotation that contains no members. Its sole purpose is to mark a declaration. Thus, its presence as an annotation is sufficient. The best way to determine if a marker annotation is present is to use the method isAnnotationPresent( ), which is a defined by the AnnotatedElement interface.

```
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();
}
}
```

(b) What are the different retention policies that can be specified to an annotation? Illustrate with an example.

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();
```

}

3 (a) What is an enumeration? Explain how to create an enumeration and access the values of enumeration with an example

An enumeration is a list of named constants.In Java, an enumeration defines a class type. By making enumerations into classes, the concept of the enumeration is greatly expanded. For example, in Java, an enumeration can have constructors, methods, and instance variables. Therefore, although enumerations were several years in the making

```
// An enumeration of apple varieties.
enum Apple {
Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
class EnumDemo {
public static void main(String args[])
{
}
Apple ap;
ap = Apple.RedDel;
// Output an enum value.
System.out.println("Value of ap: " + ap);
System.out.println();
ap = Apple.GoldenDel;
// Compare two enum values.
if(ap == Apple.GoldenDel)
System.out.println("ap contains GoldenDel.\n");
// Use an enum to control a switch statement.
switch(ap) {
}
}
case Jonathan:
System.out.println("Jonathan is red.");
break;
case GoldenDel:
System.out.println("Golden Delicious is yellow.");
break;
case RedDel:
System.out.println("Red Delicious is red.");
break;
case Winesap:
System.out.println("Winesap is red.");
break;
case Cortland:
System.out.println("Cortland is red.");
break;
```

The output from the program is shown here:

```
Value of ap: RedDel
ap contains GoldenDel.
Golden Delicious is yellow.
```

(b) What is the use of values () and valueOf() methods? Illustrate with an example.

All enumerations automatically contain two predefined methods: **values( )** and **valueOf().**Their general forms are shown here:

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

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

The **values( )** method returns an array that contains a list of the enumeration constants. The **valueOf( )** method returns the enumeration constant whose value corresponds to the string passed in *str*. In both cases, *enum-type* is the type of the enumeration. For example, in the case of the **Apple** enumeration shown earlier, the return type of **Apple.valueOf("Winesap")** is **Winesap**.

The following program demonstrates the **values( )** and **valueOf( )** methods:

```
// Use the built-in enumeration methods.
// An enumeration of apple varieties.
enum Apple {
Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
class EnumDemo2 {
public static void main(String args[])
{
Apple ap;
System.out.println("Here are all Apple constants:");
// use values()
Apple allapples[] = Apple.values();
for(Apple a : allapples)
System.out.println(a);
System.out.println();
// use valueOf()
ap = Apple.valueOf("Winesap");
System.out.println("ap contains " + ap);
}
}
```

This program uses a for-each style **for** loop to cycle through the array of constants obtained by calling **values( )**. For the sake of illustration, the variable **allapples** was created and assigned a reference to the enumeration array.

eliminating the need for **allapples** variable:

```
for(Apple a : Apple.values())
System.out.println(a);
```

The value corresponding to the name **Winesap** was obtained by calling **valueOf( )**.

```
ap = Apple.valueOf("Winesap");
```

valueOf( ) returns the enumeration value associated with the name of the constant represented as a string.

4 Explain with an example, How do you obtain annotations at run time by use of Reflection

Although annotations are designed mostly for use by other development or deployment tools, if they specify a retention policy of **RUNTIME**, then they can be queried at run time by any Java program through the use of *reflection.* Reflection is the feature that enables information about a class to be obtained at run time. The reflection API is contained in the **java.lang.reflect** package.

The first step to using reflection is to obtain a **Class** object that represents the class whose annotations we want to obtain. **Class** is one of Java's built-in classes and is defined in **java.lang.** We can call **getClass( )** to obtain a **Class** object. Its general form is shown here:

```
final Class getClass( )
```

It returns the **Class** object that represents the invoking object.

After we have obtained a **Class** object, we can use its methods to obtain information about the various items declared by the class, including its annotations. If we want to obtain the annotations associated with a specific item declared within a class, we must first obtain an object that represents that item. For example, **Class** supplies (among others) the **getMethod(**

),**getField( )**, and **getConstructor( )** methods, which obtain information about a method, field and constructor, respectively. These methods return objects of type **Method**, **Field**, and **Constructor**.

We can call **getMethod( )** on that **Class** object, specifying the name of the method. **getMethod( )** has this general form:

Method getMethod(String *methName*, Class ... *paramTypes*)

The name of the method is passed in *methName.* If the method has arguments, then **Class** objects representing those types must also be specified by *paramTypes.* Notice that *paramTypes* is a varargs parameter. This means that we can specify as many parameter types as needed,including zero. **getMethod( )** returns a **Method** object that represents the method. If the methodcan't be found, **NoSuchMethodException** is thrown.

From a **Class**, **Method**, **Field**, or **Constructor** object, we can obtain a specific annotation associated with that object by calling **getAnnotation( )**. Its general form is shown here:

Annotation getAnnotation(Class *annoType*)

Here, *annoType* is a **Class** object that represents the annotation in which we want to retrieve. The method returns a reference to the annotation. Using this reference, we can obtain the values associated with the annotation's members. The method returns **null** if the annotation is not found, which will be the case if the annotation does not have **RUNTIME** retention.

Here is a program that 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();
}
  }
```

5 What is auto boxing? Explain with an example how auto boxing can be performed in methods.

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.

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

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

autoboxing automatically occurs whenever a primitive type must be converted into an object; auto-unboxing takes place whenever an object must be converted into a primitive type. Thus, autoboxing/unboxing might occur when an argument is passed to a method, or when a value is returned by a method.

consider this example:

```
// Autoboxing/unboxing takes place with
// method parameters and return values.
class AutoBox2 {
// Take an Integer parameter and return
// an int value;
static int m(Integer v) {

  return v ; // auto-unbox to int
}
public static void main(String args[]) {
// Pass an int to m() and assign the return value
// to an Integer.  Here, the argument 100 is autoboxed
// into an Integer.  The return value is also autoboxed
// into an Integer.
Integer iOb = m(100);
System.out.println(iOb);
  }
  }
```

6 What is un boxing? With an example, explain how un boxing occurs in expressions

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.

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
```

```
// 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
}
}
```

Auto-unboxing also allows you to mix different types of numeric objects in an expression.

Once the values are unboxed, the standard type promotions and conversions are applied. For example, the following program is perfectly valid:

```
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);
}
}
```