Module 1

Enumerations, Autoboxing and Annotations(metadata)

Presented By: Pankaj Kumar

Module objective

- Identify the need for advanced Java concepts like Enumerations , Wrapper class, Annotations

Enumerations

Presented By: Pankaj Kumar

Enumerations

- Means A list of named constant.
- Defines a class type.
- Have constructors, methods and instance variables.
- Created using **enum** keyword.
- Enumeration constant is *public*, *static* and *final* by default.
- Can define an enum either inside the class or outside the class.

Why And When To Use Enums?

- Use enums when you have values that you know aren't going to change, like month days, days, colors, etc.

How to Define and Use an Enumeration

```
enum Subject
{
Java, Cpp, C, Dbms
}
```

- 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.

  syntax : enum variable as enum_variable = enum_type.enum_constant;

  Ex: sub = Subject.Java;

Example of Enumeration

```
enum WeekDays
{
sun, mon, tues, wed, thurs, fri, sat
}
class Test { public static void main(String args[])
{
WeekDays wk;
WeekDays wk = WeekDays.sun;
System.out.println("Today is "+wk);
} }
```

output:
Today is sun

Enum toString()

- enum class automatically gets a toString() method in the class when compiled.
- toString() method returns a string value of the name of the given enum instance.

Example for toString in ENUM

```
enum WeekDays
{
  sun, mon, tues, wed, thurs, fri, sat
   }
class Test { public static void main(String args[])
{        WeekDays wk;
         WeekDays wk = WeekDays.sun.toString;
         System.out.println("Today is "+wk);
}
}
```

Example of Enumeration using switch statement

```
enum Restaurants
{
dominos, kfc, pizzahut, mc-d
}
class Test {
public static void main(String args[])
{
Restaurants r;
r = Restaurants.kfc;
 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 mc-d: System.out.println("I AM " + r.mc-d);
break;
} } }
```

Values( ) and ValueOf( ) method

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

  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 the method.

  public static enum-type valueOf (String str)

Example of enumeration using values() and valueOf()

```
enum Restaurants

{    dominos, kfc, pizzahut    }

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

 Restaurants r;

System.out.println("All  constants of enum type Restaurants are:");

Restaurants rArray[] = Restaurants.values();

for(Restaurants a : rArray)

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

I AM dominos

Defining the enum inside the class

```
class EnumExample

{

public enum Season { WINTER, SPRING, SUMMER, FALL }

public static void main(String[] args) {

for (Season s : Season.values())

System.out.println(s);

}}
```

Output:

WINTER

SPRING

SUMMER

FALL

Enumeration with Constructor, instance variable and Method

```
enum Student {
advith(20), abhishek(21), ajith(19), bharath(18);
 private int age;
 int getage     {   return age;  }
 Student(int age)
 {
        this.age= age;
 } }
 class EnumDemo
 { public static void main( String args[] )
 {
      Student S;
      System.out.println("Age of bharath is " +Student.bharath.getage()+ "years");  }
```

```
enum Student
 {
indrajith(20), avinash(21), likith(19), chetan,
bagesh(22);
 private int age;
 int getage()
      {
              return age;
       }
 Student(int age)
 {
         this.age= age;
 }
Student()
 {
        age= -1;
}
}
class EnumDemo
{
public static void main( String args[] )
{
      Student S;
      System.out.println("Age of chetan is "
+Student.chetan.getage()+ "years"); }
```

Output:

Age of chetan is -1 years

Enemuration Inherit Enum

- All enum automatically inherit java.lang.Enum.
- We 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:

```
final int ordinal( )
```

- Ordinal values begin at zero
- can compare the ordinal value of two constants of the same enumeration by using the **compareTo( )** method.

```
final int compareTo(enum-type e)
```

- can compare for equality an enumeration constant with any other object by using **equals( ).**

```
enum Direction
{   East, South, West, North   }
public class Main { public static void main(String
args[])
{
Direction ap, ap2, ap3;
for (Direction a : Direction.values()){
System.out.println(a + " " + a.ordinal()); }
ap = Direction.West;
ap2 = Direction.South;
ap3 = Direction.West;
System.out.println();
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);
} }
```

output

East 0

South 1

West 2

North 3

South comes before west

West equals West

West equals West

West == West

Type Wrapper

Presented By: Pankaj Kumar

Introduction

- Java is an object-oriented language and can view everything as an object.
- A simple file can be treated as an object , an address of a system can be seen as an object , an image can be treated as an object (with java.awt.Image) and a simple data type can be converted into an object (with wrapper classes)

- The primitive data types are not objects; they do not belong to any class; they are defined in the language itself.
- Sometimes, it is required to convert data types into objects in Java language
- A data type is to be converted into an object and then added to a Stack or Vector etc.
- For this conversion, the designers introduced wrapper classes.

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Boxing and unboxing

- Converting primitive datatype to object is called **boxing**.
- converting an object into corresponding primitive datatype is known as **unboxing.**

Example Boxing and unboxing

```
class Wrap {

public static void main(String args[]) {

Integer iOb = new Integer(100);

int i = iOb.intValue();

System.out.println(i  + " " + iOb);}

}
```

Result:
100 100

Wrapper class

- provides the mechanism ***to convert primitive into object*** and ***object into primitive*.**

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

Example for autoboxing and 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);
}
}
```

Output:

100 100

Autoboxing/unboxing takes place with method parameter and return values

```
class AutoBox2 {
static int m(Integer v) {
return v ;                    // auto-unbox to int
}
public static void main(String args[]) {
Integer iOb = m(100);
System.out.println(iOb);
}
}
```

Result:
100

Autoboxing/Unboxing Occurs in Expressions

```
Integer iOb, iOb2;

int i;

iOb = 100;

System.out.println("Original value of iOb: " + iOb);

++iOb;

System.out.println("After ++iOb: " + iOb);

iOb2 = iOb + (iOb / 3);

System.out.println("iOb2 after expression: " + iOb2);

i = iOb + (iOb / 3);

System.out.println("i after expression: " + i);
```

Output:
Original value of iOb: 100
After ++iOb: 101
iOb2 after expression: 134
i after expression: 134

Autoboxing/Unboxing Boolean and Character Values

```
Boolean b = true;

if(b)

System.out.println("b  is true");

Character ch = 'x';                    // box a char

char ch2 = ch;                         // unbox a char

System.out.println("ch2  is " + ch2);
```

Output:

b is true

ch2 is x

Java Annotations

Presented By: Pankaj Kumar

Java Annotations

- A tag that represents the *metadata* i.e. attached with class, interface, methods or fields to indicate some additional information
- Used to provide additional information, which can be used by java compiler and JVM.
- Annotations start with '@'.
- Annotations do not change action of a compiled program.

Annotation Basics

- An annotation is created through a mechanism based on the **interface**

```
@interface MyAnno
{
String str();
int val();
}
```

- When we apply an annotation, we give values to its members.

```
@MyAnno(str = "Annotation Example", val = 100)
```

Specifying a Retention Policy : Annotations

- A retention policy determines at what point an annotation is discarded
- Java defined 3 types of retention policies through java.lang.annotation.RetentionPolicy enumeration.

1. SOURCE,
2. CLASS
3. RUNTIME

- **SOURCE**: annotation retained only in the source file and is discarded during compilation.
- **CLASS**: annotation stored in the .class file during compilation, not available in the run time.
- **RUNTIME**: annotation stored in the .class file and available in the run time.

- A retention policy is specified using Java's built-in annotations:

  @Retention(retention-policy)

- The default policy is CLASS

- **Example:**

```
@Retention(RetentionPolicy.RUNTIME)
public @interface MySampleAnn
{
    String name();
    String desc();
}
```

Obtaining Annotations at Run Time by Use of Reflection

- Reflection is the feature that enables information about a class to be obtained at run time
- Reflection API is contained in the java.lang.reflect package

Contd…

- The first step to using reflection is to obtain a Class object that represents the class whose annotations you want to obtain

```
final Class getClass( )
```

- After you have obtained a Class object, we can use its methods to obtain information about the various items declared by the class, including its annotations

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

- We can obtain a specific annotation associated with that object by calling getAnnotation( )

```
Annotation getAnnotation(Class annoType)
```

```
import java.lang.annotation.*;
import java.lang.reflect.*;
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
String str();
int val();
}
class Meta {.
@MyAnno(str = "Annotation Example", val = 100)
public static void myMeth() {
Meta ob = new Meta();
try {
Class c = ob.getClass();
Method m = c.getMethod("myMeth");
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();
}
}
```

A Second Reflection Example

```
import java.lang.annotation.*;
import java.lang.reflect.*;
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
String str();
int val();
}
class Meta {
@MyAnno(str = "Two Parameters", val = 19)
public static void myMeth(String str, int i)
{
Meta ob = new Meta();
try {
Class c = ob.getClass();
Method m = c.getMethod("myMeth", String.class,
int.class);
MyAnno anno = m.getAnnotation(MyAnno.class);
System.out.println(anno.str() + " " + anno.val());
} catch (NoSuchMethodException exc) {
System.out.println("Method Not Found.");
}
}
public static void main(String args[]) {
myMeth("test", 10);
}
}
```

Obtaining All Annotations

- We can obtain all annotations that have RUNTIME retention that are associated with an item by calling getAnnotations( ) on that item.
- It has this general form:

```
Annotation[ ] getAnnotations( )
```

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

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

@Retention(RetentionPolicy.RUNTIME)
@interface What
{ String description(); }

@What(description = "An annotation test class")
@MyAnno(str = "Meta2", val = 99)

class Meta2
{
@What(description = "An annotation test method")
@MyAnno(str = "Testing", val = 100)
public static void myMeth()
{
Meta2 ob = new Meta2();
try
{
Annotation annos[] =
ob.getClass().getAnnotations();
System.out.println("All annotations for Meta2:");

for(Annotation a : annos)
System.out.println(a);
System.out.println();

Method m = ob.getClass( ).getMethod("myMeth");
annos = m.getAnnotations();
System.out.println("All annotations for myMeth:");

for(Annotation a : annos)
System.out.println(a);
}

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

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

output

All annotations for Meta2:

@What(description=An annotation test class)

@MyAnno(str=Meta2, val=99)

All annotations for myMeth:

@What(description=An annotation test method)

@MyAnno(str=Testing, val=100)

Using Default Values

- We can give annotation members default values that will be used if no value is specified when the annotation is applied.
- general form is:

```
type member( ) default value;
```

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno
{
String str() default "Testing";
 int val() default 9000;
 }
class Meta3
{
@MyAnno()
 public static void myMeth()
{
Meta3 ob = new Meta3();
try
{
 Class c = ob.getClass();
 Method m = c.getMethod("myMeth");
MyAnno anno = m.getAnnotation(MyAnno.class);
System.out.println(anno.str() + " " + anno.val());
}
catch (NoSuchMethodException exc) {
System.out.println("Method Not Found."); } }

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

output is

Testing 9000

Marker Annotations

- Only purpose is to mark a declaration.
- Contain no members and do not consist any data.
- Its presence as an annotation is sufficient
- To determine if a marker annotation is present is to use the method named **isAnnotationPresent()**

```
import java.lang.annotation.*;
import java.lang.reflect.*;
@Retention(RetentionPolicy.RUNTIME)
@interface MyMarker
 { }

class Marker {
@MyMarker
public static void myMethod() {
Marker obj = new Marker();
try {
Method m =
obj.getClass().getMethod("myMethod");
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[]) {
myMethod(); } }
```

Output:

MyMarker is present.

Single value Annotations

- contains only one member

```
@Retention(RetentionPolicy.RUNTIME)
@interface MySingle
{
        int value();
 }

class Single {
@MySingle(100)
public static void myMethod()
{
Single obj = new Single();
 try  {
Method m =
obj.getClass().getMethod()("myMethod");
 MySingle anno =
m.getAnnotation(MySingle.class);
System.out.println(anno.value());
 } catch(NoSuchMethodException exc) {
System.out.println("Method not found..!!"); } }
 public static void main(String args[])
{ myMethod(); } }
```

Output:

@MySingle(100)

Java Built In Annotations

- Java defines many built-in annotations. Most are specialized, but seven are general purpose.
    - @Retention
    - @Documented
    - @Target
    - @Inherited
    - @Override
    - @Deprecated
    - @SuppressWarnings

Manoj Dept. of CS & JKMSFC

@Override

- @Override annotation assures that the subclass method is overriding the parent class method.
- If it is not so, compile time error occurs

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

class Dog extends Animal{
@Override
void eatsomething()
{    System.out.println("eating foods ");  }
}
class TestAnnotation1{
public static void main(String args[]){
Animal a=new Dog();
a.eatSomething();
}}
```

Output:
Compile Time Error

@Deprecated(disapproval)

- @Deprecated annotation marks that this method is deprecated so compiler prints warning.
- Informs user that it may be removed in the future versions.
- So, it is better not to use such methods.

```
class A{
void m(){  System.out.println("hello  m");  }
  @Deprecated
void n(){   System.out.println("hello  n");  }
}
class TestAnnotation3{
public static void main(String args[]){
 A a=new A();
a.n();
}}
```

At compile time:

Compile by: javac TestAnnotation3.java
5/TestAnnotation3.java uses or overrides a deprecated API.

@Target

- **@Target** tag is used to specify at which type, the annotation is used.
- **@Target** takes only one argument, which is an array of constants of the **ElementType** enumeration.
- This argument determines the types of declarations to which 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

Example to specify annoation for a class

```
@Target(ElementType.TYPE)
@interface MyAnnotation{
int value1();
String value2();
}
```

@SuppressWarnings

- The **@SuppressWarnings** specifies that one or more warnings that might be issued by the compiler are to be suppressed
- Just to tell compiler to ignore specific warnings they produce,
- The warnings to suppress are specified by name in string form

Example : @SuppressWarnings

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

public class SuppressWarningTest
{
@SuppressWarnings("checked")
public static void main(String args[])
{
DeprecatedTest d1 = new DeprecatedTest();
d1.Display();
} }
```

Output-
Deprecatedtest display()

@Documented

- The @Documented annotation is a marker interface
- Indicates that new annotation should be included into java documents

Example: @Documented

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

@Documented
public @interface MyCustomAnnotation
{
   //Some other code
}
```

@Inherited

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

Example: @ Inherited

```
java.lang.annotation.Inherited
@Inherited
 public  @interface  MyAnnotation  { }

@MyAnnotation
public class MySuperClass
 { ... }

public class MySubClass  extends MySuperClass
 { ... }
```