Module -1

Enumerations, Autoboxing, and Annotations(Metadata)

Enumerations

- Enumerations included in JDK 5. An enumeration is a list of named constants. It is similar to final variables.
- **Enum in java** is a data type that contains fixed set of constants.
- An enumeration defines a class type in Java. By making enumerations into classes, so it can have constructors, methods, and instance variables.
- An enumeration is created using the enum keyword.
  Ex:

  enum Apple { Jonathan, GoldenDel, RedDel, Winesap, Cortland }

  The identifiers Jonathan, GoldenDel, and so on, are called enumeration constants.
  Each is implicitly declared as a public, static final member of Apple.

  Enumeration variable can be created like other primitive variable. It does not use the new for creating object.
  Ex:Apple ap;

  Ap is of type Apple, the only values that it can be assigned (or can contain) are those defined by the enumeration. For example, this assigns:

  ap = Apple.RedDel;

Example Code-1

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

class EnumDemo
{
public static void main(String args[])
{
Apple ap;
ap = Apple.RedDel;
 System.out.println("Value of ap: " + ap);// Value of ap: RedDel
 ap = Apple.GoldenDel;
 if(ap == Apple.GoldenDel)
 System.out.println("ap contains GoldenDel.\n"); // ap contains GoldenDel.
 switch(ap)
{
 case Jonathan:
 System.out.println("Jonathan is red.");
 break;
 case GoldenDel:
 System.out.println("Golden Delicious is yellow.");// 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 values( ) and valueOf( ) Methods 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.

Example Code-2:

```
enum Season { WINTER, SPRING, SUMMER, FALL } ;

class EnumExample1{

public static void main(String[] args) {

for (Season s : Season.values())

        System.out.println(s);

        Season s = Season.valueOf("WINTER");

        System.out.println("S contains " + s);

}       }
```

Example Code-3

```
class EnumExample5{

enum Day{ SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDA
Y, SATURDAY}

public static void main(String args[]){

Day day=Day.MONDAY;

 switch(day){

case SUNDAY:

 System.out.println("sunday");

 break;

case MONDAY:

 System.out.println("monday");

 break;

default:

System.out.println("other day");

}

}}
```


Class Type Enumeration

```
enum Apple {

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

private int price;

 Apple(int p) { price = p; }

int getPrice() { return price; }

}
```

```
class EnumDemo3 {

public static void main(String args[])

{ Apple ap;

System.out.println("Winesap costs " + Apple.Winesap.getPrice() + " cents.\n");

System.out.println("All apple prices:");

for(Apple a : Apple.values())

 System.out.println(a + " costs " + a.getPrice() + " cents.");

}

}
```

The Class type enumeration contains three things

The first is the instance variable price, which is used to hold the price of each variety of apple.

The second is the Apple constructor, which is passed the price of an apple.

The third is the method getPrice( ), which returns the value of price.

When the variable ap is declared in main( ), the constructor for Apple is called once for each constant that is specified.

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);

These values are passed to the parameter of Apple(),which then assigns this value to price. The constructor is called once for each constant.

Because each enumeration constant has its own copy of price, you can obtain the price of a specified type of apple by calling getPrice().

For example, in main() the price of a Winesap is obtained by the following call:Apple.Winesap.getPrice()

Enum Super class

All enumerations automatically inherit one: java.lang.Enum.

Enum class defines several methods that are available for use by all enumerations.

ordinal( )

To 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, shown here:

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.

compareTo( )

To compare the ordinal value of two constants of the same enumeration by using the compareTo( ) method. It has this general form:

final int compareTo(enum-type e)

equals()

equals method is overridden method from Object class, it is used to compare the enumeration constant. Which returns true if both constants are same.

Program to demonstrate the use of ordinal(), compareTo(), equals()

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

class EnumDemo4

{ public static void main(String args[])

{ Apple ap, ap2, ap3;

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();

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);

}

}
```

Wrappers Classes

Java uses primitive types such as int or double, to hold the basic data types supported by the language.

The primitive types are not part of the object hierarchy, and they do not inherit Object.

Despite the performance benefit offered by the primitive types, there are times when you will need an object representation.

Many of the standard data structures implemented by Java operate on objects, which means that you can't use these data structures to store primitive types.

To handle the above situation, Java provides type wrappers, which are classes that encapsulate a primitive type within an object.

The type wrappers are Double, Float, Long, Integer, Short, Byte, Character, and Boolean. These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy.

Character:

Character is a wrapper around a char. The constructor for Character is

Character(char ch)

Here, ch specifies the character that will be wrapped by the Character object being created.

To obtain the char value contained in a Character object, call charValue( ), shown here:

char charValue( )

Boolean:

Boolean is a wrapper around boolean values. It defines these constructors:

Boolean(boolean boolValue)

Boolean(String boolString)

In the first version, boolValue must be either true or false.

In the second version, if boolString contains the string "true" (in uppercase or lowercase), then the new Boolean object will be true. Otherwise, it will be false.

To obtain a boolean value from a Boolean object, use booleanValue( ), shown here:

boolean booleanValue( )

It returns the boolean equivalent of the invoking object.

Integer Wrapper class example code:

Integer(int num)

Integer(String str)

```
class Wrap
{ public static void main(String args[])
 {
   Integer iOb = new Integer(100);
```

```
  int i = iOb.intValue();
  System.out.println(i + " " + iOb);
 }
}
```

This program wraps the integer value100 inside an Integer object called iOb.

The program then obtains this value by calling intValue() and stores the result in i.

The process of encapsulating a value within an object is called boxing.

Thus, in the program, this line boxes the value 100 into an Integer:

Integer iOb = new Integer(100);

The process of extracting a value from a type wrapper is called unboxing.

The program unboxes the value in iOb with this statement:

int i = iOb.intValue();

AutoBoxing

Auto boxing 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.

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

Auto-unboxing

Auto- unboxing is the process by which the value of a boxed object is automatically extracted from a type wrapper when it is assigned to primitive type value is needed.

There is no need to call a method such as intValue( ).

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

Example Program:

```
class AutoBoxUnBox
{
 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
}
}
```

Explain auto boxing and auto unboxing during method call

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

In the program, notice that m( ) specifies an Integer parameter and returns an int result.

Inside main( ), m( ) is passed the value 100.

Because m( ) is expecting an Integer, this value is automatically boxed.

Then, m( ) returns the int equivalent of its argument. This causes v to be auto-unboxed.

Next, this int value is assigned to iOb in main( ), which causes the int return value to be autoboxed.

Explain auto boxing and unboxing during expression evaluation

Autoboxing/Unboxing Occurs in Expressions autoboxing and unboxing take place whenever a conversion into an object or from an object is required.

This applies to expressions. Within an expression, a numeric object is automatically unboxed.

The outcome of the expression is reboxed, if necessary. For example, consider the following program:

```
class AutoBox3

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

Integer iOb, iOb2; int i;

iOb = 100;

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

 ++iOb; // auto unbox and rebox

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

iOb2 = iOb + (iOb / 3);

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

i = iOb + (iOb / 3); // auto unbox and rebox

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

}

}
```



```
++iOb;
```

This causes the value in iOb to be incremented.

It works like this: iOb is unboxed, the value is incremented, and the result is reboxed.

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:

```
Integer iOb = 100; Double dOb = 98.6;

dOb = dOb + iOb; // type promoted to double
```

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

Integer iOb = 2;

switch(iOb) {

case 1: System.out.println("one");

        break;

case 2: System.out.println("two");

       break;

default:

    System.out.println("error");

}
```

When the switch expression is evaluated, iOb is unboxed and its int value is obtained.

As the examples in the program show, because of autoboxing/unboxing, using numeric objects in an expression is both intuitive and easy.

Autoboxing/unboxing a Boolean and Character.

```
class AutoBox5 { public static void main(String args[]) {

    Boolean b = true; // auto boxing boolean

   if(b)

   System.out.println("b is true");// auto unboxed when used in conditional expression

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

   char ch2 = ch; // unbox a char

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

   }}
```

The output is shown here:

b is true ch2 is x

Annotations

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.

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:

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

@ 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, you 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.

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

public static void myMeth() { // ...

Notice that no parentheses follow str in this assignment.

What is retention policy ? Explain the use of retention tag.

- 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.

A retention policy is specified for an annotation by using one of Java's built-in annotations: @Retention.

- @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(); }


```
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();
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);
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 MyAnnoannotation associated with myMeth()in the Metaclass.

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.

However, to obtain a method that has parameters, you must specify class objects representing the types of those parameters as arguments to getMethod( ). For example, here is a slightly different version of the preceding program:

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

import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)

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

class Meta {

// myMeth now has two arguments.

@MyAnno(str = "Two Parameters", val = 19)

 public static void myMeth(String str, int i)

 { Meta ob = new Meta();

try { Class c = ob.getClass();

// Here, the parameter types are specified.

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); }

}
```

The output from this version is shown here:

```
Two Parameters 19
```

myMeth( ) takes a String and an int parameter.

To obtain information about this method, getMethod( ) must be called as shown here:

Method m = c.getMethod("myMeth", String.class, int.class);

Here, the Class objects representing String and int are passed as additional arguments.

Obtaining All Annotations

You 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( )

It returns an array of the annotations.

getAnnotations( ) can be called on objects of type Class, Method, Constructor, and Field. Here is another reflection example that shows how to obtain all annotations associated with a class and with a method.

It declares two annotations.

It then uses those annotations to annotate a class and a method.

Example code:

```
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();
```

```
// Display all annotations for Meta2.
System.out.println("All annotations for Meta2:");
 for(Annotation a : annos) System.out.println(a);
System.out.println();
// Display all annotations for myMeth.
 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(); }
}
```

The output is shown here:

```
 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)
```

The program uses getAnnotations( ) to obtain an array of all annotations associated with the Meta2 class and with the myMeth( ) method. As explained, getAnnotations( ) returns an array of Annotation objects.

Recall that Annotation is a super-interface of all annotation interfaces and that it overrides toString( ) in Object.

Thus, when a reference to an Annotation is output, its toString( ) method is called to generate a string that describes the annotation, as the preceding output shows.

The AnnotatedElement Interface

The methods declared in AnnotatedElement Interface

1. getAnnotation( ) --- It can be invoked with method, class. It return the used annotation.
2. getAnnotations( ) --- It can be invoked with method, class. It return the used annotations.
3. getDeclaredAnnotations( ) -- It returns all non-inherited annotations present in the invoking object.
4. isAnnotationPresent( ), which has this general form:
   It returns true if the annotation specified by annoType is associated with the invoking object. It returns false otherwise.

Default Values in annotation

You can give annotation members default values that will be used if no value is specified when the annotation is applied.

A default value is specified by adding a default clause to a member's declaration. It has this general form:

type member( ) default value;

// An annotation type declaration that includes defaults.

```
@interface MyAnno { String str() default "Testing"; int val() default 9000; }

@MyAnno() // both str and val default

@MyAnno(str = "some string") // val defaults

@MyAnno(val = 100) // str defaults

@MyAnno(str = "Testing", val = 100) // no defaults
```

Example:

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

import java.lang.reflect.*;

@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:

Testing 9000

Marker Annotations

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.

Here is an example that uses a marker annotation.

Because a marker interface contains no members, simply determining whether it is present or absent is sufficient.

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

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyMarker { }
class Marker {
@MyMarker
public static void myMeth() { Marker ob = new Marker();
try { Method m = ob.getClass().getMethod("myMeth");
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(); }
}
```

Output

```
MyMarker is present.
public static void main(String args[]) { myMeth(); }
}
```

Built in Annotations

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.

Built-In Java Annotations used in java code

- @Override
- @SuppressWarnings
- @Deprecated

Built-In Java Annotations used in other annotations

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

Example:
```
class Animal{
    void eatSomething()
    {System.out.println("eating something");}
    }
    class Dog extends Animal{
    @Override
    void eatsomething()
    {
    System.out.println("eating foods");
    }//Compile time error  }
```

@SuppressWarnings

annotation: is used to suppress warnings issued by the compiler.

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 TestAnnotation2{

@SuppressWarnings("unchecked")

public static void main(String args[]){

ArrayList list=new ArrayList();

list.add("sonoo");

}}
```

@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.

```
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();

    }}
```

Error message: `Test.java uses or overrides a deprecated API.`

@Inherited

is a marker annotation that can be used only on another annotation declaration. Furthermore, it affects only annotations that will be used on class declarations. @Inherited causes the annotation for a superclass to be inherited by a subclass.

```
@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.

@Documented

@Documented annotation indicates that elements using this annotation should be documented by JavaDoc.

```
@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

@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.

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

@Target({ElementType.METHOD})
public @interface MyCustomAnnotation {

}
public class MyClass {
  @MyCustomAnnotation
  public void myMethod()
  {
     //Doing something
  }
}
```

If you do not define any Target type that means annotation can be applied to any element.

@Retention is mention in earlier topic.

BRIDGE TO THE FUTURE
CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)