Enumerations

In its simplest form, an enumeration is a list of named constants. In their simplest form, Java enumerations appear similar to enumerations in other languages. In Java, an enumeration defines a class type. By making enumerations into classes, the capabilities of the enumeration are greatly expanded. For example, in Java, an enumeration can have constructors, methods, and instance variables.

An enumeration is created using the **enum** keyword.

```
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. Furthermore, their type is the type of the enumeration in which they are declared, which is Apple in this case. Thus, in the language of Java, these constants are called self-typed, in which "self" refers to the enclosing enumeration.

Once you have defined an enumeration, you can create a variable of that type. We do not instantiate an enum using new.

```
        Apple ap;
```

- Two enumeration constants can be compared for equality by using the = = relational operator.
- An enumeration value can also be used to control a switch statement.

```
        switch (ap) {
                case Jonathan:    //...
                case Winesap:
   //...
```

```
enum Apple
{
	Jonathan, GoldenDel, RedDel, Winesap
}
class enumdemo
{
	public static void main(String args[])
	{
		Apple ap;	ap = Apple.RedDel;
		System.out.println("Value of ap: " + ap);
		ap = Apple.Winesap;
		// Compare two enum values.
		if (ap == Apple.Winesap)
			System.out.println("ap contains Winesap.\n");
		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;
		}
	}
}
```

```
C:\Users\Nimesh\Desktop\enum>java enumdemo
Value of ap: RedDel

ap contains Winesap.

Winesap is red.
```

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.

```
enum country
{
	Nepal, pakistan, USA, Japan, Finland;
}
class enumdemo2
{
	public static void main(String args[])
	{
		country c;
		System.out.println("Here are all country constants:");
		for(country a : country.values())
			System.out.println(a);
		c = country.valueOf("USA");
		System.out.println("selected country is: " + c);
	}
}
```

```
C:\Users\Nimesh\Desktop\enum>java enumdemo2
Here are all country constants:
Nepal
pakistan
USA
Japan
Finland
selected country is: USA
```

Java Enumerations Are Class Types:

```
// Use an enum constructor, instance variable, and method.
enum book
{
        Java(1000), C(900), English(670), Database(15);
        private int price;
        book(int p)
        {
                price = p;
        }
        int getPrice()
        {
                return price;
        }
}
```

```
class enumdemo3
{
	public static void main(String args[])
	{
		book b;
		System.out.println("Database costs :Rs. "+book.Database.getPrice());
		// Display all books and prices.
		System.out.println("All book prices:");
		for(book a : book.values())
			System.out.println(a + " costs :Rs. "+ a.getPrice());
	}
}
```

```
Database costs :Rs. 15
All book prices:
Java costs :Rs. 1000
C costs :Rs. 900
English costs :Rs. 670
Database costs :Rs. 15
```

Enumerations Inherit Enum:

Although you can't inherit a superclass when declaring an enum, all enumerations automatically inherit one: java.lang.Enum. This class defines several methods that are available for use by all enumerations.

Ordinal value is the value that indicates an enumeration constant's position in the list of constants. It is retrieved by calling the ordinal( ) method.

```
// Demonstrate ordinal(), compareTo(), and equals().
enum car
{
	Hundai, Ferrari, Mercedes
}
class enumdemo4
{
	public static void main(String args[])
	{
		car ap, ap2, ap3;
		System.out.println("Here are all car and their ordinal values: ");
		for(car a : car.values())
			System.out.println(a + " " + a.ordinal());
		ap =  car.Mercedes;
		ap2 = car.Ferrari;
		ap3 = car.Mercedes;
		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);
    }
}
```

```
Here are all car and their ordinal values:
Hundai 0
Ferrari 1
Mercedes 2

Ferrari comes before Mercedes
Mercedes equals Mercedes

Mercedes equals Mercedes
```

Type Wrappers:

primitive types: int, double

- you can't pass a primitive type by reference to a method
- you can't use many of the standard data structures implemented by Java operate on objects to store primitive types

To handle such situations, Java provides type wrappers.

Type wrappers are classes that encapsulate a primitive type within an object. They relate directly to Java's autoboxing feature.

The type wrappers: Double, Float, Long, Integer, Short, Byte, Character and Boolean.

Character:

Character is a wrapper around a char. The constructor for Character is: Character(char ch)

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

    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( )

boolean booleanValue( )

The Numeric Type Wrappers:

Byte:	byte byteValue( )	Short :	short shortValue( )
Integer:	int intValue( )	Long:	long longValue( )
Float:	float floatValue( )	Double:	double doubleValue( )

```
// 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
        }
 }
```

```
C:\Users\Nimesh\Desktop\enum>java Wrap
100 100
```

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**. For example, the program unboxes the value in iOb with this statement:

int i = iOb.intValue();

Autoboxing & Auto-unboxing:

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.

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

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

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

Autoboxing and Auto-unboxing greatly streamlines the coding of several algorithms, removing the tedium of manually boxing and unboxing values. It also helps prevent errors.

Autoboxing and Methods:

```
// Autoboxing/unboxing takes place with method parameters and return values.

class autobox1

{       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);

        }

}
```

Autoboxing/unboxing occurs in expressions:

```
class autoBox2
{
	public static void main(String args[])
	{
		Integer iOb, iOb2;	int i;
		iOb = 100;
		System.out.println("Original value of iOb: " + iOb);
		// The following automatically unboxes iOb,performs the increment, and then
		// reboxes the result back into iOb.
		++iOb;
		System.out.println("After ++iOb: " + iOb);
		// Here, iOb is unboxed, the expression is evaluated, and the result is reboxed
		// and assigned to iOb2.
		iOb2 = iOb + (iOb / 3);
		System.out.println("iOb2 after expression: " + iOb2);
		// The same expression is evaluated, but the result is not reboxed.
		i = iOb + (iOb / 3);
		System.out.println("i after expression: " + i);
	}
}
```

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

Autoboxing/Unboxing Boolean and Character Values:

```
class AutoBox4
{
	public static void main(String args[])
	{
		// Autobox/unbox a boolean.
		Boolean b = true;
		// Below, b is auto-unboxed when used in a conditional expression, such as an if.
		if(b)
			System.out.println(b);
			// Autobox/unbox a char.
			Character ch = 'x';			// box a char
			char ch2 = ch;				// unbox a char
			System.out.println("ch2 is " + ch2);
	 }
}
```

```
true
ch2 is x
```

Autoboxing/Unboxing Helps Prevent Errors:

```
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 !
	}
}
```

```
C:\Users\Nimesh\Desktop\enum>java UnboxingError
-24
```

Annotations(Metadata):

Java Annotation allow us to add metadata information into our source code, although they are not a part of the program itself.

Annotations were added to the java from JDK 5.

Annotation has no direct effect on the operation of the code they annotate (i.e. it does not affect the execution of the program).

Thus, an annotation leaves the semantics of a program unchanged.

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:

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

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

Specifying a Retention Policy:

- 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.
  1. SOURCE:
     retained only in the source file and is discarded during compilation
  2. CLASS:
     stored in the .class file during compilation. However, it is not available through the JVM during run time.
  3. RUNTIME:
     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)

- If no retention policy is specified for an annotation, then the default policy of CLASS is used.

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

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.

The reflection API is contained in the java.lang.reflect package.

getClass( ) : to obtain a Class object

getMethod(): to obtain a Method

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

```
import java.lang.annotation.*;
import java.lang.reflect.*;
@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 of 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);
                System.out.println(anno.str() + " " + anno.val());
            }
            catch (NoSuchMethodException exc)
            {
                System.out.println("Method Not Found.");
            }
        }
        public static void main(String args[])
        {
            myMeth();
        }
}
```

```
C:\Users\Nimesh\Desktop\enum>java Meta
Annotation Example 100
```