Module - 1

Enumerations, Autoboxing, Annotations.

* Enumerations:

An enumeration is a list of named constants.
An enumeration is created using the enum keyword.

Eg: 1)

```
enum Apple
{
Jonathan, GoldenDel, RedDel, Winesap, Cortland.
}
Apple ap;
ap= Apple.RedDel;
if (ap== Apple.GoldenDel)
switch(ap)
{
case Jonathan:
case Winesap:
SOP (Apple.Winesap);
enum Apple
{
Jonathan, GoldenDel; RedDel, Winesap, Cortland
}
class EnumDemo
{
public static void main(string args[])
{
Apple ap;
ap= Apple.RedDel;
SOP ("value of ap:" +ap);
SOP ();
ap = Apple.GoldDel;
if (ap== Apple.GoldenDel)
SOP ("ap contains GoldDel");
```

Scanned by CamScanner

```
switch(ap)
{
 case Jonathan:
 SOP("Jonathan is red.");
 break;
 case GoldenDel:
 SOP("GoldenDelicious is yellow.");
 break;
 }
 }
}
```

O/p: Value of ap : RedDel
ap contains GoldenDel.
Golden Delicious is yellow

* The values() and valueOf() methods.

All enumerations automatically contain two predefined methods: values() & valueOf()

syntax:

- values() : public static enum-type[] values()
- valuesOf() : public static enum-type valueOf(string str)

→ The values() method returns an array that contains a list of the enumeration constants.

→ The valuesOf() method returns the enumeration constant whose value corresponds to the string passed in str.

Scanned by CamScanner

Eg: 2)

```
enum Apple
{
Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
class EnumDemo2
{
public static void main(String args[])
{
 Apple ap;
 SOP("Here are all Apple constants:");
 Apple allapples[] = Apple.values();
 for(Apple a : all apples)
 SOP(a);
 SOP();
 ap = Apple.valueOf("winesap");
 SOP("ap contains "+ ap);
}
}
```

O/P: Here are all apple constants:

```
Jonathan
GoldenDel
RedDel
Winesap
Cortland
ap contains winesap
```

Eg: 3) use an enum constructor, instance variable, and method

```
enum Apple
{
Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15),
                cortland(8);
Apple(int p)
{
 price = p;
}
int getPrice()
{
 return price;
}
}
```

Scanned by CamScanner

```
class EnumDemo3
{
public static void main(String args[])
{
Apple ap;
SOP ("Winesap costs " + Apple.Winesap.getPrice() + "cents. \n");
SOP("All apple prices :");
for (Apple a : Apple.values())
SOP (a + " costs " + a.getPrice() + " cents.");
}
}
```

O/P: Winesap costs 15 cents.
All apple prices:
Jonathan costs 10 cents
GoldenDel. costs 9 cents
RedDel costs 12 cents
Winesap costs 15 cents.
Cortland costs 8 cents.

*Enumerations Inherit Enum

All enumerations automatically inherit one: java.lang.Enum. ~~This~~ You can obtain a value that indicates an enumeration constants position in the list of constants. This is called its ordinal value.
And it is retrieved by calling the ordinal() method,

final int ordinal()

You can compare the ordinal value of two constants of the same enumerations by using the compareTo() method.

final int compareTo(enum-type)

Here, enum-type is the type of the enumerations and e is the constant being compared to the invoking constant.

If the invoking constant has an ordinal value ~~are~~ ~~the~~ less than e's, then compareTo() returns a negative value.

Scanned by CamScanner

If the two ordinal values are the same, then zero is returned. (5)

If the invoking constant has an ordinal value greater than e's, then a positive value is returned.

You can compare for equality an enumeration constant with any other object by using equals(), which overrides the equals() method defined by Object.

Eg: 4)

Demonstrate ordinal(), compareTo(), equals()

```
enum Apple
{
 Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
class EnumDemo4
{
public static void main(String args[])
 {
  Apple ap, ap2, ap3;
   // ordinal values using ordinal().
SOP("Here are all apples constants"+"and their
            ordinal values:");
for (Apple a : Apple.values())
SOP (a + "  " + a.ordinal());
ap = Apple.RedDel;
ap2 = Apple.GoldenDel;
ap3 = Apple.RedDel;
SOP();
```

P.T.O

Scanned by CamScanner

```
// Demonstrate compareTo() and equals()
  if (ap.compareTo(ap2) < 0)
  SOP(ap + " comes before " + ap2);
  if (ap.compareTo(ap2) > 0)
  SOP(ap2 + " comes before " + ap);
  if (ap.compareTo(ap3) == 0)
  SOP(ap + " equals " + ap3);
  SOP();
 if (ap.equals(ap2))
  SOP("Error!");
 if (ap.equals(ap3))
  SOP(ap + " equals " + ap3);
 if (ap == ap3)
  SOP(ap + " == " + ap3);
 }
}.
```

<u>Wrapper classes</u> (converting datatypes to object & classes & viceversa)

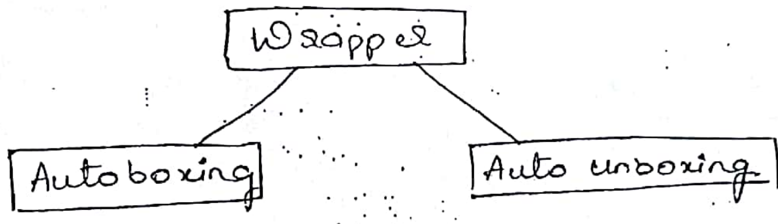
<u>datatypes</u>

- byte → Bytes
- short → Short
- char → Character
- int → Integer
- float → Float
- boolean → Boolean.

<u>Eg:5)</u>

```
class Wrap
{
 public static void main(String args[])
 {
  Integer iOb = new Integer(100);
  int i = iOb.intValue();
  SOP(i+" "+iOb);
 }
}
```

Scanned by CamScanner



Autoboxing:

Autoboxing is the process by which datatype is automatically encapsulated into its equivalent type wrapper. (converting datatype to wrapper object).
We need to call the value of method.

eg: 6): Datatype to object.

```
class Wrap
{
public static void main(String[] args)
int i=10;
Integer iOb=new Integer(i);
SOP(iOb);          // Autoboxing.
}
```

o/p: 10

Auto-Unboxing.

Auto-unboxing is the process by which the value of a boxed object is automatically extracted from a type wrapper when its value is needed.
(converting wrapper object to datatype).
No need to call a method such as intValue() (or doubleValue().

Eg: 7) Object to datatype.

```
class Wrap!
{
Public static void main(String[] args)
Integer iOb=new Integer(10);
int i=iOb;   // Auto-unboxing.
SOP(i)
}
```

o/p: 10.

Scanned by CamScanner

Annotations (Metadata).

* A new facility was added to Java that enables you to embed supplemental information into a source file.
This information is called an annotation.

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

For. eg: 8)

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

* Notice that @ that precedes the keyword interface (This tells the compiler that an annotation type is being declared).

* The two members str() and val()
(All annotations consist solely of method declarations).

* Annotation is a super-interface of all annotations.
It is declared within the java.lang.annotation package.
It overrides hashCode(), equals(), toString(), which are defined by Object.

Re

Scanned by CamScanner

Retention Policy.

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

Eg: 9)

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

Eg:10) Using Default values.

An Annotation type declaration that includes defaults.

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

This declaration gives a default value of "Testing" to str and 9000 to val.

This means that neither value needs to be specified when @MyAnno is used.

@MyAnno can be used in four following ways:

```
@MyAnno // both str and val default.
@MyAnno(str = "some string") // val defaults.
@MyAnno(val=100)             // str defaults.
@MyAnno(str="Testing", val =100) // no defaults.
```

Scanned by CamScanner

Eg. 11) The program demonstrates the use of default values in an annotation.

```
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);
 SOP (anno.str() + " " + anno.val());
 }
 catch (NoSuchMethodException exc)
 {
 SOP("Method Not Found.");
 }
}
public static void main(String args[])
 {
 myMeth();
 }
}
```

O/P: 9000

Scanned by CamScanner

Marker Annotations.

A marker annotation is a special kind of annotation that contains no member. Its sole purpose is to mark a declaration.

Eg: 12)

```
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))
 SOP("MyMarker is Present.");
 }
 catch(NoSuchMethodException exc)
 {
 SOP("Method Not Found.");
 }
}
public static void main(String args[])
{
 myMeth();
}
}
```

O/P:

MyMarker is present.

Single-Member Annotation

A single-member annotation contains only one member. It works like a normal annotation except that it allows a shorthand form of specifying the value of the member.

Eg:13)

```
import java.lang.annotation.*;
import java.lang.reflect.*;
@Retention(RetentionPolicy.RUNTIME)
@interface MySingle
{
int value();
}
class Single
{
@MySingle(100)
public static void myMeth()
{
single ob =new Single();
try
{
Method m = ob.getClass().getMethod("myMeth");
MySingle anno =m.getAnnotation(MySingle.class);
SOP(anno.value());
}
catch(NoSuchMethodException exc)
{
System.out.println("Method not found.");
}
}
public static void main(string args[])
{
myMeth();
}
}
```

O/p:

@MySingle(100).

Scanned by CamScanner

The Built-In Annotations

① @Retention.

@Retention is designed to be used only as an annotation to another annotation. It specifies the retention policy as described earlier in this chapter.

② @Documented.

The @Documented annotation is a marker interface that tells a tool that an annotation is to be documented. It is designed to be used only as an annotation to an annotation declaration.

③ @Target.

The @Target annotation specifies the types of declarations to which an annotation can be applied. It is designed to be used only as an annotation to another annotation. @Target takes one argument, which ~~cannot~~ must be a constant from the ElementType enumeration.

Scanned by CamScanner

4) @Inherited

@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.
∴ when a request for a specific annotation is made to the subclass, if that annotation is not present in the subclass, then its superclass is checked.
If that annotation is present in the superclass, and if it is annotated with @Inherited, then that annotation will be returned.

5) @Override

@Override 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.

6) @Deprecated

@Deprecated is a marker annotation. It indicates that a declaration is obsolete and has been replaced by a newer form.

7) @SuppressWarnings

@SuppressWarnings specifies that one or more warnings that might be issued by the compiler are to be suppressed.
The warnings to suppress are specified by name, in string form. This annotation can be applied to any type of declaration.

Scanned by CamScanner

// Eg.14) For inbuilt Annotations

```
@RetentionPolicy(Retention.RUNTIME)
@Target
@suppressWarning("unchecked")
@ interface Smartphone
{
String OS() default OS = "symphon";
int version() default version = 1
}
@smartphone(OS = "Android" version = 5)
class Nokia
{
String Model;
int size;
}
Nokia(String Model, int size)
{
this.Model = Model;
this.size = size;
}
class Demo
{
public static void main(String args[])
Nokia obj = new Nokia("FIVE", 5)
class C = obj.getclass()
annotation an = e.getannotation(smartphone())
smartphone s = (smartphone) an
SOP(an.OS());
SOP(an.version());
}
```

Scanned by CamScanner