Enumerations

- an *enumeration* is a data type that contains list of named constants.
- An enumeration is created using the **enum** keyword.
- enum may implement many interfaces but cannot extend any class because it internally extends Enum class.
- Enum constants are implicitly static and final and can not be changed once created.
- enum improves type safety
- enum can be traversed
- enum can be easily used in switch
- enum can have fields, constructors and methods
- Example:

```
public enum Direction { NORTH, SOUTH, EAST, WEST }
```

  - o

Enumerations are class types

- *An* enumeration defines a class type. An enumeration can have constructors, methods, and instance variables.
- Example:

```
//An enumeration of Week days.
enum Week {
      MONDAY, TUESDAY, .., SATURDAY;
}
```

- **MONDAY**, TUESDAY are called *enumeration constants*.
- Each is implicitly declared as a public, static final member of `Week`
- Creating variable of enumeration
  - o **Week day;**
- Assigning value
  - o **day = Week.SUNDAY;**
- Enumeration constants can be compared:
  - o **if (day == Week.Monday)**
- When enumeration constant is displayed it prints the name of constant
  - o **System.out.pri ntln(Week.MONDAY);** // *outputs MONDAY*

Restrictions on enum

- Even though enumerations define a class type,
  - o We cannot instantiate an **enum** using **new**.
- Using enumeration
  - o Declare and use an enumeration variable in the same way as of the primitive types.

The values( ) and valueOf( ) Methods

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

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

- **valueOf() method**
  - *The java.lang.Enum.valueOf()* method returns the enum constant for the specified *enumtype* passed as parameter.
  - The **name** must match exactly an identifier used to declare an enum constant in this type.
  - General forms are shown here:

```
public static enum-type valueOf (String name)
```

- **Example:** consider following enumeration

```
enum Week {
        MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FR I DAY, SATURDAY;
}
```

- *values() can be used to access all the constants of enum Week*

```
Week allDays[] = Week.values();
  for (Week eachDay : allDays )
    System.out.prinln(eachDay + ",");  // shows all  values
```

- valuesOf can be used to convert string "FRIDAY" to enum-type

```
Week today = Week.valueOf ("FRIDAY");
    System.out.println ("Basket contains: " + today );
```

Constructors and methods, instance variables and enumerations

- enum t ype can have a ***privat e const ruct ors*** that can be used t o initialize instance fields. Whenever you don ' t declare a constructor in enum, a default constructor is provided by compiler.
- *Constructor executes repetitively for each constant in enum.*

```
enum Basket {
     Apple(100), Banana(50), Grapes(30);
     int price;

         Basket ( int val ) {
               price = val;
     }
     int showPr ice() {
              return price;
     }
}

class EnumDemo {
public static void main(String args[])   {
        System.out.println("Output by values():");
              Basket allFruits[] = Basket.values();

            for ( Basket fruit  : allFruits )
              System.out.println(fruit  + ", ");

           System.out.println("Output by valueOf():");
          Basket item = Basket.valueOf ("Apple");
          System.out.println ("Basket  cont ains: " + item);
     }
}
```

Output:

```
Output by values(): Apple, Bannana, Grapes
Output by valueOf():   Basket contains Apple
```

- Additional points on enum
  - An *enum class* can have "public static void main(String[] args)" and run as standalone application.
  - Enum can be used in switch

Wrapper Classes

- Type wrappers are classes that encapsulate a primitive type within an object.
- Constructors of different type wrappers are

Wrapper class	Constructor	Methods
Character	**Character** (char *ch*)	char charValue()
Boolean	**Boolean** (boolean *boolValue*) **Boolean** (String *boolString*)	boolean booleanValue()
Byte	**Byte** (byte *byteVal*) **Byte** (String *byte*String)	byte byteValue()
Short	**Short** (short *shortVal*) **Short** (String *shortString*)	short shortValue()
Integer	**Integer** (int *intVal*) **Integer** (String *intString*)	int intValue()
Long	**Long** (long *longVal*) **Long** (String *longString*)	long longValue()
Float	**Float** (float *floatVal*) **Float** (String *floatString*)	float floatValue()
Double	**Double** (double *doubleVal*) **Double** (String *doubleString*)	double doubleValue()

- If string does not contain a valid numeric value, then a ***NumberFormatException*** is thrown.
- All of the type wrappers ***override toString( ),*** which returns the value contained within the wrapper in proper format.
- ***toString( )*** *is* called whenever wrapper object is concatenated with string.
- ***Demonstrate a type wrapper***

```
class Wrap {
   public static void main(String args[]) {
        Integer intOb = new Integer (100);  //called boxing
        int iVal = intOb.intValue();        //called Unboxing
        System.out.println( iVal + " " + intOb);
                  // displays 100 100
   }
}
```

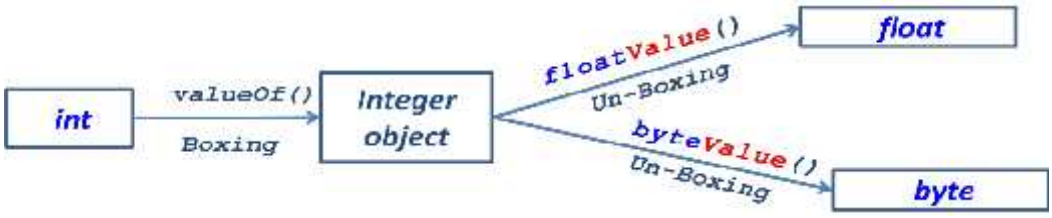
- This program wraps the integer value 100 inside an Integer object called iOb.
- ***intValue***( ) converts ***class-type*** into ***int*** type and returns the value in ***iVal***.
- The process of encapsulating a value within an object is called ***boxing***.
- *Thus, in the program, this line* ***boxes the value 100 into an Integer:***

Boxing and UnBoxing

- **Boxing**
  - The process of encapsulating a value within an object is called boxing.
    - **Integer i ntOb = new I nteger(100);**
- ***UnBoxing:***
  - The process of extracting a value from a type wrapper is called unboxing.
    - **int i Val = intOb.intValue();**
- Steps to convert one datatype to other type

*1. **Boxing**: Box it in the object of Wrapper class using valueOf() method*

*2. **UnBoxing**: Apply respective method and Unwrap to required type*



Example:shows Boxing and UnBoxing

```
class WrapperDemo {
    public static void main (string[] args) {
        Integer intObject = Integer.valueOf(42);     //Boxing

        float fVal = intObject.intValue();           //UnBoxing
        double dVal = intObject.doubleValue();       //UnBoxing
        byte bVal = intObject.byteValue();           //UnBoxing

        System.out.println ( iVal ) ;
        System.out.println ( dVal ) ;
        System.out.println ( bVal ) ;
    }
}
```

AutoBoxing

- **Autoboxing** is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes.
- ***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.
- ***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( ).***
- With autoboxing
  - it is no longer need to manually construct an object to wrap a primitive type.

```
Integer intOb = 100;        // autobox an int
int iVal = intOb;           // auto-unbox
```

Example:

```
class AutoBox {
    public static void main(String args[]) {
        Integer intOb = 100;     // Autobox an int
        int iVal = intOb;        // Auto-Unbox
        System.out.println( iVal + " " + intOb);
    }
}
```

AutoBoxing and Methods

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

```
class AutoBox {
// Take an Integer parameter and return an int value;
     static int myMethod(Integer intOb) {  //AutoBox to class
return intOb ;  // auto-Unbox to int
}
public static void main(String args[]) {
Integer iOb = myMethod(100);
System.out.println(iOb);
}
}
```

AutoBoxing/Un Boxing happens in expression

- Within an expression, a numeric object is automatically unboxed
- The outcome of the expression is reboxed, if necessary

```
Integer iOb, iOb2;
iOb = 4;
iOb2 = iOb + 2;
```

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

```
Integer intOb = 100; // Autoboxing int into object
Float floatOb = 98.6; // Autoboxing float into object
floatOb = floatOb + intOb; // Automatically Unboxed and Reboxed
System.out.println("Result after expression: " + floatOb);

Boolean boolOb = true;
if(boolOB)   // boolOB is auto-unboxed in if condition to boolean-type
    System.out.println("object  is true");
```

AutoBoxing/Un Boxing helps prevent errors

```
class UnboxingError {
 public static void main(String args[]) {
 Integer intOb = 1000; // autobox the value 1000
 int ival1 = intOb.byteValue(); // manually unbox as byte !!!Wrong
 int ival2 = intOb.intValue(); // manually unbox as int !!!
 int ival3 = intOb; // Auto-Unbox
 System.out.println(i); // does not display 1000 !
}
}
```

- Helps `overcome manual errors in statement`
- ***int ival1 = intOb.byteValue();*** *// manually Unbox as byte !!!**Wrong***
- Byte value returned by byteValue( ) method, causes the truncation of the value 1,000. This results in the garbage value of –24.
- `Auto-Unboxing prevents this type of error`

Annotations

- Beginning with JDK 5, Java enables you to embed supplemental information into a source file, called an annotation or Metadata.
- Annotation does not change the actions of a program.
- An annotation leaves the semantics of a program unchanged. However, this information can be used by various tools like an annotation might be processed by a source-code generator.
- Example:

```
// A simple annot ation t ype.
@interface MyAnno {
        String str();
        int val();
}
```

- The symbol **@** that precedes the keyword interface.
  - This tells the compiler that an annotation type is being declared.
  - the two members str( ) and val( ) act much like fields in annotations,
- An annotation cannot include an extends clause
- all annotation types automatically extend the Annotation interface.
- Thus, Annotation is a super-interface of all annotations
- Annotations is declared within the ***j ava. l ang.annotat i on*** 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.
- classes, methods, fields, parameters, and enum constants can have an annotation associated with it. ***Even an annotation can be annotated***.
- Example of **MyAnno** being applied to a method:
- When an annotation member is given a value, only its name is used. Thus, annotation members look like fields in this context.

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

- This annotation is linked with the method ***myMethod( )***
- 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.

Built-in Annotations

- Java defines many built-in annotations. Most are specialized, but seven are general purpose.
- Four are imported from ***java.lang.annotation****:* ***@Retention***, ***@Documented***, ***@Target***, *and* ***@Inherited***.
- ***Three—@Override, @Deprecated, and @SuppressWarnings***—are included in ***java.lang***.
- Each is described here.

Annotations	Description
@Retention	Specifies the retention policy that will be associated with the annotation. The retention policy determines how long an annotation is present during the compilation and deployment process.
@Documented	A marker annotation that tells a tool an annotation is to be documented. It is designed to be used only as an annotation declaration.
@Target	Specifies the types of the declarations to which an annotation can be applied. It is designed to be used only as an annotation to another annotation.
@Inherited	A marker annotation that causes the annotation for a superclass to be inherited by a subclass.
@Override	A method annotated with @Override must override a method from a superclass. If it does't, a compile-time error will result. It is used to ensure that a superclass method is actually overridden, and not simply overloaded. This is a marker annotation.
@Deprecated	A marker annotation that indicates that a feature is obsolete and has been replaced by a newer form.
@Safe Varargs	A marker annotation that indicates that no unsafe actions related to varargs parameter in a method or constructor occur.
@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.