ACHARYA INSTITUTE OF TECHNOLOGY
DR. SARVEPALLI RADHAKRISHNAN ROAD,
BENGALURU-560107

Department of Computer Science and Engineering

**VI SEMESTER**

# ADVANCED JAVA PROGRAMMING

[21CS642]

## Module – 1 : Enumerations, Autoboxing and Annotations

Prepared by:

Mrs. Nagamma Aravalli

Assistant Professor,
Dept, of Computer Science & Engineering,AIT

# Module – 1 : Enumerations, Autoboxing and Annotations

## Enumerations

1. Enumerations were added to the Java language in JDK5.
2. **Enumerations** (enum) in Java are a special type of class used to define a collection of constants. They provide a way to represent a group of named constants.
3. An Enumeration can have constructors, methods and instance variables.
4. It is created using the **enum** keyword. Each enumeration constant is *public, static* and *final* by default.
5. Even though enumeration defines a class type and has constructors, you do not instantiate an **enum** using **new**.
6. Enumeration variables are used and declared in much a same way as you do a primitive var

### How to Define and Use an Enumeration

1. An enumeration can be defined simply by creating a list of enum variables. Let us take an example for the **WeekDays** variable with different days in the Week.

```
enum WeekDays   //Enumeration defined
{
   SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
```

2. Identifiers *SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY* are called enumeration constants and are public, static and final by default.
3. Variables of enumeration can be defined directly without using a **new** key word.

```
WeekDays day;
```

4. Variables of Enumeration type can have only enumeration constants as value. We define an enum variable as enum_variable = enum_type.enum_constant;

```
day = WeekDays.SUNDAY;
```

5. Two enumeration constants can be compared for equality by using the = = relational operator.

**Example:**

```
 if(day == WeekDays.SUNDAY) { ..... }
```

**Example of Enumeration**

**Ex1:**

```
enum WeekDays   //Enumeration defined
{
   SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}

class EnumDemo1
{
     public static void main(String args[])
      {
        WeekDays wk; //wk is an enumeration variable of type WeekDays
        wk = WeekDays.SUNDAY; //only WeekDays enum type value can be assigned
        System.out.println("Today is "+wk);
       }
}
```

**Output :**

```
Today is SUNDAY
```

6. The enum can be defined within or outside the class.
   **Note**: Above **Ex1.** demonstrate the use of enum outside the class

**Example of enum within the class**

**Ex2.**

```
class EnumDemo2
{
  enum WeekDays   //Enumeration defined within the class
  {
   SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
  }

  public static void main(String args[])
   {
      WeekDays wk =  WeekDays.SUNDAY;
       System.out.println("Today is "+wk);
    }
}
```

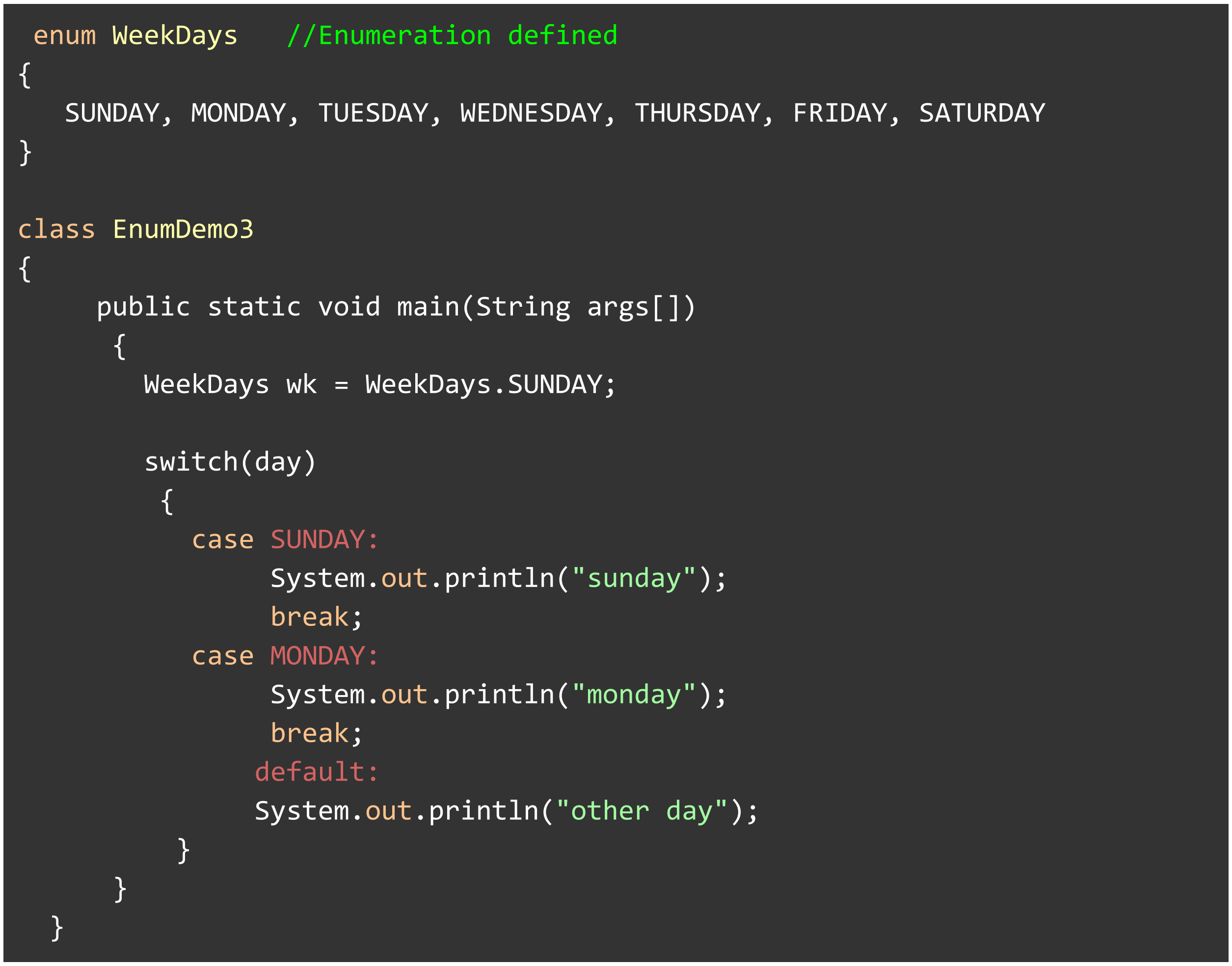
**Output :**

```
Today is SUNDAY
```

7. An enumeration value can also be used to control a switch statement

   **Example of Enumeration using switch statement**

   **Ex3.**



```
 enum WeekDays   //Enumeration defined
{
   SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}

class EnumDemo3
{
     public static void main(String args[])
      {
        WeekDays wk = WeekDays.SUNDAY;

        switch(day)
         {
           case SUNDAY:
                System.out.println("sunday");
                break;
           case MONDAY:
                System.out.println("monday");
                break;
               default:
               System.out.println("other day");
          }
      }
  }
```

**Output:** sunday

## values() and valueOf() methods

1. All the enumerations have predefined methods values() and valueOf().
2. **values():** This method returns an array of enum-type containing all the enumeration constants in it.

   **Generic form:** `public static enum-type[] values()`
3. **valueOf():** This method is used to return the enumeration constant whose value is equal to the string passed in as argument while calling this method.

   **Generic form:** `public static enum-type valueOf(String str)`

**Example of enumeration using  values() and valueOf() methods**
**Ex4.**

```
// use of enumeration built-in methods
// An enumeration of apple varieties
enum Apple {
    Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
class EnumDemo4
 {
    public static void main(String args[])
     {
      Apple ap;
      System.out.println("Here are all Apple constants:");
      // use values()
      Apple allapples[] = Apple.values();
      for(Apple a : allapples)
      System.out.println(a);
      System.out.println();
      // use valueOf()
      ap = Apple.valueOf("Winesap");
      System.out.println("ap contains " + ap);
    }
 }
```

**Output:**

```
Here are all Apple constants:
Jonathan
GoldenDel
RedDel
Winesap
Cortland

ap contains Winesap
```

4. Instead of creating an array we can directly obtain all values as shown in below code snippet From **Ex4**.

```
for(Apple a : Apple.values())
 System.out.println(a);
```

## Java Enumerations Are Class Types

1. Enumerations are defined using the **enum** keyword which resembles the syntax used to create a class using the keyword **class.**
2. Just like classes, enums can have instance variables, methods and constructors, even enums can implement interfaces.
3. Each enumeration constant is an object of its enumeration type.
4. The constructor is called when each enum constant is created.
5. Each enum constant has its own copy of any instance variable defined by the enumeration.

But there are two restrictions that apply to enumerations:

1. Enumeration cannot **inherit** another class
2. An **enum** cannot be a **superclass**,this means that an **enum** can't be **extended.**

**Example for default constructor of enum**
**Ex5.**

```
enum WeekDays   //Enumeration defined
{
   SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;
   //Overloaded default constructor
   WeekDays(){
     System.out.println("Default constructor called");
    }
}

class EnumDemo5
 {
    public static void main(String args[])
     {
        WeekDays day; //No constructor called
        day = WeekDays.MONDAY; //default constructor called for constant MONDAY
     }
 }
```

**Output:**

```
Default constructor called
Default constructor called
Default constructor called
Default constructor called
Default constructor called
Default constructor called
Default constructor called
```

- The default constructor is called for each enum constant (SUNDAY,MONDAY,.....,SATURDAY) because enum constants are implicitly instantiated when an enum type is loaded.
- Even though we are only explicitly assigning ‘`WeekDays.MONDAY`’ to ‘`day`’, the default constructor for all enum constants is still executed.
- So, when we run the program , we will see “`Default constructor called`” printed 7 times, once for each enum constant, regardless of how many of them we explicitly use in our code.

**Example for enum as class type**

### Ex6.

```
// Use an enum constructor, instance variable, and method.
enum Apple {
   Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);

   //instance variable
   private int price; // price of each apple

   // Constructor
    Apple(int p){
     price = p;
   }

   //instance method
    int getPrice(){
     return price;
    }
}

class EnumDemo6 {
   public static void main(String args[])
   {

   Apple ap;
   // Display price of Winesap.
   System.out.println("Winesap costs " +
   Apple.Winesap.getPrice() + " cents.\n");
   // Display all apples and prices.
   System.out.println("All apple prices:");
   for(Apple a : Apple.values())
     System.out.println(a + " costs " + a.getPrice() + " cents.");
   }
}
```

**Output:**

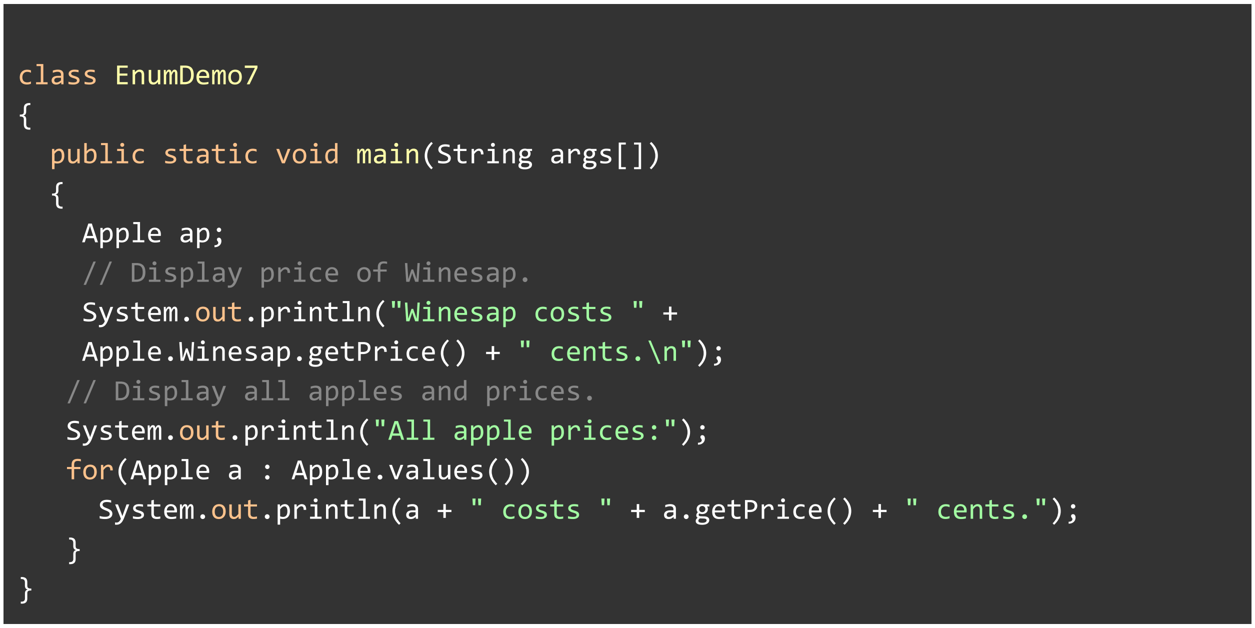
```
Winesap costs 15 cents.

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

- This example contains 3 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.
- As soon as the enum variable is declared and loaded , the parameterized constructor of each constant is called and it initializes the price with values specified with them in parenthesis.
- As we know , each enumeration constant has its own copy of **`price`**, we can obtain the price of a specified type of apple by calling **`getPrice()`.**

  `Apple.Winesap.getPrice()` returns price of Winesap apple i.e `15`
- In above Apple enum, overload the constructor and initialize the price to -1, to indicate the no price data is available:

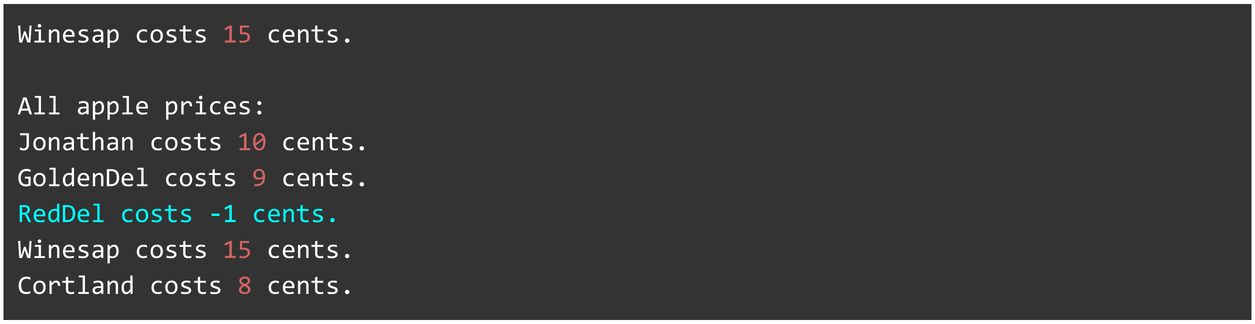
  **Ex7:**

```
// Use an enum constructor.
enum Apple {
 Jonathan(10),GoldenDel(9),RedDel,Winesap(15),Cortland(8);

  private int price; // price of each apple

  // Constructor
  Apple(int p){
     price = p;
  }
  //Overloaded constructor
  Apple(){
    price = -1;
  }
   int getPrice(){
     return price;
   }
}
```



```
class EnumDemo7
{
  public static void main(String args[])
  {
    Apple ap;
    // Display price of Winesap.
    System.out.println("Winesap costs " +
    Apple.Winesap.getPrice() + " cents.\n");
   // Display all apples and prices.
   System.out.println("All apple prices:");
   for(Apple a : Apple.values())
     System.out.println(a + " costs " + a.getPrice() + " cents.");
   }
}
```

**output:**



```
Winesap costs 15 cents.

All apple prices:
Jonathan costs 10 cents.
GoldenDel costs 9 cents.
RedDel costs -1 cents.
Winesap costs 15 cents.
Cortland costs 8 cents.
```

- In this example , **RedDel** is not given an argument, that means the default constructor is called, and **RedDel**'s price variable returns the value -1.

## Enumerations inherit Enum

1. All enumerations automatically inherit one class that is **java.lang.Enum**.
2. The **Enum** class defines several methods such as **ordinal( )**, **compareTo( )**, **equals( )** and so on, 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.

**Example using ordinal() method:**
**Ex8.**

```
enum Apple3 {
Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo8
{
 public static void main(String args[])
 {
        Apple3 ap1, ap2, ap3;
      // Obtain all ordinal values using ordinal().
        System.out.println("Here are all apple constants" +
        " and their ordinal values: ");
        for(Apple3 a : Apple3.values())
        System.out.println(a + " " + a.ordinal());
  }
}
```

**Output:**

```
Here are all apple constants and their ordinal values:
Jonathan 0
GoldenDel 1
RedDel 2
Winesap 3
Cortland 4
```

**<u>compareTo( ):</u>**

- The ordinal value of two constants of the same enumeration can be compared by using the **compareTo( )** method.
- Its general form is `final int compareTo(enum-type e)`

  Here,`enum-type` is the type of the enumeration, and `e` is the constant being compared to the invoking constant.
- Both the invoking constant and `e` must be of the same enumeration
- **compareTo()** method returns following 3 values
  a. positive value (+) : If the invoking constant's ordinal value greater than `e'`s
  b. negative value (-) : If the invoking constant's ordinal value is less than `e'`s
  c. Zero (0) : If the two ordinal values are the same.

## Example using compareTo() method:

## Ex9.

```
 enum Apple4 {
Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
public class EnumDemo9 {
      public static void main(String args[])
        {
               Apple4 ap1, ap2, ap3;
               ap1 = Apple4.RedDel;
               ap2 = Apple4.GoldenDel;
               ap3 = Apple4.RedDel;
               System.out.println();
             // Demonstrate compareTo()
             if(ap1.compareTo(ap2) < 0)
             System.out.println(ap1 + " comes before " + ap2);
             if(ap1.compareTo(ap2) > 0)
             System.out.println(ap2 + " comes before " + ap1);
             if(ap1.compareTo(ap3) == 0)
             System.out.println(ap1 + " equals " + ap3);
        }
}
```

## Output:

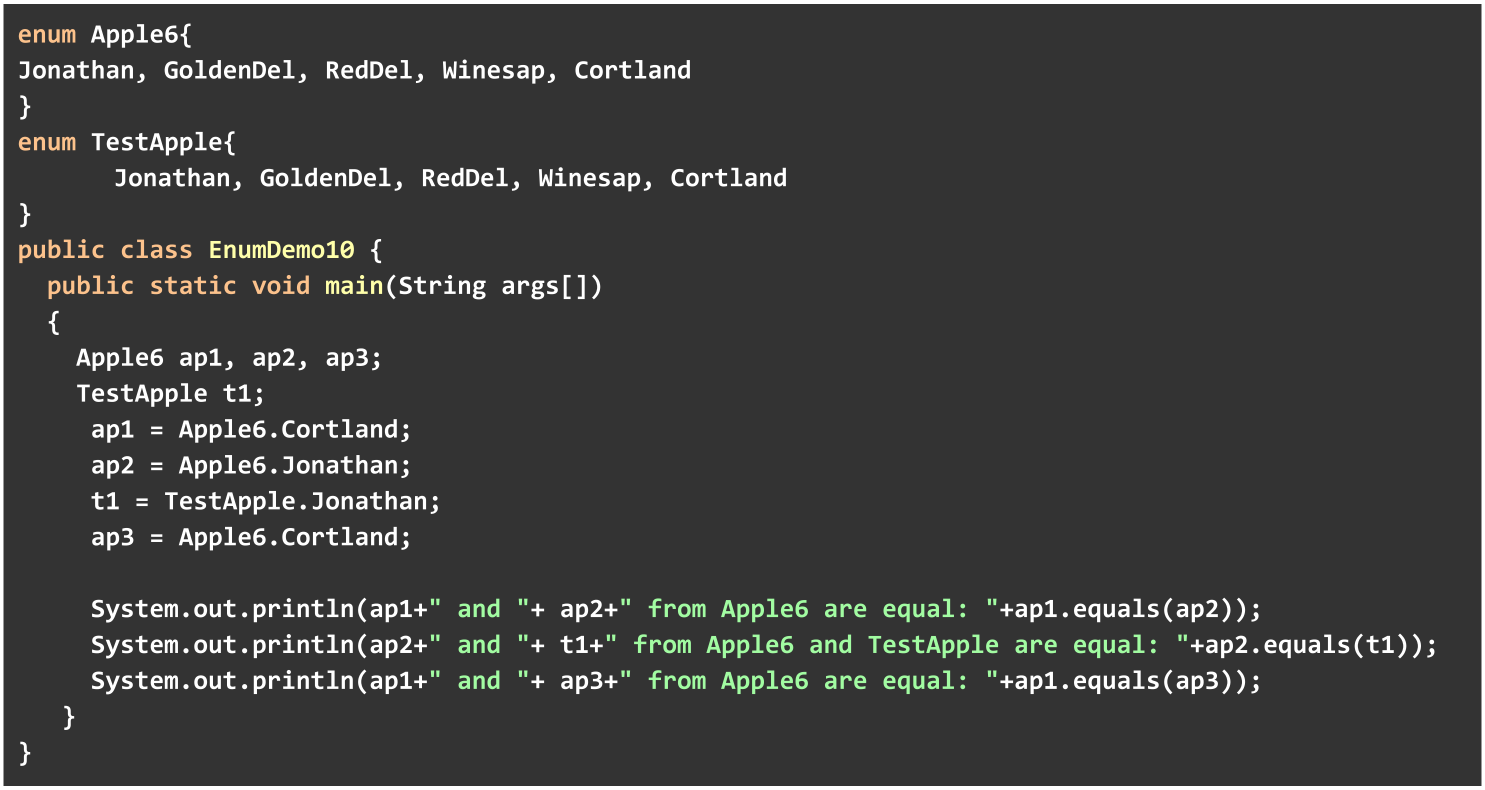
```
GoldenDel comes before RedDel
RedDel equals RedDel
```

## equals():

- An enumeration constant can be compared with any other object by using **equals( ),** which overrides the **equals( )** method defined by **Object**.
- Although **equals()** can compare an enum constant to any other object, the two objects will only be considered equal if they refer to the same constant within the same enum.

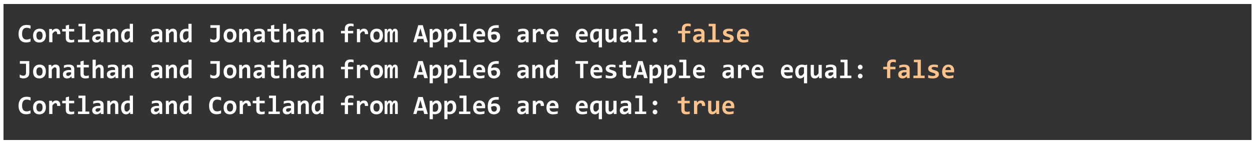
**Example using equals() method**

**Ex10.**



```java
enum Apple6{
Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
enum TestApple{
      Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
public class EnumDemo10 {
  public static void main(String args[])
  {
    Apple6 ap1, ap2, ap3;
    TestApple t1;
     ap1 = Apple6.Cortland;
     ap2 = Apple6.Jonathan;
     t1 = TestApple.Jonathan;
     ap3 = Apple6.Cortland;

     System.out.println(ap1+" and "+ ap2+" from Apple6 are equal: "+ap1.equals(ap2));
     System.out.println(ap2+" and "+ t1+" from Apple6 and TestApple are equal: "+ap2.equals(t1));
     System.out.println(ap1+" and "+ ap3+" from Apple6 are equal: "+ap1.equals(ap3));
   }
}
```

**Output:**

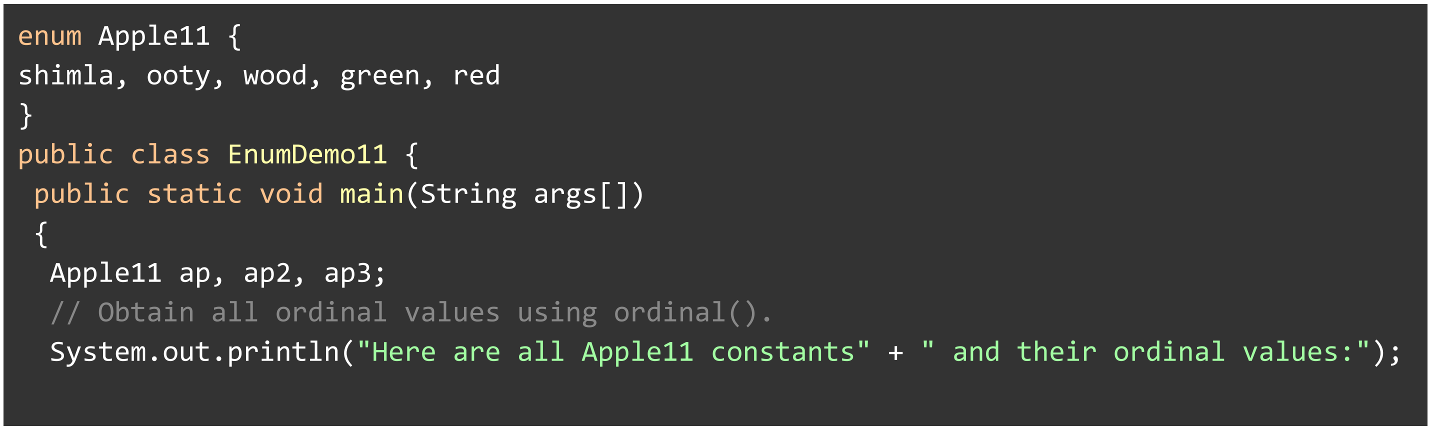


```
Cortland and Jonathan from Apple6 are equal: false
Jonathan and Jonathan from Apple6 and TestApple are equal: false
Cortland and Cortland from Apple6 are equal: true
```

3. We can compare 2 enumeration references for equality using ==( operator).

**Example using ordinal(), compareTo(), equals() and == operator**

**Ex11.**



```java
enum Apple11 {
shimla, ooty, wood, green, red
}
public class EnumDemo11 {
 public static void main(String args[])
 {
  Apple11 ap, ap2, ap3;
  // Obtain all ordinal values using ordinal().
  System.out.println("Here are all Apple11 constants" + " and their ordinal values:");
for(Apple11 a : Apple11.values())
    System.out.println(a + " " + a.ordinal());
  ap = Apple11.wood;
  ap2 = Apple11.ooty;
  ap3 = Apple11.wood;
  System.out.println();
  // Demonstrate compareTo() and equals()
  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:**

```
Here are all Apple11 constants and their ordinal values:
shimla 0
ooty 1
wood 2
green 3
red 4
ooty comes before wood
wood equals wood
wood equals wood
wood == wood
```

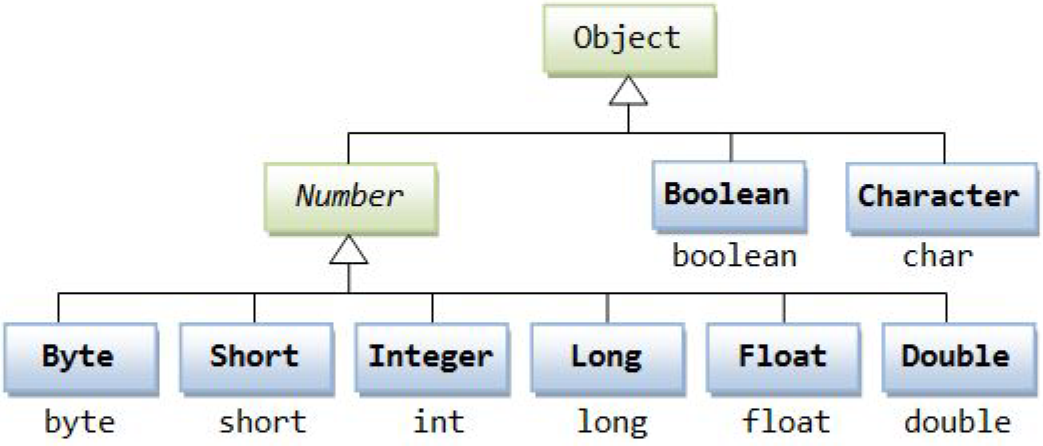
## Type Wrappers

1. Java uses primitive data types such as int, double, float etc. to hold the basic data types.
   **Ex.**
   Int a =10;
   Float f=24.7;
   Char ch='c';
2. Despite the performance benefits offered by the primitive data types, there are situations when you will need an object representation of the primitive data type.
3. For example, many data structures in Java operate on objects. So you cannot use primitive data types with those data structures.
4. To handle these situations ,Java provides ***type wrappers*** , which are classes that encapsulate a primitive type within an object.

**List of Primitive Data types and its corresponding Wrapper classes**

| **sl.no** | **Primitive** | **Wrapper Class** | **Constructor Argument** | **Methods to get Primitive Values** |
|---|---|---|---|---|
| 1 | byte | Byte | Byte(byte b) **or** Byte(String str) | byteValue() |
| 2 | short | Short | Short(short s) **or** Short(String str) | shortValue() |
| 3 | int | Integrer | Integer(int i) **or** Integer(String str) | intValue() |
| 4 | float | Float | Float(float f) **or** Float(float str) | floatValue() |
| 5 | double | Double | Double(double d) **or** Double(String str) | doubleValue() |
| 6 | long | Long | Long(long l) **or** Long(String str) | longValue() |
| 7 | char | Character | Character(char ch) **or** Character(String str) | charValue() |
| 8 | boolean | Boolean | Boolean(boolean b) **or** Boolean(String str) | booleanValue() |

**Type Wrapper Hierarchy**



## Character:

- It encapsulates primitive type char within an object. The constructor for character is
  **`Character(char` *`ch`*`)`**
  Here, ***ch*** specifies the primitive character for which you want to create an object and **`Character`** specifies the Wrapper class.
- To obtain the char value contained in a **Character** object, the **charValue()** method is used.
  **`char charValue()`**
  It returns the encapsulated character

## Boolean:

- It encapsulates primitive type boolean within an object.
  **`Boolean (boolean` *`b`*`)`**
  To obtain primitive bool value contained in **Boolean** object, call
  **`boolean booleanValue()`**

## The Numeric Type Wrappers:

- The Numeric Type Wrappers in Java, including Byte, Short, Integer, Long, Float, and Double, inherit the abstract class Number.
- These wrappers provide methods like **byteValue(), doubleValue(), floatValue(), intValue(), longValue(), and shortValue()** to return the value in different number formats.
- Constructors like **Integer(int *num*)** and **Integer(String *str*)** allow object creation from given values or string representations, throwing a **NumberFormatException** if the string is invalid.
- Overriding **toString()** provides a human-readable form of the value, enabling direct output using **println()**.
- **Boxing**: The process of encapsulating a value within an object is called *boxing*.

  ```
  Integer iob = new Integer(100);
  ```
- **Unboxing**: The process of extracting value from a type wrapper is called *unboxing*.

  ```
  int i = iob.intValue();
  ```

**Ex1**: Program to demonstrate how to use numeric wrapper to encapsulate a value and then extract that value.

```
public class BoxingUnboxing {
      public static void main(String[] args) {
            Integer iob = new Integer(100); // boxing
            int i = iob.intValue(); //unboxing
            System.out.println("value from primitive variable:"+i);
            System.out.println("value from wrapper object: "+iob);
      }
}
```

**Output:**

```
value from primitive variable:100
value from wrapper object: 100
```

Following example shows constructors in wrapper classes.

**Ex2.**

```
public class WrapperDemo1 {
public static void main(String[] args)
{
Byte B1 = new Byte((byte) 10); //Constructor which takes byte value as an argument
Byte B2 = new Byte("10"); //Constructor which takes String as an argument
//Byte B3 = new Byte("abc"); //Run Time Error : NumberFormatException
//Because, String abc can not be parse-able to byte
Short S1 = new Short((short) 20); //Constructor which takes short value as an argument
Short S2 = new Short("10"); //Constructor which takes String as an argument
Integer I1 = new Integer(30); //Constructor which takes int value as an argument
Integer I2 = new Integer("30"); //Constructor which takes String as an argument
Long L1 = new Long(40); //Constructor which takes long value as an argument
Long L2 = new Long("40"); //Constructor which takes String as an argument
Float F1 = new Float(12.2f); //Constructor which takes float value as an argument
Float F2 = new Float("15.6"); //Constructor which takes String as an argument
Float F3 = new Float(15.6d); //Constructor which takes double value as an argument
Double D1 = new Double(17.8d); //Constructor which takes double value as an argument
Double D2 = new Double("17.8"); //Constructor which takes String as an argument
Boolean BLN1 = new Boolean(false); //Constructor which takes boolean value as an argument
Boolean BLN2 = new Boolean("true"); //Constructor which takes String as an argument
Character C1 = new Character('D'); //Constructor which takes char value as an argument
Character C2 = new Character("abc"); //Compile time error : String abc can not be
converted to character
}
}
```

## Autoboxing and Unboxing

- Autoboxing and Unboxing features were 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.

**Autoboxing:**

- With autoboxing there is no need to explicitly construct an object to wrap a primitive type.
- You need to assign that value to a type-wrapper reference, Java automatically constructs an object..

**Auto-unboxing:**

- To unbox an object ,just assign that object reference to a primitive type reference variable.

**Ex3.** Demonstrate autoboxing and unboxing

```
public class AutoBox {
      public static void main(String[] args) {
            Integer iob = 100; // autobox
            int i = iob; //auto-unbox
            System.out.println(i+" "+iob);
      }
}
```

**Output:**

```
100 100
```

### Autoboxing and Methods:

Autoboxing and auto-unboxing might occur when an argument is passed to a method or when a value is returned by a method.

**Ex4.**

```
class AutoBox2 {
// Take an Integer parameter and return an int value;
static int m(Integer v) {
return v ; // auto-unbox to int
}
public static void main(String args[]) {
// Pass an int to m() and assign the return value to an Integer. Here, the
argument 100 is autoboxed into an Integer. The return value is also autoboxed
// into an Integer.
Integer iOb = m(100);
System.out.println(iOb);
}
}
```

**Output:**

```
100
```

- In Program the method **m()** specifies an **Integer** parameter and returns an **int** result.
- In **main()**, method **m()** is passed with value 100, as method **m()** expecting an **Integer** this value is automatically boxed.
- Then, **m()** auto-unboxes the value of **v** and returns the **int** equivalent of its argument.
- This **int** value is assigned to **iob** in **main(),** this causes the returned **int** value to be autoboxed.

## Autoboxing/Unboxing Occurs in Expressions

- Whenever we use an object of the Wrapper class in an expression, automatic unboxing and boxing is done by JVM.

**Ex5.**

```
public class Autobox3 {
      public static void main(String args[]) {
            Integer iOb;
            iOb = 100; //Autoboxing of int
            ++iOb;
            System.out.println(iOb);
      }
}
```

- When we perform an increment operation on an Integer object, it is first unboxed, then incremented and then again reboxed into an Integer type object.
- This will always happen, when we will use Wrapper class objects in expressions or conditions etc.

**Ex: 6**

```
class Autobox4 {
public static void main(String args[]) {
Integer i = 35;
Double d = 33.3;
d = d + i;
System.out.println("Value of d is " + d);
}
}
```

**Output:**

```
Value of d is 68.3
```

- Because of auto-unboxing, we can use Integer numeric objects to control a switch statement

```
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.

### Autoboxing/Unboxing Boolean and Character Values

- Because of auto-unboxing, a Boolean object can be used in conditional statements and looping statements in Java.
  **Ex**:

```
// Autoboxing/unboxing a Boolean and Character.
class AutoBox5 {
      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 is true");
      // Autobox/unbox a char.
      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
```

### Autoboxing/Unboxing Helps Prevent Errors

- Autoboxing and unboxing in Java offer convenience and error prevention.
- They automatically convert primitive types to their corresponding wrapper objects and vice versa.
- This automation helps prevent errors, as shown in the example below:

```
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 !
    }
}
```

- In this example, the program tries to unbox an **Integer** object **iOb** to a **byte**.
- However, the manual unboxing truncates the value, resulting in an unexpected output of -24 instead of the expected 1000.
- Auto-unboxing, on the other hand, prevents such errors by always producing a value compatible with the target type.

### A Word of Warning

- Autoboxing and auto-unboxing can make code less efficient if used excessively.
- While they allow primitives to be treated like objects, using them for simple operations, like in the example below, adds unnecessary overhead:

```
Double a, b, c;
a = 10.0;
b = 4.0;
c = Math.sqrt(a*a + b*b);
System.out.println("Hypotenuse is " + c);
```

- In this code, **Double** objects are used for basic arithmetic, which is less efficient than using primitive types like **double**.
- It's best to reserve wrapper types for cases where object representation is needed, rather than replacing primitives entirely.

## Annotations (Metadata)

1. Java **Annotations** allow us to add metadata information into our source code, Annotations were added to the java from JDK 5.
2. Annotations in Java provide additional information to the compiler and Java.
3. Annotations do not impact the execution of the code that they annotate. However they can change the way a program is treated by the compiler.
4. Annotations are used in tools during both development and deployment.

## Annotation Basics

1. They can be attached with various java elements such as classes, methods, fields,constructors, enum and even to an annotation.
2. Annotations are created through a mechanism based on the interface.
3. Annotation is defined using the '@' symbol followed by the annotation name.
4. Annotation can include elements with values, which can be specified when using the annotation.

**Syntax**:

```
@interface <annotation-name>{  element1,element2,....} //declaration
@<annotation-name>(element1=value,element2=value2..) //applying
annotation with its member values
```

5. All annotations consist solely of method declaration and methods act much like fields.
6. An annotation cannot include an **'extends'** clause**.** All Annotation types automatically extends an *'**Annotation'*** interface which is a super-interface for all the annotations.
7. It is declared within the **java.lang.annotation** package.

let's take the example of an annotation called **MyAnno:**

```
// A simple annotation type.
@interface MyAnno {
    String str();
    int val();
}
```

- In this declaration, the @ symbol before interface indicates that it's an annotation type.
- Annotations contain method declarations, like **str()** and **val(),** but without method bodies.Java handles these methods behind the scenes.
- When we apply an annotation we give value to its members.For instance:

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

- Here, **@MyAnno** is linked to the **myMeth()** method.
- Annotations are followed by a list of member initializations in parentheses.
- Each member is assigned a value using its name, without parentheses. This makes annotation members resemble fields.

## Specifying a Retention Policy

- Annotation retention policies determine when an annotation is discarded.
- Java defines 3 policies which are encapsulated within the '**java.lang.annotation.RetentionPolicy'** enumeration.
- Java has three retention policies: SOURCE, CLASS, and RUNTIME.
    - SOURCE: Retained only in the source file, discarded during compilation.
    - CLASS: Stored in the .class file during compilation but not available at runtime.

- RUNTIME: Stored in the .class file and available at runtime, offering the longest persistence.

- To specify a retention policy for an annotation, use the **@Retention** annotation with the desired policy. If no policy is specified, CLASS is used by default.
- Its general form is shown here:

  `@Retention(retention-policy)`
  Here, `retention-policy` must be one of the enumeration constants

- 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();
}
```

## Types of Annotations

### 1. Marker Annotations

- A marker annotation is a special type of annotation with no members, used solely to mark an item.
- Its presence indicates a certain characteristic or behavior.
- Marker Annotation's primary purpose is to convey information about the annotated element without providing any additional parameters or values.
- To check if a marker annotation is present, we use **isAnnotationPresent()** method, which is defined by the **AnnotatedElement** interface.

```
import java.lang.annotation.*;
import java.lang.reflect.Method;
//A marker annotation.
@Retention(RetentionPolicy.RUNTIME)
@interface MyMarker { }
class MarkerAnnotation {
// Annotate a method using a marker.
// Notice that no () is needed.
@MyMarker
public static void myMeth() {
        MarkerAnnotation ob = new MarkerAnnotation();
    try {
        Method m = ob.getClass().getMethod("myMeth");
        // Determine if the annotation is present.
```

```
            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.
```

- Note that when applying the marker annotation, like **@MyMarke**r, no parentheses are needed.

**2. Single-Member Annotations**

- A single-member annotation is an annotation with only one member.
- It behaves like a regular annotation but allows a shortcut for specifying the value of its single member.
- When there's only one member, you can directly provide its value without specifying the member's name.
- However, for this shortcut to work, the member must be named **value**.

```
 import java.lang.annotation.*;
import java.lang.reflect.*;
// Define a single-member annotation
@Retention(RetentionPolicy.RUNTIME)
@interface MySingle {
   int value(); // The member must be named 'value'
}
class Single {
   // Annotate a method using the single-member annotation
   @MySingle(100)
   public static void myMeth() {
       Single ob = new Single();
       try {
           Method m = ob.getClass().getMethod("myMeth");
           MySingle anno = m.getAnnotation(MySingle.class);
           System.out.println(anno.value()); // Displays 100
       } catch (NoSuchMethodException exc) {
           System.out.println("Method Not Found.");
       }
```

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

- In this example, **@MySingle(100)** annotates the method **myMeth().**
- Since it's a single-member annotation and the member is named **value**, we can directly specify its value without using the member's name.
- Additionally, single-value syntax can be used for annotations with default values for other members. For instance:

```
@interface SomeAnno {
    int value();
    int xyz() default 0;
}

// Applying the annotation with single-value syntax
@SomeAnno(88)
```

- In this case, if **xyz** is not specified, it defaults to zero, and only the value for **value** needs to be provided using the single-member syntax.
- If you want to specify a value for both **value** and **xyz**, you must explicitly name both members. **Ex**, **@SomeAnno(value = 88, xyz = 99)**

## The Built-In Annotations

- There are many built-in annotations in java. Some annotations are applied to java code and some to other annotations.
- Some annotations are specialized and 9 are general purpose
- Four are imported from **java.lang.annotation**:
    a. @Retention
    b. @Documented
    c. @Target, and
    d. @Inherited.
- Five are imported from j**ava.lang:**
    a. @Override
    b. @Deprecated
    c. @FunctionalInterface,
    d. @SafeVarargs and
    e. @SuppressWarnings

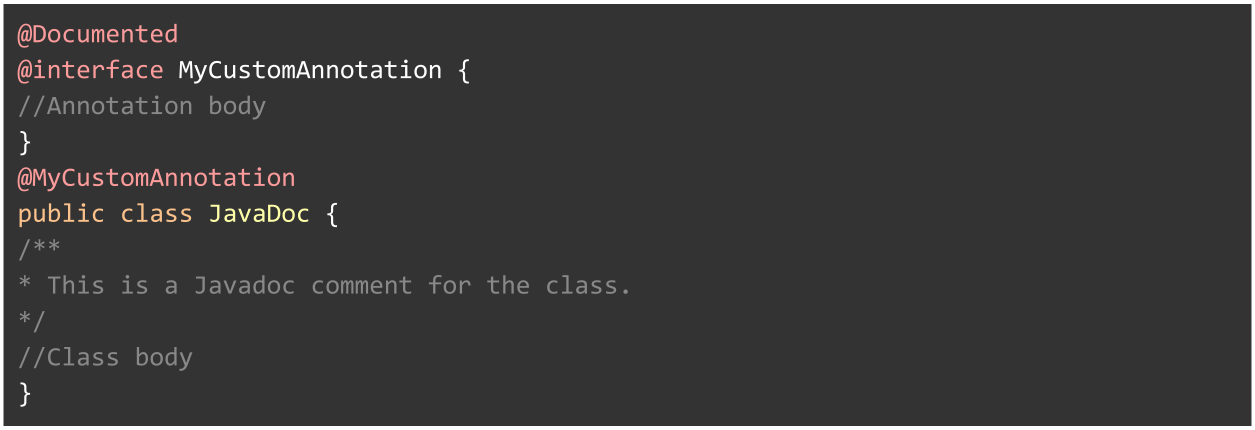
**@Retention:** It is designed to be used only as an annotation to another annotation. It specifies the retention policy.

**Example**: @Retention(RetentionPolicy.RUNTIME)

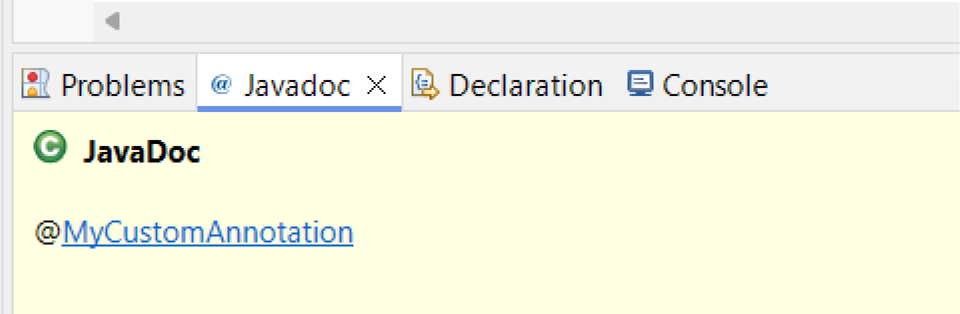
## @Documented:

- It is a marker interface that tells a tool that an annotation is to be documented.
- Use of **@Documented** annotation in the code enables tools like Javadoc to process it and include the annotation type information in the generated document.

Example:



```
@Documented
@interface MyCustomAnnotation {
//Annotation body
}
@MyCustomAnnotation
public class JavaDoc {
/**
* This is a Javadoc comment for the class.
*/
//Class body
}
```

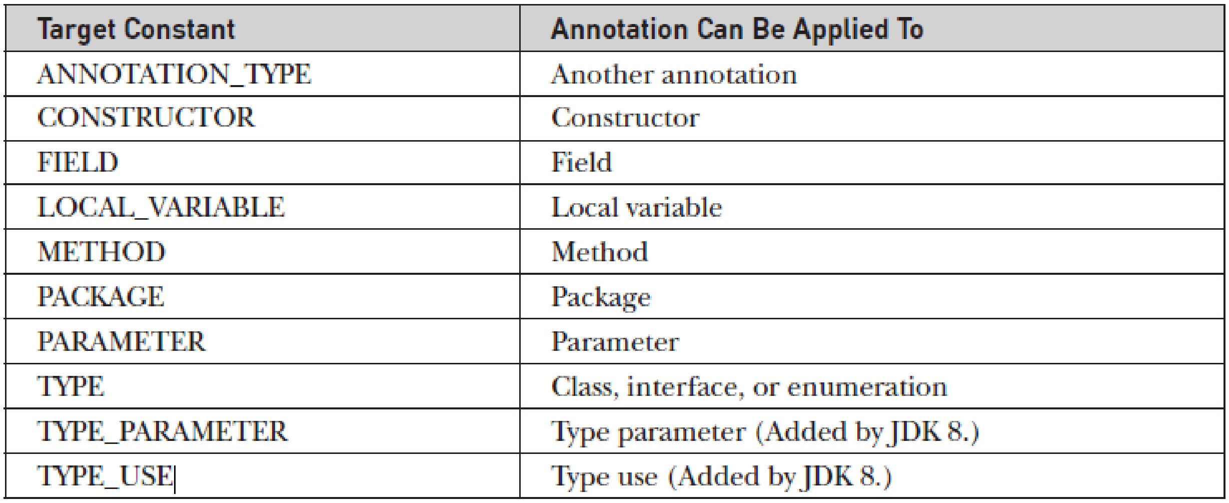


JavaDoc

@MyCustomAnnotation

## @Target :

- This annotation specifies the types of items to which an annotation can be applied.
- It's used only to annotate other annotations.
- When using **@Target**, we provide an argument, which is an array of constants from the **ElementType** enumeration.
- This argument specifies the types of things (like classes, methods, or fields) to which the annotation can be applied.



| Target Constant | Annotation Can Be Applied To |
|---|---|
| ANNOTATION_TYPE | Another annotation |
| CONSTRUCTOR | Constructor |
| FIELD | Field |
| LOCAL_VARIABLE | Local variable |
| METHOD | Method |
| PACKAGE | Package |
| PARAMETER | Parameter |
| TYPE | Class, interface, or enumeration |
| TYPE_PARAMETER | Type parameter (Added by JDK 8.) |
| TYPE_USE | Type use (Added by JDK 8.) |

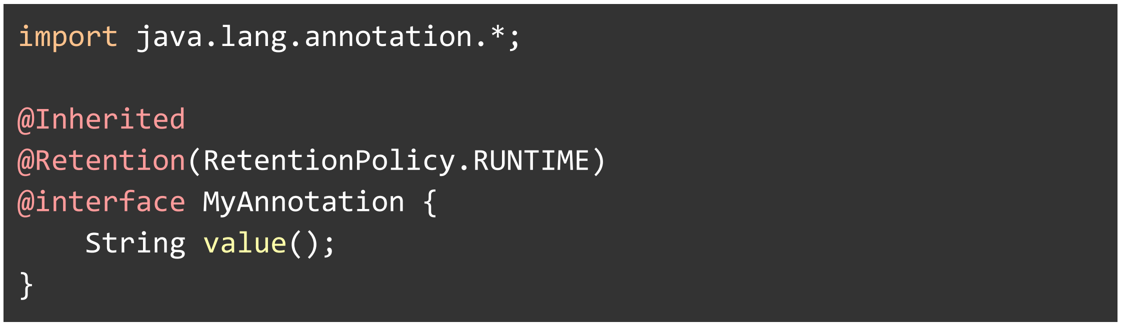
For example, to specify that an annotation applies only to fields and local variables, you can use this @Target annotation:

@Target( { ElementType.FIELD, ElementType.LOCAL_VARIABLE } )

### @Inherited:

- @Inherited is a marker annotation that can be used only on another annotation declaration.
- It affects only annotations that will be used on class declarations.
- The @Inherited annotation in Java is used to indicate that an annotation should be inherited from the superclass to the subclass.
- This means that if a class does not directly have a particular annotation, the Java runtime will check if its superclass has the annotation, and if so, it will be inherited by the subclass.

  Example:



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

@Inherited
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation {
    String value();
}
```

## @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.

## @Deprecated:

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

## @FunctionalInterface

- **@FunctionalInterface** is a marker annotation introduced in JDK 8 for interfaces.
- It signals that the annotated interface is a functional interface, which means it has only one abstract method. Functional interfaces are essential for lambda expressions.
- If an interface annotated with **@FunctionalInterface** doesn't meet the criteria of having exactly one abstract method, it will cause a compilation error.
- It's important to note that **@FunctionalInterface** is not necessary to create a functional interface; any interface with just one abstract method automatically qualifies as a functional interface.
- Therefore, **@FunctionalInterface** serves as informational, providing clarity about the functional nature of an interface.

## @SafeVarargs

- **@SafeVarargs** is a marker annotation that can be applied to methods and constructors.
- It is used to indicate that a method or constructor is safe to be called with a variable number of arguments (varargs) without causing any unsafe actions related to type safety.
- It helps suppress unchecked warnings that might occur due to varargs usage with non-reifiable types or parameterized array instantiation.

## @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.