enum in Java

Enumerations serve the purpose of representing a group of named constants in a programming language. For example the 4 suits in a deck of playing cards may be 4 enumerators named Club, Diamond, Heart, and Spade, belonging to an enumerated type named Suit. Other examples include natural enumerated types (like the planets, days of the week, colors, directions, etc.).

Enums are used when we know all possible values at **compile time**, such as choices on a menu, rounding modes, command line flags, etc. It is not necessary that the set of constants in an enum type stay **fixed** for all time.

In Java (from 1.5), enums are represented using **enum** data type. Java enums are more powerful than [C/C++ enums](http://quiz.geeksforgeeks.org/enumeration-enum-c/). In Java, we can also add variables, methods and constructors to it. The main objective of enum is to define our ow datatypes(Enumerated Data Types).

**Declaration of enum in java :**

* Enum declaration can be done outside a Class or inside a Class but not inside a Method.

|  |
| --- |
| // A simple enum example where enum is declared  // outside any class (Note enum keyword instead of  // class keyword)  enum Color  {      RED, GREEN, BLUE;  }    public class Test  {      // Driver method      public static void main(String[] args)      {          Color c1 = Color.RED;          System.out.println(c1);      }  } |

* Output :
* RED

|  |
| --- |
| // enum declaration inside a class.  public class Test  {      enum Color      {          RED, GREEN, BLUE;      }        // Driver method      public static void main(String[] args)      {          Color c1 = Color.RED;          System.out.println(c1);      }  } |

* Output :
* RED
* First line inside enum should be list of constants and then other things like methods, variables and constructor.
* According to [Java naming conventions](http://www.oracle.com/technetwork/java/codeconventions-135099.html), it is recommended that we name constant with all capital letters

**Important points of enum :**

* Every enum internally implemented by using Class.
* /\* internally above enum Color is converted to
* class Color
* {
* public static final Color RED = new Color();
* public static final Color BLUE = new Color();
* public static final Color GREEN = new Color();
* }\*/
* Every enum constant represents an **object** of type enum.
* enum type can be passed as an argument to **switch**statement.

|  |
| --- |
| // A Java program to demonstrate working on enum  // in switch case (Filename Test. Java)  import java.util.Scanner;    // An Enum class  enum Day  {      SUNDAY, MONDAY, TUESDAY, WEDNESDAY,      THURSDAY, FRIDAY, SATURDAY;  }    // Driver class that contains an object of "day" and  // main().  public class Test  {      Day day;        // Constructor      public Test(Day day)      {          this.day = day;      }        // Prints a line about Day using switch      public void dayIsLike()      {          switch (day)          {          case MONDAY:              System.out.println("Mondays are bad.");              break;          case FRIDAY:              System.out.println("Fridays are better.");              break;          case SATURDAY:          case SUNDAY:              System.out.println("Weekends are best.");              break;          default:              System.out.println("Midweek days are so-so.");              break;          }      }        // Driver method      public static void main(String[] args)      {          String str = "MONDAY";          Test t1 = new Test(Day.valueOf(str));          t1.dayIsLike();      }  } |

* Output:
* Mondays are bad.
* Every enum constant is always implicitly **public static final**. Since it is **static**, we can access it by using enum Name. Since it is **final**, we can’t create child enums.
* We can declare **main() method** inside enum. Hence we can invoke enum directly from the Command Prompt.

|  |
| --- |
| // A Java program to demonstrate that we can have  // main() inside enum class.  enum Color  {      RED, GREEN, BLUE;        // Driver method      public static void main(String[] args)      {          Color c1 = Color.RED;          System.out.println(c1);      }  } |

* Output :
* RED

**Enum and Inheritance :**

* All enums implicitly extend **java.lang.Enum class**. As a class can only extend **one** parent in Java, so an enum cannot extend anything else.
* **toString() method** is overridden in **java.lang.Enum class**,which returns enum constant name.
* enum can implement many interfaces.

**values(), ordinal() and valueOf() methods :**

* These methods are present inside **java.lang.Enum**.
* **values() method** can be used to return all values present inside enum.
* Order is important in enums.By using **ordinal() method**, each enum constant index can be found, just like array index.
* **valueOf() method** returns the enum constant of the specified string value, if exists.

|  |
| --- |
| // Java program to demonstrate working of values(),  // ordinal() and valueOf()  enum Color  {      RED, GREEN, BLUE;  }    public class Test  {      public static void main(String[] args)      {          // Calling values()          Color arr[] = Color.values();            // enum with loop          for (Color col : arr)          {              // Calling ordinal() to find index              // of color.              System.out.println(col + " at index "                               + col.ordinal());          }            // Using valueOf(). Returns an object of          // Color with given constant.          // Uncommenting second line causes exception          // IllegalArgumentException          System.out.println(Color.valueOf("RED"));          // System.out.println(Color.valueOf("WHITE"));      }  } |

* Output :
* RED at index 0
* GREEN at index 1
* BLUE at index 2
* RED

**enum and constructor :**

* enum can contain constructor and it is executed separately for each enum constant at the time of enum class loading.
* We can’t create enum objects explicitly and hence we can’t invoke enum constructor directly.

**enum and methods :**

* enum can contain **concrete** methods only i.e. no any **abstract** method.

|  |
| --- |
| // Java program to demonstrate that enums can have constructor  // and concrete methods.    // An enum (Note enum keyword inplace of class keyword)  enum Color  {      RED, GREEN, BLUE;        // enum constructor called separately for each      // constant      private Color()      {          System.out.println("Constructor called for : " +          this.toString());      }        // Only concrete (not abstract) methods allowed      public void colorInfo()      {          System.out.println("Universal Color");      }  }    public class Test  {      // Driver method      public static void main(String[] args)      {          Color c1 = Color.RED;          System.out.println(c1);          c1.colorInfo();      }  } |

Output:

Constructor called for : RED

Constructor called for : GREEN

Constructor called for : BLUE

RED

Universal Color

# Java Enum

**Enum in java** is a data type that contains fixed set of constants.

It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY and SATURDAY) , directions (NORTH, SOUTH, EAST and WEST) etc. The java enum constants are static and final implicitly. It is available from JDK 1.5.

Java Enums can be thought of as classes that have fixed set of constants.

## Points to remember for Java Enum

* enum improves type safety
* enum can be easily used in switch
* enum can be traversed
* enum can have fields, constructors and methods
* enum may implement many interfaces but cannot extend any class because it internally extends Enum class

### Simple Example of Java Enum

1. **class** EnumExample1{
2. **public** **enum** Season { WINTER, SPRING, SUMMER, FALL }
4. **public** **static** **void** main(String[] args) {
5. **for** (Season s : Season.values())
6. System.out.println(s);
8. }}

Output:WINTER

SPRING

SUMMER

FALL

### What is the purpose of values() method in enum?

The java compiler internally adds the values() method when it creates an enum. The values() method returns an array containing all the values of the enum.

### Internal code generated by the compiler for the above example of enum type

The java compiler internally creates a static and final class that extends the Enum class as shown in the below example:

1. **public** **static** **final** **class** EnumExample1$Season **extends** Enum
2. {
3. **private** EnumExample1$Season(String s, **int** i)
4. {
5. **super**(s, i);
6. }
8. **public** **static** EnumExample1$Season[] values()
9. {
10. **return** (EnumExample1$Season[])$VALUES.clone();
11. }
13. **public** **static** EnumExample1$Season valueOf(String s)
14. {
15. **return** (EnumExample1$Season)Enum.valueOf(EnumExample1$Season, s);
16. }
18. **public** **static** **final** EnumExample1$Season WINTER;
19. **public** **static** **final** EnumExample1$Season SPRING;
20. **public** **static** **final** EnumExample1$Season SUMMER;
21. **public** **static** **final** EnumExample1$Season FALL;
22. **private** **static** **final** EnumExample1$Season $VALUES[];
24. **static**
25. {
26. WINTER = **new** EnumExample1$Season("WINTER", 0);
27. SPRING = **new** EnumExample1$Season("SPRING", 1);
28. SUMMER = **new** EnumExample1$Season("SUMMER", 2);
29. FALL = **new** EnumExample1$Season("FALL", 3);
30. $VALUES = (**new** EnumExample1$Season[] {
31. WINTER, SPRING, SUMMER, FALL
32. });
33. }
35. }

## Defining Java enum

The enum can be defined within or outside the class because it is similar to a class.

### Java enum example: defined outside class

1. **enum** Season { WINTER, SPRING, SUMMER, FALL }
2. **class** EnumExample2{
3. **public** **static** **void** main(String[] args) {
4. Season s=Season.WINTER;
5. System.out.println(s);
6. }}

Output:WINTER

### Java enum example: defined inside class

1. **class** EnumExample3{
2. **enum** Season { WINTER, SPRING, SUMMER, FALL; }//semicolon(;) is optional here
3. **public** **static** **void** main(String[] args) {
4. Season s=Season.WINTER;//enum type is required to access WINTER
5. System.out.println(s);
6. }}

Output:WINTER

### Initializing specific values to the enum constants

The enum constants have initial value that starts from 0, 1, 2, 3 and so on. But we can initialize the specific value to the enum constants by defining fields and constructors. As specified earlier, Enum can have fields, constructors and methods.

### Example of specifying initial value to the enum constants

1. **class** EnumExample4{
2. **enum** Season{
3. WINTER(5), SPRING(10), SUMMER(15), FALL(20);
5. **private** **int** value;
6. **private** Season(**int** value){
7. **this**.value=value;
8. }
9. }
10. **public** **static** **void** main(String args[]){
11. **for** (Season s : Season.values())
12. System.out.println(s+" "+s.value);
14. }}

Output:WINTER 5

SPRING 10

SUMMER 15

FALL 20

#### Constructor of enum type is private. If you don't declare private compiler internally creates private constructor.

1. **enum** Season{
2. WINTER(10),SUMMER(20);
3. **private** **int** value;
4. Season(**int** value){
5. **this**.value=value;
6. }
7. }

### Internal code generated by the compiler for the above example of enum type

1. **final** **class** Season **extends** Enum
2. {
3. **public** **static** Season[] values()
4. {
5. **return** (Season[])$VALUES.clone();
6. }
7. **public** **static** Season valueOf(String s)
8. {
9. **return** (Season)Enum.valueOf(Season, s);
10. }
11. **private** Season(String s, **int** i, **int** j)
12. {
13. **super**(s, i);
14. value = j;
15. }
16. **public** **static** **final** Season WINTER;
17. **public** **static** **final** Season SUMMER;
18. **private** **int** value;
19. **private** **static** **final** Season $VALUES[];
20. **static**
21. {
22. WINTER = **new** Season("WINTER", 0, 10);
23. SUMMER = **new** Season("SUMMER", 1, 20);
24. $VALUES = (**new** Season[] {
25. WINTER, SUMMER
26. });
27. }
28. }

### Can we create the instance of enum by new keyword?

|  |
| --- |
| No, because it contains private constructors only. |

### Can we have abstract method in enum?

Yes, ofcourse! we can have abstract methods and can provide the implementation of these methods.

## Java enum in switch statement

We can apply enum on switch statement as in the given example:

### Example of applying enum on switch statement

1. **class** EnumExample5{
2. **enum** Day{ SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY}
3. **public** **static** **void** main(String args[]){
4. Day day=Day.MONDAY;
6. **switch**(day){
7. **case** SUNDAY:
8. System.out.println("sunday");
9. **break**;
10. **case** MONDAY:
11. System.out.println("monday");
12. **break**;
13. **default**:
14. System.out.println("other day");
15. }
16. }}

Output:monday

The Enumeration interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects.

This legacy interface has been superceded by Iterator. Although not deprecated, Enumeration is considered obsolete for new code. However, it is used by several methods defined by the legacy classes such as Vector and Properties, is used by several other API classes, and is currently in widespread use in application code.

The methods declared by Enumeration are summarized in the following table −

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **boolean hasMoreElements( )**  When implemented, it must return true while there are still more elements to extract, and false when all the elements have been enumerated. |
| 2 | **Object nextElement( )**  This returns the next object in the enumeration as a generic Object reference. |

## Example

Following is an example showing usage of Enumeration.

import java.util.Vector;

import java.util.Enumeration;

public class EnumerationTester {

public static void main(String args[]) {

Enumeration days;

Vector dayNames = new Vector();

dayNames.add("Sunday");

dayNames.add("Monday");

dayNames.add("Tuesday");

dayNames.add("Wednesday");

dayNames.add("Thursday");

dayNames.add("Friday");

dayNames.add("Saturday");

days = dayNames.elements();

while (days.hasMoreElements()) {

System.out.println(days.nextElement());

}

}

}

This will produce the following result −

## Output

Sunday

Monday

Tuesday

Wednesday

Thursday

Friday

Saturday

# Java Annotations tutorial with examples

[Java Annotations](https://docs.oracle.com/javase/tutorial/java/annotations/index.html) 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).

In this tutorial we are going to cover following topics: Usage of annotations, how to apply annotations, what predefined annotation types are available in the Java and how to create custom annotations.

## What’s the use of Annotations?

**1) Instructions to the compiler**: There are three built-in annotations available in Java (@Deprecated, @Override & @SuppressWarnings) that can be used for giving certain instructions to the compiler. For example the @override annotation is used for instructing compiler that the annotated method is overriding the method. More about these built-in annotations with example is discussed in the next sections of this article.

**2) Compile-time instructors**: Annotations can provide compile-time instructions to the compiler that can be further used by sofware build tools for generating code, XML files etc.

**3) Runtime instructions**: We can define annotations to be available at runtime which we can access using [java reflection](https://docs.oracle.com/javase/tutorial/reflect/) and can be used to give instructions to the program at runtime. We will discuss this with the help of an example, later in this same post.

## Annotations basics

An annotation always starts with the symbol @ followed by the annotation name. The symbol @indicates to the compiler that this is an annotation.

For e.g. @Override  
Here @ symbol represents that this is an annotation and the Override is the name of this annotation.

**Where we can use annotations?**  
Annotations can be applied to the classes, interfaces, methods and fields. For example the below annotation is being applied to the method.

@Override

void myMethod() {

//Do something

}

What this annotation is exactly doing here is explained in the next section but to be brief it is instructing compiler that myMethod() is a overriding method which is overriding the method (myMethod()) of super class.

## Built-in Annotations in Java

Java has three built-in annotations:

* @Override
* @Deprecated
* @SuppressWarnings

#### 1) @Override:

While overriding a method in the child class, we should use this annotation to mark that method. This makes code readable and avoid maintenance issues, such as: while changing the method signature of parent class, you must change the signature in child classes (where this annotation is being used) otherwise compiler would throw compilation error. This is difficult to trace when you haven’t used this annotation.

Example:

public class MyParentClass {

public void justaMethod() {

System.out.println("Parent class method");

}

}

public class MyChildClass extends MyParentClass {

@Override

public void justaMethod() {

System.out.println("Child class method");

}

}

I believe the example is self explanatory..

#### 2) @Deprecated

@Deprecated annotation indicates that the marked element (class, method or field) is deprecated and should no longer be used. The compiler generates a warning whenever a program uses a method, class, or field that has already been marked with the @Deprecated annotation. When an element is deprecated, it should also be documented using the Javadoc @deprecated tag, as shown in the following example. Make a note of case difference with @Deprecated and @deprecated. @deprecated is used for documentation purpose.

Example:

/\*\*

\* @deprecated

\* reason for why it was deprecated

\*/

@Deprecated

public void anyMethodHere(){

// Do something

}

Now, whenever any program would use this method, the compiler would generate a warning..

#### 3) @SuppressWarnings

This annotation instructs compiler to ignore specific warnings. For example in the below code, I am calling a deprecated method (lets assume that the method deprecatedMethod() is marked with @Deprecated annotation) so the compiler should generate a warning, however I am using @SuppressWarnings annotation that would suppress that deprecation warning.

@SuppressWarnings("deprecation")

void myMethod() {

myObject.deprecatedMethod();

}

## Creating Custom Annotations

* Annotations are created by using @interface, followed by annotation name as shown in the below example.
* An annotation can have elements as well. They look like methods. For example in the below code, we have four elements. We should not provide implementation for these elements.
* All annotations extends java.lang.annotation.Annotation interface. Annotations cannot include any extends clause.

import java.lang.annotation.Documented;

import java.lang.annotation.ElementType;

import java.lang.annotation.Inherited;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Documented

@Target(ElementType.METHOD)

@Inherited

@Retention(RetentionPolicy.RUNTIME)

public @interface MyCustomAnnotation{

int studentAge() default 18;

String studentName();

String stuAddress();

String stuStream() default "CSE";

}

Note: All the elements that have default values set while creating annotations can be skipped while using annotation. For example if I’m applying the above annotation to a class then I would do it like this :

@MyCustomAnnotation(

studentName="Chaitanya",

stuAddress="Agra, India"

)

public class MyClass {

...

}

As you can see, we have not given any value to the studentAge and stuStream elements as it is optional to set the values of these elements (default values already been set in Annotation definition, but if you want you can assign new value while using annotation just the same way as we did for other elements). However we have to provide the values of other elements (the elements that do not have default values set) while using annotation.

**Note**: We can also have array elements in an annotation. This is how we can use them:  
Annotation definition:

@interface MyCustomAnnotation {

int count();

String[] books();

}

Usage:

@MyCustomAnnotation(

count=3,

books={"C++", "Java"}

)

public class MyClass {

}

Lets back to the topic again: In the custom annotation example we have used these four annotations: @Documented, @Target, @Inherited & @Retention. Lets discuss them in detail.

#### @Documented

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

java.lang.annotation.Documented

@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

}

}

**Note**: 1) If you do not define any Target type that means annotation can be applied to any element.  
2) Apart from ElementType.METHOD, an annotation can have following possible Target values.  
ElementType.METHOD  
ElementType.PACKAGE  
ElementType.PARAMETER  
ElementType.TYPE  
ElementType.ANNOTATION\_TYPE  
ElementType.CONSTRUCTOR  
ElementType.LOCAL\_VARIABLE  
ElementType.FIELD

#### @Inherited

The @Inherited annotation signals that a custom annotation used in a class should be inherited by all of its sub classes. For example:

java.lang.annotation.Inherited

@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.

#### @Retention

It indicates how long annotations with the annotated type are to be retained.

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)

@interface MyCustomAnnotation {

}

Here we have used RetentionPolicy.RUNTIME. There are two other options as well. Lets see what do they mean:  
RetentionPolicy.RUNTIME: The annotation should be available at runtime, for inspection via java reflection.  
RetentionPolicy.CLASS: The annotation would be in the .class file but it would not be available at runtime.  
RetentionPolicy.SOURCE: The annotation would be available in the source code of the program, it would neither be in the .class file nor be available at the runtime.

# Annotations in Java

Annotations are used to provide supplement information about a program.

* Annotations start with ‘**@**’.
* Annotations do not change action of a compiled program.
* Annotations help to associate metadata (information) to the program elements i.e. instance variables, constructors, methods, classes, etc.
* Annotations are not pure comments as they can change the way a program is treated by compiler. See below code for example.

|  |
| --- |
| /\* Java program to demonstrate that annotations are     not barely comments (This program throws compiler     error because we have mentioned override, but not     overridden, we haver overloaded display) \*/  class Base  {       public void display()       {           System.out.println("Base display()");       }  }  class Derived extends Base  {       @Override       public void display(int x)       {           System.out.println("Derived display(int )");       }         public static void main(String args[])       {           Derived obj = new Derived();           obj.display();       }  } |

* Output :
* 10: error: method does not override or implement
* a method from a supertype
* If we remove parameter (int x) or we remove @override, the program compiles fine.

**Categories of Annotations**

There are 3 categories of Annotations:-  
**1. Marker Annotations:**  
The only purpose is to mark a declaration. These annotations contain no members and do not consist any data. Thus, its presence as an annotation is sufficient. Since, marker interface contains no members, simply determining whether it is present or absent is sufficient. **@Override** is an example of Marker Annotation.

Example: - @TestAnnotation()

**2. Single value Annotations:**  
These annotations contain only one member and allow a shorthand form of specifying the value of the member. We only need to specify the value for that member when the annotation is applied and don’t need to specify the name of the member. However in order to use this shorthand, the name of the member must be **value.**

Example: - @TestAnnotation(“testing”);

**3. Full Annotations:**  
These annotations consist of multiple data members/ name, value, pairs.

Example:- @TestAnnotation(owner=”Rahul”, value=”Class Geeks”)

**Predefined/ Standard Annotations**

Java defines seven built-in annotations.

* Four are imported from java.lang.annotation: **@Retention**, **@Documented**, **@Target**, and **@Inherited**.
* Three are included in java.lang: **@Deprecated, @Override**and **@SuppressWarnings**

**@Deprecated Annotation**

* It is a marker annotation. It indicates that a declaration is obsolete and has been replaced by a newer form.
* The Javadoc [@deprecated tag](http://docs.oracle.com/javase/1.5.0/docs/guide/javadoc/deprecation/deprecation.html#javadoc_tag)should be used when an element has been deprecated.
* @deprecated tag is for documentation and @Deprecated annotation is for runtime reflection.
* @deprecated tag has higher priority than @Deprecated annotation when both are together used.

|  |
| --- |
| public class DeprecatedTest  {      @Deprecated      public void Display()      {          System.out.println("Deprecatedtest display()");      }        public static void main(String args[])      {          DeprecatedTest d1 = new DeprecatedTest();          d1.Display();      }  } |

**Output:**

Deprecatedtest display()

**@Override Annotation**  
It 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 (see [this](https://ide.geeksforgeeks.org/ElmP5S) for example). It is used to ensure that a superclass method is actually overridden, and not simply overloaded.

Example:-

|  |
| --- |
| class Base  {       public void Display()       {           System.out.println("Base display()");       }         public static void main(String args[])       {           Base t1 = new Derived();           t1.Display();       }  }  class Derived extends Base  {       @Override       public void Display()       {           System.out.println("Derived display()");       }  } |

**Output:**

Derived display()

**@SuppressWarnings**  
It is used to inform the compiler to suppress specified compiler warnings. The warnings to suppress are specified by name, in string form. This type of annotation can be applied to any type of declaration.

Java groups warnings under two categories. They are : **deprecation**and **unchecked.**. Any unchecked warning is generated when a legacy code interfaces with a code that use generics.

|  |
| --- |
| class DeprecatedTest  {      @Deprecated      public void Display()      {          System.out.println("Deprecatedtest display()");      }  }    public class SuppressWarningTest  {      // If we comment below annotation, program generates      // warning      @SuppressWarnings({"checked", "deprecation"})      public static void main(String args[])      {          DeprecatedTest d1 = new DeprecatedTest();          d1.Display();      }  } |

**Output:**

Deprecatedtest display()

**@Documented Annotations**  
It is a marker interface that tells a tool that an annotation is to be documented. Annotations are not included by Javadoc comments. Use of @Documented annotation in the code enables tools like Javadoc to process it and include the annotation type information in the generated document.

**@Target**  
It is designed to be used only as an annotation to another annotation. **@Target**takes one argument, which must be constant from the **ElementType**enumeration. This argument specifies the type of declarations to which the annotation can be applied. The constants are shown below along with the type of declaration to which they correspond.

|  |  |
| --- | --- |
| **Target Constant** | **Annotations 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 |

We can specify one or more of these values in a **@Target**annotation. To specify multiple values, we must specify them within a braces-delimited list. 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}) **@Retention Annotation** It determines where and how long the annotation is retent. The 3 values that the @Retention annotation can have:

* **SOURCE:**Annotations will be retained at the source level and ignored by the compiler.
* **CLASS:**Annotations will be retained at compile time and ignored by the JVM.
* **RUNTIME:**These will be retained at runtime.

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

**User-defined/ Custom Annotations**

User-defined annotations can be used to annotate program elements, i.e. variables, constructors, methods, etc. These annotations can be applied just before declaration of an element (constructor, method, classes, etc).

Syntax of Declaration:-

[Access Specifier] @interface<AnnotationName>

{

DataType <Method Name>() [default value];

}

* **AnnotationName**is an identifier.
* Parameter should not be associated with method declarations and **throws**clause should not be used with method declaration.
* Parameters will not have a null value but can have a default value.
* **default value**is optional.
* Return type of method should be either primitive, enum, string, class name or array of primitive, enum, string or class name type.

|  |
| --- |
| package source;  // A Java program to demonstrate user defined annotations  import java.lang.annotation.Documented;  import java.lang.annotation.Retention;  import java.lang.annotation.RetentionPolicy;    // user-defined annotation  @Documented  @Retention(RetentionPolicy.RUNTIME)  @ interface TestAnnotation  {      String Developer() default "Rahul";      String Expirydate();  } // will be retained at runtime    // Driver class that uses @TestAnnotation  public class Test  {      @TestAnnotation(Developer="Rahul", Expirydate="01-10-2020")      void fun1()      {          System.out.println("Test method 1");      }        @TestAnnotation(Developer="Anil", Expirydate="01-10-2021")      void fun2()      {          System.out.println("Test method 2");      }        public static void main(String args[])      {          System.out.println("Hello");      }  } |

Output :

Hello