**Java Lambda Expressions Tutorial with examples**

BY CHAITANYA SINGH | FILED UNDER: [JAVA 8 FEATURES](https://beginnersbook.com/category/java-8-features/)

Lambda expression is a new feature which is introduced in Java 8. A lambda expression is an anonymous function. A function that doesn’t have a name and doesn’t belong to any class. The concept of lambda expression was first introduced in LISP programming language.

**Java Lambda Expression Syntax**

To create a lambda expression, we specify input parameters (if there are any) on the left side of the lambda operator ->, and place the expression or block of statements on the right side of lambda operator. For example, the lambda expression (x, y) -> x + y specifies that lambda expression takes two arguments x and y and returns the sum of these.

//Syntax of lambda expression

(parameter\_list) -> {function\_body}

**Lambda expression vs method in Java**

A method (or function) in Java has these main parts:  
1. Name  
2. Parameter list  
3. Body  
4. return type.

A lambda expression in Java has these main parts:  
Lambda expression **only has body and parameter list**.  
1. **No** name – function is anonymous so we don’t care about the name  
2. Parameter list  
3. Body – This is the main part of the function.  
4. **No** return type – The java 8 compiler is able to infer the return type by checking the code. you need not to mention it explicitly.

**Where to use the Lambdas in Java**

To use lambda expression, you need to either create your own functional interface or use the pre defined functional interface provided by Java. An interface with **only single abstract method** is called functional interface(or Single Abstract method interface), for example: Runnable, callable, ActionListener etc.

**To use function interface:**  
Pre Java 8: We create anonymous inner classes.  
Post Java 8: You can use lambda expression instead of anonymous inner classes.

**Java Lambda expression Example**

**Without using Lambda expression:** Prior to java 8 we used the anonymous inner classe to implement the only abstract method of functional interface.

import java.awt.\*;

import java.awt.event.\*;

public class ButtonListenerOldWay {

public static void main(String[] args) {

Frame frame=new Frame("ActionListener Before Java8");

Button b=new Button("Click Here");

b.setBounds(50,100,80,50);

b.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent e){

System.out.println("Hello World!");

}

});

frame.add(b);

frame.setSize(200,200);

frame.setLayout(null);

frame.setVisible(true);

}

}

**By using Lambda expression:** Instead of creating anonymous inner class, we can create a lambda expression like this:

import java.awt.\*;

public class ButtonListenerNewWay {

public static void main(String[] args) {

Frame frame=new Frame("ActionListener java8");

Button b=new Button("Click Here");

b.setBounds(50,100,80,50);

b.addActionListener(e -> System.out.println("Hello World!"));

frame.add(b);

frame.setSize(200,200);

frame.setLayout(null);

frame.setVisible(true);

}

}

**Note:**  
1. As you can see that we used less code with lambda expression.  
2. Backward compatibility: You can use the lambda expression with your old code. Lambdas are backward compatible so you can use them in existing API when you migrate your project to java 8.

Lets see few more examples of Lambda expressions.

**Example 1: Java Lambda Expression with no parameter**

@FunctionalInterface

interface MyFunctionalInterface {

//A method with no parameter

public String sayHello();

}

public class Example {

public static void main(String args[]) {

// lambda expression

MyFunctionalInterface msg = () -> {

return "Hello";

};

System.out.println(msg.sayHello());

}

}

Output:

Hello

**Example 2: Java Lambda Expression with single parameter**

@FunctionalInterface

interface MyFunctionalInterface {

//A method with single parameter

public int incrementByFive(int a);

}

public class Example {

public static void main(String args[]) {

// lambda expression with single parameter num

MyFunctionalInterface f = (num) -> num+5;

System.out.println(f.incrementByFive(22));

}

}

Output:

27

**Example 3: Java Lambda Expression with Multiple Parameters**

interface StringConcat {

public String sconcat(String a, String b);

}

public class Example {

public static void main(String args[]) {

// lambda expression with multiple arguments

StringConcat s = (str1, str2) -> str1 + str2;

System.out.println("Result: "+s.sconcat("Hello ", "World"));

}

}

Output:

Result: Hello World

**Example 4: Iterating collections using foreach loop**

import java.util.\*;

public class Example{

    public static void main(String[] args) {

      List<String> list=new ArrayList<String>();

       list.add("Rick");

list.add("Negan");

  list.add("Daryl");

list.add("Glenn");

list.add("Carl");

  list.forEach(

// lambda expression

    (names)->System.out.println(names)

);

}

}

# Java Functional Interfaces

BY CHAITANYA SINGH | FILED UNDER: [JAVA 8 FEATURES](https://beginnersbook.com/category/java-8-features/)

An interface with **only single abstract method** is called functional interface. You can either use the predefined functional interface provided by Java or create your own functional interface and use it. You can check the predefined functional interfaces here: [predefined functional interfaces](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) they all have only one abstract method. That is the reason,they are also known as Single Abstract Method interfaces (SAM Interfaces).

To use [lambda expression in Java](https://beginnersbook.com/2017/10/java-lambda-expressions-tutorial-with-examples/), you need to either create your own functional interface or use the pre defined functional interface provided by Java. While creating your own functional interface, mark it with **@FunctionalInterface** annotation, this annotation is introduced in Java 8. Although its optional, you should use it so that you get a compilation error if the interface you marked with this annotation is not following the rules of functional interfaces.

## What are the rules of defining a functional interface?

The functional interface should have **Only one** abstract method. Along with the one abstract method, they can have any number of default and static methods.

## Example 1: Creating your own functional interface

@FunctionalInterface

interface MyFunctionalInterface {

public int addMethod(int a, int b);

}

public class BeginnersBookClass {

public static void main(String args[]) {

// lambda expression

MyFunctionalInterface sum = (a, b) -> a + b;

System.out.println("Result: "+sum.addMethod(12, 100));

}

}

Output:

Result: 112

## Example 2: Using predefined functional interface

import java.util.function.IntBinaryOperator;

public class BeginnersBookClass {

public static void main(String args[]) {

// lambda expression

IntBinaryOperator sum = (a, b) -> a + b;

System.out.println("Result: " + sum.applyAsInt(12, 100));

}

}

Output:

Result: 112

## Functional interface example: using anonymous inner class vs using lambda expression

**We have been using functional interfaces even prior to java8**, they were used by creating anonymous inner classes using these interfaces. You must have seen functional interfaces such as Runnable, ActionListener, Comparator etc. They all have single abstract method. Lets see an example of ActionListener to see how it was used with Anonymous inner class and how it can be implemented using lambda expression.  
**ActionListener Example: Before Java 8: Using anonymous inner class**

import javax.swing.\*;

import java.awt.\*;

import java.awt.event.\*;

class Example extends JFrame

{

JButton button;

public Example()

{

setTitle("Button Action Example without Lambda Expression");

setSize(400,300);

setVisible(true);

setLayout(new FlowLayout());

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

button = new JButton("Button");

button.setBounds(100,100,90,40);

button.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent e){

System.out.println("You clicked the button.");

}

});

add(button);

}

public static void main(String args[])

{

new Example();

}

}

**ActionListener Example: Lambda Expression**

import javax.swing.\*;

import java.awt.\*;

class Example extends JFrame

{

JButton button;

public Example()

{

setTitle("Button Action Example using Lambda Expression");

setSize(400,300);

setVisible(true);

setLayout(new FlowLayout());

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

button = new JButton("Button");

button.setBounds(100,100,90,40);

//Lambda expression

button.addActionListener(e->

System.out.println("You clicked the button."));

add(button);

}

public static void main(String args[])

{

new Example();

}

}

# Method References in Java 8

BY CHAITANYA SINGH | FILED UNDER: [JAVA 8 FEATURES](https://beginnersbook.com/category/java-8-features/)

In the previous tutorial we learned [lambda expressions in Java 8](https://beginnersbook.com/2017/10/java-lambda-expressions-tutorial-with-examples/). Here we will discuss another new feature of java 8, **method reference**. Method reference is a shorthand notation of a lambda expression to call a method. For example:  
If your lambda expression is like this:

str -> System.out.println(str)

then you can replace it with a method reference like this:

System.out::println

The :: operator is used in method reference to separate the class or object from the method name(we will learn this with the help of examples).

## Four types of method references

1. Method reference to an instance method of an object – object::instanceMethod  
2. Method reference to a static method of a class – Class::staticMethod  
3. Method reference to an instance method of an arbitrary object of a particular type – Class::instanceMethod  
4. Method reference to a constructor – Class::new

## 1. Method reference to an instance method of an object

@FunctionalInterface

interface MyInterface{

void display();

}

public class Example {

public void myMethod(){

System.out.println("Instance Method");

}

public static void main(String[] args) {

Example obj = new Example();

// Method reference using the object of the class

MyInterface ref = obj::myMethod;

// Calling the method of functional interface

ref.display();

}

}

Output:

Instance Method

## 2. Method reference to a static method of a class

import java.util.function.BiFunction;

class Multiplication{

public static int multiply(int a, int b){

return a\*b;

}

}

public class Example {

public static void main(String[] args) {

BiFunction<Integer, Integer, Integer> product = Multiplication::multiply;

int pr = product.apply(11, 5);

System.out.println("Product of given number is: "+pr);

}

}

Output:

Product of given number is: 55

## 3. Method reference to an instance method of an arbitrary object of a particular type

import java.util.Arrays;

public class Example {

public static void main(String[] args) {

String[] stringArray = { "Steve", "Rick", "Aditya", "Negan", "Lucy", "Sansa", "Jon"};

/\* Method reference to an instance method of an arbitrary

\* object of a particular type

\*/

Arrays.sort(stringArray, String::compareToIgnoreCase);

for(String str: stringArray){

System.out.println(str);

}

}

}

Output:

Aditya

Jon

Lucy

Negan

Rick

Sansa

Steve

## 4. Method reference to a constructor

@FunctionalInterface

interface MyInterface{

Hello display(String say);

}

class Hello{

public Hello(String say){

System.out.print(say);

}

}

public class Example {

public static void main(String[] args) {

//Method reference to a constructor

MyInterface ref = Hello::new;

ref.display("Hello World!");

}

}

Output:

Hello World!

# Java 8 Interface Changes – default method and static method

BY CHAITANYA SINGH | FILED UNDER: [JAVA 8 FEATURES](https://beginnersbook.com/category/java-8-features/)

Prior to java 8, [interface in java](https://beginnersbook.com/2013/05/java-interface/) can only have abstract methods. All the methods of interfaces are public & abstract by default. Java 8 allows the interfaces to have default and static methods. The reason we have default methods in interfaces is to allow the developers to add new methods to the interfaces without affecting the classes that implements these interfaces.

## Why default method?

For example, if several classes such as A, B, C and D implements an interface XYZInterface then if we add a new method to the XYZInterface, we have to change the code in all the classes(A, B, C and D) that implements this interface. In this example we have only four classes that implements the interface which we want to change but imagine if there are hundreds of classes implementing an interface then it would be almost impossible to change the code in all those classes. This is why in java 8, we have a new concept “default methods”. These methods can be added to any existing interface and we do not need to implement these methods in the implementation classes mandatorily, thus we can add these default methods to existing interfaces without breaking the code.

We can say that concept of default method is introduced in java 8 to add the new methods in the existing interfaces in such a way so that they are backward compatible. Backward compatibility is adding new features without breaking the old code.

**Static methods** in interfaces are similar to the default methods except that we cannot override these methods in the classes that implements these interfaces.

## Java 8 Example: Default method in Interface

The method newMethod() in MyInterface is a default method, which means we need not to implement this method in the implementation class Example. This way we can add the default methods to existing interfaces without bothering about the classes that implements these interfaces.

interface MyInterface{

/\* This is a default method so we need not

\* to implement this method in the implementation

\* classes

\*/

default void newMethod(){

System.out.println("Newly added default method");

}

/\* Already existing public and abstract method

\* We must need to implement this method in

\* implementation classes.

\*/

void existingMethod(String str);

}

public class Example implements MyInterface{

// implementing abstract method

public void existingMethod(String str){

System.out.println("String is: "+str);

}

public static void main(String[] args) {

Example obj = new Example();

//calling the default method of interface

obj.newMethod();

//calling the abstract method of interface

obj.existingMethod("Java 8 is easy to learn");

}

}

Output:

Newly added default method

String is: Java 8 is easy to learn

## Java 8 Example: Static method in Interface

As mentioned above, the static methods in interface are similar to default method so we need not to implement them in the implementation classes. We can safely add them to the existing interfaces without changing the code in the implementation classes. Since these methods are static, we cannot override them in the implementation classes.

interface MyInterface{

/\* This is a default method so we need not

\* to implement this method in the implementation

\* classes

\*/

default void newMethod(){

System.out.println("Newly added default method");

}

/\* This is a static method. Static method in interface is

\* similar to default method except that we cannot override

\* them in the implementation classes.

\* Similar to default methods, we need to implement these methods

\* in implementation classes so we can safely add them to the

\* existing interfaces.

\*/

static void anotherNewMethod(){

System.out.println("Newly added static method");

}

/\* Already existing public and abstract method

\* We must need to implement this method in

\* implementation classes.

\*/

void existingMethod(String str);

}

public class Example implements MyInterface{

// implementing abstract method

public void existingMethod(String str){

System.out.println("String is: "+str);

}

public static void main(String[] args) {

Example obj = new Example();

//calling the default method of interface

obj.newMethod();

//calling the static method of interface

MyInterface.anotherNewMethod();

//calling the abstract method of interface

obj.existingMethod("Java 8 is easy to learn");

}

}

Output:

Newly added default method

Newly added static method

String is: Java 8 is easy to learn

## Java 8 – Abstract classes vs interfaces

With the introduction of default methods in interfaces, it seems that the [abstract classes](https://beginnersbook.com/2013/05/java-abstract-class-method/) are same as interface in java 8. However this is not entirely true, even though we can now have concrete methods(methods with body) in interfaces just like abstract class, this doesn’t mean that they are same. There are still few differences between them, one of them is that abstract class can have constructor while in interfaces we can’t have constructors.

The purpose of interface is to provide full abstraction, while the purpose of abstract class is to provide partial abstraction. This still holds true. The interface is like a blueprint for your class, with the introduction of default methods you can simply say that we can add additional features in the interfaces without affecting the end user classes.

## Default Method and Multiple Inheritance

The [multiple inheritance](https://beginnersbook.com/2013/05/java-multiple-inheritance/) problem can occur, when we have two interfaces with the default methods of same signature.

## Java 8 – forEach to iterate a Map

import java.util.Map;

import java.util.HashMap;

public class Example {

   public static void main(String[] args) {

    Map<Integer, String> hmap = new HashMap<Integer, String>();

   hmap.put(1, "Monkey");

   hmap.put(2, "Dog");

   hmap.put(3, "Cat");

 hmap.put(4, "Lion");

 hmap.put(5, "Tiger");

 hmap.put(6, "Bear");

/\* forEach to iterate and display each key and value pair

    \* of HashMap.

\*/

 hmap.forEach((key,value)->System.out.println(key+" - "+value));

/\* forEach to iterate a Map and display the value of a particular

  \* key

\*/

   hmap.forEach((key,value)->{

    if(key == 4){

    System.out.println("Value associated with key 4 is: "+value);

    }

  });

   /\* forEach to iterate a Map and display the key associated with a

    \* particular value

\*/

   hmap.forEach((key,value)->{

    if("Cat".equals(value)){

    System.out.println("Key associated with Value Cat is: "+key);

    }

   });

  }

}

## Java 8 – forEach to iterate a List

In this example, we are iterating an [ArrayList](https://beginnersbook.com/2013/12/java-arraylist/) using forEach() method. Inside forEach we are using a [lambda expression](https://beginnersbook.com/2017/10/java-lambda-expressions-tutorial-with-examples/) to print each element of the list.

import java.util.List;

import java.util.ArrayList;

public class Example {

public static void main(String[] args) {

List<String> fruits = new ArrayList<String>();

fruits.add("Apple");

fruits.add("Orange");

fruits.add("Banana");

fruits.add("Pear");

fruits.add("Mango");

//lambda expression in forEach Method

fruits.forEach(str->System.out.println(str));

}

}

Java Annotations

Java **Annotation** is a tag that represents the *metadata* i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.

Annotations in Java are used to provide additional information, so it is an alternative option for XML and Java marker interfaces.

First, we will learn some built-in annotations then we will move on creating and using custom annotations.

Built-In Java Annotations

There are several built-in annotations in Java. Some annotations are applied to Java code and some to other annotations.

Built-In Java Annotations used in Java code

* @Override
* @SuppressWarnings
* @Deprecated

Built-In Java Annotations used in other annotations

* @Target
* @Retention
* @Inherited
* @Documented

Understanding Built-In Annotations

Let's understand the built-in annotations first.

@Override

@Override annotation assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.

Sometimes, we does the silly mistake such as spelling mistakes etc. So, it is better to mark @Override annotation that provides assurity that method is overridden.

1. **class** Animal{
2. **void** eatSomething(){System.out.println("eating something");}
3. }
5. **class** Dog **extends** Animal{
6. @Override
7. **void** eatsomething(){System.out.println("eating foods");}//should be eatSomething
8. }
10. **class** TestAnnotation1{
11. **public** **static** **void** main(String args[]){
12. Animal a=**new** Dog();
13. a.eatSomething();
14. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestAnnotation1)

Output:Comple Time Error

@SuppressWarnings

@SuppressWarnings annotation: is used to suppress warnings issued by the compiler.

1. **import** java.util.\*;
2. **class** TestAnnotation2{
3. @SuppressWarnings("unchecked")
4. **public** **static** **void** main(String args[]){
5. ArrayList list=**new** ArrayList();
6. list.add("sonoo");
7. list.add("vimal");
8. list.add("ratan");
10. **for**(Object obj:list)
11. System.out.println(obj);
13. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestAnnotation2)

Now no warning at compile time.

If you remove the @SuppressWarnings("unchecked") annotation, it will show warning at compile time because we are using non-generic collection.

@Deprecated

@Deprecated annoation marks that this method is deprecated so compiler prints warning. It informs user that it may be removed in the future versions. So, it is better not to use such methods.

1. **class** A{
2. **void** m(){System.out.println("hello m");}
4. @Deprecated
5. **void** n(){System.out.println("hello n");}
6. }
8. **class** TestAnnotation3{
9. **public** **static** **void** main(String args[]){
11. A a=**new** A();
12. a.n();
13. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestAnnotation3)

At Compile Time:

Note: Test.java uses or overrides a deprecated API.

Note: Recompile with -Xlint:deprecation for details.

At Runtime:

hello n

Java Custom Annotations

**Java Custom annotations** or Java User-defined annotations are easy to create and use. The *@interface* element is used to declare an annotation. For example:

1. **@interface** MyAnnotation{}

Here, MyAnnotation is the custom annotation name.

Points to remember for java custom annotation signature

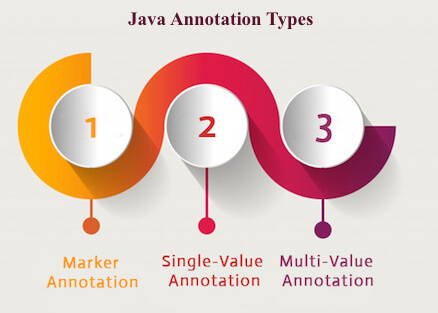
There are few points that should be remembered by the programmer.

1. Method should not have any throws clauses
2. Method should return one of the following: primitive data types, String, Class, enum or array of these data types.
3. Method should not have any parameter.
4. We should attach @ just before interface keyword to define annotation.
5. It may assign a default value to the method.

Types of Annotation

There are three types of annotations.

1. Marker Annotation
2. Single-Value Annotation
3. Multi-Value Annotation



1) Marker Annotation

An annotation that has no method, is called marker annotation. For example:

1. **@interface** MyAnnotation{}

The @Override and @Deprecated are marker annotations.

2) Single-Value Annotation

An annotation that has one method, is called single-value annotation. For example:

1. **@interface** MyAnnotation{
2. **int** value();
3. }

We can provide the default value also. For example:

1. **@interface** MyAnnotation{
2. **int** value() **default** 0;
3. }

How to apply Single-Value Annotation

Let's see the code to apply the single value annotation.

1. @MyAnnotation(value=10)

The value can be anything.

3) Multi-Value Annotation

An annotation that has more than one method, is called Multi-Value annotation. For example:

1. **@interface** MyAnnotation{
2. **int** value1();
3. String value2();
4. String value3();
5. }
6. }

We can provide the default value also. For example:

1. **@interface** MyAnnotation{
2. **int** value1() **default** 1;
3. String value2() **default** "";
4. String value3() **default** "xyz";
5. }

How to apply Multi-Value Annotation

Let's see the code to apply the multi-value annotation.

1. @MyAnnotation(value1=10,value2="Arun Kumar",value3="Ghaziabad")

Built-in Annotations used in custom annotations in java

* @Target
* @Retention
* @Inherited
* @Documented

@Target

**@Target** tag is used to specify at which type, the annotation is used.

The java.lang.annotation.**ElementType** enum declares many constants to specify the type of element where annotation is to be applied such as TYPE, METHOD, FIELD etc. Let's see the constants of ElementType enum:

|  |  |
| --- | --- |
| **Element Types** | **Where the annotation can be applied** |
| TYPE | class, interface or enumeration |
| FIELD | fields |
| METHOD | methods |
| CONSTRUCTOR | constructors |
| LOCAL\_VARIABLE | local variables |
| ANNOTATION\_TYPE | annotation type |
| PARAMETER | parameter |

Example to specify annoation for a class

1. @Target(ElementType.TYPE)
2. **@interface** MyAnnotation{
3. **int** value1();
4. String value2();
5. }

Example to specify annotation for a class, methods or fields

1. @Target({ElementType.TYPE, ElementType.FIELD, ElementType.METHOD})
2. **@interface** MyAnnotation{
3. **int** value1();
4. String value2();
5. }

@Retention

**@Retention** annotation is used to specify to what level annotation will be available.

|  |  |
| --- | --- |
| **RetentionPolicy** | **Availability** |
| RetentionPolicy.SOURCE | refers to the source code, discarded during compilation. It will not be available in the compiled class. |
| RetentionPolicy.CLASS | refers to the .class file, available to java compiler but not to JVM . It is included in the class file. |
| RetentionPolicy.RUNTIME | refers to the runtime, available to java compiler and JVM . |

Example to specify the RetentionPolicy

1. @Retention(RetentionPolicy.RUNTIME)
2. @Target(ElementType.TYPE)
3. **@interface** MyAnnotation{
4. **int** value1();
5. String value2();
6. }

Example of custom annotation: creating, applying and accessing annotation

Let's see the simple example of creating, applying and accessing annotation.

*File: Test.java*

1. //Creating annotation
2. **import** java.lang.annotation.\*;
3. **import** java.lang.reflect.\*;
5. @Retention(RetentionPolicy.RUNTIME)
6. @Target(ElementType.METHOD)
7. **@interface** MyAnnotation{
8. **int** value();
9. }
11. //Applying annotation
12. **class** Hello{
13. @MyAnnotation(value=10)
14. **public** **void** sayHello(){System.out.println("hello annotation");}
15. }
17. //Accessing annotation
18. **class** TestCustomAnnotation1{
19. **public** **static** **void** main(String args[])**throws** Exception{
21. Hello h=**new** Hello();
22. Method m=h.getClass().getMethod("sayHello");
24. MyAnnotation manno=m.getAnnotation(MyAnnotation.**class**);
25. System.out.println("value is: "+manno.value());
26. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCustomAnnotation1)

Output:value is: 10

[download this example](https://static.javatpoint.com/src/newjdk/annotation.zip)

How built-in annotaions are used in real scenario?

In real scenario, java programmer only need to apply annotation. He/She doesn't need to create and access annotation. Creating and Accessing annotation is performed by the implementation provider. On behalf of the annotation, java compiler or JVM performs some additional operations.

@Inherited

By default, annotations are not inherited to subclasses. The @Inherited annotation marks the annotation to be inherited to subclasses.

1. @Inherited
2. **@interface** ForEveryone { }//Now it will be available to subclass also
4. **@interface** ForEveryone { }
5. **class** Superclass{}
7. **class** Subclass **extends** Superclass{}

@Documented

The @Documented Marks the annotation for inclusion in the documentation.

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.