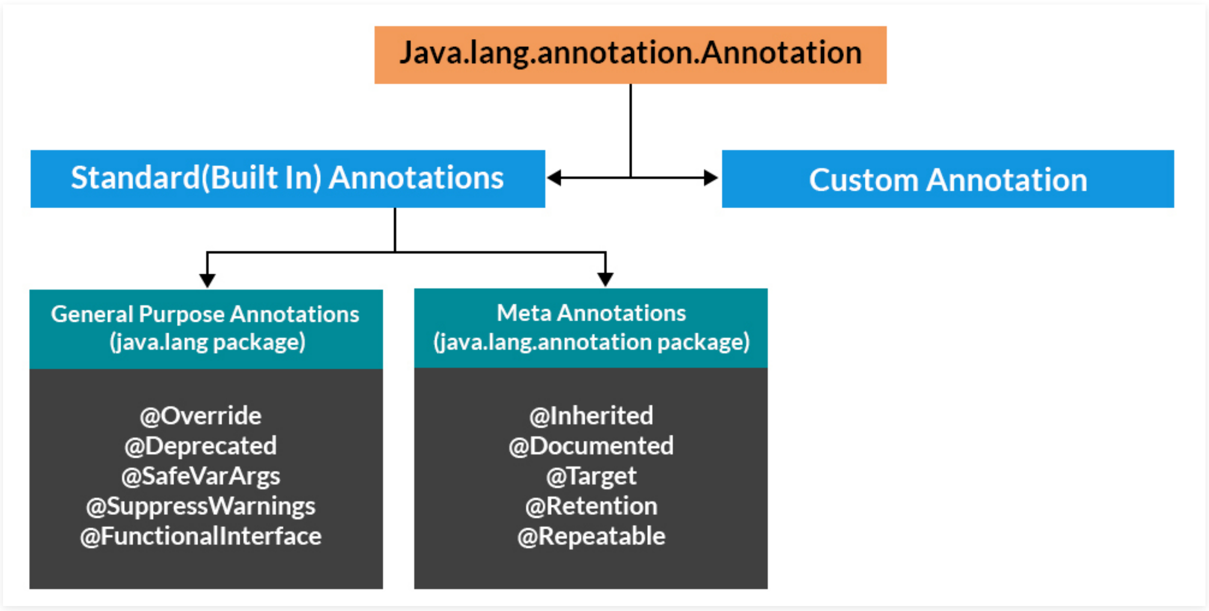
Annotations in Java

Difficulty Level : Hard Last Updated : 10 Nov, 2021

Annotations are used to provide supplemental information about a program.

- Annotations start with '**@**'.
- Annotations do not change the 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 the compiler. See below code for example.

Hierarchy of Annotations in Java


Java.lang.annotation.Annotation
Standard(Built In) Annotations
Custom Annotation
General Purpose Annotations (java.lang package)
@Override
@Deprecated
@SafeVarArgs
@SuppressWarnings
@FunctionalInterface
Meta Annotations (java.lang.annotation package)
@Inherited
@Documented
@Target
@Retention
@Repeatable

Implementation:

> ***Note:*** *This program throws compiler error because we have mentioned override, but not ov erridden, we have overloaded display.*

Example:

```
// Java Program to Demonstrate that Annotations
// are Not Barely Comments

// Class 1
class Base {

    // Method
    public void display()
    {
        System.out.println("Base display()");
    }
}

// Class 2
// Main class
class Derived extends Base {

    // Overriding method as already up in above class
    @Override public void display(int x)
    {
        // Print statement when this method is called
        System.out.println("Derived display(int )");
    }

    // Method 2
    // Main driver method
    public static void main(String args[])
    {
        // Creating object of this class inside main()
        Derived obj = new Derived();

        // Calling display() method inside main()
        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 broadly 5 categories of annotations as listed:

1. Marker Annotations
2. Single value Annotations

3. Full Annotations
4. Type Annotations
5. Repeating Annotations

Let us discuss and we will be appending code where ever required if so.

Category 1: Marker Annotations

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

Example

```
@TestAnnotation()
```

Category 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 a value.

Example

```
@TestAnnotation(“testing”);
```

Category 3: Full Annotations

These annotations consist of multiple data members, names, values, pairs.

Example

```
@TestAnnotation(owner=”Rahul”, value=”Class Geeks”)
```

Category 4: Type Annotations

These annotations can be applied to any place where a type is being used. For example, we can annotate the return type of a method. These are declared annotated with ***@Target** annotation*.

Example

```java
// Java Program to Demonstrate Type Annotation

// Importing required classes
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

// Using target annotation to annotate a type
@Target(ElementType.TYPE_USE)

// Declaring a simple type annotation
@interface TypeAnnoDemo{}

// Main class
public class GFG {

    // Main driver method
    public static void main(String[] args) {

        // Annotating the type of a string
        @TypeAnnoDemo String string = "I am annotated with a type annotation";
        System.out.println(string);
        abc();
    }

    // Annotating return type of a function
    static @TypeAnnoDemo int abc() {

        System.out.println("This function's  return type is annotated");

        return 0;
    }
}
```

Output:

```
I am annotated with a type annotation
This function's  return type is annotated
```

Category 5: Repeating Annotations

These are the annotations that can be applied to a single item more than once. For an annotation to be repeatable it must be annotated with the **@Repeatable** annotation, which is defined in the **java.lang.annotation** package. Its value field specifies the **container type** for the repeatable annotation. **The container is specified as an annotation whose value field is an array of the repeatable annotation type.** Hence, to create a repeatable annotation, firstly the container annotation is created, and then the annotation type is specified as an argument to the @Repeatable annotation.

Example:

```java
// Java Program to Demonstrate a Repeatable Annotation

// Importing required classes
import java.lang.annotation.Annotation;
import java.lang.annotation.Repeatable;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.reflect.Method;

// Make Words annotation repeatable
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(MyRepeatedAnnos.class)
@interface Words
{
    String word() default "Hello";
    int value() default 0;
}

// Create container annotation
@Retention(RetentionPolicy.RUNTIME)
@interface MyRepeatedAnnos
{
    Words[] value();
}
public class Main {

    // Repeat Words on newMethod
    @Words(word = "First", value = 1)
    @Words(word = "Second", value = 2)
    public static void newMethod()
    {
        Main obj = new Main();

        try {
            Class<?> c = obj.getClass();

            // Obtain the annotation for newMethod
            Method m = c.getMethod("newMethod");

            // Display the repeated annotation
            Annotation anno
                = m.getAnnotation(MyRepeatedAnnos.class);
            System.out.println(anno);
        }
        catch (NoSuchMethodException e) {
            System.out.println(e);
        }
    }
    public static void main(String[] args) { newMethod(); }
}
```

Output:

```
@MyRepeatedAnnos(value={@Words(value=1, word="First"), @Words(value=2, word="
```

Predefined/ Standard Annotations

Java popularly defines seven built-in annotations as we have seen up in the hierarchy diagram.

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

Annotation 1: @Deprecated

- It is a marker annotation. It indicates that a declaration is obsolete and has been replaced by a newer form.
- The Javadoc @deprecated 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.

Example:

```
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()
```

Annotation 2: @Override

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 for example). It is used to ensure that a superclass method is actually overridden, and not simply overloaded.

Example

```
// Java Program to Illustrate Override Annotation

// Class 1
class Base
{
     public void Display()
     {
         System.out.println("Base display()");
     }

     public static void main(String args[])
     {
         Base t1 = new Derived();
         t1.Display();
     }
}

// Class 2
// Extending above class
class Derived extends Base
{
     @Override
     public void Display()
     {
         System.out.println("Derived display()");
     }
}
```

Output

```
Derived display()
```

Annotation 3: @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 **deprecated** and **unchecked**. Any unchecked warning is generated when a legacy code interfaces with a code that uses generics.

Example:

```
// Java Program to illustrate SuppressWarnings Annotation

// Class 1
```

```
class DeprecatedTest
{
    @Deprecated
    public void Display()
    {
        System.out.println("Deprecatedtest display()");
    }
}

// Class 2
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()
```

Annotation 4: @Documented

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

Annotation 5: @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

Target Constant	Annotations Can Be Applied To
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.

Annotation 6: @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.

Annotation 7: User-defined (Custom)

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

Syntax: Declaration

```
[Access Specifier] @interface<AnnotationName>
{
   DataType <Method Name>() [default value];
}
```

Do keep these certain points as rules for custom annotations before implementing user-defined annotations.

1. **AnnotationName** is an identifier.
2. The parameter should not be associated with method declarations and **throws** clause should not be used with method declaration.
3. Parameters will not have a null value but can have a default value.
4. ***default value*** is optional.
5. The return type of method should be either primitive, enum, string, class name, or array of primitive, enum, string, or class name type.

Example:

```java
// Java Program to Demonstrate User-defined Annotations

package source;

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
```

This article is contributed by **Rahul Agrawal.** If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.