An **annotation is a kind of meta data** which can be applied at various elements of java source code. We can annotate classes, methods, variables, parameters and packages OR almost everything. Annotations applied on java source code is compiled into bytecode with other class members, and using reflection programmer can query this meta data information to decide the appropriate action to perform in any particular context

**Purposes**

* **Compiler instructions**
* **Build-time instructions**
* **Runtime instructions**

Java has 5 built-in annotations. Java annotations can be used at build-time. The build process includes generating source code, compiling the source, generating XML files (e.g. deployment descriptors), packaging the compiled code and files into a JAR file etc. Build tools(Ant or Maven) may scan your Java code for specific annotations and generate source code or other files based on these annotations.

Normally, Java annotations are not present in your Java code after compilation. It is possible, however, to define your own annotations that are available at runtime. These annotations can then be accessed via [**Java Reflection**](http://tutorials.jenkov.com/java-reflection/index.html), and used to give instructions to your program, or some third party API.

**Annotation Elements**

A Java annotation can have elements for which you can set values. An element (example- tableName) is like an attribute or parameter

@Entity(tableName = "vehicles")

No need to put element name in case of single element, value only is enough

**Built-in Java Annotations**

**@Deprecated**

**@Override**

**@SuppressWarnings**

It suppresses warnings for a given method or data member. For instance, if a method calls a deprecated method, or makes an insecure type cast, the compiler may generate a warning.

**@SafeVarargs**

Introduced in java 7, this ***annotation ensures that the body of the annotated method or constructor does not perform potentially unsafe operations*** on its varargs parameter. Applying this annotation to a method or constructor suppresses unchecked warnings about a non-reifiable variable arity (vararg) type and suppresses unchecked warnings about parameterized array creation at call sites.

|  |
| --- |
| public static <T> List<T> list( final T... items )  {      return Arrays.asList( items );  } |

### @FunctionalInterface

This annotation is used to ***mark an interface as functional interface*** which are introduced in java 8. To read more about [**functional interfaces**](http://howtodoinjava.com/java-8/functional-interface-tutorial/) please follow the linked post.

|  |
| --- |
| @FunctionalInterface  public interface MyFirstFunctionalInterface {      public void doSomeWork();  } |

# Creating Custom Annotations

* use the keyword “**@interface**
* Each method declaration defines an element of the annotation type.
* Method declarations must not have any parameters or a throws clause.
* Return types are restricted to primitives, String, Class, enums, annotations, and arrays of the preceding types.
* Methods can have default values.

@interface MyAnnotation {

String value(); String name();

int age(); String[] newNames(); }

This example defines an annotation called MyAnnotation which has four elements. Notice that each element is defined similarly to a method definition in an interface. It has a data type and a name. You can use all primitive or arrays data types as element data types. You cannot use complex objects as data type.

@MyAnnotation(

value="123",

name="Jakob",

age=37,

newNames={"Jenkov", "Peterson"}

)

public class MyClass { }

Specify default values for an element and it makes the element optional.

@interface MyAnnotation {

String value() default "";

String name(); int age(); String[] newNames(); }

The value element can now be left out when using the annotation

@MyAnnotation(

name="Jakob", age=37, newNames={"Jenkov", "Peterson"}

)

public class MyClass { }

# Annotations Applied To Other Annotations

5 annotations are used inside other annotations to hint compiler that how new annotation should be treated by JVM.

**@Retention: specifies how the marked annotation is stored in java runtime.** Whether it is limited to source code only, embedded into the generated class file, or available at runtime through reflection as well.

import java.lang.annotation.Retention; import java.lang.annotation.RetentionPolicy;

[//@Retention(RetentionPolicy.CLASS)](mailto://@Retention(RetentionPolicy.CLASS))

@Retention(RetentionPolicy.RUNTIME) [//@Retention(RetentionPolicy.SOURCE)](mailto://@Retention(RetentionPolicy.SOURCE))

public @interface MyCustomAnnotation { //some code}

**RetentionPolicy.CLASS (**default) means that the annotation is stored in the .class file, but not available at runtime.

**RetentionPolicy.SOURCE** Only available in the source code, and not in the .class or not a runtime.

**RetentionPolicy.CLASS** It will be available to the JVM through runtime.

**Use:** If you create your own annotations for use with build tools that scan the code, you can use this . That way the .class files are not polluted unnecessarily.

**@Target** Use @Target annotation to ***restrict the usage of new annotation on certain java elements*** such as class, interface or methods. After specifying the targets, you will be able to use the new annotation on given elements only. Below annotation that can only be used to annotate methods.

import java.lang.annotation.ElementType; import java.lang.annotation.Target;

@Target({ElementType.METHOD})

public @interface MyAnnotation { String value(); }

The ElementType class contains the following possible targets:

* ElementType.ANNOTATION\_TYPE
* ElementType.CONSTRUCTOR
* ElementType.FIELD
* ElementType.LOCAL\_VARIABLE
* ElementType.METHOD
* ElementType.PACKAGE
* ElementType.PARAMETER
* ElementType.TYPE

The **ANNOTATION\_TYPE** target means Java annotation definitions. Thus, the annotation can only be used to annotate other annotations. Like the @Target and @Retention annotations.

The TYPE target means any type. A type is either a class, interface, enum or annotation.

**@Inherited**

The @Inherited annotation signals that a custom Java annotation used in a class should be inherited by subclasses inheriting from that class. Here is an @Inherited Java annotation example:

java.lang.annotation.Inherited

@Inherited

public @interface MyAnnotation { }

@MyAnnotation

public class MySuperClass { ... }

public class MySubClass extends MySuperClass { ... }

In this example the class MySubClass inherits the annotation @MyAnnotation because MySubClass inherits from MySuperClass, and MySuperClass has a @MyAnnotation annotation.

**@Documented:** This annotation ***indicates that new annotation should be included into java documents*** generated by java document generator tools.

|  |
| --- |
| import java.lang.annotation.Documented;  @Documented  public @interface MyCustomAnnotation {   //Some other code } |

### @Repeatable:

By default, an annotation is applied on a java element only once. But, by any requirement, you have to ***apply a annotation more than once***, then use @Repeatable annotation on your new annotation.

@Repeatable has been added in latest java 8 release.

|  |
| --- |
| @Repeatable(Schedules.class)  public @interface Schedule { ... } |

Now use above annotation as below:

|  |
| --- |
| @Schedule(dayOfMonth="last")  @Schedule(dayOfWeek="Fri", hour="23")  public void doPeriodicCleanup() { ... } |

**Processing Annotations Using Reflection**

The main reason we are using annotations are because they are metadata. So we should be able to fetch this metadata to utilize the annotation information when we need it.

In java, you have to use **reflection API** to access annotations on any type (i.e. class or interface) or methods.

|  |
| --- |
| package test.core.annotations;   import java.lang.annotation.Annotation;  import java.lang.reflect.AnnotatedElement;  import java.lang.reflect.Method;   public class ProcessAnnotationExample  {     public static void main(String[] args) throws NoSuchMethodException, SecurityException     {        new DemoClass();        Class<DemoClass> demoClassObj = DemoClass.class;        readAnnotationOn(demoClassObj);        Method method = demoClassObj.getMethod("getString", new Class[]{});        readAnnotationOn(method);     }       static void readAnnotationOn(AnnotatedElement element)     {        try        {           System.out.println("\n Finding annotations on " + element.getClass().getName());           Annotation[] annotations = element.getAnnotations();           for (Annotation annotation : annotations)           {              if (annotation instanceof JavaFileInfo)              {                 JavaFileInfo fileInfo = (JavaFileInfo) annotation;                 System.out.println("Author :" + fileInfo.author());                 System.out.println("Version :" + fileInfo.version());              }           }        } catch (Exception e)        {           e.printStackTrace();        }     }  }  Output:  Finding annotations on java.lang.Class  Author :unknown  Version :0.0    Finding annotations on java.lang.reflect.Method  Author :Lokesh  Version :1.0 |

## Summary

Before the advent of Annotations, you don’t need to define your sourcecode metadata outside in some properties file. Now, they can directly define this meta-data information in the source code itself. If used this feature wisely (as it is used in latest java frameworks like [**Spring**](http://howtodoinjava.com/) and [**Struts**](http://howtodoinjava.com/struts-2/struts-2-hello-world-with-annotations/)), benefits are countless.

Let’s summarize our learning from this post in some bullet points:

1. Annotations are metadata which can be applied on either annotations OR other java element in java sourcecode.
2. Annotations do not directly affect program semantics, but they do affect the way programs are treated by tools and libraries, which can in turn affect the semantics of the running program.
3. Annotations can be read from source files, class files, or reflectively at run time.
4. There are 10 in-built annotations as of today. 5 of them are meant to be applied on custom annotations and other 5 are meant to be applied on java source code elements. Read respective sections for more details.
5. Because annotation types are compiled and stored in byte code files just like classes, the annotations returned by these methods can be queried just like any regular Java object. You saw an example above.

That’s all for this lovely and very powerful feature i.e. **Annotation**. Let me know of your thoughts/queries in comments section.

**Happy Learning !!**