Annotations

In Java, an annotation is a form of syntactic metadata that can be added to Java source code. Classes, methods, variables, parameters, and packages may be annotated. Unlike Javadoc tags, Java annotations can be reflective in that they can be embedded in class files generated by the compiler and may be retained by the Java VM to be made retrievable at run-time. It's used for providing information about the code which is not part of the program itself but can be used by Java compiler and JVM. Annotations have a number of uses, among them:

- ★ Information for the compiler
- ★ Compile-time and deployment-time processing
- ★ Runtime processing.

Common Built-in Annotations

1. `@Override`
2. `@SuppressWarnings`
3. `@Deprecated`
4. `@FunctionalInterface`

`@Override`

The `@Override` annotation in Java is used to indicate that a method is intended to override a method in a superclass. If a method marked with `@Override` fails to correctly override a method in one of its superclasses, the compiler generates an error. Using this annotation can help catch errors at compile time rather than at runtime.

Suppose we have a superclass in Java

```
public class TimePrinter {
    public TimePrinter(){}

    public void printTime() {
        System.out.println("The time is: 10 AM");
    }
}
```

Now a class `myTime` wants to extend this class and override the method here.

```
public class myTime extends TimePrinter{
    public myTime(){}

    public void printTine() {
        System.out.println("The time is: 9 PM");
    }
}
```

Now if we write the main function like this,

```
public static void main(String[] args) {
        myTime t = new myTime();
        t.printTime();
    }
```

It will show output,

```
The time is: 10 AM
```

But what happened? We have overridden the method. Look closely. There is a typing mistake. If we had used @override as annotation then it would throw an error because though we are saying we are overriding actually for typing mistake. We are not.

```
@Override
public void printTine() {
    System.out.println("The time is: 9 PM");
}

! Error => not overriding
```

It will only work if I make my spelling correct

```
@Override
public void printTime() {
    System.out.println("The time is: 9 PM");
}
```

Now the output is,

```
The time is: 9 PM
```

@Deprecated

The `@Deprecated` annotation in Java is used to indicate that the marked element (class, method, etc.) is deprecated and should no longer be used. The compiler generates a warning whenever a program uses a method, class, or field with the `@Deprecated` annotation. It informs the user that it might be removed in future versions and that it's better to stop using it.

```
@Deprecated
public int add(int a, int b) {
    return a + b;
}
```

Now if I want to call the method from any other place it will suggest not to use this because this method is now deprecated.

Now I want to use this method at any cost and I do not want an error. So the way is using another annotation.

@SuppressWarnings("deprecation")

The `@SuppressWarnings` annotation in Java is used to suppress specified compiler warnings in the annotated element and all program elements inside that element. The `"deprecation"` argument indicates that the deprecation warning should be suppressed. This is typically used when a deprecated method or class is still in use intentionally, and the developer wants to avoid a compile-time warning.

```
@SuppressWarnings("deprecation")
public void calculator(int a, int b){
    Adder ad = new Adder();
    ad.add(a, b);
}
```

Now the code will not throw any warning while compiling deprecated methods.

The `@SuppressWarnings` annotation in Java is a powerful tool that allows developers to control the warnings issued by the compiler. While it's commonly used to suppress deprecation warnings, it can also be used to suppress a variety of other warning types. Here are some of the most common ones:

1. **Unchecked**: This warning type is issued when the compiler encounters an operation that is unchecked, such as the use of raw types. By suppressing this warning, the compiler will not issue a warning when such operations are encountered.

```
@SuppressWarnings("unchecked")
public void uncheckedWarningExample() {
    List list = new ArrayList();
    list.add("test");
    List<String> stringList = list; // Compiler would normally warn here
}
```

2. **Cast**: This warning is issued when the compiler encounters a cast operation that it deems unsafe. Suppressing this warning will prevent the compiler from issuing a warning when such operations are encountered.

```
@SuppressWarnings("cast")
public void castWarningExample() {
    Object object = new String("test");
    String string = (String) object; // Compiler would normally warn here
}
```

3. **Divzero**: This warning is issued when the compiler encounters a division by zero operation. Suppressing this warning will prevent the compiler from issuing a warning when such operations are encountered.

```
@SuppressWarnings("divzero")
public void divZeroWarningExample() {
    int a = 10;
    int b = 0;
    int c = a / b; // Compiler would normally warn here
}
```

4. **Empty**: This warning is issued when the compiler encounters a statement with an empty body. Suppressing this warning will prevent the compiler from issuing a warning when such operations are encountered.

```
@SuppressWarnings("empty")
public void emptyWarningExample() {
    // Compiler would normally warn here
    if (true) {
    }
}
```

5. **Fallthrough**: This warning is issued when the compiler encounters a case in a switch statement that does not have a break, allowing execution to 'fall through' to the next case. Suppressing this warning will prevent the compiler from issuing a warning when such operations are encountered.

```java
@SuppressWarnings("fallthrough")
public void fallthroughWarningExample() {
    int num = 1;
    switch (num) {
        case 1:
            System.out.println("1");
            // Compiler would normally warn here
        case 2:
            System.out.println("2");
            break;
    }
}
```

6. **Hiding**: This warning is issued when a local variable in a method hides a field with the same name. Suppressing this warning will prevent the compiler from issuing a warning when such operations are encountered.

```java
public class HidingWarningExample {
    private int x;

    @SuppressWarnings("hiding")
    public void hidingWarningExample() {
        int x = 10; // Compiler would normally warn here
        System.out.println(x);
    }
}
```

7. **Unused**: This warning is issued when the compiler encounters a declaration or parameter that is never used. Suppressing this warning will prevent the compiler from issuing a warning when such operations are encountered.

```java
@SuppressWarnings("unused")
public void unusedWarningExample() {
    int x = 10; // Compiler would normally warn here
}
```

Remember while suppressing warnings can be useful in certain situations, it's generally best practice to address the issues causing the warnings if possible. Warnings are there for a reason - they often indicate potential problems or bad practices in your code.

Custom Annotations

We can create our very own annotations as we like. But before that, we have to know about 2 built-in annotations in JAVA.

The @Target Annotation

The `@Target` annotation is a **meta-annotation** in Java, which means it can be used on other annotations to provide metadata about them. It is used to specify the **types of elements** an annotation can be applied to. These elements can include:

- **TYPE**: Class, interface, or enumeration
- **FIELD**: Field, including enum constants
- **METHOD**: Method level annotation
- **PARAMETER**: Parameters
- **CONSTRUCTOR**: Constructors
- **LOCAL_VARIABLE**: Local variables
- **ANNOTATION_TYPE**: Annotation type declaration
- **PACKAGE**: Package declarations

Here's an example of how to use the `@Target` annotation when creating a custom annotation:

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

@Target(ElementType.METHOD)
public @interface CustomAnnotation {
    String value() default "";
}
```

In this example, the `@Target` annotation is used to specify that the `CustomAnnotation` can only be applied to methods. If you try to apply `CustomAnnotation` to a class, field, or any other element type not specified in the `@Target` annotation, the compiler will throw an error.

The `@Target` annotation is a powerful tool for creating **custom annotations** in Java, allowing developers to control how and where their annotations can be used. This helps to prevent misuse and enhances the readability and maintainability of the code.

Remember, the `@Target` annotation is not mandatory when creating a custom annotation. If it is not present, the custom annotation can be used on any element.

The @Retention Annotation in Java

The `@Retention` annotation is another **meta-annotation** in Java. It is used to indicate how long the annotation should be retained by the Java Virtual Machine (JVM). The `@Retention` annotation takes one argument, which is a value from the `java.lang.annotation.RetentionPolicy` enumeration. This enumeration has three values:

- **SOURCE**: The annotation will be retained only in the source code and is discarded during the compile time.
- **CLASS**: The annotation will be retained in the .class file, but the JVM does not retain it at runtime. This is the default retention policy if none is specified.
- **RUNTIME**: The annotation will be retained in the .class file, and the JVM retains it at runtime so it can be read reflectively.

Here's an example of how to use the `@Retention` annotation when creating a custom annotation:

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

@Retention(RetentionPolicy.RUNTIME)
public @interface CustomAnnotation {
    String value() default "";
}
```

In this example, the `@Retention` annotation is used to specify that the `CustomAnnotation` should be retained by the JVM at runtime. This means you can use reflection to read the `CustomAnnotation` annotations during the execution of your program.

The `@Retention` annotation is crucial when you want to create annotations that will be queried at runtime. Without it, the JVM does not retain the annotation data, making it impossible to use reflective techniques to read the annotation.

Reflection in Java

Reflection is a powerful feature in Java that allows an executing Java program to examine or "introspect" upon itself, and manipulate internal properties of the program. It is a part of `Java.lang` package and provides the ability to inspect and modify the runtime behavior of applications.

Reflection can be used for various purposes, but it's most commonly used for,

- **Inspecting classes, interfaces, fields, and methods at runtime.** For example, you can use reflection to determine the class of an object, the methods that the class declares, the fields that it has, etc.
- **Instantiating new objects, invoking methods, and getting/setting field values.** Reflection allows you to do things like creating an object of a type which you don't know until runtime, or calling a method of an object that was declared in a class you don't have access to at compile time.

Here's a simple example of using reflection to inspect a class:

```
import java.lang.reflect.Method;

public class ReflectionExample {
    public static void main(String[] args) {
        Class<?> c = String.class;
        Method[] methods = c.getDeclaredMethods();

        for (Method method : methods) {
            System.out.println(method.getName());
        }
    }
}
```

In this example, we're using reflection to get all declared methods in the `String` class and print their names. This is something that would not be possible without reflection, as these details are typically hidden away during normal usage of the class.

However, while powerful, reflection should be used sparingly. Overuse of reflection can lead to code that is hard to understand and maintain, and it can also lead to poor performance if not used carefully. It's also worth noting that reflection operations have slower performance than their non-reflective counterparts, and they can compromise security and expose private information.

Structure of Annotation

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

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)

public @interface VeryImportant {

}
```

The code defines a custom annotation in Java named `VeryImportant`. Let's break it down:

- `@Target(ElementType.TYPE)`: This is a meta-annotation that specifies this annotation can be used on a type (class, interface, or enum).
- `@Retention(RetentionPolicy.RUNTIME)`: This is another meta-annotation that indicates the annotation should be retained at runtime. This means the annotation will be readable in the JVM at runtime, which is necessary if you want to process the annotation using reflection.
- `public @interface VeryImportant {}`: This defines the annotation itself. Right now, `VeryImportant` doesn't have any elements, so it's just a marker annotation. You can use it to mark classes (or interfaces or enums) that are, well, very important.

Class Level Custom Annotation

```
@VeryImportant
public class Cat {
    // Your code here...
}
```

In this example, `CriticalClass` is marked as `@VeryImportant`. You could then use reflection at runtime to check if a class has been marked with `@VeryImportant` and handle it differently if so. For instance, you might log a message, throw an error, or perform some special initialization.

Now let's check which class is very important and which class is not, using the reflection technique in the main.

```
public static void main(String[] args) {
    Cat myCat = new Cat();
    dog myDog = new dog();

    if(myCat.getClass().isAnnotationPresent(VeryImportant.class)){
        System.out.println("This is a very important class!");
    } else {
        System.out.println("This is not a very important class!");
    }
}
```

The output is

```
This is a very important class!
```

But there is a class dog that is not annotated with the annotation `@VeryImportant` . That's why if I run the code,

```
public static void main(String[] args) {
    Cat myCat = new Cat();
    dog myDog = new dog();

    if(myDog.getClass().isAnnotationPresent(VeryImportant.class)){
        System.out.println("This is a very important class!");
    } else {
        System.out.println("This is not a very important class!");
    }
}
```

Output is,

```
This is not a very important class !
```

Method Level Custom Annotation

With our previous Cat example let's modify the cat class while building a `RunNow` Annotation.

```
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)

public @interface RunNow {

}
```

Now using Reflection I can run the methods in a class immediately which are annotated with this annotation.

But first, let's see the implementation of the `cat class`

```
@VeryImportant
public class Cat {
    public Cat(){}

    @RunNow
    public void meow() {
        System.out.println("meow");
    }

    public void purr() {
        System.out.println("purr");
    }
}
```

Now in main,

```
public static void main(String[] args){
    Cat myCat = new Cat();

    for(Method method: myCat.getClass().getMethods()) {
        if(method.isAnnotationPresent(RunNow.class)) {
            method.invoke(myCat);
        }
    }
}
```

The output is,

```
meow
```

So it invokes the method that is annotated with `@RunNow` Annotation.

Parameterized Annotation

We can put variables in annotation to have some value and in reflection, we can use that value to perform some operation based on that annotation parameter.

Let's try to make `@RunNow` Annotation with a `times` variable. In reflection, we will invoke that method `times` times.

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)

public @interface RunNow {
    int times() default 1;
}
```

Here `default 1` means if there is no parameter passed on annotation it will be 1 by default.

Now in the `Cat` Class,

```
@VeryImportant
public class Cat {
    public Cat(){}

    @RunNow(times = 4)
    public void meow() {
        System.out.println("meow");
    }

    public void purr() {
        System.out.println("purr");
    }
}
```

Here `@RunNow(times = 4)` specifies the value of the annotation for this method is 4.

In the reflection, we have to retrieve the variable `times()` stored in the annotation. So,

```
public static void main(String[] args) {
    Cat myCat = new Cat();

    for(Method method : myCat.getClass().getMethods()) {
        if(method.isAnnotationPresent(RunNow.class)) {
            RunNow rn = method.getAnnotation(RunNow.class);
            for(int i = 0; i < rn.times(); i++){
                method.invoke(myCat);
            }
        }
    }
}
```

Now the output is,

```
meow
meow
meow
meow
```

That is what we wanted from our annotation.

Conclusion

Java annotations provide a powerful way to add metadata to your Java code. They allow you to embed supplementary information directly in your code, without changing its logic. Annotations such as `@Target` and `@Retention` offer control over where and how long your custom annotations are retained, respectively.

The `@Target` annotation allows you to specify where your custom annotation can be used, whether on classes, methods, fields, parameters, etc. On the other hand, the `@Retention` annotation controls how long your annotation is kept around by the JVM.

Reflection, in combination with annotations, provides a dynamic way of inspecting and manipulating your code at runtime. This can be particularly useful in frameworks that rely on configuration through annotations.

However, while annotations and reflection are powerful tools, they should be used judiciously. Overuse can lead to code that is hard to understand and maintain. But when used correctly, they can make your code more flexible and easier to work with.

In conclusion, understanding and using Java annotations effectively can greatly improve the quality of your Java code, making it more robust, flexible, and maintainable. Happy coding!