Annotation

What is it?

- Java annotations are a mechanism for adding metadata information to source code.
- They're a powerful part of Java that was added in JDK5.
- annotations are Java types that are preceded by an “@” symbol.
- Spring and Hibernate are great examples of frameworks that rely heavily on annotations to enable various design techniques.
- Applying annotations consistently is a good practice since adding them can prevent future programmer error.
- An annotation needs to be interpreted in one way or another to be useful. Annotations can be interpreted at development-time by the IDE or the compiler, or at run-time by a framework.

Example of annotation use

```
@Override
public String toString() {
    return "Student{" + "ID='" + ID + '\'' + ", name='" + name + '}';
}


 @Test
 public void sumTest() {
     Calculator calculator = new Calculator();
     assertEquals(9, calculator.sum(5, 4));
 }
```

Advantages

- Inform the compiler about warnings and errors
- Manipulate source code at compilation time
- Modify or examine behavior at runtime

Example of built-in annotations

`@Override` a method overrides or replaces the behavior of an inherited method.

`@SuppressWarnings` to ignore certain warnings from a part of the code.

`@Deprecated` to mark an API as not intended for use anymore.

`@SafeVarargs` acts on a type of warning related to using varargs.

`@FunctionalInterface` to write code in a more functional way.

Where can we use them?

We can use them to

- classes, interfaces, methods, fields, parameters ,constructors etc.

Creating Custom Annotation

We can create custom annotation using @interface keyword

Example 1

```
public @interface NameOfAnnotation {

}
```

Example 2

```
public @interface NameOfAnnotation {
    String dosomething();
}
```

Program Execution Flow

Normally,

Source code → parser → Type Checker → Class File Writer → .class file

With annotation

Source code → parser → Type Checker → Annotation Checker → Class File Writer → .class file

Annotations Type

- Marker
    - Take no parameters. Used to mark an element to process in a particular way.
- Single-value
    - Provides a single piece of data. Can be used as data=value pair or only value within parenthesis
- Multi-value
    - Have multiple data members. Have to specify data=value comma separated.

Example:

Meta-Annotation

- Meta-annotations are annotations that can be applied to other annotations.
- These meta-annotations are used for annotation configuration:
    - @Target
    - @Retention
    - @Inherited
    - @Documented
    - @Repeatable

Meta-Annotation: Target

- The scope of annotations can vary based on the requirements.
- While one annotation is only used with methods, another annotation can be consumed with constructor and field declarations.
- `@Target` is used to define the scope of custom annotations.

```
@Target(ElementType.METHOD)

public @interface NameOfAnnotation {

    String dosomething();

}
```

Meta-Annotation: Target

`@Target(ElementType.TYPE)` [class and interface]

```
@Target(ElementType.METHOD)

@Target(ElementType.FIELD)

@Target(ElementType.CONSTRUCTOR)

@Target(ElementType.PARAMETER)

@Target(ElementType.LOCAL_VARIABLE)

@Target(ElementType.ANNOTATION_TYPE)

@Target(ElementType.PACKAGE)
```

Meta-Annotation

- `@Retention` where the annotation will be applied in the program's lifecycle
- @Retention with one of three retention policies:
  - RetentionPolicy.SOURCE – The annotation is used at compile time and discarded at runtime.
  - RetentionPolicy.CLASS – The annotation is stored in the class file at compile time and discarded at run time.
  - RetentionPolicy.RUNTIME – The annotation is retained at runtime.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(TYPE)
public @interface RetentionAnnotation {
}
```

Meta-Annotation

- `@Inherited` annotation to make our annotation propagate from an annotated class to its subclasses.
- `@Documented` By default, Java doesn't document the usage of annotations in Javadocs. But, we can use the @Documented annotation to change Java's default behavior.
- `@Repeatable` annotation, we can make an annotation repeatable:

Creating Custom Annotation

Class level annotation

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.Type)
public @interface JsonSerializable {
}
```

Field level annotation

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface JsonElement {
    public String key() default "";
}
```

Reflection

Introspection

- Asking an object for its meta-object is called introspection. The ability to inspect code in the system.
- Example:

```
Student student = new Student();
student.getClass()

Student.class.getAnnotations();
```

- Introspection and annotations belong to what is called reflection and meta-programming.

Reflection

- Reflection is then the ability to make modifications at runtime by making use of introspection.
- Java reflection allows us to inspect and/or modify runtime attributes of classes, interfaces, fields and methods.
- we can instantiate new objects, invoke methods and get or set field values using reflection
- One very common use case in Java is the usage with annotations.

Example

```
public class Person {
    private String name;
    private int age;
}

@Test
public void givenObject_whenGetsFieldNamesAtRuntime_thenCorrect() {
    Object person = new Person();
    Field[] fields = person.getClass().getDeclaredFields();

    List<String> actualFieldNames = getFieldNames(fields);

    assertTrue(Arrays.asList("name", "age")
      .containsAll(actualFieldNames));
}
```

we are able to get an array of *Field* objects from our *person* object, even if the reference to the object is a parent type of that object.

Functional Interface →Lambdas

Lambdas

- Any interface with a SAM(Single Abstract Method) is a functional interface, and its implementation may be treated as lambda expressions.

Example

Declaration of a functional interface

```
@FunctionalInterface
public interface Adder {
    int add(int a, int b);
}
```

Using the functional interface as lambdas

```
Adder adder = (a,b) -> a + b;
int result = adder.add(4,5);
```