Annotations

Intro, story & reasoning

Imagine reading a new interesting book. You carefully go through the pages; you enjoy the language, the metaphors; sometimes you even try to remember expressions you liked; sometimes the sentences remind you of another book you've read or a movie you've watched.

Now imagine studying a book. The topic may be hard, you need to make notes to support your brain while preparing for the exam or to link the information you already know. You use sticky notes to leave hints for future self, you use different colors for different notes.

You may have been studying for a while. On your shelf you'll find lots of books containing lots of your notes. Whenever you need to recap something you grab the books and pick up the necessary ones based on notes colors and follow the hints on those notes, very much similar to a primitive Zettelkasten system.

Now what has the above to do with Java annotations?

The answer is - it's the very same/similar concept!

They are not sticky notes…

Annotations, a form of metadata, provide data about a program that is not part of the program itself. Annotations have no direct effect on the operation of the code they annotate.

Annotations have a number of uses, among them:

- **Information for the compiler** — Annotations can be used by the compiler to detect errors or suppress warnings.
- **Compile-time and deployment-time processing** — Software tools can process annotation information to generate code, XML files, and so forth.
- **Runtime processing** — Some annotations are available to be examined at runtime.

First example

```
public class Calc {
    /**
     * ...
     * @deprecated do not use this method, it's unsafe (vulnarable to underflow and overflow)
     */
    @Deprecated
    public static int add(final int a, final int b) {
        return a + b;
    }
}
```

!

Your IDE can help you understand what’s going on:

```java
/**
 * @deprecated use builder instead
 */
1 usage
@Deprecated
public Employee(String name) {
    this.name = name;
}
```

```java
    var employee = new Employee( name: "wim");
```

'Employee(java.lang.String)' is deprecated

Boring theory/ Java Language Spec

- Annotations start with @

```
@Entity
```

- Annotations are usually used on class, method, parameter levels
    - Several other "targets", but those are not widely used

```
public static int add(@NotNull final Integer a, @NotNull final Integer b) {
    return a + b;
}
```

- Annotations can include elements/ properties/ fields

```
@Table(name = "T_USER", schema = "production")
```

- Annotations can be repeated if defined as such

```
@Author(name = "Kent Beck")
@Author(name = "Martin Fowler")
...
class AgileManifesto { ... }
```

!

- Annotations can be nested, if defined as such

```
@NamedQueries({
    @NamedQuery(name = "Owner.findAll", query = "SELECT o FROM Owner o"),
    @NamedQuery(name = "Owner.findById", query = "SELECT o FROM Owner o WHERE o.id = :id"),
    @NamedQuery(name = "Owner.findByFirstName", query = "SELECT o FROM Owner o WHERE o.firstName =
:firstName")
})
class OwnerRepository { ... }
```

Exciting demo

```
@Retention(RetentionPolicy.SOURCE) // we only want this annotation to be a marker
@Target({
    ElementType.TYPE,
    ElementType.FIELD,
    ElementType.METHOD,
    ElementType.PARAMETER,
    ElementType.CONSTRUCTOR,
    ElementType.LOCAL_VARIABLE,
    ElementType.ANNOTATION_TYPE,
    ElementType.PACKAGE,
    ElementType.TYPE_PARAMETER,
    ElementType.TYPE_USE,
    ElementType.MODULE,
    ElementType.RECORD_COMPONENT
}) // we want to be able to use this annotation everywhere
@Repeatable(value = Refactorings.class) // we want to be able to mark code for multiple refactorings due to
different reasons/issues
public @interface ToBeRefactored {
    String assignee();
    String cause() default "Clean code violation";
    String description();
}
```

Task: identify all usages of annotation `@ToBeRefactored` in your IDE

Note: in reality, one would use `TODO:` or `FIXME:`, but above is a good example of categorizing the code using annotations. Same approach is used in e.g. testing frameworks to define different types of tests or in JAX-RS frameworks to enhance code with additional functionality or to mark it for auto-generation during compile- or runtime, more on it later

Built-in annotations

As seen on the previous slide, Java already comes with a set of predefined annotations and can be divided in two categories:

- Annotations used by compiler
  - **`@Deprecated`**
  - **`@Override`**
  - **`@SupressWarnings`**
    - is also used to supress warnings from static code analysis tools (e.g. Sonar), be sure to know what you’re doing
  - `@SafeVarargs`
  - *`@FunctionalInterface`*
    - this annotation closely relates to functional programming concept, which is a topic for itself

!

- Annotations applied to other annotations (some of them used in the previous demo)
  - `@Retention`
  - `@Documented`
  - **`@Target`**
  - `@Inherited`
  - `@Repeatable`

Welcome to the (annotation) jungle!

JUnit 5

See `CalcTest` in `java-demo-annotations/nl.yoink.courses.dev.java.annotation.demo2`

```
public class CalcTest {

    ...

    @BeforeAll
    public static void beforeAll() {
        System.out.println("Now starting Calc tests");
    }

    @ParameterizedTest
    @MethodSource(value = "provideArgsAndExpectedResult")
    void calc(int a, int expected) {
        assertEquals(expected, Calc.computeExpression(a));
    }

    @Test
    @Disabled
    void disabled() {

    }
}
```

Lombok

Prior to `record` there was Lombok (https://projectlombok.org/), reducing the amount of boiler-plate code to write and maintain.

See https://projectlombok.org/features/Data and `java-demo-annotations/nl.yoink.courses.dev.java.annotation.demo3/Person.java` (together with decompiled `Person.class`)

```
@NoArgsConstructor
@AllArgsConstructor
@Builder(toBuilder = true)
@Data
public class Person {
    private long id;
    private String name;
    private String address;
}
```

!

```
//
// Source code recreated from a .class file by IntelliJ IDEA
// (powered by FernFlower decompiler)
//

package nl.yoink.courses.dev.java.annotation.demo3;

public class Person {
    private long id;
    private String name;
    private String address;
```

```
    public static PersonBuilder builder() {
        return new PersonBuilder();
    }

    public PersonBuilder toBuilder() {
        return (new PersonBuilder()).id(this.id).name(this.name).address(this.address);
    }

    public Person() {
    }

    public Person(long id, String name, String address) {
        this.id = id;
        this.name = name;
        this.address = address;
    }

    public long getId() {
        return this.id;
    }

    public String getName() {
        return this.name;
    }

    public String getAddress() {
        return this.address;
    }

    public void setId(long id) {
        this.id = id;
    }

    public void setName(String name) {
        this.name = name;
```

```
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public boolean equals(Object o) {
        if (o == this) {
            return true;
        } else if (!(o instanceof Person)) {
            return false;
        } else {
            Person other = (Person)o;
            if (!other.canEqual(this)) {
                return false;
            } else if (this.getId() != other.getId()) {
                return false;
            } else {
                Object this$name = this.getName();
                Object other$name = other.getName();
                if (this$name == null) {
                    if (other$name != null) {
                        return false;
                    }
                } else if (!this$name.equals(other$name)) {
                    return false;
                }

                Object this$address = this.getAddress();
                Object other$address = other.getAddress();
                if (this$address == null) {
                    if (other$address != null) {
                        return false;
                    }
                } else if (!this$address.equals(other$address)) {
```

```
                    return false;
                }

                return true;
            }
        }
    }

    protected boolean canEqual(Object other) {
        return other instanceof Person;
    }

    public int hashCode() {
        int PRIME = true;
        int result = 1;
        long $id = this.getId();
        result = result * 59 + (int)($id >>> 32 ^ $id);
        Object $name = this.getName();
        result = result * 59 + ($name == null ? 43 : $name.hashCode());
        Object $address = this.getAddress();
        result = result * 59 + ($address == null ? 43 : $address.hashCode());
        return result;
    }

    public String toString() {
        long var10000 = this.getId();
        return "Person(id=" + var10000 + ", name=" + this.getName() + ", address=" + this.getAddress() + ")";
    }

    public static class PersonBuilder {
        private long id;
        private String name;
        private String address;

        PersonBuilder() {
```

```
        }

        public PersonBuilder id(long id) {
            this.id = id;
            return this;
        }

        public PersonBuilder name(String name) {
            this.name = name;
            return this;
        }

        public PersonBuilder address(String address) {
            this.address = address;
            return this;
        }

        public Person build() {
            return new Person(this.id, this.name, this.address);
        }

        public String toString() {
            return "Person.PersonBuilder(id=" + this.id + ", name=" + this.name + ", address=" + this.address
+ ")";
        }
    }
}
```

JPA

```
@Entity
@Table(name = "T_PERSON")
public class Person {
    @Id
    private long id;

    @Column(name = "NAME")
    private String name;

    @Column(name = "LOCATION")
    private String address;
}
```

CDI

AOP

AOP (aspect oriented programming)

See `java-demo-annotations/nl.yoink.courses.dev.java.annotation.demo4`

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MeasureExecTime {
}

...

@Aspect
public class MeasureExecTimeAspect {

    @Pointcut("@annotation(measureExecTime)")
    public void callAt(MeasureExecTime measureExecTime) {
    }

    @Around("callAt(measureExecTime)")
    public Object around(ProceedingJoinPoint pjp,
                         MeasureExecTime measureExecTime) throws Throwable {
        long start = System.currentTimeMillis();
        Object result = pjp.proceed();
        System.out.println(pjp.toLongString() + " took " + (System.currentTimeMillis() - start) + " ms to
execute");
        return result;
    }
}

...

public class DemoAopStyleExecTimeMeasurement {
```

```
    @MeasureExecTime
    public static void main(String[] args) throws InterruptedException {
//        Thread.sleep(1_000L);
        System.out.println("Hello world!");
    }
}
```

Sample output:

```
Hello world!
execution(public static void
nl.yoink.courses.dev.java.annotation.demo4.DemoAopStyleExecTimeMeasurement.main(java.lang.String[])) took 2
ms to execute
```

Note: remember to execute `package` Maven goal in IntelliJ each time you change the code (e.g. `DemoAopStyleExecTimeMeasurement`)

Annotations

`@Annotation`s are a very useful tool to improve maintainability of the code and/or to induce additional behaviour without having to repeatedly write boiler-plate code.

Further reading:

- https://docs.oracle.com/javase/tutorial/java/annotations/basics.html
- https://docs.oracle.com/javase/tutorial/java/annotations/type_annotations.html
- https://docs.oracle.com/javase/tutorial/java/annotations/repeating.html

Questions?