Session 2 : Annotations, Dynamic Proxies

Annotations

```
@Override
@Test
```

- A way of commenting which is compiler-readable
- Java program can use the annotation to do something useful
  - Checking that the method is overridden, not overloaded
  - JUnit: Making a test method runnable

Declaring annotation types

- Annotate a method to measure its running time
- Measure the running time of a method and print the defined message

```
@interface Measure {
    String message() default "Hello world"; // Element
}
```

Using annotations

- Annotations can be used on different elements: Class, method, variable, parameters etc.

```
@Measure
public void someMethod() {
    // ...
}

@Measure(message = "Exceeded running time")
public void someMethod() {
    // ...
}
```

Marker annotation type

- Annotation type without any elements
- Don't have any elements
- Used by annotation processing tools

```
@interface Marker {
    // No elements
}

@Marker
class TestClass {

}
```

Meta-annotations

- Annotations used in declaring other annotation types
- Part of Java language

```
@Target
@Retention
@Inherited
@Documented
@Repeatable
```

Meta-annotation: `@Target`

https://docs.oracle.com/javase/8/docs/api/java/lang/annotation/ElementType.html

```
@Target({ElementType.METHOD})
@interface Measure {
    String message();
}
```

Meta-annotation: @Retention

- An annotation can be retained in three levels
  - ✓ Source code only SOURCE
  - ✓ Class file only CLASS
  - ✓ Class file and the runtime RUNTIME

```
@Retention(RetentionPolicy.RUNTIME)
@interface Measure {
    String message();
}
```

Meta-annotation: @Inherited

```
@Inherited
@interface AnnotatinTest {

}

@AnnotatinTest
class A {

}

class B extends class A {
    // @AnnotationTest is not declared
    // but Class B will have it
}
```

Commonly used annotations

- From Java API (java.lang package)
  - ✓ Deprecated
  - ✓ Override
  - ✓ SuppressWarnings
  - ✓ FunctionalInterface

Functional interfaces

- An interface with one abstract method
- Can be written as Lambda expressions

```
@FunctionalInterface
interface Job {
    void perform();
}

Job job = () -> { /* implement */ };
job.perform();
```

Some functional interfaces in java.util.function

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}

@FunctionalInterface
public interface BiFunction<T, U, R> {
    R apply(T t, U u);
}

@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

Some functional interfaces in java.util.function

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
}

@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}

@FunctionalInterface
public interface BiConsumer<T, U> {
    void accept(T t, U u);
}
```

Accessing annotations at runtime

- Only annotations with `RetentionPolicy.RUNTIME` are accessible
- It is possible to rely on annotations to add some sort of logic
- Using a technique called "reflection", a.k.a. meta-programming

Dynamic Proxies

Dynamic Proxies: InvocationHandler

- Recall **Proxy Pattern**
- Downside: We need to implement a new class
- Java provides an interface, InvocationHandler to build dynamic proxy objects
- Proxies add some functionalities to existing objects without changing the source code
- Happens during the runtime

Defining a Dynamic Proxy with InvocationHandler

- Create your original class instance: `new ApplicationServiceImpl()`
- Use `Proxy.newProxyInstance(...)` to create a run-time proxy class

Dynamic Proxy example

```
final ApplicationServiceImpl service = new ApplicationServiceImpl();
Object proxyInstance = Proxy.newProxyInstance(
    App.class.getClassLoader(),
    new Class[]{ApplicationService.class},
    // Lambda expression for new InvocationHandler() {...}
    (proxy, method, args) -> {
        System.out.println("Before");
        Object result = method.invoke(service, args);
        System.out.println("After");
        return result;
    }
 );

// Can also cast
ApplicationService custom =
    (ApplicationService) proxyInstance;
```

Generic Usage

- The InvocationHandler can be a generic task, i.e. logging or authentication
- How to plug the logging functionality to any object type?
- How to secure any object type?

Generic Usage

1-Implement the InvocationHandler interface

```
public class GenericLoggingProxy implements InvocationHandler { ... }
```

2-Store the original object

```
private final Object delegate;  // Original object

// Constructor
public GenericLoggingProxy(Object delegate) {
    this.delegate = delegate;
}
```

3-Implement the invoke(...) method, which defines the generic task (e.g. logging)

Generic Factory Method

- Recall **Factory Method Pattern**
- 4-Provide a generic factory method

```
public static <T> T create(
    Class<T> interfaceType, // any class of type T
    Object realObject) {    // an instance of type T

    return (T) Proxy.newProxyInstance(
        realObject.getClass().getClassLoader(),
        new Class[]{ interfaceType },
        new GenericLoggingProxy(realObject));
}
```

Generic proxy creation

```
ApplicationService proxyService = GenericLoggingProxy.create(
    ApplicationService.class,
    new ApplicationServiceImpl()
);

Map<String, String> proxyMap = GenericLoggingProxy.create(
    Map.class,
    new HashMap<>()
);
```

Combining Dynamic Proxies and Annotations

- Create an annotation Measure for methods
- When annotating a method, we want its running time to be measured
- Provide a message for the annotation to print with the running time

Steps:

- ✓ Define the annotation
- ✓ Implement an InvocationHandler
- ✓ Create a proxy: Proxy.newProxyInstance(...)
- ✓ Use the proxy object in source code

Homework 2

Project **Lombok** is a widely-used library that provides many convinient features for the developers via annotations.

Examples of Lombok annotations are **@Getter / @Setter** that free a developer from writing boilerplate getter and setter methods.

1-Include the Lombok project as a Maven dependency in a project and use the @Geter and @Setter annotations in a sample app.

2-Investigate the retention policy of the annotations and explain how it works.

3-Find the class files that are generated when you compile the Lombok-enabled project. Are there any annotations in the class files? Explain how the class files are generated.

Homework 2

4-Implement an annotation @Retry that has two properties: int limit and String message. Apply the annotation on some method that throws any Exception and implement a dynamic proxy that uses the annotation so that when the method throws the exception, it is automatically retried up to number specified in limit and if still failed it throws a RuntimeException with the given message. Write some test methods that demonstrate that your implemented annotation and the proxy work correctly.

Exemple :

```
@Retry(limit = 3, message = "Epic fail")
public void myMethod() {

    if (/* some creative condition */) {
        throw new Exception();
    }
}
```

Push the source code of your implemented task and a text file containing your answers to 1-3 in your repository.

End

sourcemind