ECS 160 – Annotations & Reflections

Instructor: Tapti Palit

Teaching Assistant: **Gabe Bai**

Agenda

- **Frameworks**
- Annotations
- Reflections

Frameworks

- An environment you build your application *inside of*
- Examples in other languages:
  - ReactJS in JavaScript
  - Python FastAPI

Main Idea

- Frameworks are structured and organized
- We write code that fits into the **framework's** structure
- The framework decides how to run the code we've written
  - Framework handles mundane, low-end tasks, such as **glue code**
  - Devs can handle the high-level stuff

Glue Code

- Code connecting your system together
  - Not business logic
- Boring, repetitive, error-prone

Agenda

- Frameworks
- **Annotations**
- Reflections

What Are Annotations?

- Labels for our code – that's metadata [@Label]
  - Extra information about
    - Classes
    - Methods
    - Variables
    - Parameters
- We can then pass this metadata on to anything that reads our code
  - Compiler
  - Frameworks

Simple Example

```
@Override
public String toString() { return "Hi"; }
```

Annotations

- Compiler instructions
  - E.g. @Override, @Deprecated
- Runtime processing
  - Frameworks heavily rely on annotations – think JUnit
- Code configuration
  - How do we connect our objects?

Configuration

```
@Component
public class Car {
    @Autowired
    private Engine engine;
}
```

```
Engine engine = new Engine();
Car car = new Car();
car.setEngine(engine);
```

Uses

- Tells frameworks what your code does
  - Is it a service?
  - Is it meant to be a Rest API?
- Control behavior
  - Injections?
  - Validation?
- Reduces config and boilerplate
  - Automatic getter/setter generation

Built-in Compiler Annotations

- @Override
- @Deprecated
- @SuppressWarnings
- These communicate with the compiler
  - They don’t affect runtime

@Override

- **Overrides** a parent method
- Use case: helps catch errors early
  - If you make a typo/mismatch a method signature
- Enforces compile-time safety, documentation of intent
- Best practice: use this anywhere you override

Example

```
public class Animal {

    public void speak() {}


    public String getType() {

        return "Generic animal";

    }

}
```

```
public class Cat extends Animal {
    @Override
    public void speak() {
        // println("Meow.");
    }

    @Override
    public String gettype() {
        // Compile-time error due
    to typo: should be getType()
    not gettype().
        return "Cat";
    }
}
```

@Deprecated

- Marks code as **deprecated**, i.e. old/unsafe
- Use case: warns other developers to not use old code
- Allows compatibility, but warns against it

@SuppressWarnings

- Directs compiler to ignore/**suppress warnings**
- Use case: silence particular warnings
  - Use with caution, when you ***understand the issue***

Runtime Annotations

- These annotations are used in conjunction with frameworks
  - Spring
  - Hibernate
  - JUnit
- Frameworks read them via **reflection** to decide how your code behaves
  - Annotations **express intent**!

Custom Annotations

- Create custom annotations with @interface
- Add meta-annotations
- Inspect it with reflection

Custom Example: @LogExecutionTime

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


@LogExecutionTime
public void serve() { ... }
```

serve() Originally

```
public void serve() {
    long start = System.currentTimeMillis();
    doSomething();
    System.out.println(System.currentTimeMillis() - start + "ms");
}
```

Solution

```
@LogExecutionTime
public void serve() {
    doSomething();
}
```

- This tells the framework to wrap that method call in that time-logging mechanism

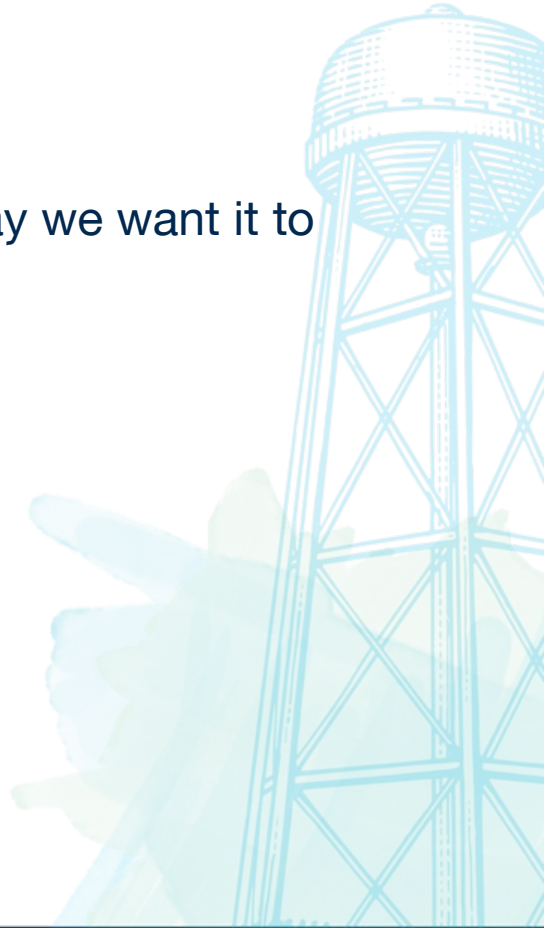
@LogExecutionTime

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


@Retention(RetentionPolicy.RUNTIME) // note to compiler
@Target(ElementType.METHOD) // ensure only allowed on methods
public @interface LogExecutionTime {}
```

Transition

- By itself, @LogExecutionTime does nothing
  - If we use **reflection**, we can make it respond the way we want it to



Agenda

- Frameworks
- Annotations
- **Reflections**

Reflection

- The ability for a program to inspect, understand, and modify itself at **runtime**
- java.lang.reflect.* package
  - *Class, Field, Method, Constructor, Parameter, Annotation*
- With reflection, we can discover classes at runtime
- Enables us to configure/reconfigure our software dynamically

Potential Use Cases

- Swap behavior by environment
  - E.g. MockPayment in tests, StripePayment in prod
- Auto-detect things
  - E.g. Find all @Test methods in a test suite

To Reiterate…

- We allow program to reason about itself
  - Find classes in a package
  - Read their annotations
  - Find field/methods to inject/call
  - Invoke them
- Declarative programming: we tell the code what we want done,
  - The framework via reflections figures out how to do it

Next Steps

- This becomes the foundation for:
  - Dependency injection – Spring
  - Object-Relational Mapping – Hibernate
  - Testing – JUnit
  - Serialization – Jackson

Advantages

- Cleaner, more declarative code
- Dynamic behavior at runtime
- Simplifies configuration

Disadvantages

- Slower reflection
- Breaks encapsulation
- Harder to debug or maintain
- Not always type-safe

```
public static void main(String[] args) throws Exception {
    Service service = new Service();
    for (Method method : Service.class.getDeclaredMethods()) {
        if (method.isAnnotationPresent(LogExecutionTime.class)) {
            long start = System.currentTimeMillis();
            method.invoke(service);
            long end = System.currentTimeMillis();
            System.out.println(
                method.getName() + " took " + (end - start) + " ms"
            );
        } else {
            method.invoke(service);
        }
    }
}
```