Aspect Oriented Programming
AOP

Why AOP?

- Some concerns in an application are cross cutting
  - the same code would be repeated at many places
  - e.g. security, logging, retry-on-error etc.
- AOP complements the Object Oriented paradigm

AOP in Java

- AspectJ is the most used AOP implementation
- Needs an additional weaving compiler after the Java compiler
- Makes the build process more complicated
- Spring AOP is easier but less powerful

AOP definitions

- Join point
  - a point during execution of code (a method execution)
- Advice
  - Action taken at a join point
  - e.g. “around”, “before” or “after”
  - this is where you implement your code
- Pointcut
  - predicate that matches join points
  - e.g. “all methods in a certain package”

POM dependencies

```
<dependency>
  <groupid>org.springframework.boot</groupid>
  <artifactid>spring-boot-starter-aop</artifactid>
</dependency>
```

Example class

```
@Component("student")
public class AOPStudent {
  private String name = "Wim";

  public String getName() {
    return name;
  }
}
```

Declaring an Aspect

```
@Aspect
@Component
public class AOPStudentAspect {

  //Pointcut and advice

  @Before("execution(* com.example.*.get*())")
  public void getGetterAdvice(){
    System.out.println("Getter is called");
  }
}
```

Triggering the aspect

```
@SpringBootApplication
public class AOPApplication implements CommandLineRunner {
  @Autowired
  private AOPStudent student;

  @Override
  public void run(String... arg0) throws Exception {
    System.out.println(student.getName());
  }

  public static void main(String[] args) {
    // Used to make sure Tomcat is not started
    new SpringApplicationBuilder(
      AOPApplication.class).web(false).run(args);
  }
}
```

Pointcut definitions

```
execution(public String lab2.MovieCatalog.toString(..))
```

- `public`: access modifier
- `String`: return type
- `MovieCatalog`: type
- `toString`: method
- `..`: arguments

Pointcut examples

Pointcut	**Explanation**
execution(public (..))	All public methods
execution(String *(..))*	All methods returning a String
execution(public save(..))	All public save methods
execution(public *a.b.dao.* (..))	All public methods in the dao package

Pointcut examples

Pointcut	**Explanation**
execution(public String *(String, Integer))	All public methods returning a String and arguments of type String and Integer
@annotation(a.b.MyAnnotation)	All methods annotated a.b.MyAnnotation

Before advice

```
@Before("execution(public String hello())")
public void before(JoinPoint joinpoint){
}
```

- Make sure you import the correct JoinPoint

```
import org.aspectj.lang.JoinPoint;
```

JoinPoint

```
@Before("execution(public String hello())")
public void before(JoinPoint joinPoint){
  System.out.println(joinPoint.getStaticPart().toString());
  System.out.println("Begin");
}
```

Passing arguments

```
@Before("execution(* *(..)) && args(message)") {
public void before(JoinPoint joinPoint,
    String message ) {
  System.out.println("Calling method with String: " + message);
}
```

- Declare the arguments: `args` (**`message`**)
- The **`message`** in the pointcut must match the **`message`** argument of the `before` method

After advices

```
@AfterReturning(value = "execution(* * (..))", returning = "retval")
public void afterSucces(Object retval){
  System.out.println("I returned succesfully: " + retval);
}

@AfterThrowing(value = "execution(* * (..))", throwing = "exception")
public void afterException(Exception exception){
  System.out.println("An exception was thrown " +
    exception.getMessage());
}

@After(value = "execution(* * (..))")
public void after(Object retval){
  System.out.println("I'm done");
}
```

Around advice

- Call `proceed` to continue the original method call

```
@Around(value = "execution(public * *(..))")
public Object trace(ProceedingJoinPoint proceedingJoinPoint)
    throws Throwable {
  long startTime = System.currentTimeMillis();

  Object result = proceedingJoinPoint.proceed();

  long elapsedTime = System.currentTimeMillis() - startTime;
  System.out.println(proceedingJoinPoint.getStaticPart().
    getSignature().getName() + " " + elapsedTime);
  return result;
}
```

Retry example

```
@Around("execution(public * *(..))")
public Object retry(ProceedingJoinPoint proceedingJoinPoint)
    throws Throwable {
  Object result = null;

  try {
    result = proceedingJoinPoint.proceed();
  } catch (Throwable e) {
    System.out.println("Sleep 2 seconds before retry");
    TimeUnit.SECONDS.sleep(2);
    result = proceedingJoinPoint.proceed();
  }
  return result;
}
```

Reusable pointcut definitions

```
@Aspect
@Component
public class MyPointcuts {

  @Pointcut("execution(* a.b.dao(..))")
  public void inDAOLayer() {}
}

@After("lab2.aop.MyPointcuts.inDAOLayer")
public void after() {
  System.out.println("I'm done!");
}
```

Introductions

- Introduce new methods and interfaces to a class
- How to make a car fly and shoot?

- Plain car implementation

```
@Component
public class CarImpl implements Car {
  @Override
  public void drive() {
    System.out.println("Driving...");
  }
}
```

Introduction example

```
@Component
@Aspect
public class AbilityIntroduction {
  @DeclareParents(value="com.example.CarImpl",
    defaultImpl = com.example.FlyerImpl.class)
  public Flyer flyer;

  @DeclareParents(value="com.example.CarImpl",
    defaultImpl = com.example.ShooterImpl.class)
  public Shooter shooter;
}
```

Introduction example

```
public class FlyerImpl implements Flyer {
  @Override
  public void fly() {
    System.out.println("Flying!");
  }
}

public class ShooterImpl implements Shooter {
  @Override
  public void shoot() {
    System.out.println("Shooting!");
  }
}
```

Using the introduction

```
@Component
public class IntroductionExample {
  @Autowired
  private Car car;

  public void testCarAbilities() {
    car.drive();

    Flyer flyer = (Flyer) car;
    flyer.fly();

    Shooter shooter = (Shooter) car;
    shooter.shoot();
  }
}
```