Aspect Oriented Programming

Charles Zhang

Fall 2019

Topics covered in last lecture

- Injection of static/instance factories
- Injection of static/instance fields
- Singletons
  - Traditional vs Spring singletons
- Constructor/Setter/Member injections
- Guice
- Dagger

Outline

- Crosscutting concerns
- Aspect oriented programming (AOP)
- AOP in Spring
    - Method advice
    - Pointcut definition and expressions
    - How AOP actually works?

Application concerns

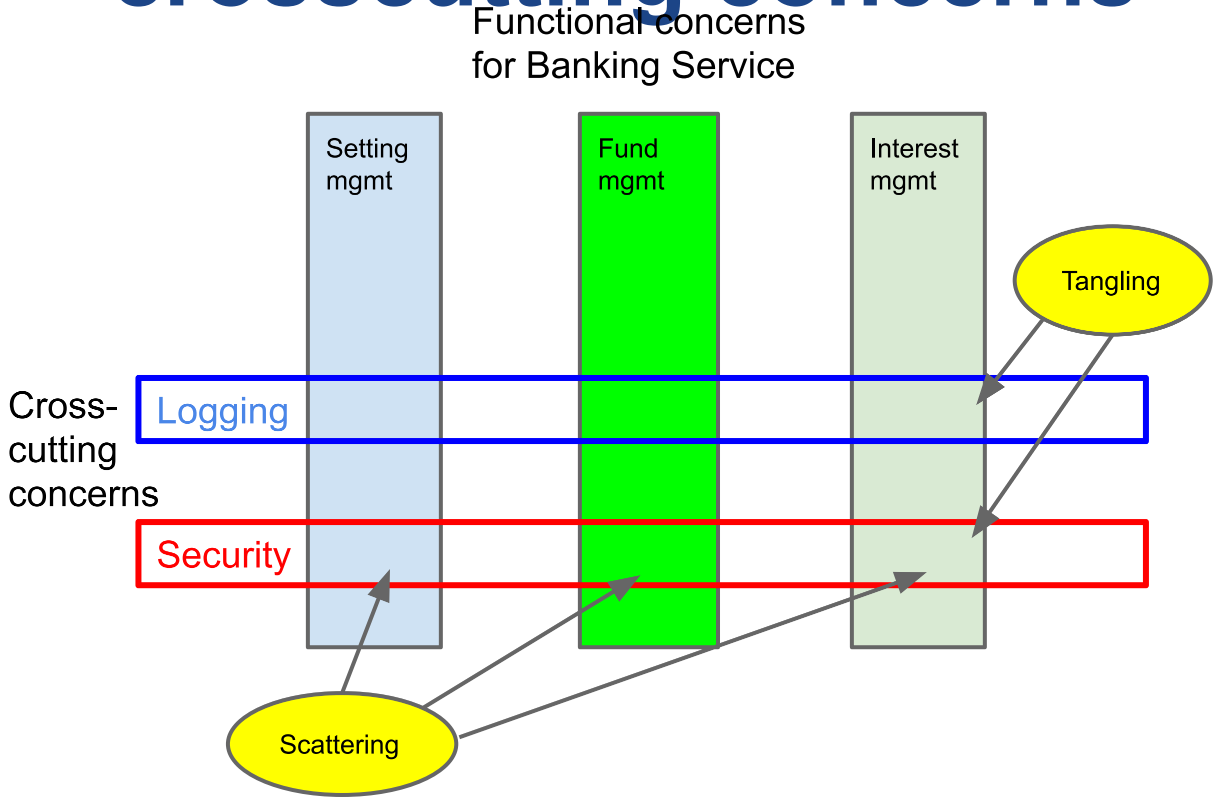
- Concerns are categories of requirements for an application
  - Functional concerns: all features working
  - Quality of service concerns: performance, robustness, scalability
  - Security
  - Compliance

Address multiple concerns

```
public double getBalance(String userId, long accountNumber){
        ...
}

public double deposit(String userId, long accountNumber, double amount){
        ...
}

public double withdraw(String userId, long accountNumber, double amount){
        ...
}
```

- How do we address more concerns?
  - Authorization
  - Logging
  - Transaction

Functional concerns and crosscutting concerns



Crosscutting concerns

- Can be scattered across a number of different functional concerns
- Can be tangled with other concerns
- Common, and orthogonal

More crosscutting concerns

- Synchronization
- Real-time constraints
- Error detection and correction
- Data validation
- Transaction processing
- Internationalization and localization which includes language localisation
- Caching
- Monitoring

*:Source: Wikipedia

Aspect Oriented Programming (AOP)

- Aspect
  - Abstraction of a crosscutting concern
- AOP
  - Programming model that encapsulates crosscutting concerns into aspects to retain modularity
  - Reduces tangling and scattering

Why AOP? Separation of concerns

- The principle of *separation of concerns* (*SoC*)
  - Organize computer software into distinct elements, each taking care of one separate concern
  - Easy to understand, develop, and maintain
- The *KISS* principle
  - Keep it simple and stupid
  - Keep it simple and straightforward
  - Keep it simple and strong

Keep it simple!!!

Key AOP concepts: aspect, advice, joinpoint, and pointcut

- Aspect: abstraction of a crosscutting concern
- Advice: code to implement a concern
- Joinpoint: a point in the flow of program execution
- Pointcut: a set of joinpoints, usually expressed as an expression

World without AOP

```java
package com.apress.springrecipes.calculator;

public interface ArithmeticCalculator {

    public double add(double a, double b);
    public double sub(double a, double b);
    public double mul(double a, double b);
    public double div(double a, double b);
}
```

```java
public class ArithmeticCalculatorImpl implements ArithmeticCalculator {

    public double add(double a, double b) {
        double result = a + b;
        System.out.println(a + " + " + b + " = " + result);
        return result;
    }

    public double sub(double a, double b) {
        double result = a - b;
        System.out.println(a + " - " + b + " = " + result);
        return result;
    }
```

Tangling and scattered

Define an Aspect in Spring

```java
@Aspect
public class CalculatorLoggingAspect {

    private Log log = LogFactory.getLog(this.getClass());

    @Before("execution(* ArithmeticCalculator.add(..))")
    public void logBefore() {
        log.info("The method add() begins");
    }
}
```

Different types of advices

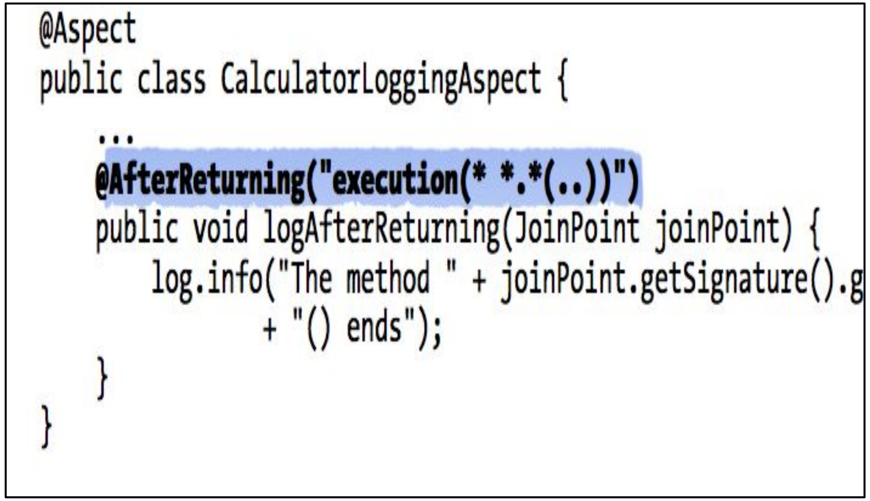
- @Before
- @After
- @AfterReturning
- @AfterThrowing
- @Around

@After and JoinPoint

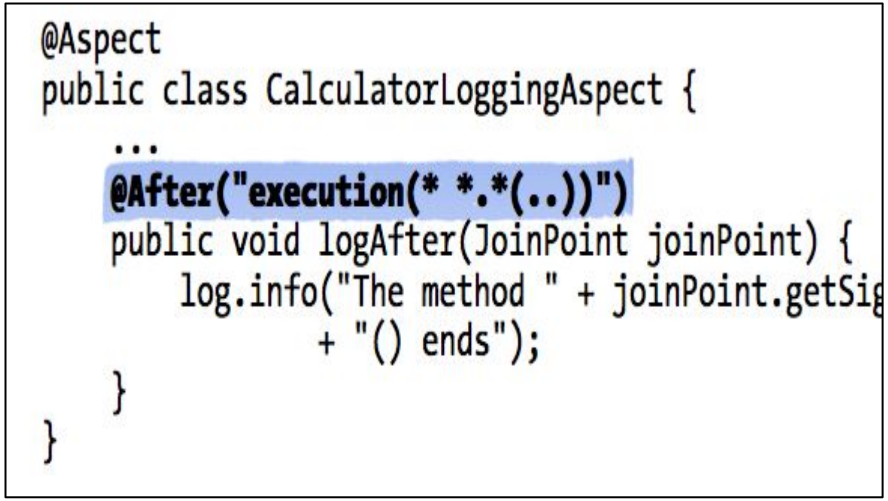
```
@Aspect
public class CalculatorLoggingAspect {
    ...
    @After("execution(* *.*(..))")
    public void logAfter(JoinPoint joinPoint) {
        log.info("The method " + joinPoint.getSignature().getName()
                + "() ends");
    }
}
```

The Joinpoint object captures the context info for the execution point

@AfterReturning vs @After



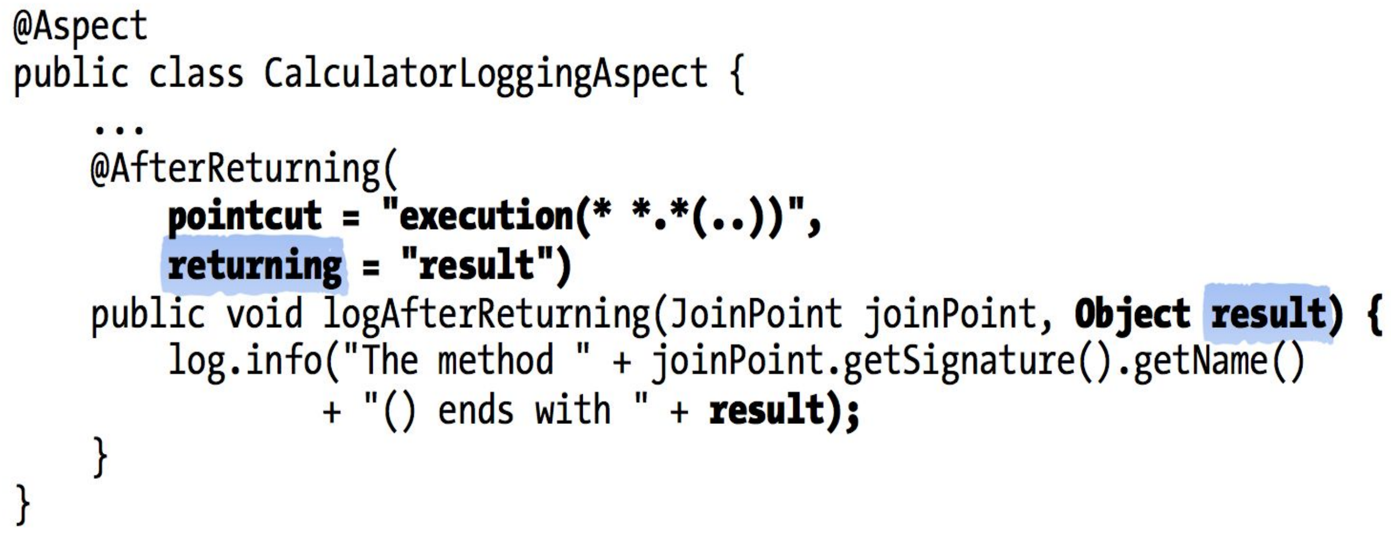
```
@Aspect
public class CalculatorLoggingAspect {
    ...
    @AfterReturning("execution(* *.*(..))")
    public void logAfterReturning(JoinPoint joinPoint) {
        log.info("The method " + joinPoint.getSignature().g
                + "() ends");
    }
}
```



```
@Aspect
public class CalculatorLoggingAspect {
    ...
    @After("execution(* *.*(..))")
    public void logAfter(JoinPoint joinPoint) {
        log.info("The method " + joinPoint.getSig
                + "() ends");
    }
}
```

@After is called no matter it returns or not

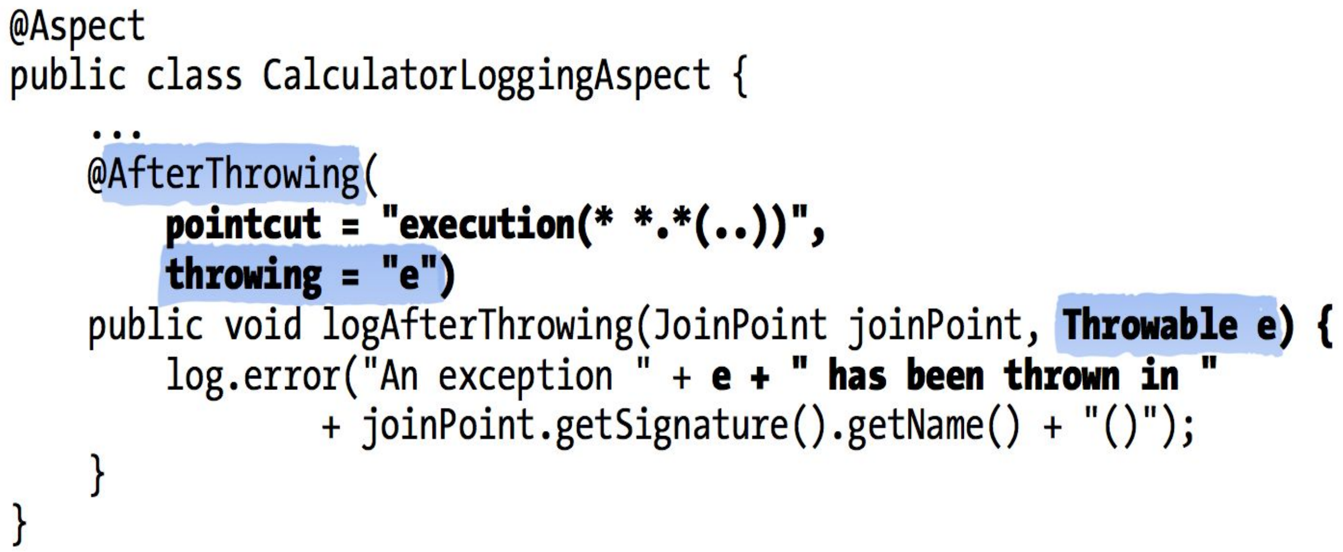
@AfterReturning: access the returned value



```
@Aspect
public class CalculatorLoggingAspect {
    ...
    @AfterReturning(
        pointcut = "execution(* *.*(..))",
        returning = "result")
    public void logAfterReturning(JoinPoint joinPoint, Object result) {
        log.info("The method " + joinPoint.getSignature().getName()
                + "() ends with " + result);
    }
}
```

How is the “result” passed in?

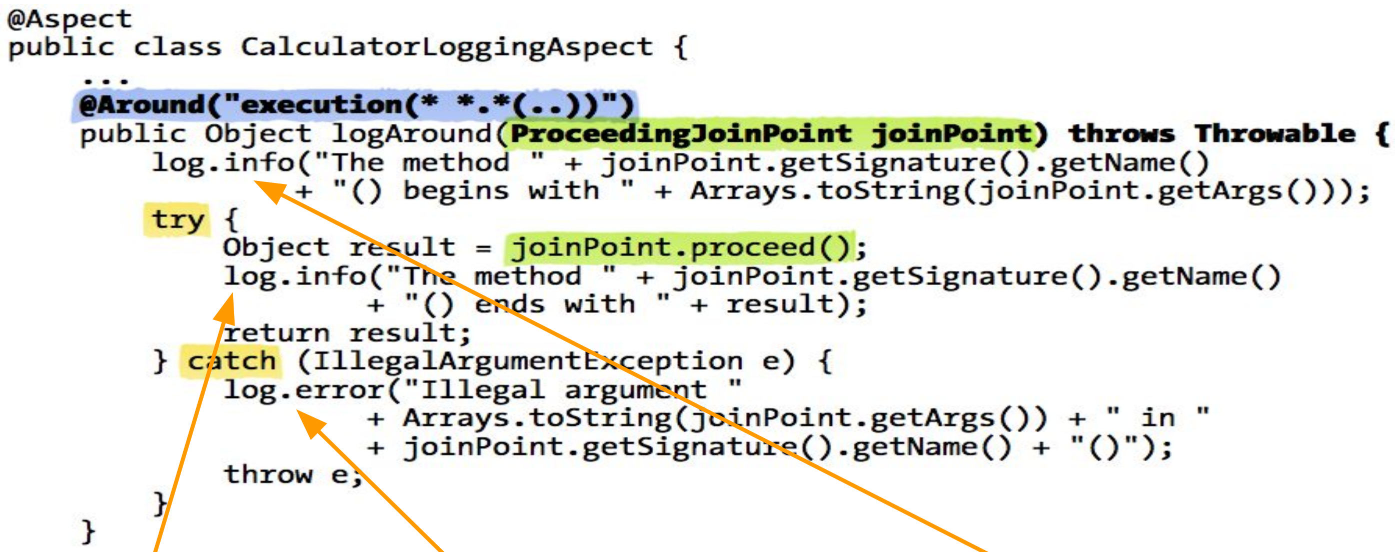
@AfterThrowing



```
@Aspect
public class CalculatorLoggingAspect {
    ...
    @AfterThrowing(
        pointcut = "execution(* *.*(..))",
        throwing = "e")
    public void logAfterThrowing(JoinPoint joinPoint, Throwable e) {
        log.error("An exception " + e + " has been thrown in "
                + joinPoint.getSignature().getName() + "()");
    }
}
```

Can @After and @AfterThrowing both be called?

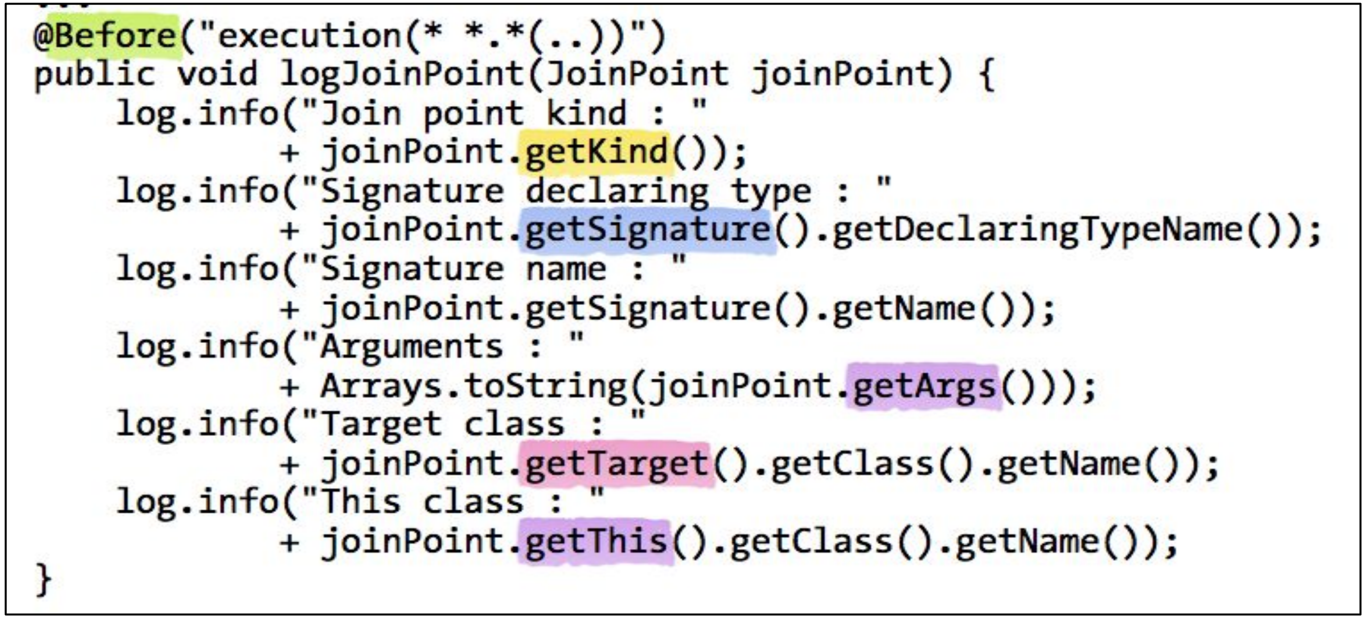
@Around - most powerful advice



```
@Aspect
public class CalculatorLoggingAspect {
    ...
    @Around("execution(* *.*(..))")
    public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
        log.info("The method " + joinPoint.getSignature().getName()
                + "() begins with " + Arrays.toString(joinPoint.getArgs()));
        try {
            Object result = joinPoint.proceed();
            log.info("The method " + joinPoint.getSignature().getName()
                    + "() ends with " + result);
            return result;
        } catch (IllegalArgumentException e) {
            log.error("Illegal argument "
                    + Arrays.toString(joinPoint.getArgs()) + " in "
                    + joinPoint.getSignature().getName() + "()");
            throw e;
        }
    }
```

Full control of a joint point: capable of @Before, @AfterReturning, @AfterThrowing, and @After

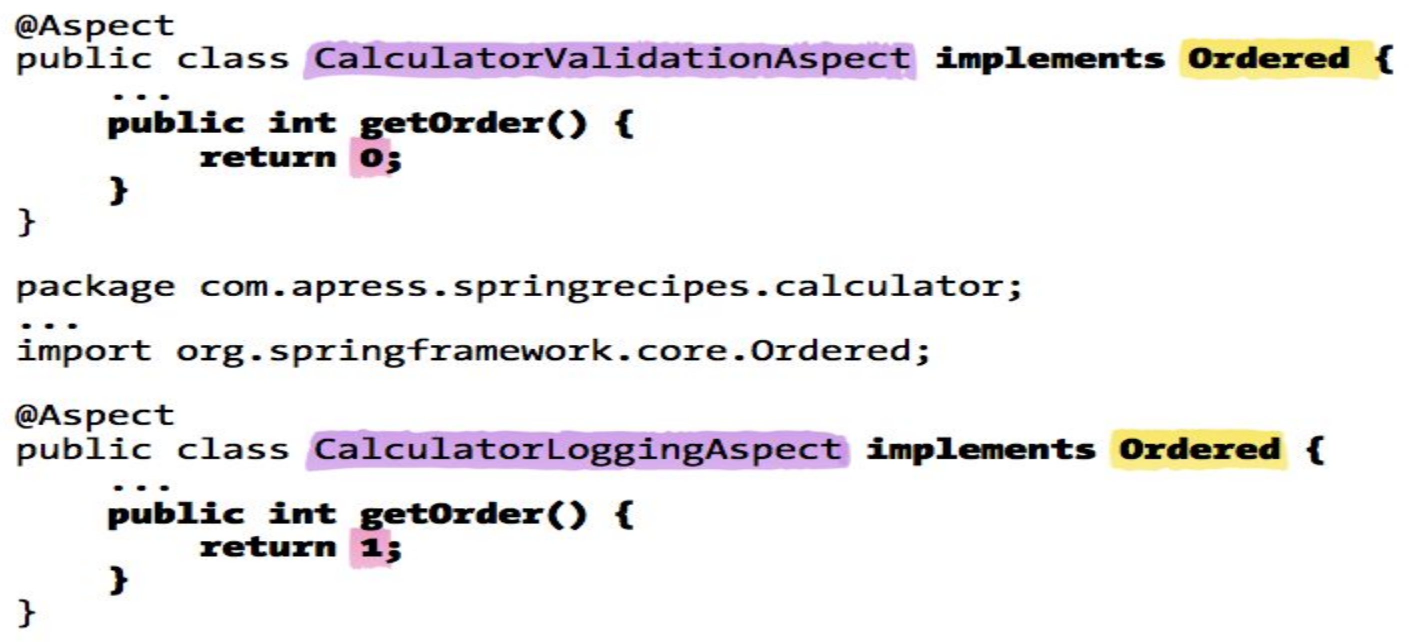
Access the joint point info



```
@Before("execution(* *.*(..))")
public void logJoinPoint(JoinPoint joinPoint) {
    log.info("Join point kind : "
            + joinPoint.getKind());
    log.info("Signature declaring type : "
            + joinPoint.getSignature().getDeclaringTypeName());
    log.info("Signature name : "
            + joinPoint.getSignature().getName());
    log.info("Arguments : "
            + Arrays.toString(joinPoint.getArgs()));
    log.info("Target class : "
            + joinPoint.getTarget().getClass().getName());
    log.info("This class : "
            + joinPoint.getThis().getClass().getName());
}
```

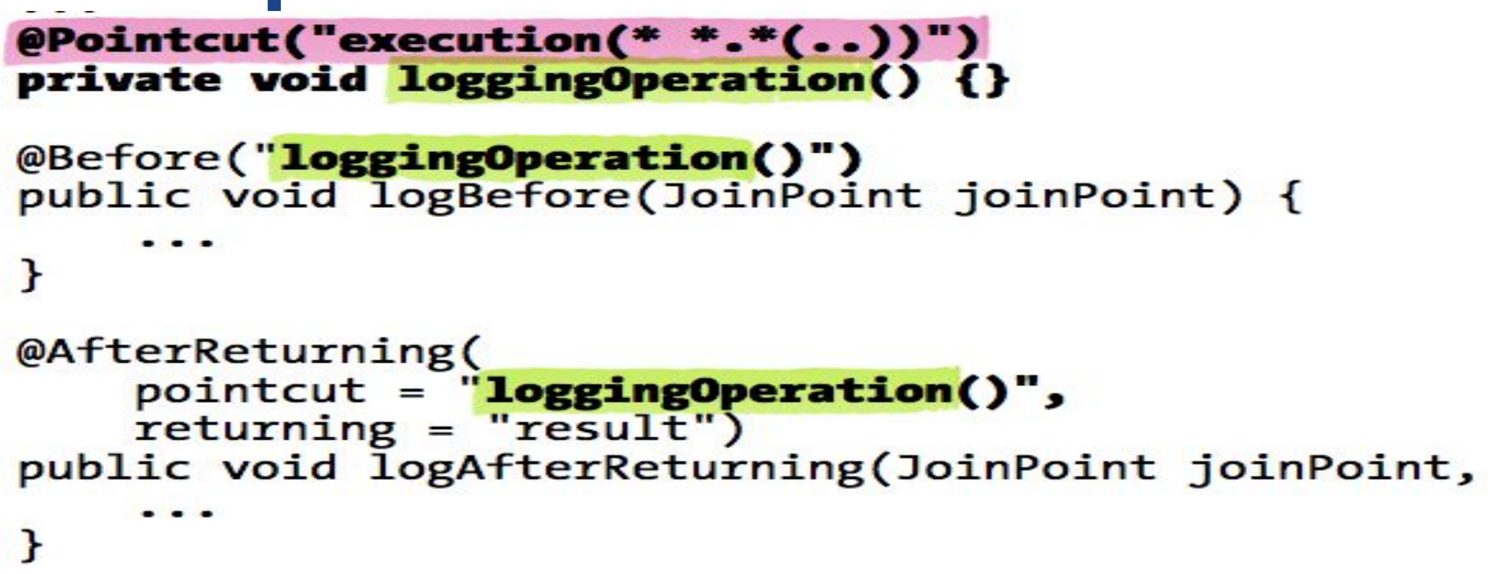
What’s the difference between execution and call?

Specifying aspect precedence



```
@Aspect
public class CalculatorValidationAspect implements Ordered {
    ...
    public int getOrder() {
        return 0;
    }
}

package com.apress.springrecipes.calculator;
...
import org.springframework.core.Ordered;

@Aspect
public class CalculatorLoggingAspect implements Ordered {
    ...
    public int getOrder() {
        return 1;
    }
}
```

Reuse pointcut definitions



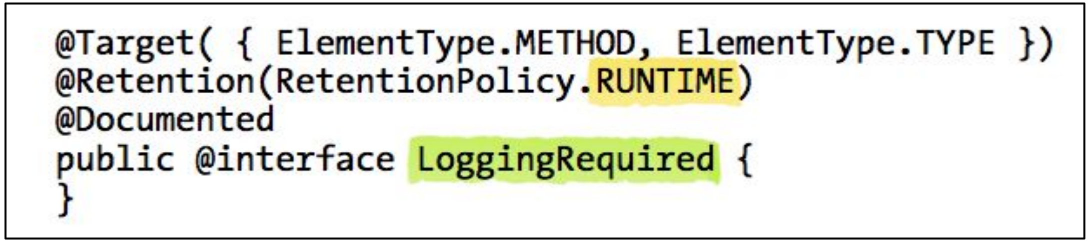
```
@Pointcut("execution(* *.*(..))")
private void loggingOperation() {}

@Before("loggingOperation()")
public void logBefore(JoinPoint joinPoint) {
    ...
}

@AfterReturning(
    pointcut = "loggingOperation()",
    returning = "result")
public void logAfterReturning(JoinPoint joinPoint,
    ...
}
```

Use @Pointcut to annotate a method, then the method can be used for pointcut expressions

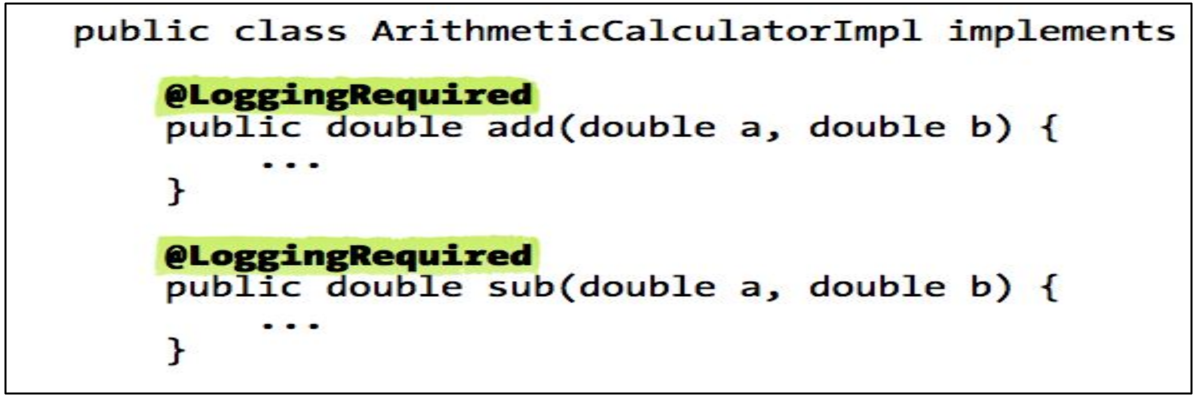
Pointcut expressions

- Any method
  - execution(* com.foo.Bar.*(..))
- Any public method
  - execution(public * com.foo.Bar.*(..))
- Any method that takes a double and return a double
  - execution(public double com.foo.Bar.*(double))

Using annotation for pointcut expression



```
@Target( { ElementType.METHOD, ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface LoggingRequired {
}
```



```
public class ArithmeticCalculatorImpl implements

    @LoggingRequired
    public double add(double a, double b) {
        ...
    }

    @LoggingRequired
    public double sub(double a, double b) {
        ...
    }
```

Then we can use: `@Before(“@annotation(LoggingRequired)”)`

AspectJ

- What is AspectJ
  - An aspect-oriented programming (AOP) extension created for the Java programming language
  - Available in Eclipse Foundation open-source projects, both stand-alone and integrated into Eclipse
- How to enable AspectJ annotation support for your application?
  - Define an empty XML element, <aop:aspectj-autoproxy>, in your bean configuration
  - Add the aop schema definition to your <beans> root element
  - When the Spring IoC container notices the <aop:aspectj-autoproxy>, it will automatically create proxies for your beans

Aspect Weaving

The AspectJ weaver takes class files as input and produces class files as output

- Compile-time weaving. When you have the source code for an application, ajc will compile from source and produce woven class files as output. The invocation of the weaver is integral to the ajc compilation process
- Post-compile weaving (also sometimes called binary weaving) is used to weave existing class files and JAR files. As with compile-time weaving, the aspects used for weaving may be in source or binary form, and may themselves be woven by aspects
- Load-time weaving (LTW) is simply binary weaving deferred until the point that a class loader loads a class file and defines the class to the JVM. To support this, one or more "weaving class loaders" are required

Summary

- What’s a crosscutting concern?
- What is AOP?
- What is a joinpoint?
- What is a pointcut?
- What is an advice?
- What are the method advice types?
- How to express a pointcut?
- What is Aspect Weaving?

Mockito - simpler & better mocking

- Project page
  - https://github.com/mockito/mockito
- Introduction
  - http://www.slideshare.net/fwendt/using-mockito