TS Spring

Day 3: Aspect-oriented programming

Introduction

- https://github.com/banckaert/ts-spring-aop
- Why this course?
  - AOP is a powerful tool
  - AOP is a fundamental part of the Spring Framework (and Java EE)

  ⇔
  - Seldom used to its full potential
  - Trainings and books on the subject are often academic and unpractical

Overview

- Why Aspect-oriented Programming (AOP)?
  - How AOP helps to solve problems in enterprise applications
- My First Aspect
  - First example of an aspect
- Advices
  - What code can be used in aspects?
- Pointcuts
  - How to define where aspects are added to your application

Overview

- Aspects and Architecture
  - Expressing architecture using Pointcuts
  - Make architecture a part of your application
- Spring AOP: under the hood
  - How Aspects are added to Objects
  - Deeper understanding about Spring’s AOP internals
- Spring AOP: Spring Aspect Library
  - Out-of-the-box aspects for your application
- Aspects, the real world
  - Some ideas for your own aspects

Why Aspect-oriented Programming (AOP)?

- Why AOP?
- Reducing Boiler Plate Code using AOP
- How AOP Works
- Cross-Cutting Concerns
- Summary

Why AOP?

- What is AOP?
  - Aspect Oriented Programming (AOP)
  - Complements Object Oriented Programming (OOP)
  - Unit of modularisation: OOP Class vs. AOP Aspect
  - Modularisation across types and objects (Cross Cutting Concerns)
- What does AOP bring to the table?
  - AOP reduces Boiler Plate Code
  - AOP is used to add enterprise features to your application

Reducing Boiler Plate Code using AOP (1/5)

```
19    public void someMethod() {
20        final String METHODNAME = "someMethod";
21        logger.trace("entering " + CLASSNAME + "." + METHODNAME);
22        TransactionStatus tx = transactionManager
23                .getTransaction(new DefaultTransactionDefinition());
24        try {
25            // Business Logic
26        } catch (RuntimeException ex) {
27            logger.error("exception in " + CLASSNAME + "." + METHODNAME, ex);
28            tx.setRollbackOnly();
29            throw ex;
30        } finally {
31            transactionManager.commit(tx);
32            logger.trace("exiting " + CLASSNAME + "." + METHODNAME);
33        }
34    }
```

Reducing Boiler Plate Code using AOP (2/5)

- Logging
- Transactions
- Exception Handling

```
19    public void someMethod() {
          final String METHODNAME = "someMethod";
          logger.trace("entering " + CLASSNAME + "." + METHODNAME);
          TransactionStatus tx = transactionManager
                  .getTransaction(new DefaultTransactionDefinition());
24        try {
25            // Business Logic
26        } catch (RuntimeException ex) {
              logger.error("exception in " + CLASSNAME + "." + METHODNAME, ex);
              tx.setRollbackOnly();
29            throw ex;
30        } finally {
              transactionManager.commit(tx);
              logger.trace("exiting " + CLASSNAME + "." + METHODNAME);
33        }
34    }
```

Reducing Boiler Plate Code using AOP (3/5)

~~Logging~~

Transactions

Exception Handling

```
19    public void someMethod() {


          TransactionStatus tx = transactionManager
                  .getTransaction(new DefaultTransactionDefinition());
24        try {
25            // Business Logic
26        } catch (RuntimeException ex) {
              logger.error("exception in " + CLASSNAME + "." + METHODNAME, ex);
              tx.setRollbackOnly();
29            throw ex;
30        } finally {
              transactionManager.commit(tx);

33        }
34    }
```

Reducing Boiler Plate Code using AOP (4/5)

~~Logging~~

~~Transactions~~

Exception Handling

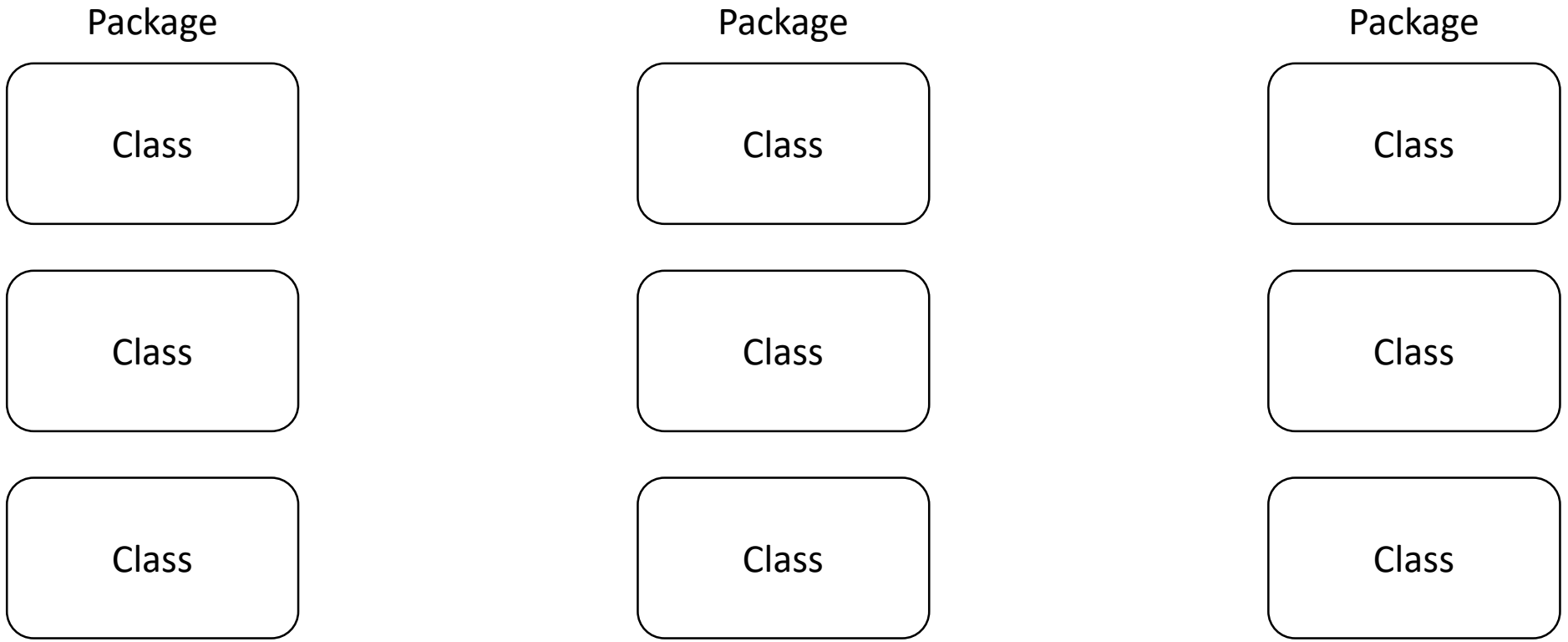
```
19    public void someMethod() {
24        try {
25            // Business Logic
26        } catch (RuntimeException ex) {
              logger.error("exception in " + CLASSNAME + "." + METHODNAME, ex);
29            throw ex;
30        } finally {
33        }
34    }
```

Reducing Boiler Plate Code using AOP (5/5)

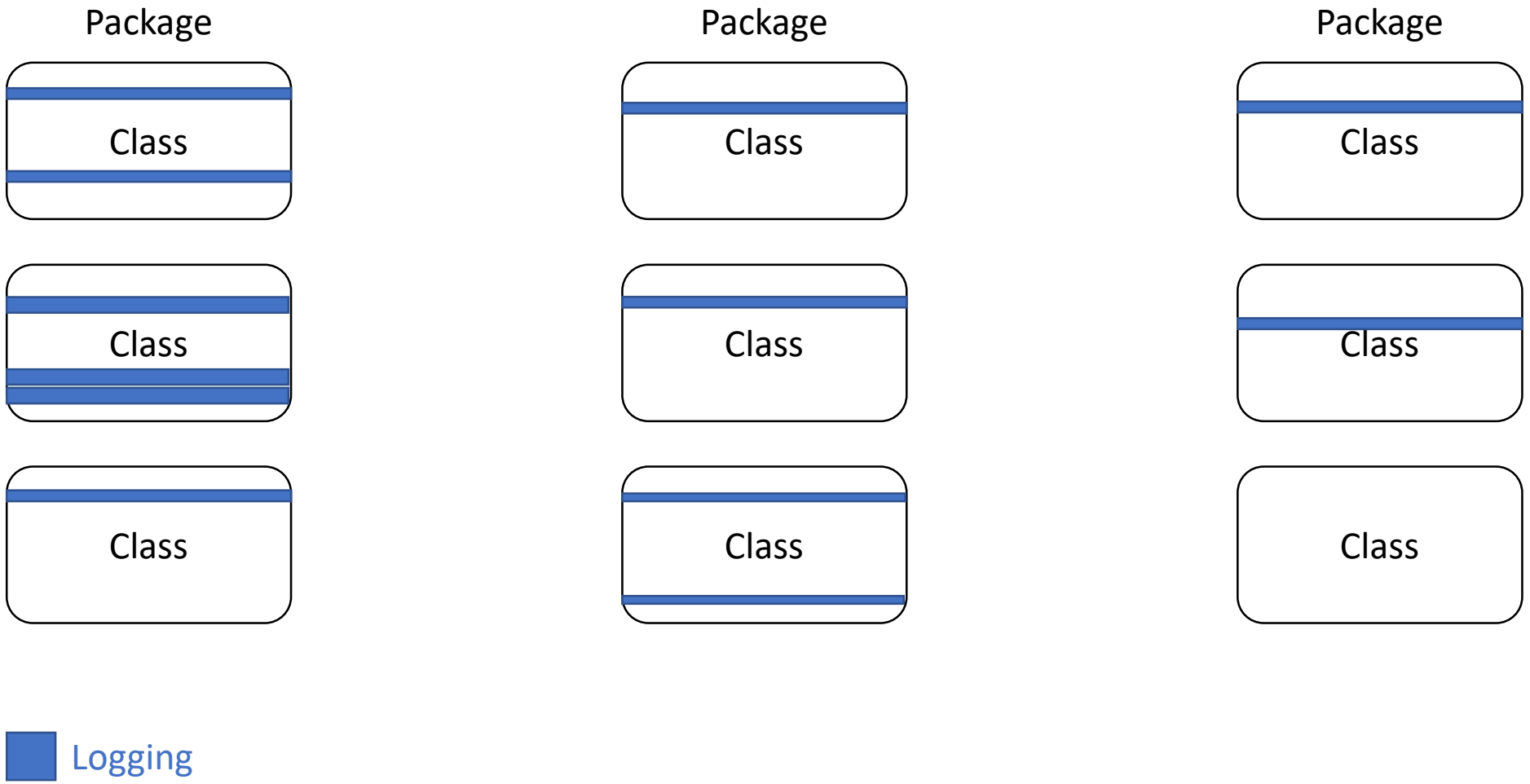
- ~~Logging~~
- ~~Transactions~~
- ~~Exception Handling~~

```
19    public void someMethod() {
25        // Business Logic
34    }
```

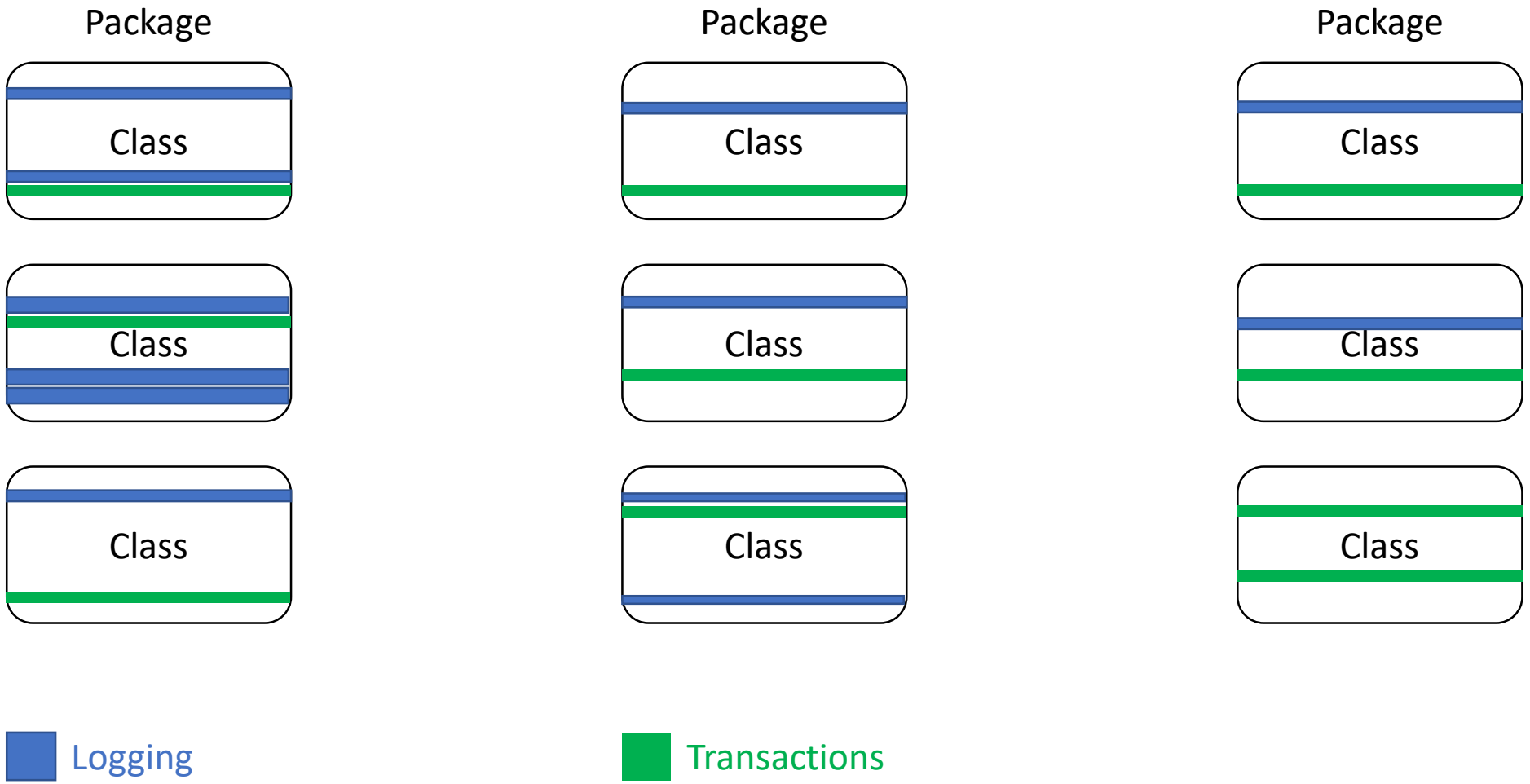
How AOP Works



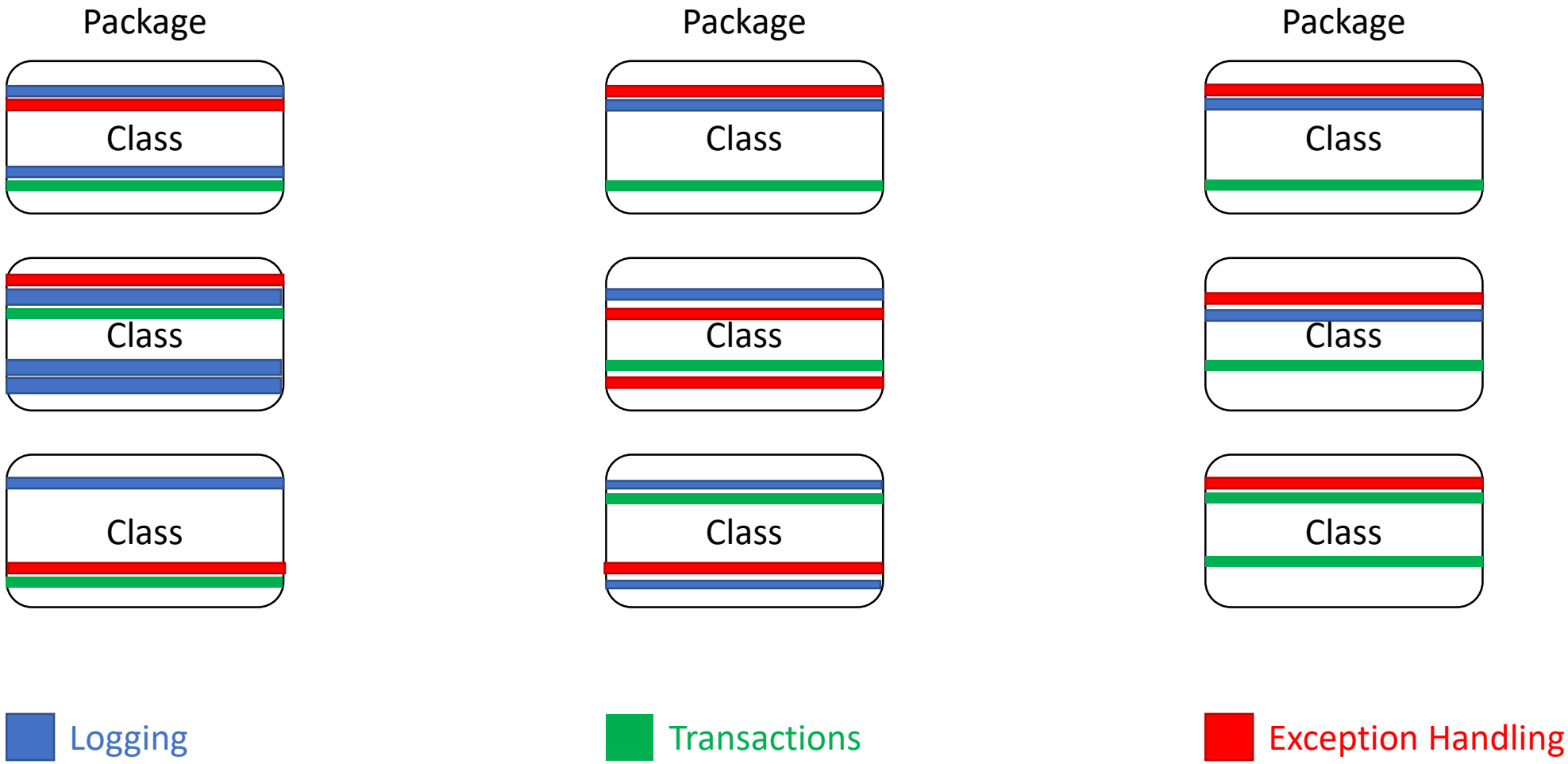
How AOP Works



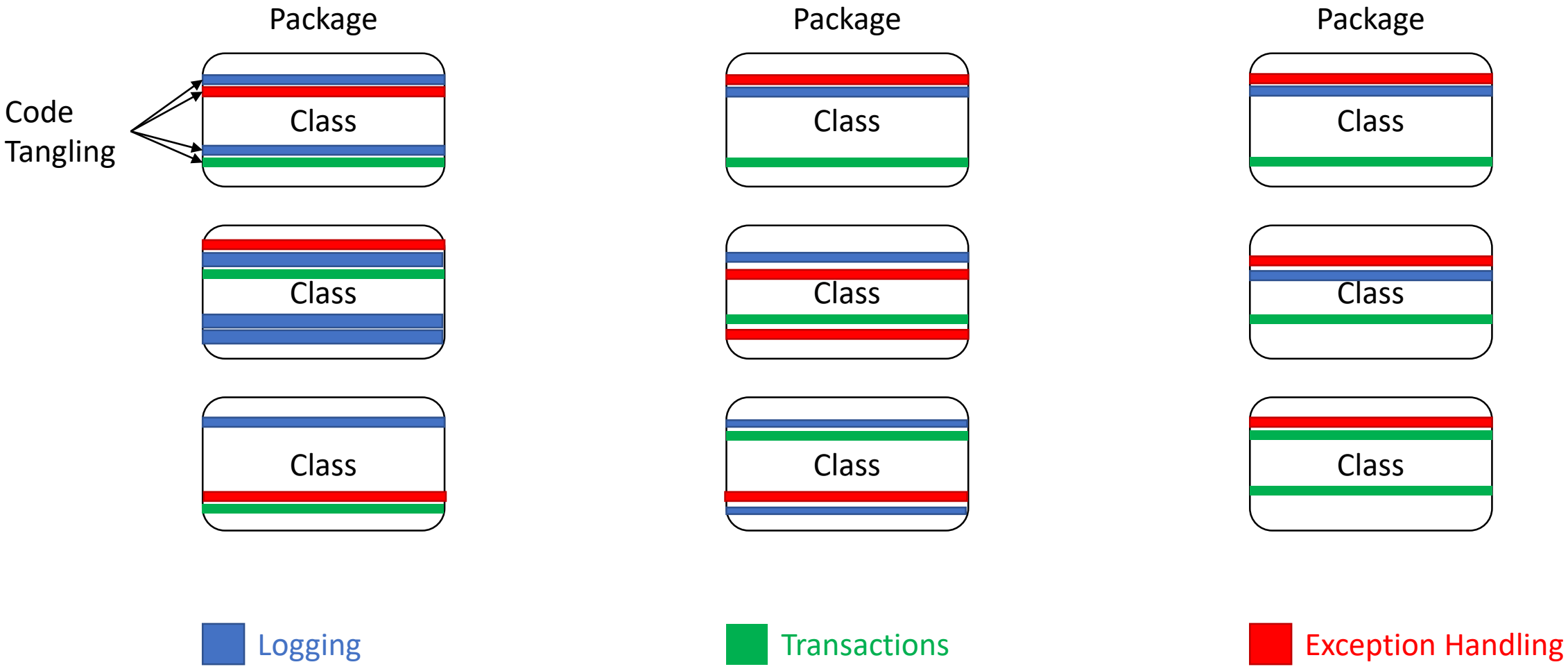
How AOP Works



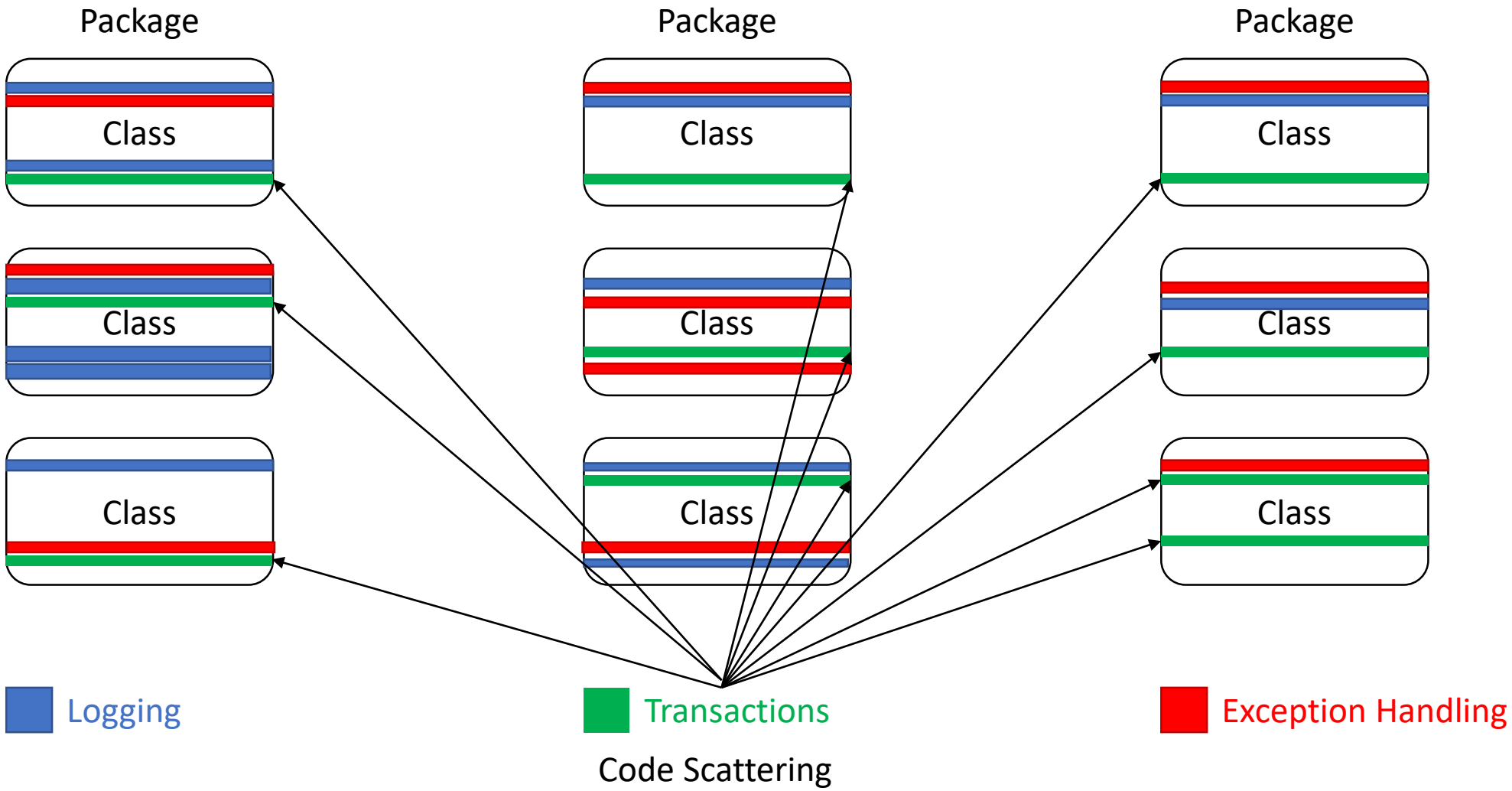
How AOP Works



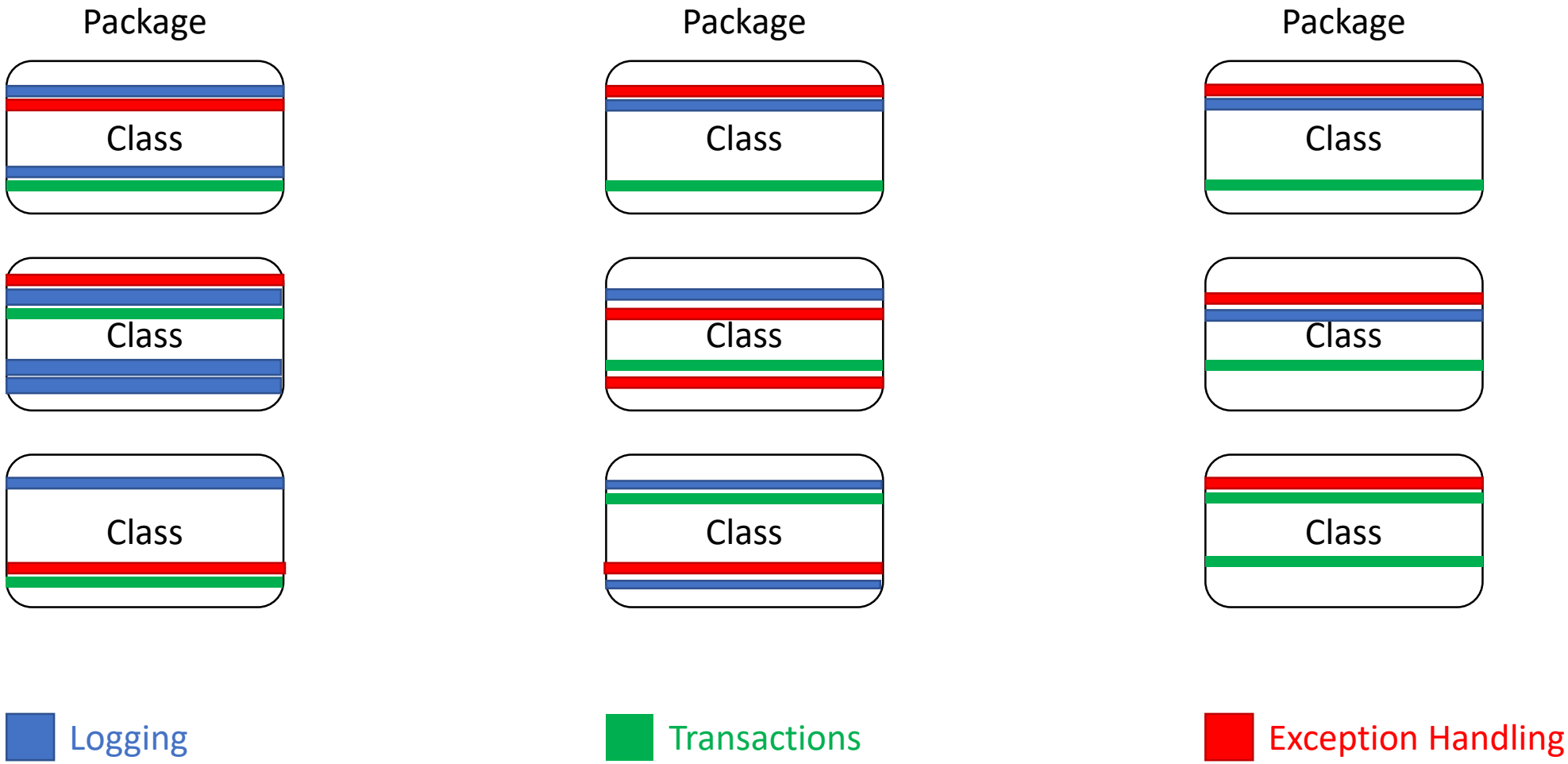
How AOP Works



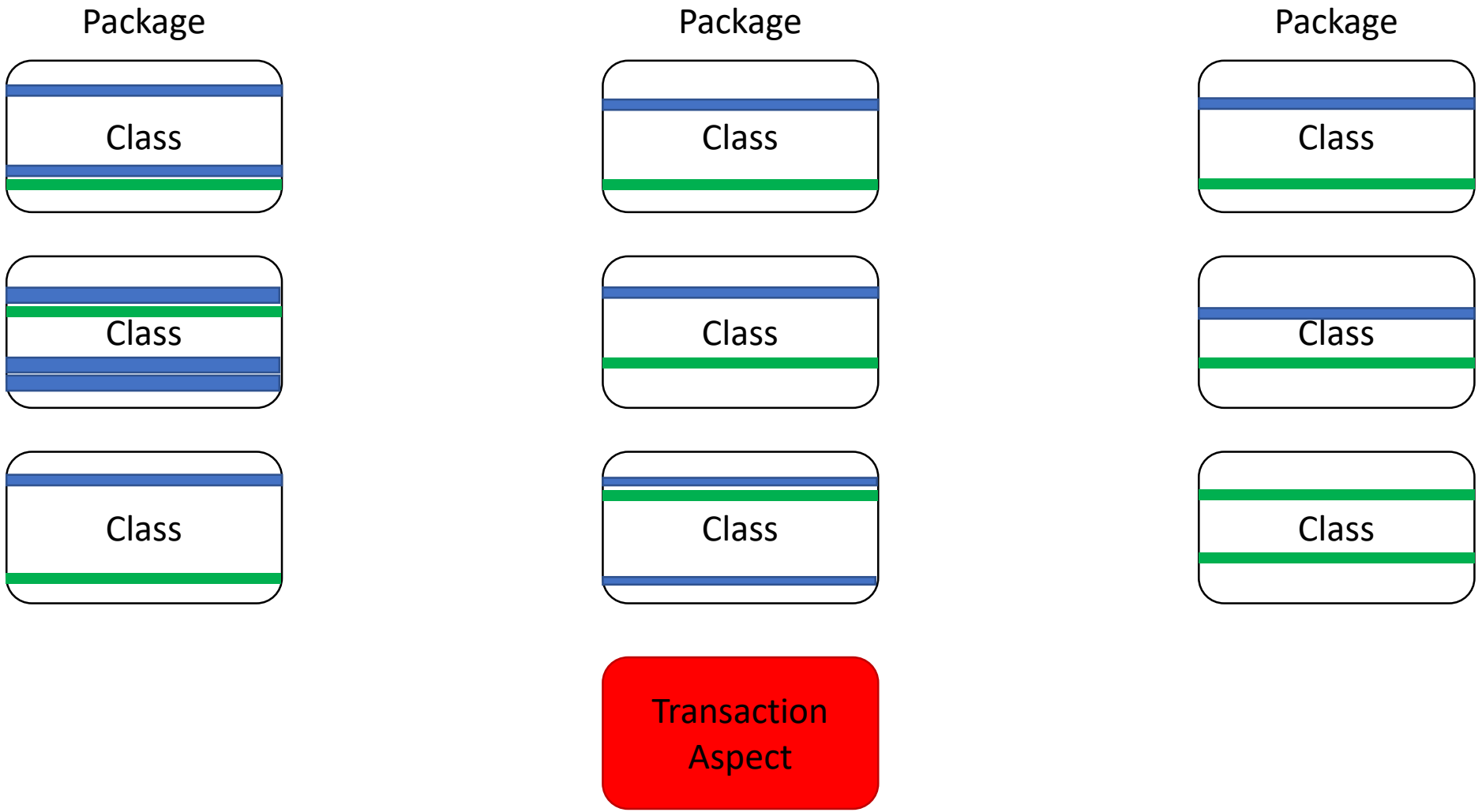
How AOP Works



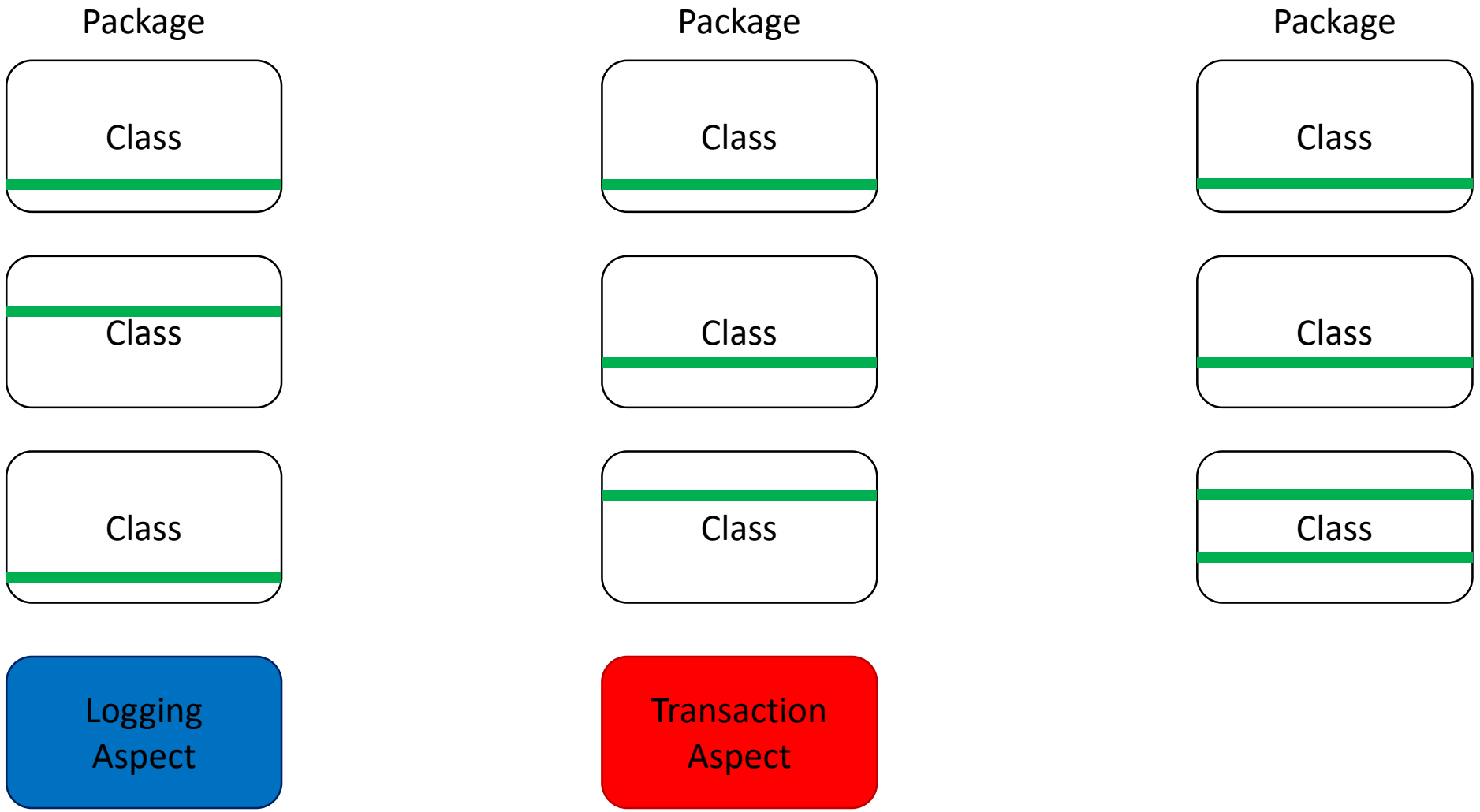
How AOP Works



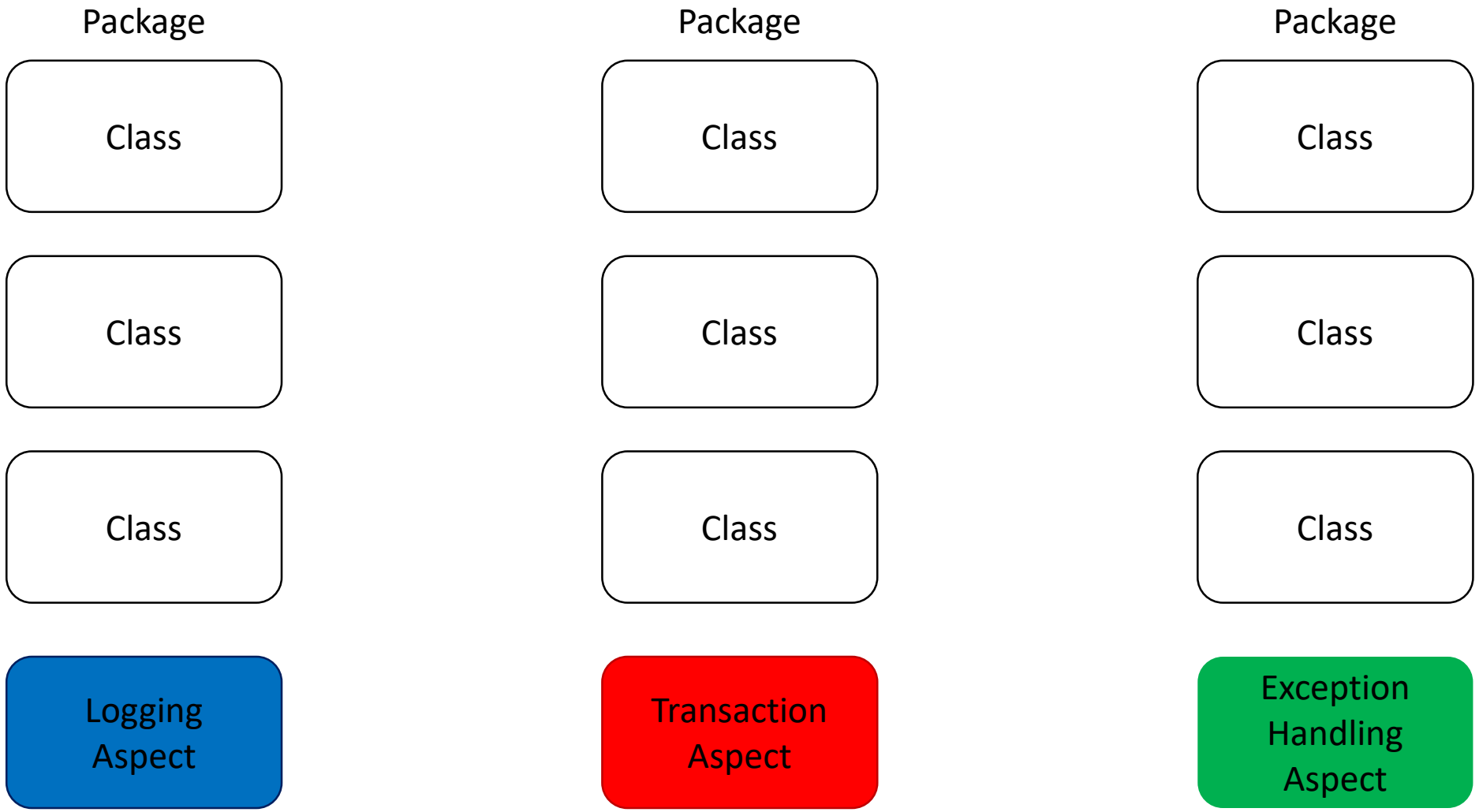
How AOP Works



How AOP Works



How AOP Works



Cross-Cutting Concerns

- Logging, Transactions and Exception Handling
- OOP: Different classes contain same/similar code

⇔

- AOP: centralized implementation

Summary

- Without AOP cross-cutting concerns are introduced across the code
- Mostly of a technical nature
  - Logging
  - Exception handling
  - Transactions
  - Security
  - …
- Result
  - Code tangling: Multiple concerns in each piece of code
  - Code scattering: Aspects are not implemented in one place
- Aspect-oriented programming resolves these issues

- Exercise - aop01

My First Aspect

- What is an Aspect?
- JoinPoint
- My First Aspect
- Weaving
- Spring configuration
- Summary

What is an Aspect?

- Aspect implements cross cutting concern
  - ⇔‘that scattered code’
  - ⇔Boiler Plate Code

Aspect	=	**Pointcut**	+	**Advice**
		Where the Aspect is applied (Set van Joinpoints)		What Aspect code is executed

JoinPoint

- Point in the control flow of a program
- Advices are provided with information about the join point
  - Class name
  - Method name

My First Aspect

- Logging

```
19    public void someMethod() {
          final String METHODNAME = "someMethod";
          logger.trace("entering " + CLASSNAME + "." + METHODNAME);
22        TransactionStatus tx = transactionManager
23                .getTransaction(new DefaultTransactionDefinition());
24        try {
25            // Business Logic
26        } catch (RuntimeException ex) {
27            logger.error("exception in " + CLASSNAME + "." + METHODNAME, ex);
28            tx.setRollbackOnly();
29            throw ex;
30        } finally {
31            transactionManager.commit(tx);
              logger.trace("exiting " + CLASSNAME + "." + METHODNAME);
33        }
34    }
```

My First Aspect

```java
@Component
@Aspect
public class TracingAspect {

    public boolean isEnteringCalled() {
        return enteringCalled;
    }

    Logger logger = LoggerFactory.getLogger(TracingAspect.class);

    boolean enteringCalled = false;

    @Before("execution(* *(..))")
    public void entering(JoinPoint joinPoint) {
        enteringCalled = true;
        logger.trace("entering "
                + joinPoint.getStaticPart().getSignature().toString());
    }

}
```

Weaving

- The process of adding Advices to the main program code at the configured JoinPoints

Spring configuration - XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.2.xsd
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.2.xsd">

    <aop:aspectj-autoproxy />

    <context:component-scan base-package="simpleaspect" />

</beans>
```

Spring configuration - Java

```java
package configuration;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;

@Configuration
@EnableAspectJAutoProxy
@ComponentScan(basePackages="simpleaspect")
public class SimpleAspectConfiguration {

}
```

Summary

- Aspect =
  - Advice – what code is executed +
  - Pointcut – where the code is executed
- Aspects are Spring Beans
  - @Aspect
- Advices are methods
- @Before annotation
  - With pointcut expression
- Activate AOP
  - XML or
  - Java Config

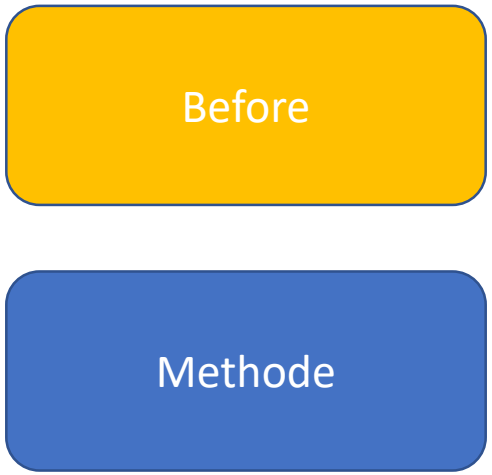
- Exercise - aop02

Advices

- So far: “Before” Advices
- There’s more

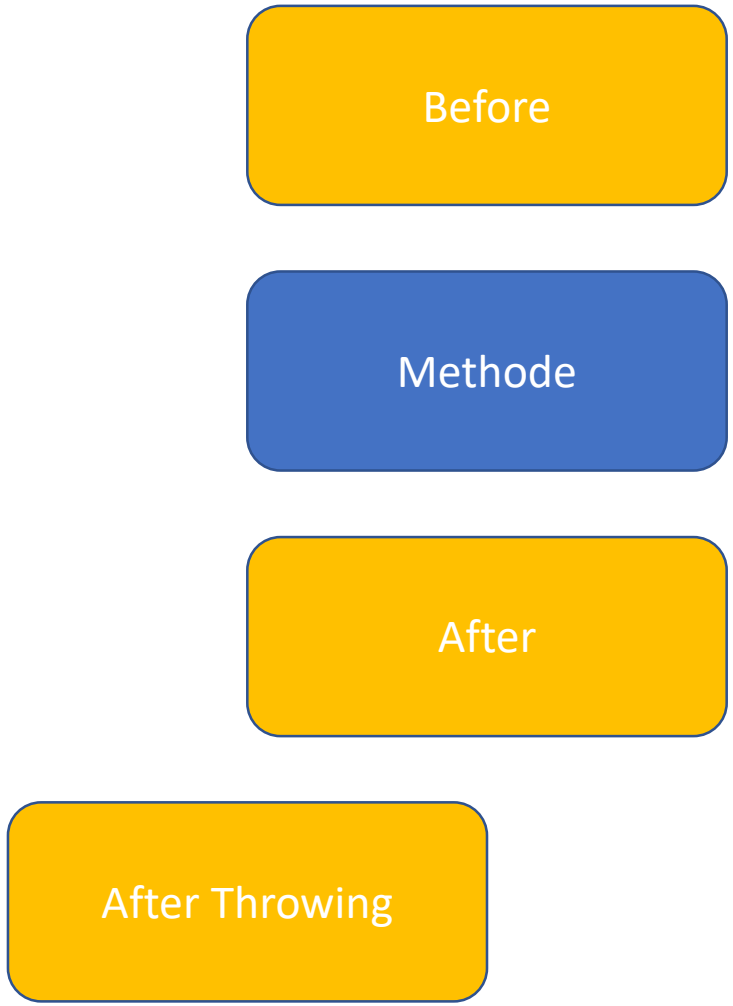
Advices – Types of Advices

Advices – Types Advices

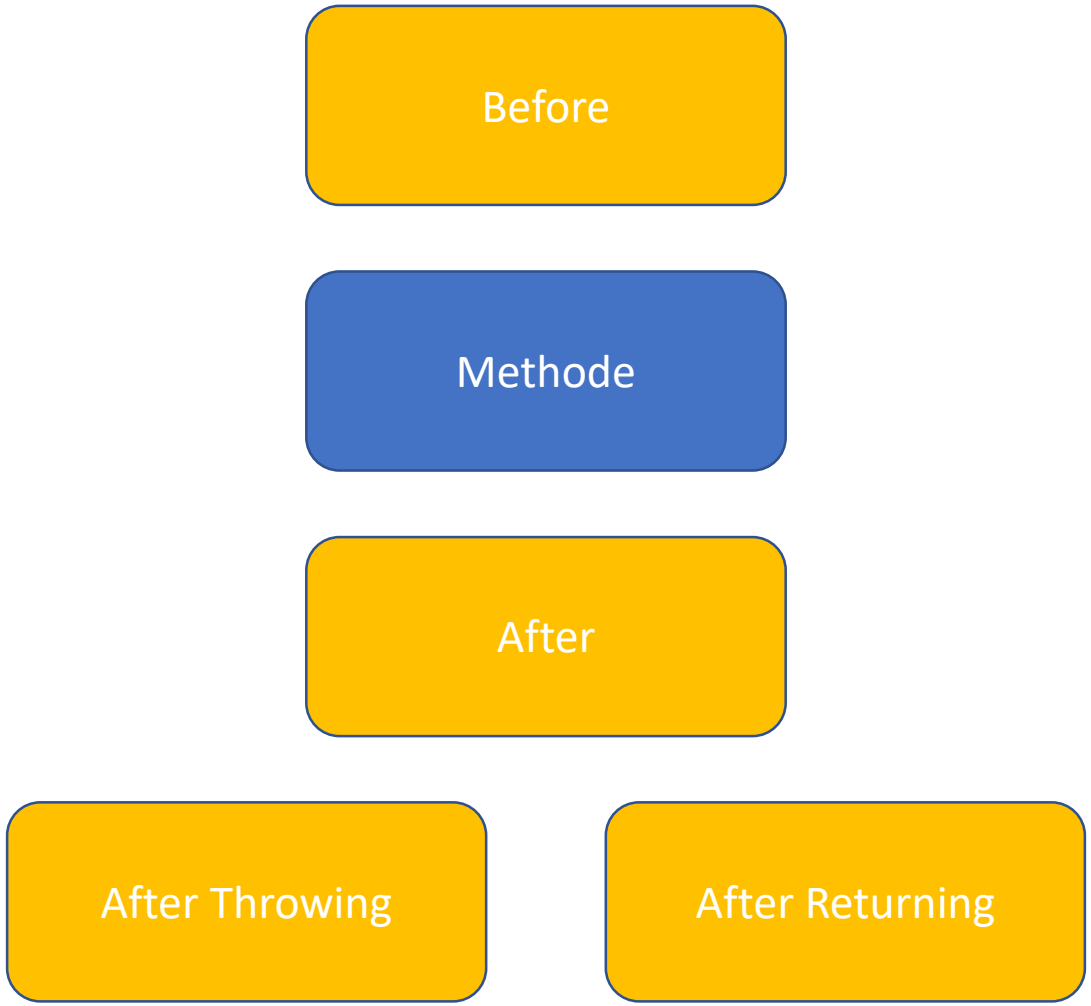


Advices – Types Advices

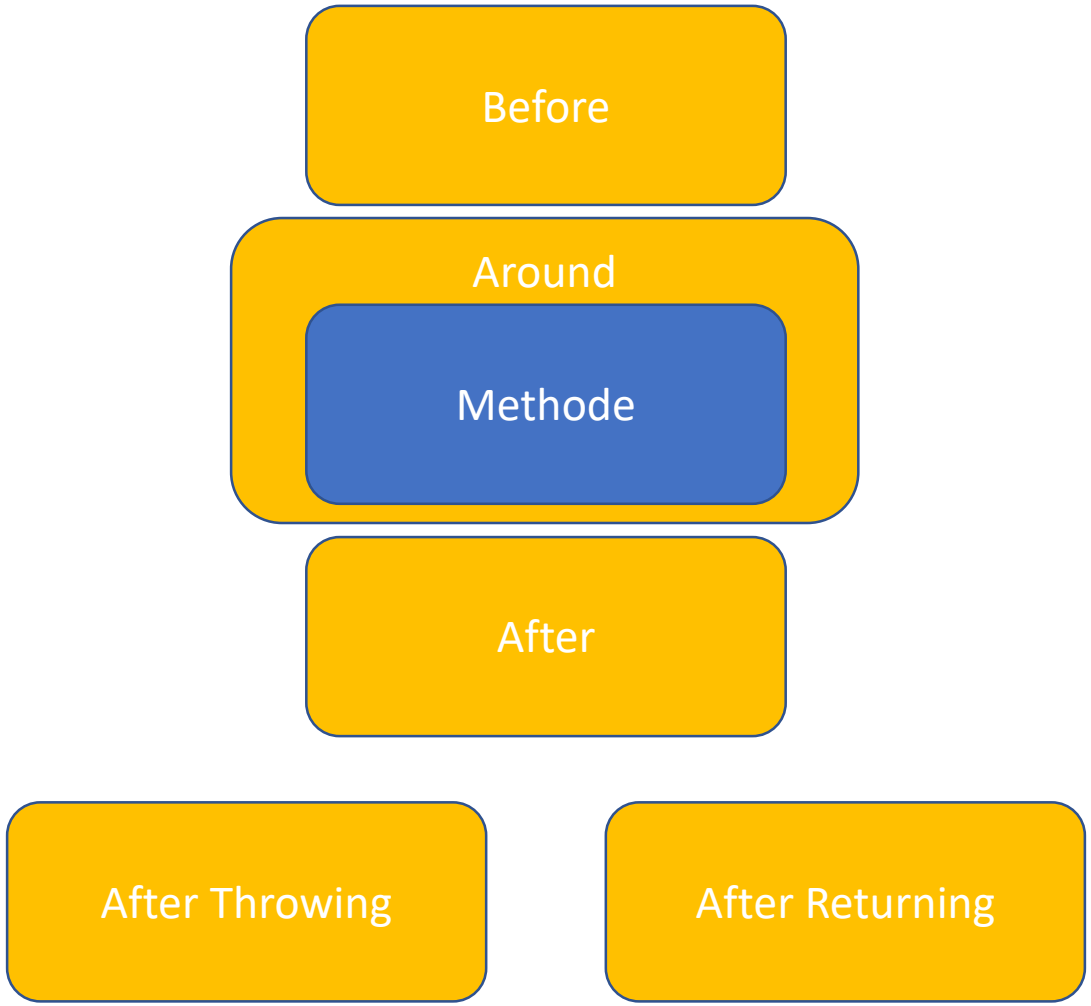
Advices – Types Advices



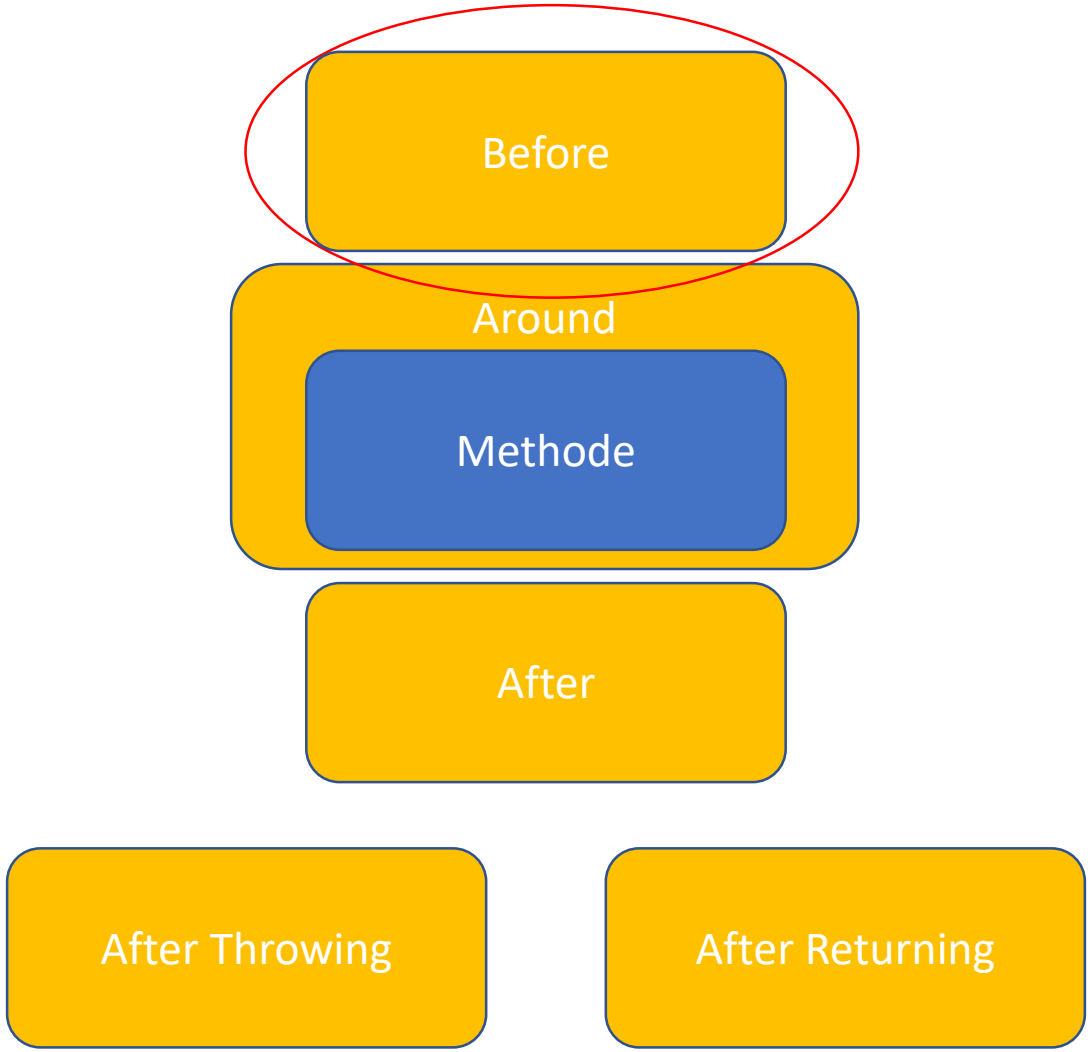
Advices – Types Advices



Advices – Types Advices



Advices – Before



Advices - Before

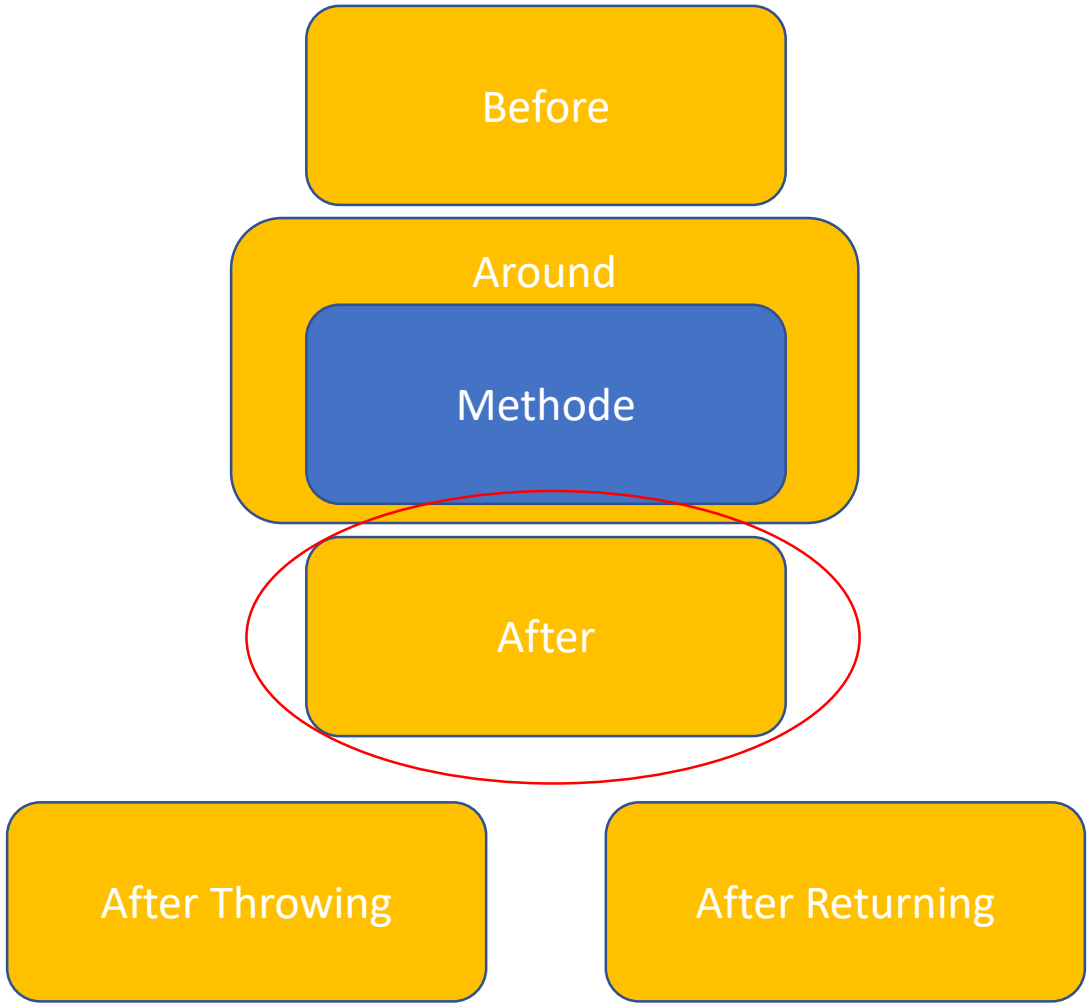
- Execution BEFORE the method
- Exception

  => prevents method execution

  => propagated to caller

```
26    @Before("execution(void someMethod())")
27    public void entering(JoinPoint joinPoint) {
28      beforeCalled = true;
29      logger.trace("entering "
30          + joinPoint.getStaticPart().getSignature().toString());
31    }
```

Advices – After



Advices – After

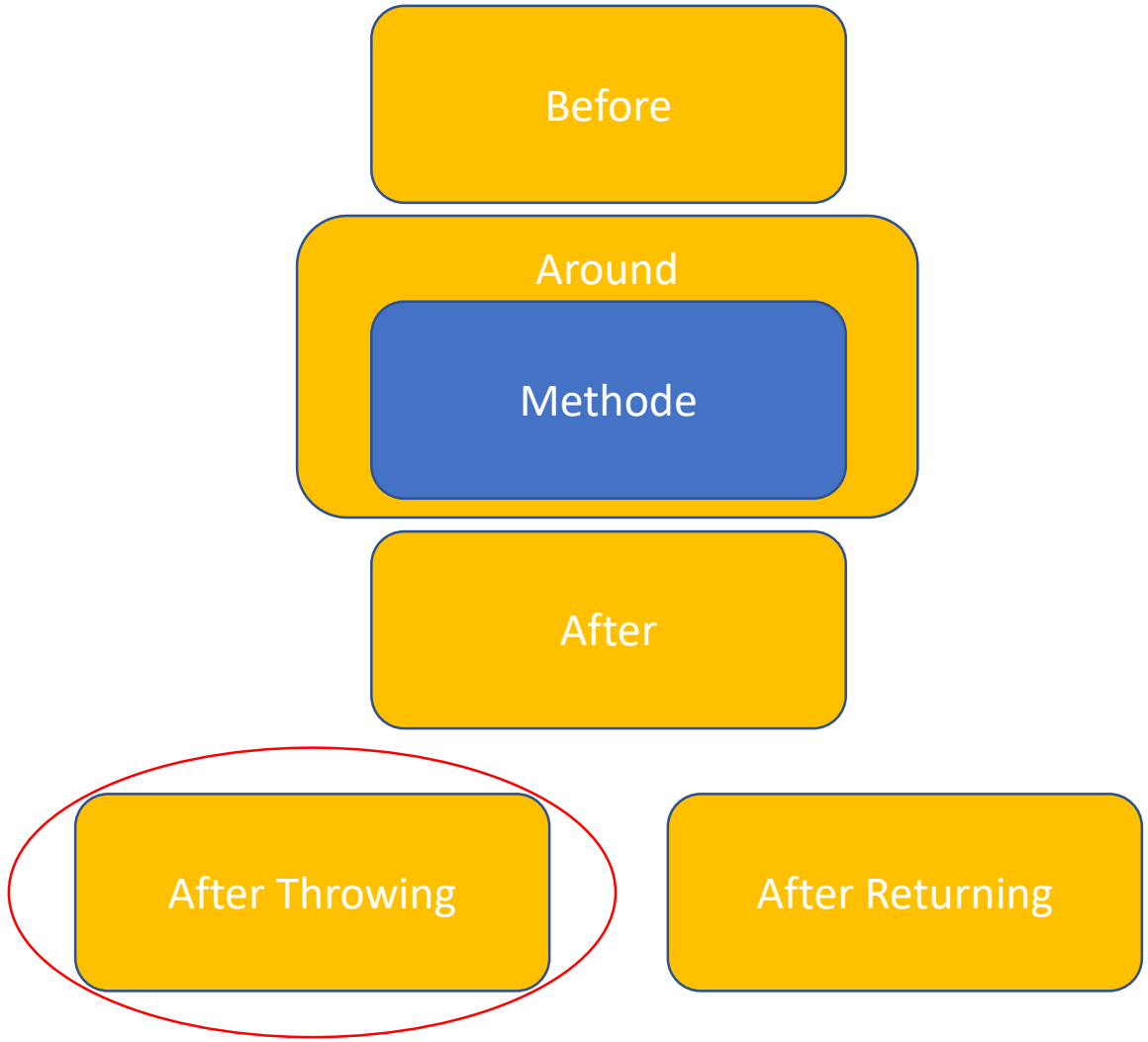
- Execution AFTER the method
- Exception

  => could have been thrown

  => or method could have been executed successfully

```
26  @After("execution(* *(..))")
27   public void exiting(JoinPoint joinPoint) {
28      afterCalled = true;
29      logger.trace("exiting "
30              + joinPoint.getSignature());
31      for (Object arg : joinPoint.getArgs()) {
32         logger.trace("Arg : " + arg);
33      }
34   }
```

Advices – After Throwing



Advices – After Throwing

- Execution AFTER the method THREW an exception
- Exception

  => propagated to caller

  => can be accessed from code

  => Type Safe: e.g. only executed if a RuntimeExceptionis thrown

```
25  @AfterThrowing(pointcut = "execution(void throwsRuntimeException())", throwing = "ex")
26  public void logException(RuntimeException ex) {
27    afterThrowingCalled = true;
28    logger.error("Exception ", ex);
29  }
```

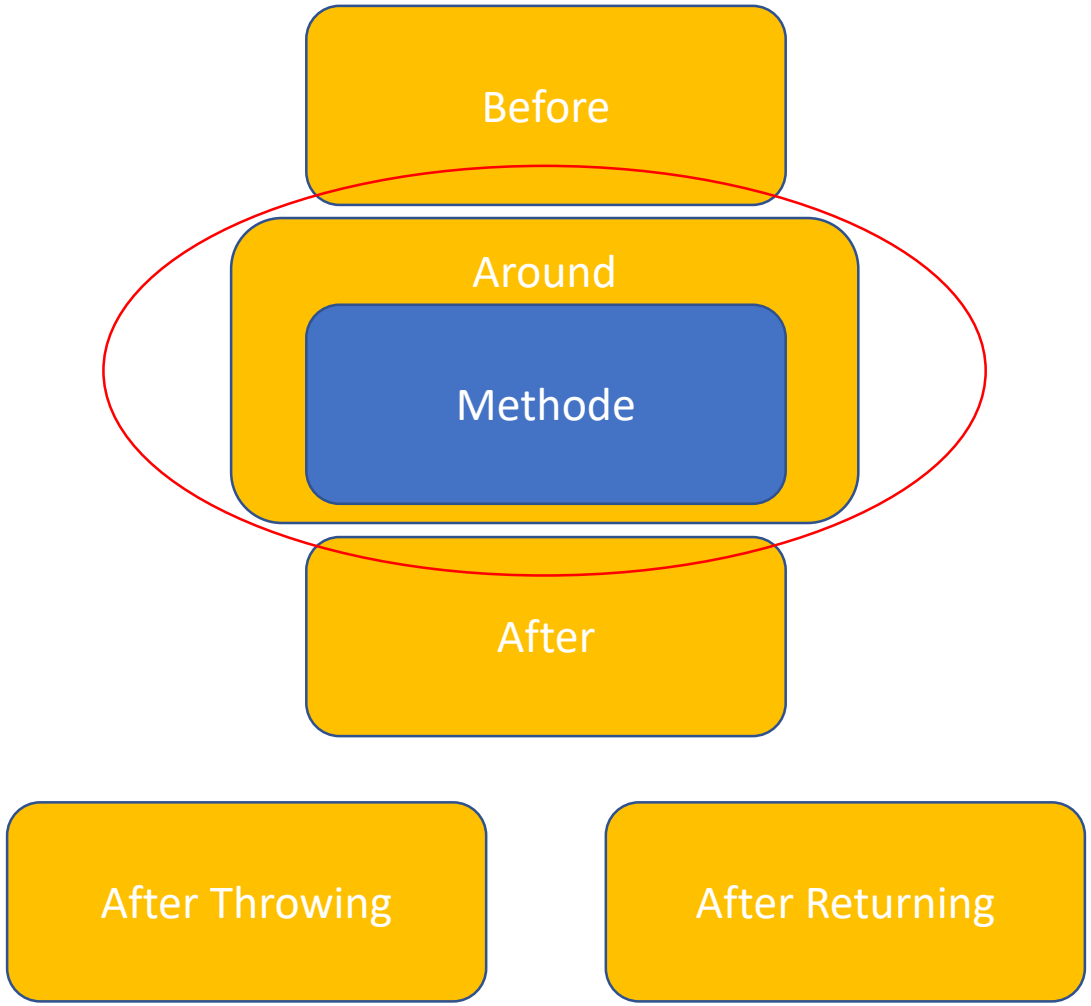
Advices – After Returning

Advices – After Returning

- Execution AFTER the method RETURNED successfully
- Result can be accessed from code
- Type Safe: e.g. only called if a String is returned

```
25    @AfterReturning(pointcut = "execution(* *(..))", returning = "string")
26     public void logResult(String string) {
27         afterReturningCalled = true;
28         logger.trace("result " + string);
29     }
```

Advices – Around



Advices – Around

- Wraps AROUND the method

- Prevent method execution
- … even without throwing an exception (<> Before Advice)

=> only advice that can catch exceptions

=> only advice that can modify return value

- Current method call is passed to the Advice
- ProceedingJoinPoint
- Can be executed or skipped

Advices – Around

```
21      @Around("execution(* *(..))")
22      public Object trace(ProceedingJoinPoint proceedingJP ) throws Throwable {
23          String methodInformation =
24                  proceedingJP.getStaticPart().getSignature().toString();
25          logger.trace("Entering "+methodInformation);
26          called=true;
27          try {
28              return proceedingJP.proceed();
29          } catch (Throwable ex) {
30              logger.error("Exception in "+methodInformation, ex);
31              throw ex;
32          } finally {
33              logger.trace("Exiting "+methodInformation);
34          }
35      }
```

Advices – Around

- Most powerful advice

=> can be used instead of Before and After

=> powerful but complex

- With great power comes great responsibility

Advices

- Exercise - aop3

Pointcuts

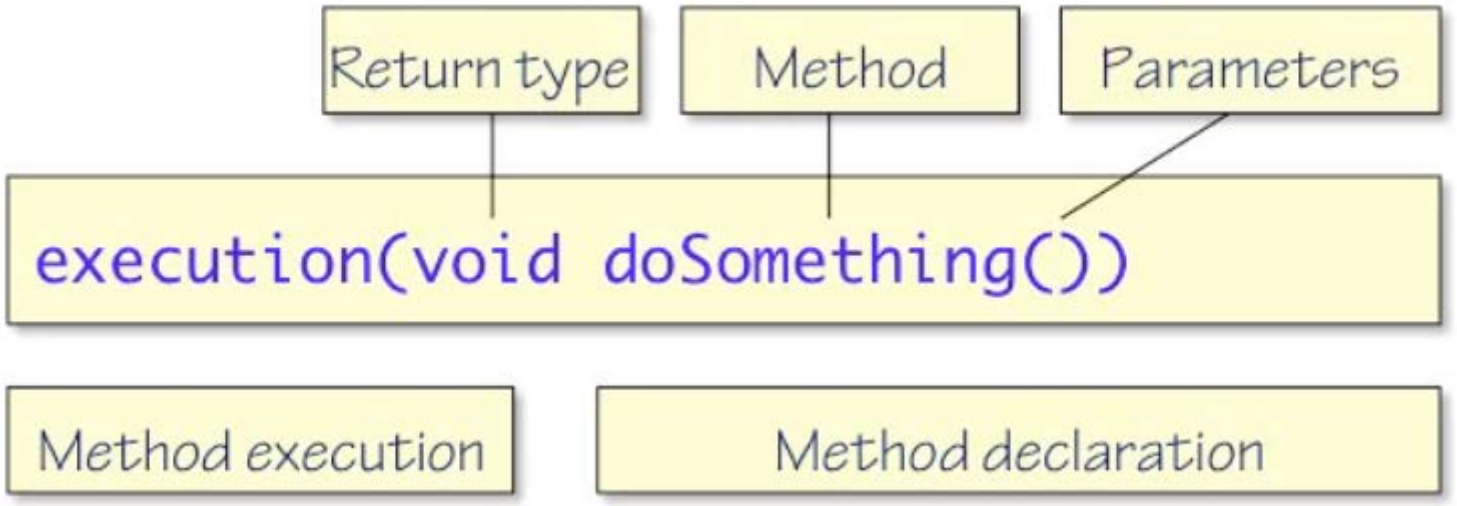
- So far: aspects on specific methods or all methods
- There’s more

Pointcuts

- Pointcut Expressions
- On Packages and Classes
- On Annotations
- On Spring Beans
- Combining Pointcuts
- Pointcut Reuse
- Summary

Pointcut Expressions

- @PointCut
- Defines where an aspect is added to the program flow
- Pointcut



- Wildcards, wildcards, wildcards…

Pointcut Expressions

execution(hello())*

- Method: hello
- Parameters: none
- Return type: any

execution(hello(int, int))*

- Method: hello
- Parameters: 2 x int
- Return type: any

Pointcut Expressions

execution(hello(*))*

- Method: hello
- Parameters: 1 parameter of any type
- Return type: any

execution(hello(..))*

- Method: hello
- Parameters: any number of parameters of any type
- Return type: any

On Packages and Classes

execution(int com.axxes.services.Service.hello(int))

- Method: hello
  - Class: Service
  - Package: com.axxes.services
- Parameters: 1 parameter of type int
- Return type: int

execution(com.axxes..*Service.*(..))*

- Method: any
  - Class: with class name ending with Service
  - Package: com.axxes or sub package
- Parameters: any number of parameters of any type
- Return type: any

On Packages and Classes

execution(*.*(..))*

- Method: any
  - Class: any
  - Package: default
- Parameters: any number of parameters of any type
- Return type: any

execution(*..*.*(..))*

- Method: any
  - Class: any
  - Package: any package or sub package
- Parameters: any number of parameters of any type
- Return type: any

On Annotations

*execution(@com.axxes.annotations.Annotation * *(..))*

- Annotated method

execution((@ com.axxes.annotations.Annotation *).*(..))*

- Annotated class

On Spring Beans

*bean(*Service)*

- Bean name default: class name
- Beans definition
  - Java Config: @Bean method name
  - Annotation: parameter to @Component, @Service, @Repository
  - XML: name / id attribute of bean element

Combining Pointcuts

- Boolean operators: &&, ||, !, …

execution(service.*.*(..)) || execution(* repository.*.*(..))*

Pointcut Reuse

- Problem: Pointcut expression repeated every time a pointcut is used
  => Solution: Around Advice on annotated stub

```
public class MyPointcuts {

        @Pointcut("execution(@annotation.Trace * *(..))")
        public void traceAnnotated() {
        }
        Use @Pointcut annotation
}       Note: Method's purpose is just to be annotated
```

```
@Around("MyPointcuts.traceAnnotated()")
public void trace(ProceedingJoinPoint proceedingJP )
 throws Throwable {
}
```

Summary

- Pointcuts: Where should an advice be added?
- Pointcut Expressions, use of wildcards
- Pointcuts through:
  - Methods (within class on classpath)
  - Annotations
  - Bean names
- Combine and reuse Pointcuts

- Exercise – aop04

AOP and Architecture

- Problems with Architecture
- Criticism on AOP
- Architecture in terms of Aspects, a plan
- Summary

Problems with Architecture

- Architecture in documents
  - Not read
  - Not followed
  - Lots of boilerplate code

< = >

- Architecture in code => AOP to the rescue!

Problems with Architecture

- For each call to a service
  - Call must be traced
  - Exceptions must be logged
- For each call to a repository
  - Call must be traced
  - Performance must be traced
  - Exceptions must be logged

=>

- Specific behavior should be added
  - Tracing, exception handling, …
- … to specific parts of the the architecture
  - Repositories, services etc

Criticism on AOP

- Control flow is obscured
- Pointcut may depend on runtime condition

=>

- AOP adds random code to random parts of the system
- It is hard to reason about the system
- What happens when?

<=>

Not true when AOP is used properly

Architecture in terms of Aspects, a plan

- Aim: Add behavior to parts of the architecture using AOP

- Methodology:
  - Step 1 : Define architecture as Pointcuts
  - Step 2 : Define behavior using Advices
  - Step 3 : Add Advices to correct Pointcuts

- Result: No more technical boiler plate

Summary

- AOP adds behavior to specific parts of the system

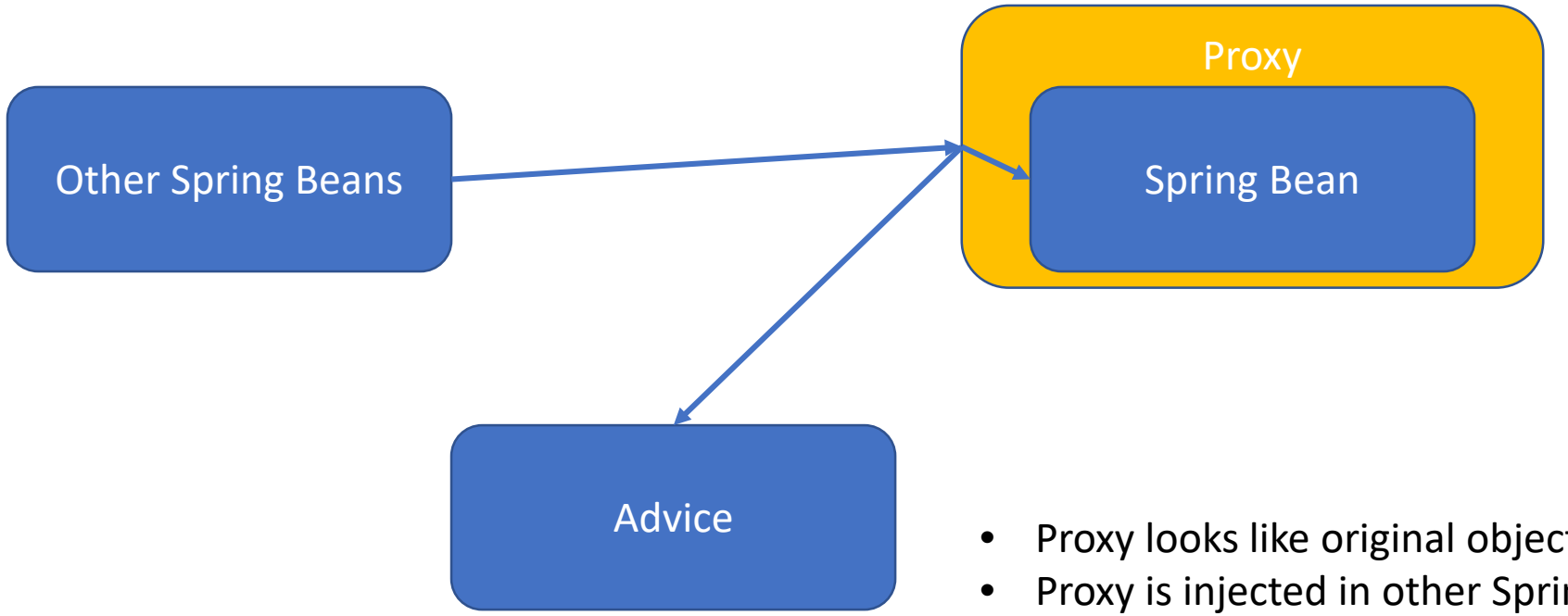
- Pointcuts can express architecture
- Should be added to the project early

- No Exercise -> LAB at the end of the session!

AOP in Spring (How It Works)

- Proxies
- Local Method Calls
- Pitfalls of proxies
- How proxies are implemented
- Limits of proxies
- Summary

Proxies



- Proxy looks like original object
- Proxy is injected in other Spring Beans
- Interfaces -> Dynamic Proxies
- Otherwise -> CGLIB generated sub classes

Local Method Calls

- Call never reaches proxy
- => No call to Advice

Pitfalls of proxies

```java
package com.axxes.demo;

import org.springframework.transaction.annotation.Transactional;

public class TxExample {

    @Transactional
    public void transactionalMethod() {
    }

    public void callsTransactionalMethod() { transactionalMethod(); }

}

```

- @Transactional is implemented using Spring AOP
- @Transactional will not be evaluated when called via callsTransactionalMethod()

How proxies are implemented

- Dynamic Proxies
  - Feature of the JDK
  - Allow dynamic method dispatch
  - For interfaces only
- CGLIB is used
  - Byte code instrumentation library
  - Subclass dynamically create subclass
  - Subclass implements the proxy
  - When no interface is implemented
  - Or if proxy-target-class is set to true

Limits of proxies

- Work only on public methods

-> No protected, private

- Works only on methods calls from outside
- Spring Dependency Injection makes it transparent

Summary

- Spring AOP uses proxy based AOP
  - CGLIB (subclasses)
  - or Dynamic Proxies (interfaces)
- DI makes proxies transparent
- Beware: Call on the local object will not go through the proxy
- Can write code to create proxies

- Exercise – aop05

AOP in Spring (Spring Aspect Library)

- Spring offers some Aspects OTB
- Apply to most common usage of Aspects
  - Caching
  - Exception handling
  - Logging
  - Transactions
  - …
- High Quality Code (Open Source)
- Save time & effort

AOP in Spring (Spring Aspect Library)

- Logging

```
19    public void someMethod() {
          final String METHODNAME = "someMethod";
          logger.trace("entering " + CLASSNAME + "." + METHODNAME);
22        TransactionStatus tx = transactionManager
23                .getTransaction(new DefaultTransactionDefinition());
24        try {
25            // Business Logic
26        } catch (RuntimeException ex) {
27            logger.error("exception in " + CLASSNAME + "." + METHODNAME, ex);
28            tx.setRollbackOnly();
29            throw ex;
30        } finally {
31            transactionManager.commit(tx);
              logger.trace("exiting " + CLASSNAME + "." + METHODNAME);
33        }
34    }
```

The Spring Context

```
10    <context:component-scan base-package="com.axxes" />
11
12    <bean id="debugInterceptor" class="org.springframework.aop.interceptor.DebugInterceptor" />
13
14    <aop:config>
15        <aop:advisor advice-ref="debugInterceptor"
16            pointcut="SystemArchitecture.Service() || SystemArchitecture.Repository()" />
17    </aop:config>
18
```

Other Tracing Aspects

- Package org.springframework.aop.interceptor
  - CustomizableTraceInterceptor: Can customize the trace output
  - SimpleTraceInterceptor: Basic information
  - DebugInterceptor: Full information
  - …
  - PerformanceMonitorInterceptor: Performance measurement in ms

=>

There’s plenty available OTB

Other Aspects

- Caching, Exception handling, Logging, Transactions,… and Security

- AsyncExecutionInterceptor to process method calls asynchronously
- ConcurrencyThrottleInterceptor to limit the number of threads in an object

- …

AOP in Spring (Spring Aspect Library)

- Transactions

```
19    public void someMethod() {
20        final String METHODNAME = "someMethod";
21        logger.trace("entering " + CLASSNAME + "." + METHODNAME);
          TransactionStatus tx = transactionManager
                  .getTransaction(new DefaultTransactionDefinition());
24        try {
25            // Business Logic
26        } catch (RuntimeException ex) {
27            logger.error("exception in " + CLASSNAME + "." + METHODNAME, ex);
              tx.setRollbackOnly();
29            throw ex;
30        } finally {
              transactionManager.commit(tx);
32            logger.trace("exiting " + CLASSNAME + "." + METHODNAME);
33        }
34    }
```

The Spring Context

```
<context:component-scan base-package="com.axxes" />

<bean id="transactionManager" class="com.axxes.transaction.StubPlatformTransactionManager" />

<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="find*" read-only="true" />
        <tx:method name="*" />
    </tx:attributes>
</tx:advice>

<aop:config>
    <aop:advisor advice-ref="txAdvice"
        pointcut="SystemArchitecture.Service()||SystemArchitecture.Repository()" />
</aop:config>
```

Summary

- Spring provides a library of aspects
- Include tracing, transactions…
- Using pointcuts enterprise services can be transparently added to the business logic

- Exercise – aop06a (tracing) and aop06b (transactions)

Aspects, the real world

- Spring’s aspect library contains some interesting aspect
- But can we do more?
- Which challenges you suppose can be solved with AOP?

Example 1:Retry

- Retry a method call if it fails
- Can help to resolve transient failures

- Exercise – aop07a

Retry

- Don’t use
  - If a service is not accessible:
    - Calls are buffered
    - Calls pile up
  - The service comes up again
    - immediately swamped with requests
    - go down again
  - Not very smart
  - Especially when calling external systems

Example 2: Circuit Breaker

- If an error occurs the circuit breaker breaks the circuit
  - service is not called anymore
  - exception is immediately forwarded to the caller
- After a while the service is called again
  - retry after some time
  - or retry after number of calls
- If the call succeeds the circuit breaker is closed again

- Exercise – aop07b

Circuit Breaker

- Usable, but room for improvement
  - After a failure slowly ramp up
  - Incorporate a STRATEGY
    - isAvailable() : boolean
    - reportSuccess()
    - reportFailure()
    - …
  - Use a fallback
    - Default value
    - Simplified service
    - Cache

Example 3: JPA / JDBC

- Problem: JPA contains a cache
- Changes are not immediately propagated to the database
- JDBC calls might see incorrect data

=>

Solution: Flush EntityManager every time a JDBC call happens

Exercise – aop07c

Example 4: Context Security

- Customer may only see his Account
- Account is a domain object – no Spring Bean

- Exercise – aop07d

Summary

- Manipulate how a method is called
  - Exception Handling
  - Record calls
  - Filter
  - Adjust return values
  - …

Aspects - LAB

- Free exercise based on the project from the first 2 days