CS544 EA

Aspect Oriented Programming

Overview

CS544 Enterprise Applications

Week	Session	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
Week 1	a.m.	Aspect Oriented Programming	Transactions	Spring MongoDB	Web Interface	In Class Review	**Exam**
	p.m.					Review	
	Eve	Homework	Homework	Homework	Homework		
Week 2	a.m.	Messaging	Scheduling, Events, Logging	Monitoring	Testing	In Class Review	**Exam**
	p.m.					Review	
	Eve	Homework	Homework	Homework	Homework		
Week 3	a.m.	Security	AI Theory	Tool Calling	MCP & RAG	In Class Review	**Exam**
	p.m.					Review	Project
	Eve	Homework	Homework	Homework	Homework		
Week 4	a.m.	Project	Project	Project	Presentations		
	p.m.						
	Eve						

Wholeness

- In this lecture we will discuss aspect oriented programming (AOP), one of the primary techniques utilized by frameworks like Spring, along with Dependency Injection and Templates
- AOP lets us write code once and then apply it to many locations, lets us do less and accomplish more.

Overview

- In this lesson we will discuss:
  - AOP Intro
  - AOP Terminology
  - Types of Advice
  - JoinPoint
  - Pointcut Expression Language
  - Data inside Advice
  - Proxy Weaving Problems


MAHARISHI
HIGHER EDUCATION
FOR HIGHER CONSCIOUSNESS
INTERNATIONAL UNIVERSITY

CS544 EA

Spring

Aspect Oriented Programming

Aspect Oriented Programming

- Aspect Oriented Programming is a way to provide separation of concerns, creating modularity
  - Certain features are hard to cleanly separate
  - Certain things you need all the time (like logging)
  - Certain code you need before and after (like Transactions)

- These are what AOP calls **cross cutting concerns**
  - You need it across the application

Cross Cutting Concerns

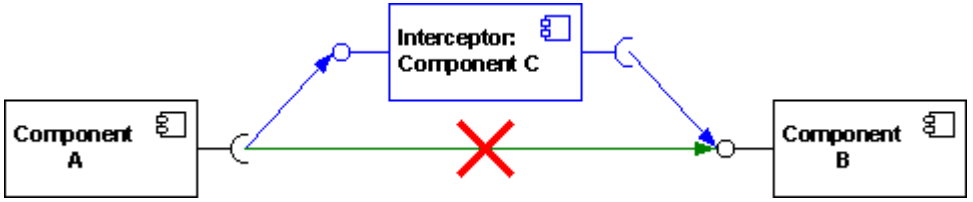
- Cross Cutting Concerns are usually scattered or tangled throughout your code

- Logging is scattered
  - Single thing, copy / pasted everywhere
- Transactions are tangled
  - A part before, a part afterwards every time you use the DB

- How can you **write code like this once** and re-use everywhere?

Interceptor

- AOP with Spring uses the **interceptor pattern**



- Because of IoC and DI, Spring can inject something else (a proxy) in between
  - Proxy then calls the desired code (advice) before and / or after

Proxies

- Spring creates proxies using either JDK dynamic proxies or using CGLIB
  - **JDK proxies** are made by implementing the same interface(s) that your object implements
  - **CGLIB proxies** are made as a subclass
    - Using Objenesis – therefore no super() constructor call!

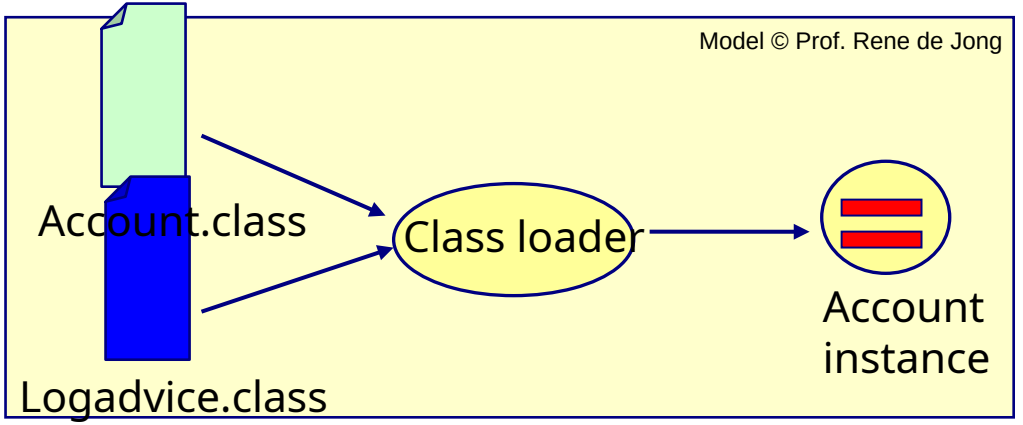
JDK > CGLIB

- Spring **prefers JDK proxies**
  - Faster to create, and you should **`P2I`** anyway
  - If your class implements 1 or more interfaces
    - Spring will make a JDK proxy based on your interface(s)
    - Leaving out methods not represented on an interface

- If there are no interfaces Spring uses CGLIB
  - You **can force Spring to always use CGLIB** proxies
    - Then you don’t have to write all those interfaces (no P2I)

AOP with ByteCode

- AOP is also possible with **ByteCode Enhancement**
  - Instead of an proxy based interceptor (will not cover in this course)
- The AspectJ project provides tools for this
  - Can be configured to be used with Spring
  - AspectJ combines your .class file with the advice .class file to create a new .class file that contains both




MAHARISHI
HIGHER EDUCATION
FOR HIGHER CONSCIOUSNESS
INTERNATIONAL UNIVERSITY

CS544 EA

Aspect Oriented Programming

AOP: Terminology

Advice

- The **implementation of the cross cutting** concern is called advice.
  - Advice is implemented as a method in a class

```
@Aspect
@Configuration
public class LogAspect {
	private static final Logger logger = LogManager.getLogger(LogAspect.class.getName());

	@Before("execution(* cs544.spring40.aop.terms.CustomerService.*(..))")
	public void logBefore(JoinPoint joinpoint) {
		logger.warn("About to exec: " + joinpoint.getSignature().getName());
	}

	@After("execution(* cs544.spring40.aop.terms.CustomerService.*(..))")
	public void logAfter(JoinPoint joinpoint) {
		logger.warn("Just execed: " + joinpoint.getSignature().getName());
	}
}
```

Advice Method

Another Advice Method

JoinPoint

- JoinPoint is a **specific point** (method) in code
  - **Where the advice will be applied**

```
@Service
public class CustomerService {

	public void doSomething() {
		System.out.println("something");
	}

	public void otherThing() {
		System.out.println("other");
	}
}
```

doSomething() is a JoinPoint

otherThing() is a JoinPoint

Target

- While executing an advice, the **object on which the joinpoint is** located is called the target
  - Here target is an object of the CustomerService class

```
@Aspect
@Configuration
public class LogAspect {
	private static final Logger logger = LogManager.getLogger(LogAspect.class.getName());

	@Before("execution(* cs544.spring40.aop.terms.CustomerService.*(..))")
	public void logTargetBefore(JoinPoint joinpoint) {
		logger.warn("About to exec a method on: " + joinpoint.getTarget());
	}

	@After("execution(* cs544.spring40.aop.terms.CustomerService.*(..))")
	public void logTargetAfter(JoinPoint joinpoint) {
		logger.warn("Just execed a method on: " + joinpoint.getTarget());
	}
}
```

```
09:35:04.004 About to exec a method on: cs544.spring40.aop.terms.CustomerService@7bd7d6d6
09:35:04.033 Just execed a method on: cs544.spring40.aop.terms.CustomerService@7bd7d6d6
```

Pointcut

- A Pointcut is a **collection of points**
  - Described in the Pointcut Expression Language

```
@Aspect
@Configuration
public class LogAspect {
	private static final Logger logger =  LogManager.getLogger(LogAspect.class.getName());

	@Before("execution(* cs544.spring40.aop.terms.CustomerService.*(..))")
	public void logBefore(JoinPoint joinpoint) {
		logger.warn("Method: " + joinpoint.getSignature().getName());
	}
}
```

This PointCut expression says that all methods of CustomerService are JoinPoints

Aspect

- Aspect is the combination of advice and pointcut
  - What (**advice**) should execute where (**pointcut**)

This class is an Aspect

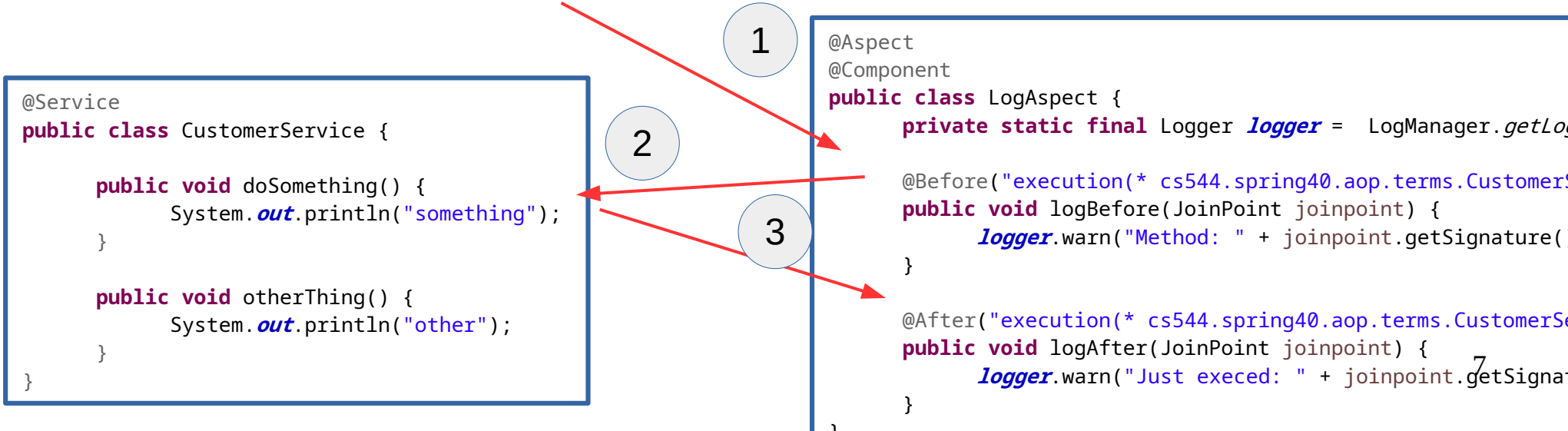
```
@Aspect
@Configuration
public class LogAspect {
	private static final Logger logger =  LogManager.getLogger(LogAspect.class.getName());

	@Before("execution(* cs544.spring40.aop.terms.CustomerService.*(..))")
	public void logBefore(JoinPoint joinpoint) {
		logger.warn("Method: " + joinpoint.getSignature().getName());
	}
	@After("execution(* cs544.spring40.aop.terms.CustomerService.*(..))")
	public void logAfter(JoinPoint joinpoint) {
		logger.warn("Just execed: " + joinpoint.getSignature().getName());
	}
}
```

Pointcut

Advice

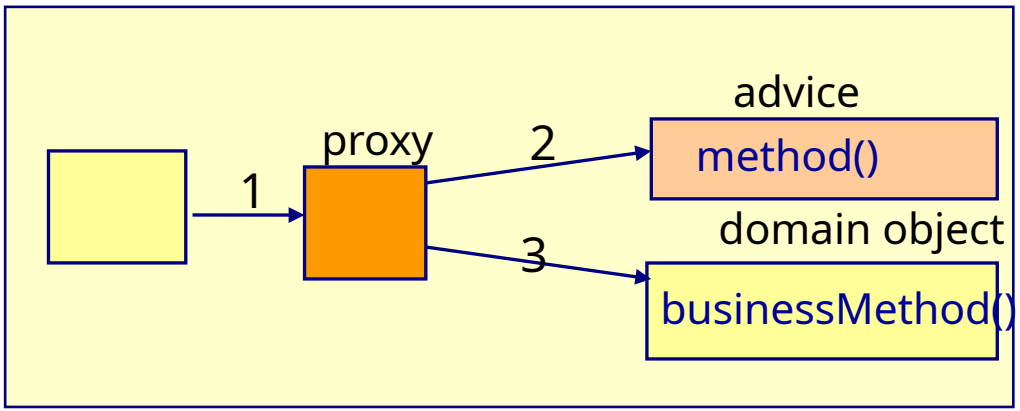
Weaving

- Weaving is seen at execution time
  - **Execution weaves** back and forth between advice and the actual method
  - For example, when calling **`doSomething()`** on **`CustomerService`**

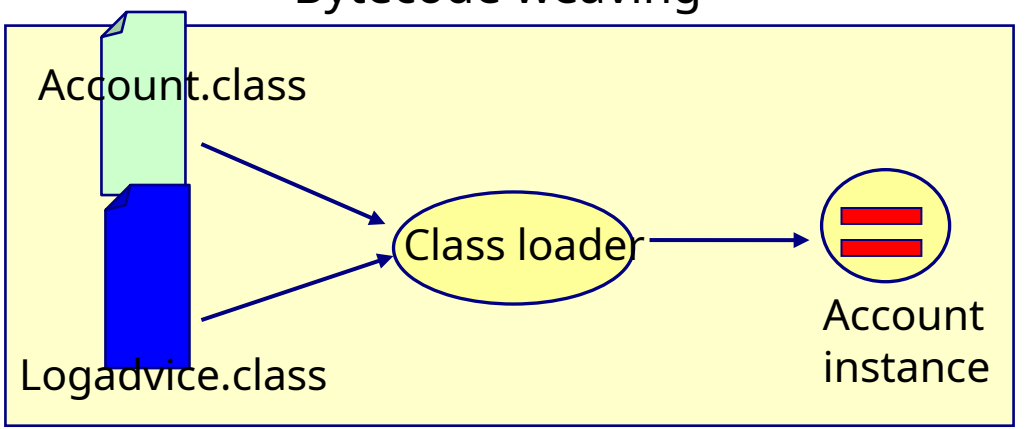


Weaving

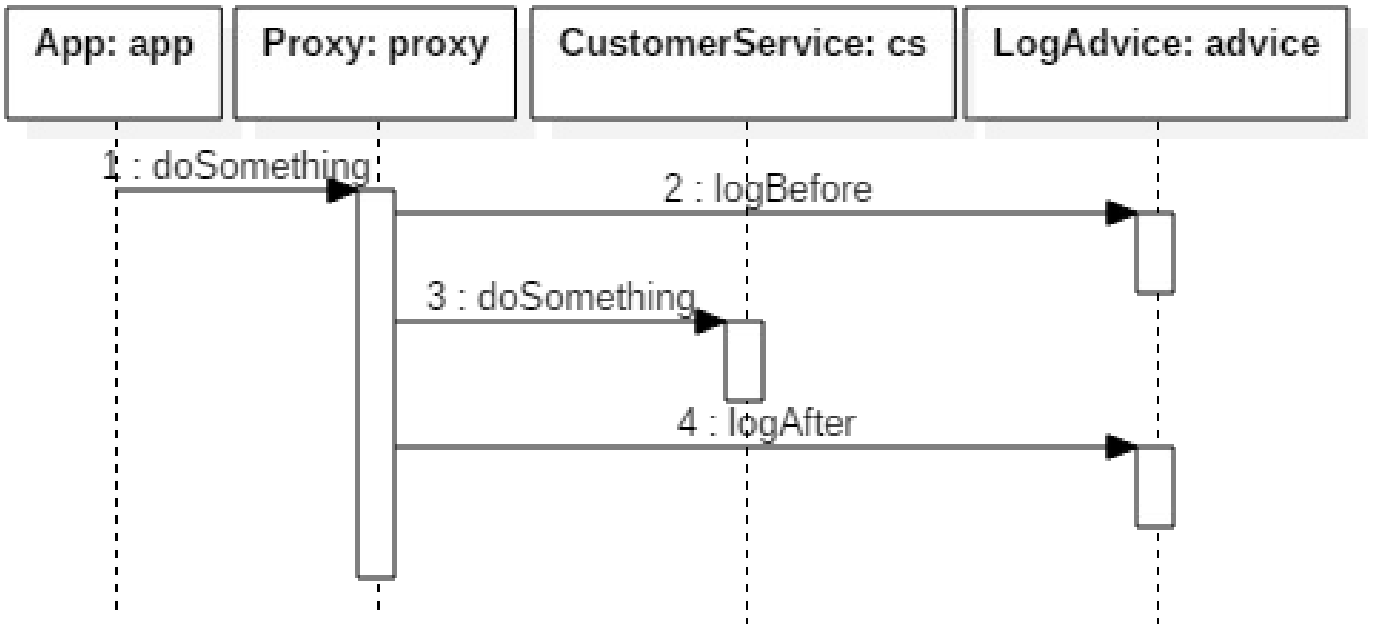
Proxy-based weaving



Bytecode weaving



Proxy Weaving Sequence Diagram



Full Example Code

```java
package cs544.spring40.aop.terms;

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

@Configuration
@ComponentScan("cs544.spring40.aop.terms")
@EnableAspectJAutoProxy
public class Config {
}
```

Tells Spring to look for AspecJ annotations on its beans and create proxies for them

```java
package cs544.spring40.aop.terms;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Aspect
@Configuration
public class LogAspect {
	private static Logger logger = LogManager.getLogger(LogAspect.class.getName());

	@Before("execution(* cs544.spring40.aop.terms.CustomerService.*(..))")
	public void logBefore(JoinPoint joinpoint) {
		logger.warn("About to exec: " + joinpoint.getSignature().getName());
	}
	@After("execution(* cs544.spring40.aop.terms.CustomerService.*(..))")
	public void logAfter(JoinPoint joinpoint) {
		logger.warn("Just execed: " + joinpoint.getSignature().getName());
	}
}
```

Needs @Configuration to be a Bean

Full Example Code

```java
package cs544.spring40.aop.terms;

import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplication
import org.springframework.context.support.ClassPathXmlApplicationContext

public class App {
  public static void main(String[] args) {
    ConfigurableApplicationContext context;
    //context = new ClassPathXmlApplicationContext("cs544/spring40/aop/terms/
    context = new AnnotationConfigApplicationContext(Config.class);
    ICustomerService cs = context.getBean("customerService", IcustomerService
    cs.doSomething();

    context.close();
  }
}
```

```
15:51:44.416 About to exec: doSomething
something
15:51:44.451 Just execed: doSomething
```

```java
package cs544.spring40.aop.terms;

public interface ICustomerService {
      void doSomething();
      void otherThing();
}
```

```java
package cs544.spring40.aop.terms;

import org.springframework.stereotype.Service;

@Service
public class CustomerService
                  implements ICustomerService{

      public void doSomething() {
            System.out.println("something");
      }
      public void otherThing() {
            System.out.println("other");
      }
}
```

Force CGLIB Proxies

- Proxy target class (instead of from interfaces)

```
package cs544.spring40.aop.terms;

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

@Configuration
@ComponentScan("cs544.spring40.aop.terms")
@EnableAspectJAutoProxy(proxyTargetClass=true)
public class Config {
}
```


MAHARISHI
HIGHER EDUCATION
FOR HIGHER CONSCIOUSNESS
INTERNATIONAL UNIVERSITY

CS544 EA

Aspect Oriented Programming

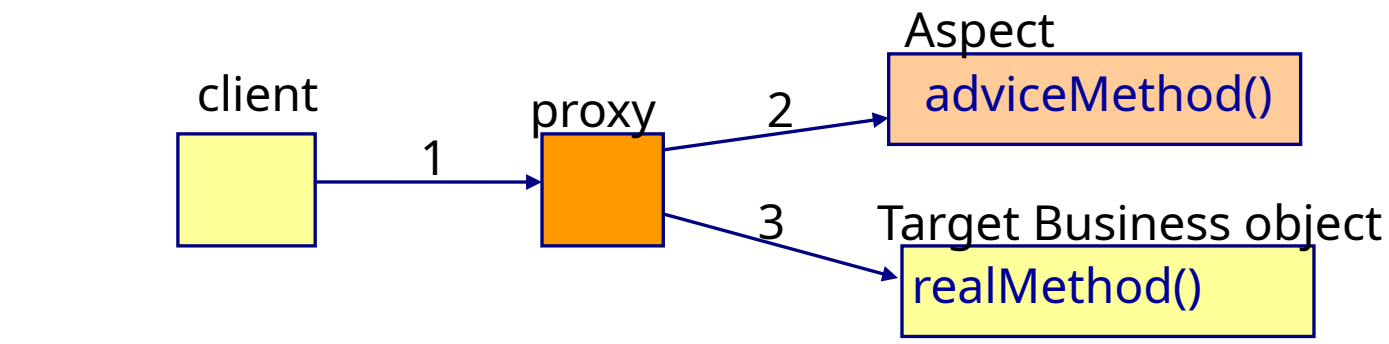
Types of Advice

5 Types of Advice

- There are **5 types** of advice:
  - @Before
  - @After
  - @AfterReturning (only execs if returns properly)
  - @AfterThrowing (only execs if exception thrown)
  - @Around (single advice method execs before and after)

@Before

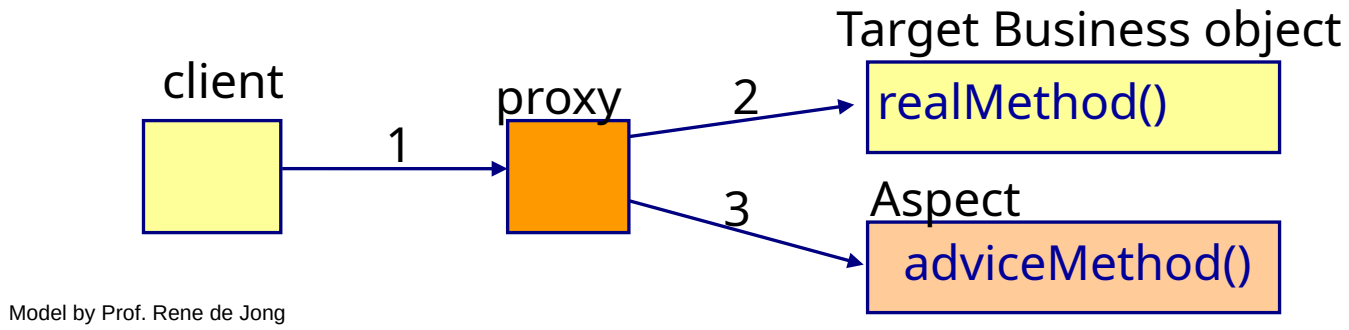
- Calls the advice method before calling the actual method



Model by Prof. Rene de Jong

@After

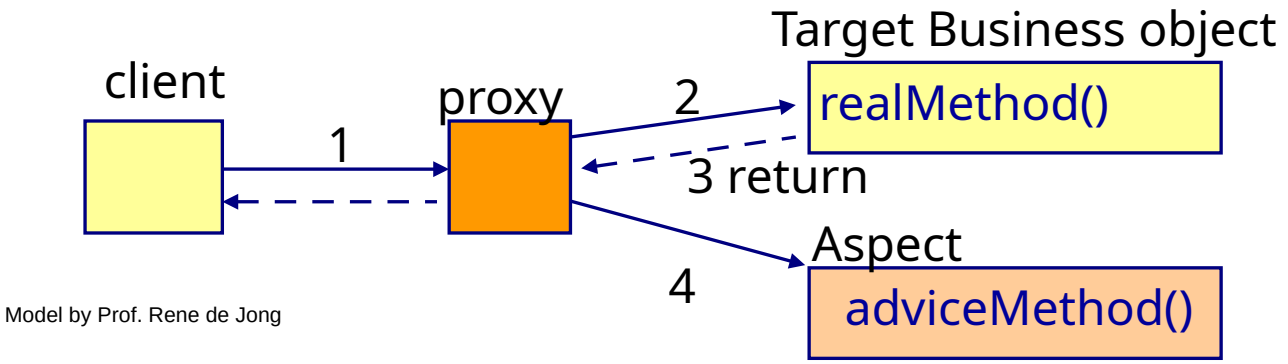
- Calls the advice method (regardless of what happens) after the real method



Model by Prof. Rene de Jong

@AfterReturning

- Calls the advice method only if the real method returned normally
  - Allows the advice to receive the returned value



@AfterReturning

We will go into this in more detail coming up

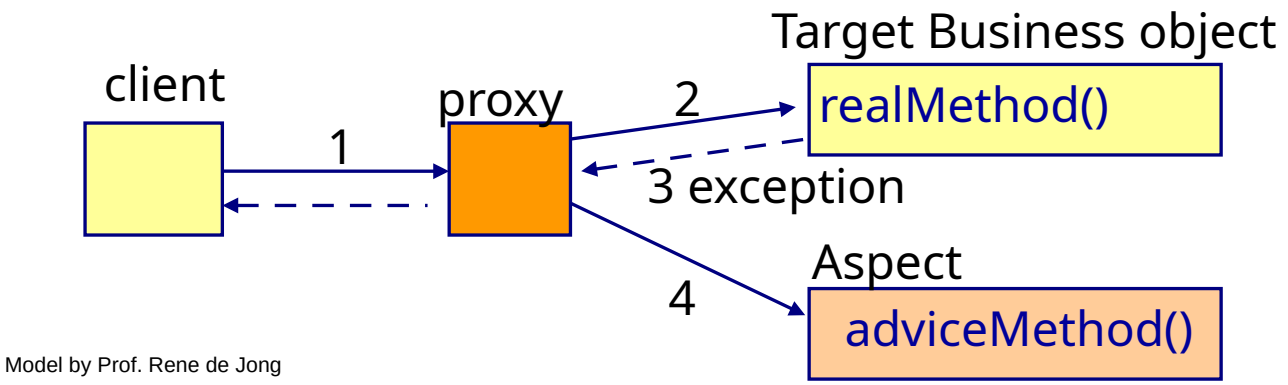
```java
package cs544.spring41.aop.advices;

import org.springframework.stereotype.Service;

@Service
public class CustomerService implements ICustomerService {
	public String getName() {
		return "John";
	}
}
```

```java
package cs544.spring41.aop.advices;

...

@Aspect
@Component
public class TestAspect {
	@AfterReturning(pointcut="execution(* cs544.spring41.aop.advices.CustomerService.getName(..))", returning="ret")
	public void afterRet(JoinPoint jp, String ret) {
		System.out.println(jp.getSignature().getName() + " returned: " + ret);
	}
}
```

```
getName returned: John
```

@AfterThrowing

- Calls the advice method only if the real method throws and exception
  - Allows the advice to receive the exception



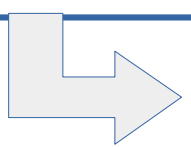
Model by Prof. Rene de Jong

@AfterThrowing

We will go into this in more detail coming up

```java
package cs544.spring41.aop.advices;

import org.springframework.stereotype.Service;

@Service
public class CustomerService implements ICustomerService {
	public String getAge() {
		throw new MyException();
	}
}
```

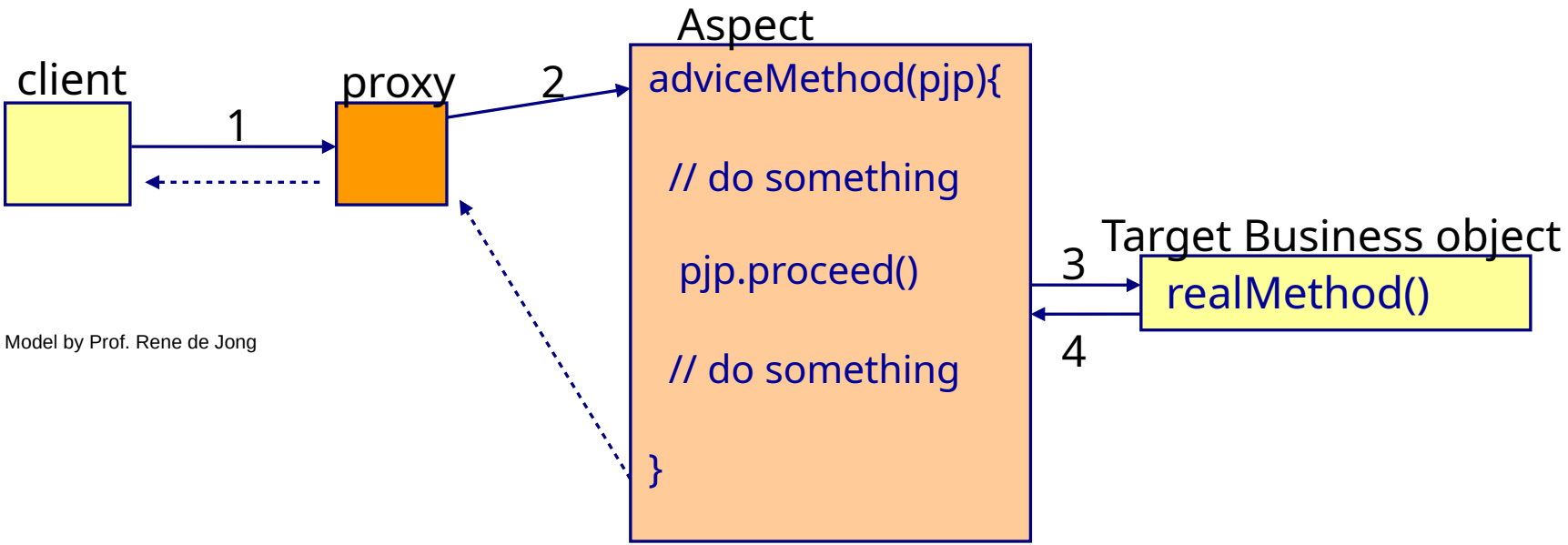
```java
package cs544.spring41.aop.advices;

...

@Aspect
@Component
public class TestAspect {
	@AfterThrowing(pointcut="execution(* cs544.spring41.aop.advices.CustomerService.getAge(..))", throwing="ex")
	public void afterThrow(JoinPoint jp, MyException ex) {
		System.out.println(jp.getSignature().getName() + " threw a: " + ex.getClass().getName());
	}
}
```



```
getAge threw a: cs544.spring41.aop.advices.MyException
```

@Around

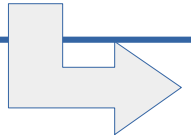
- Around has to choose when (and if) it calls the real method.
  - Receives the **parameters** to pass to the real method
  - Receives the **return** from the real method



@Around

We will go into this in more detail coming up

```
@Around("execution(* cs544.spring41.aop.advices.CustomerService.getName(..))")
public Object around(ProceedingJoinPoint pjp) {
	String m = pjp.getSignature().getName();
	System.out.println("Before " + m);
	Object ret = null;
	try {
		ret = pjp.proceed();
	} catch (Throwable e) {
		e.printStackTrace();
	}
	System.out.println("After " + m + " returned " + ret);
	return ret;
}
```



```
Before getName
After getName returned John
```

```
package cs544.spring41.aop.advices;

import org.springframework.stereotype.Service;

@Service
public class CustomerService implements ICustomerService
{
	public String getName() {
		return "John";
	}
}
```


MAHARISHI
HIGHER EDUCATION
FOR HIGHER CONSCIOUSNESS
INTERNATIONAL UNIVERSITY

CS544 EA

Aspect Oriented Programming

JoinPoint

JoinPoint

- Every advice method can optionally receive as its **first argument** a JoinPoint object
  - Not required, but it is usually nice to have

- The JoinPoint object contains info about the method (point) that will be (or was) joined for a call
  - Remember a Pointcut often specifies many points

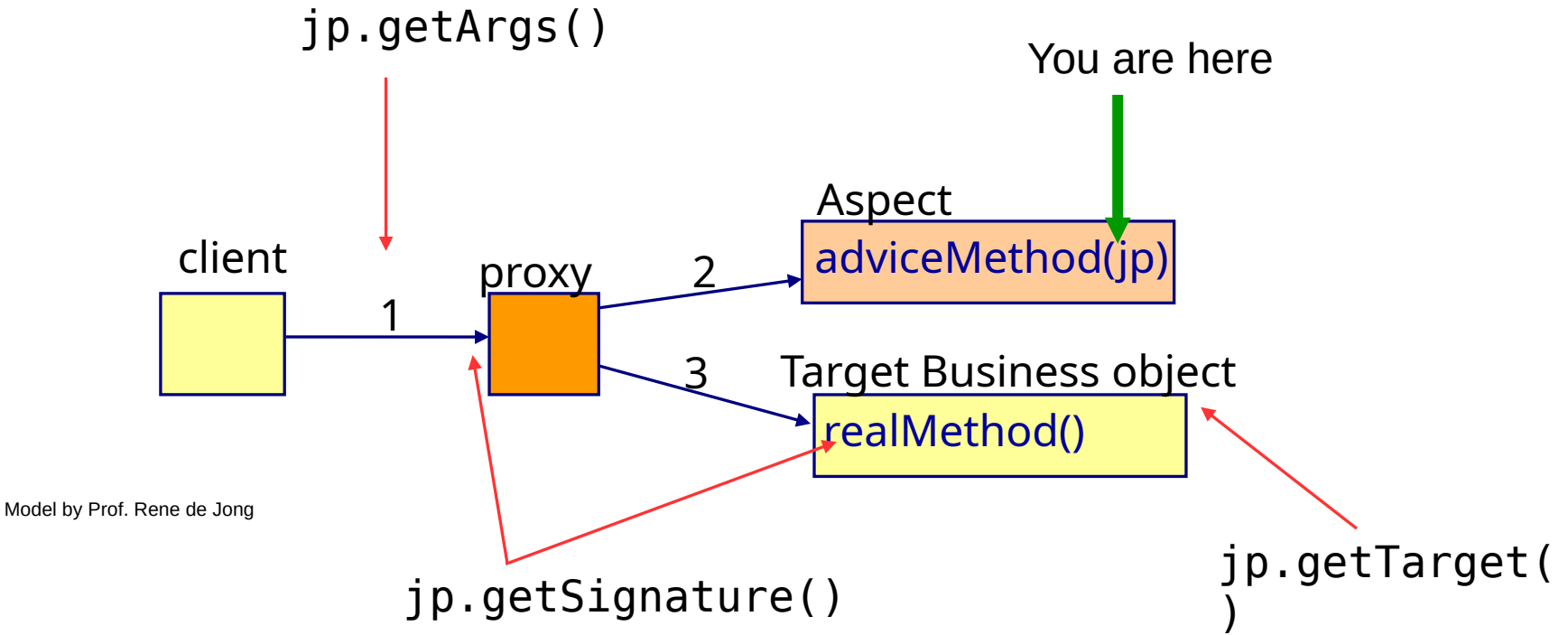
Example Code

```
@Aspect
@Configuration
public class LogAspect {
     private static final Logger logger = LogManager.getLogger(LogAspect.class.getName());

     @Before("execution(* cs544.spring40.aop.terms.CustomerService.*(..))")
     public void logBefore(JoinPoint joinpoint) {
          logger.warn("About to exec: " + joinpoint.getSignature().getName());
     }

     @After("execution(* cs544.spring40.aop.terms.CustomerService.*(..))")
     public void logAfter(JoinPoint joinpoint) {
          logger.warn("Just execed: " + joinpoint.getSignature().getName());
     }
}
```

JoinPoint API

- The most important methods on a JoinPoint
  - `Signature getSignature()`
    - Returns the method signature of the real method (name, return type, etc)
  - `Object[] getArgs()`
    - Returns the arguments passed to real method as Object[]
  - `Object getTarget()`
    - Returns the Object on which real method was / will be called

Example

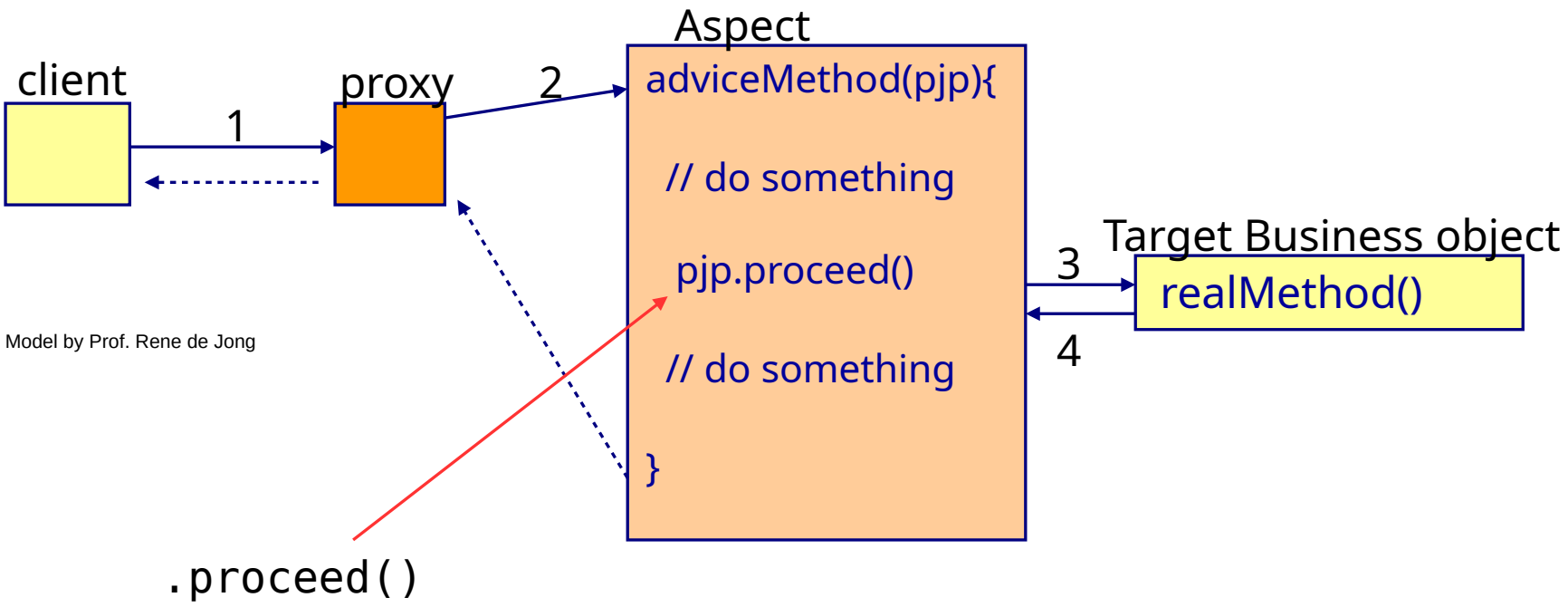

jp.getArgs()
You are here
Aspect
client
proxy
2
adviceMethod(jp)
1
3
Target Business object
realMethod()
Model by Prof. Rene de Jong
jp.getSignature()
jp.getTarget(
)

ProceedingJoinPoint

- ProceedingJoinPoint extends JoinPoint for use inside an @Around advice adding the following overloaded method:
  - `Object proceed()`
  - `Object proceed(Object[] args)`

- Proceed without args causes the real method to be called with the arguments from the client, returning an Object
  - Proceed with args allows you to **give your own version** of the args array

Proceed


client
proxy
1
2
Aspect
adviceMethod(pjp){
// do something
pjp.proceed()
// do something
}
Target Business object
3
realMethod()
4
Model by Prof. Rene de Jong
.proceed()

Not Optional for @Around

- JoinPoint is an optional argument for @Before, @After, @AfterReturning and @AfterThrowing
  - These methods do not need it to function

- ProceedingJoinPoint is **not optional for @Around**
  - Cannot function without it
  - @Around method also has to return Object
  - And declare throws Throwable

Example Code

```
@Around("execution(* cs544.spring41.aop.advices.CustomerService.getName(..))")
public Object around(ProceedingJoinPoint pjp) throws Throwable {
     String m = pjp.getSignature().getName();
     System.out.println("Before " + m);
     Object ret = null;
     try {
          ret = pjp.proceed();
     } catch (Throwable e) {
          e.printStackTrace();
          throw e;
     }
     System.out.println("After " + m + " returned " + ret);
     return ret;
}
```


MAHARISHI
HIGHER EDUCATION
FOR HIGHER CONSCIOUSNESS
INTERNATIONAL UNIVERSITY

CS544 EA

Aspect Oriented Programming

Pointcut Expression Language

PointCut Express Language

- Written as a String
  - Part of the advice annotation (@Before / …)
  - No compile time checking
  - If it doesn’t match properly it fails silently

- Expressions can be combined with boolean operators
  - && (boolean and)
  - || (boolean or)
  - ! (boolean not)

Expressions

Pointcut expressions have to start with one of the following pointcut designators

- execution
- args
- within
- target
- @annotation
- @args
- @within
- @target

Execution

- `execution` is the most used designator

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?name-pattern(param-pattern) throws-pattern?)
```

Optional modifer
(public, protected, private)

Return type
* indicates any type

Optional type
(package and class)
ending in .* includes all classes in package
ending in ..* includes classes in sub-packages

Method name
Use . to connect with type
May contain, or just be *

Parameters
Comma separated list
(..) means any parameters
(*) means one param any type
(int, *) = a int and one other type

Optional throws
Comma separated list of types
Optionally including ! (not)

See: https://docs.spring.io/spring/docs/5.1.6.RELEASE/spring-framework-reference/core.html#aop-pointcuts-examples

Execution Examples

- Examples from the Spring Documentation

  See: https://docs.spring.io/spring/docs/5.1.6.RELEASE/spring-framework-reference/core.html#aop-pointcuts-examples

```
execution(public * *(..))    // any public method
execution(* set*(..))  // any method whose name starts with set
execution(* com.xyz.service.AccountService.*(..)) // any method of AccountService
execution(* com.xyz.service.*.*(..)) // any method of any class in the service package
execution(* com.xyz.service..*.*(..)) // any method in the service package or sub packages
execution(* *(int))    // any method taking a single int
execution(* put*(String, int))  // any method starting with put, taking a String and an int
```

@annotation

- Matches any method that is annotated with the given annotation

```
@annotation(org.springframework.transaction.annotation.Transactional)
```

Matches any method that has the Spring @Transactional annotation

args and @args

- args(int, String)
  - Matches only methods that take an int and a String

- @args(org.springframework.stereotype.Service)
  - Matches only methods that take one object whose class is annotated as being a Service

within and @within

- within(cs544.spring40.aop.CustomerService)
  - Any method within this class
- within(cs544.spring40..*)
  - Any method within this package, or sub-packages

- @within(org.springframework.stereotype.Service)
  - Any methods within a class annotated as a Spring service

target and @target

- target(cs544.spring40.aop.ICustomerService)
  - Specifies what the type of the Target has to be
  - Type can be an interface (then matches all classes that implement)
  - Matches any methods in classes with the specified type

- @target(org.springframework.stereotype.Service)
  - Specifies annotation that the Target has to have
  - Matches any methods in classes annotated with it

Boolean Operators

- Boolean operators work as you would expect

```
package cs544.spring42.aop.boolops;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class TestAspect {
	private static Logger logger = LogManager.getLogger(TestAspect.class.getName());

	@Before("execution(* cs544.spring42.aop.boolops.CustomerService.*(..)) "
			+ " && @target(org.springframework.stereotype.Service)")
	public void logBefore(JoinPoint joinpoint) {
		logger.warn("About to exec: " + joinpoint.getSignature().getName());
	}
}
```

Enforces that CustomerService is annotated with @Service

Named Pointcuts

```java
package cs544.spring42.aop.boolops;
...
@Aspect
@Component
public class CheckOrderAspect {
	@Pointcut("execution(* cs544.spring42.aop.boolops.OrderService.*(..))")
	public void checkOrder() {
	}
	@Before("checkOrder()")
	public void checkOrder(JoinPoint joinpoint) {
		System.out.println("check order");
	}
	@After("checkOrder()")
	public void logOrderEvent(JoinPoint joinpoint) {
		System.out.println("log order event");
	}
}
```

```java
@Service
public class OrderService implements IOrderService {
	@Override
	public void createOrder(Customer customer, ShoppingCart shoppingCart) {
		System.out.println("Create Order");
	}
	@Override
	public void deleteOrder(String ordernumber) {
		System.out.println("Delete Order");
	}
	@Override
	public void shipOrder(String ordernumber) {
		System.out.println("Ship Order");
	}
}
```

Named PointCut (other class)

```
package cs544.spring42.aop.boolops;
...
@Aspect
@Component
public class NamedPointCuts {
	@Pointcut("execution(* cs544.spring42.aop.boolops.OrderService.*(..))")
	public void checkOrder() {
	}
}
```

```
package cs544.spring42.aop.boolops;
...
@Aspect
@Component
public class CheckOrderAspect {
	@Before("NamedPointCuts.checkOrder()")
	public void checkOrder(JoinPoint joinpoint) {
		System.out.println("check order");
	}

	@After("NamedPointCuts.checkOrder()")
	public void logOrderEvent(JoinPoint joinpoint) {
		System.out.println("log order event");
	}
}
```


MAHARISHI
HIGHER EDUCATION
FOR HIGHER CONSCIOUSNESS
INTERNATIONAL UNIVERSITY

CS544 EA

Aspect Oriented Programming

Data Inside Advice

Data and Advice Methods

- There are several ways you can receive data in an advice method
  - Through the JoinPoint (args, target)
  - Return value / Thrown exceptions
  - Injected into the Aspect object (eg. DAOs)

Injected Objects

- An Aspect class is **just another bean**
  - Can have objects injected just like any other bean
  - Useful: you can inject DAOs to retrieve additional data from DB

```
package cs544.spring43.aop.data;
...

@Configuration
@Aspect
public class InjectAspect {
	@Autowired
	private PersonDao personDao;

	@Before("execution(* cs544.spring43.aop.data.CustomerService.setName(String))")
	public void argsBefore(JoinPoint jp) {
		Object[] args = jp.getArgs();
		String name = (String)args[0];
		Person p = personDao.byName(name);
		if (p.getAge() > 18) {
			System.out.println("adult");
		}
	}
}
```

Arguments

- `jp.getArgs()` returns an `Object[]`
  - Spring does not know the types of the args
  - **You have to cast** them yourself

```
package cs544.spring43.aop.data;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Aspect
@Configuration
public class TestAspect {
	@Before("execution(* cs544.spring43.aop.data.CustomerService.setName(String))")
	public void argsBefore(JoinPoint jp) {
		Object[] args = jp.getArgs();
		String name = (String)args[0];
		System.out.println("Argument value: " + name);
	}
}
```

Pointcut args() Designator

- It is also possible to receive incoming args directly **into the advice method**
  - Use args() pointcut to specify names instead of types
  - A bit slower (more CPU) than using JoinPoint

```
package cs544.spring43.aop.data;

...

@Aspect
@Configuration
public class TestAspect {
	@Before("execution(* cs544.spring43.aop.data.CustomerService.setName(String)) && args(name))")
	public void argsBefore(JoinPoint jp, String name) {
		System.out.println("Argument value: " + name);
	}
}
```

The latest version of Spring seems to have trouble with this. You can make it work by compiling with -parameters so that the parameter names are explicitly perserved

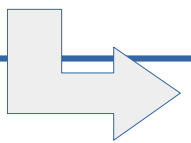
Changing Args

- The @Around advice has the additional possibility of **changing the argument** values
  - Before giving them to the real method

```
package cs544.spring43.aop.data;

...

@Aspect
@Configuration
public class TestAspect {

	@Around("execution(* cs544.spring43.aop.data.CustomerService.setName(String))")
	public Object aroundSetName(ProceedingJoinPoint pjp) throws Throwable {
		Object[] args = pjp.getArgs();
		System.out.println("Argument value: " + args[0]);
		args[0] = "James";
		return pjp.proceed(args);
	}

}
```

Changing Args Demo

```
package cs544.spring43.aop.data;

import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {
  public static void main(String[] args) {
    ConfigurableApplicationContext context;
    //context = new ClassPathXmlApplicationContext("cs544/spring43/aop/data/springconfig.xml");
    context = new AnnotationConfigApplicationContext(Config.class);
    ICustomerService cs = context.getBean("customerService", ICustomerService.class);
    cs.setName("John");
    System.out.println("Inside cs: " + cs.getName());


    context.close();
  }
}
```



```
Argument value: John
Inside cs: James
```

Return Value

- @AfterReturning can receive the return

```
package cs544.spring41.aop.advices;

...

@Aspect
@Configuration
public class TestAspect {
	@AfterReturning(pointcut="execution(* cs544.spring41.aop.advices.CustomerService.getName())", returning="ret")
	public void afterRet(JoinPoint jp, String ret) {
		System.out.println(jp.getSignature().getName() + " returned: " + ret);
	}
}
```

Changing return value

- @Around can also **change the return** value

```
package cs544.spring43.aop.data;

...

@Aspect
@Configuration
public class TestAspect {
	@Around("execution(* cs544.spring43.aop.data.CustomerService.getName())")
	public Object aroundGetName(ProceedingJoinPoint pjp) throws Throwable {
		Object name = pjp.proceed();
		return "Chris";
	}
}
```

```
public class App {
  public static void main(String[] args) {
    ConfigurableApplicationContext context;
    context = new AnnotationConfigApplicationContext(Config.class);
    ICustomerService cs = context.getBean("customerService", ICustomerService.class);
    cs.setName("John");
    System.out.println("From cs: " + cs.getName());
  }
}
```

```
From cs: Chris
```

Exception

- @AfterThrowing can receive the exception
  - Cannot stop or alter it!

```
package cs544.spring41.aop.advices;

...

@Aspect
@Configuration
public class TestAspect {
	@AfterThrowing(pointcut="execution(* cs544.spring41.aop.advices.CustomerService.getAge(..))", throwing="ex")
	public void afterThrow(JoinPoint jp, MyException ex) {
		System.out.println(jp.getSignature().getName() + " threw a: " + ex.getClass().getName());
	}
}
```

Changing the Exception

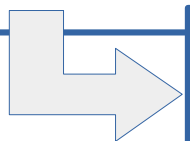
- @Around can **catch the exception** and choose:
  - Re-throw the same exception
  - Throw another exception
  - Don’t throw anything (stop the exception)

```
package cs544.spring43.aop.data;

...

@Aspect
@Configuration
public class TestAspect {
	@Around("execution(* cs544.spring43.aop.data.CustomerService.exception())")
	public Object aroundException(ProceedingJoinPoint pjp) {
		try {
			return pjp.proceed();
		} catch (Throwable e) {
			throw new OtherException();
		}
	}
}
```

Other Exception Demo

```java
package cs544.spring43.aop.data;

import org.springframework.stereotype.Service;

@Service
public class CustomerService implements ICustomerService {
	@Override
	public void exception() {
		throw new MyException();
	}
}
```

```java
public class App {
  public static void main(String[] args) {
    ConfigurableApplicationContext context;
    context = new AnnotationConfigApplicationContext(Config.class);
    ICustomerService cs = context.getBean("customerService", ICustomerService.class);
    cs.exception();
  }
}
```



```
Exception in thread "main" cs544.spring43.aop.data.OtherException
	at cs544.spring43.aop.data.TestAspect.aroundException(TestAspect.java:32)
	at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
	at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
	at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
	at java.lang.reflect.Method.invoke(Method.java:498)
```

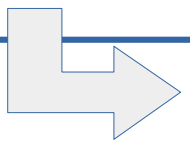
Full Power of @Around 1/2

```java
package cs544.spring43.aop.data;
...
@Component
public class Calculator implements ICalculator {
	public int add(int x, int y) {
		System.out.println("Calculator.add receiving: x= " + x + " and y= " + y);
		return x + y;
	}
}
```

```java
package cs544.spring43.aop.data;
...
@Aspect
@Configuration
public class CalcAspect {
	@Around("execution(* cs544.spring43.aop.data.Calculator.add(..))")
	public Object changeNumbers(ProceedingJoinPoint pjp) {
		Object[] args = pjp.getArgs();
		int x = (Integer) args[0];
		int y = (Integer) args[1];
		System.out.println("CalcAdvice.changeNumbers: x= " + x + " and y= " + y);
		args[0] = 5;
		args[1] = 9;
		Object object = null;
		try { object = pjp.proceed(args);
		} catch (Throwable e) { /* do nothing*/ }
		System.out.println("CalcAdvice.changeNumbers: call.proceed returns " + object);
		return 26;
	}
}
```

Full Power of @Around 2/2

```
package cs544.spring43.aop.data;
...
public class App {
	public static void main(String[] args) {
		ConfigurableApplicationContext context;

		ICalculator calc = context.getBean("calculator", ICalculator.class);
		int result = calc.add(3, 4);
		System.out.println("The result of 3 + 4 = " + result);

		context.close();
	}
}
```



```
CalcAspect.changeNumbers: x= 3 and y= 4
Calculator.add receiving: x= 5 and y= 9
CalcAdvice.changeNumbers: call.proceed returns 14
The result of 3 + 4 = 26
```

jp.getTarget()

- You can ask the JoinPoint for the target object
  - Sometimes it has **useful data or DAOs**

- Be aware though:
  - Calling methods on the target will be without AOP!
  - See next section (video) for more info


MAHARISHI
HIGHER EDUCATION
FOR HIGHER CONSCIOUSNESS
INTERNATIONAL UNIVERSITY

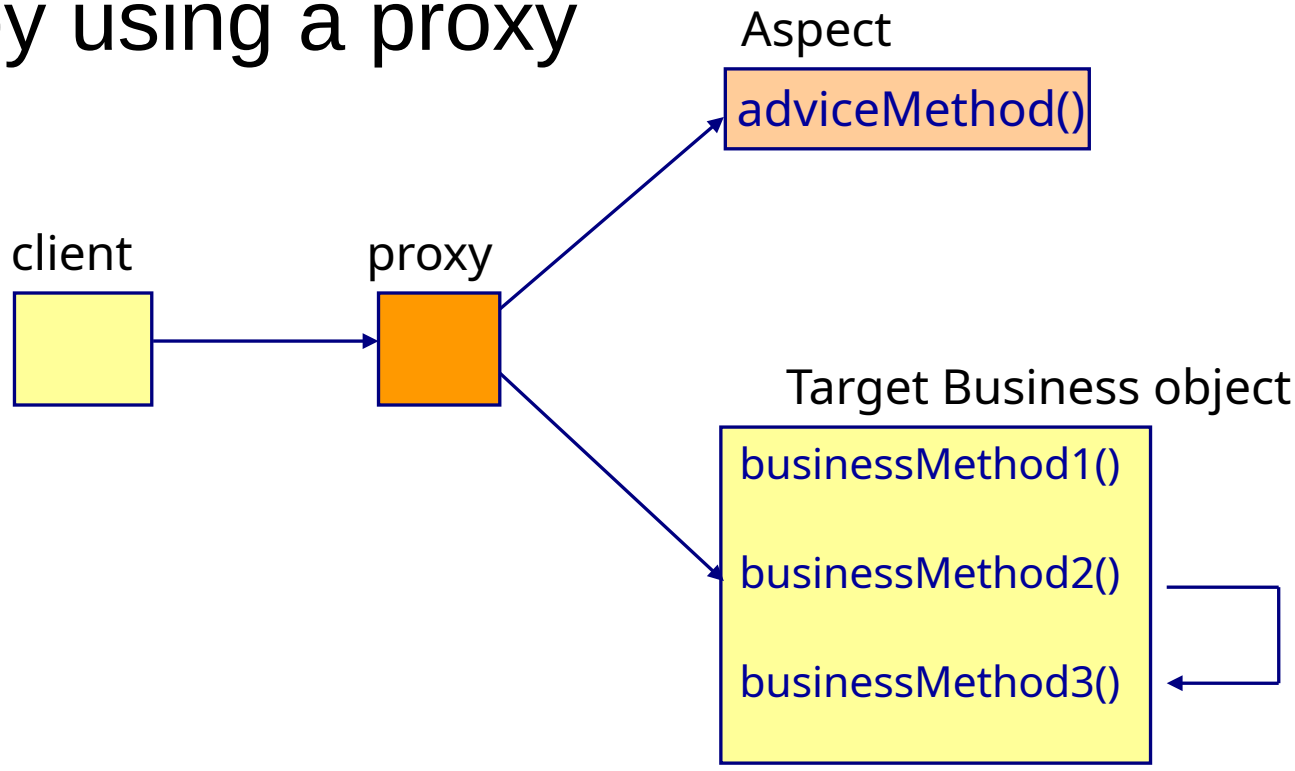
CS544 EA

Aspect Oriented Programming

Proxy Weaving Problems

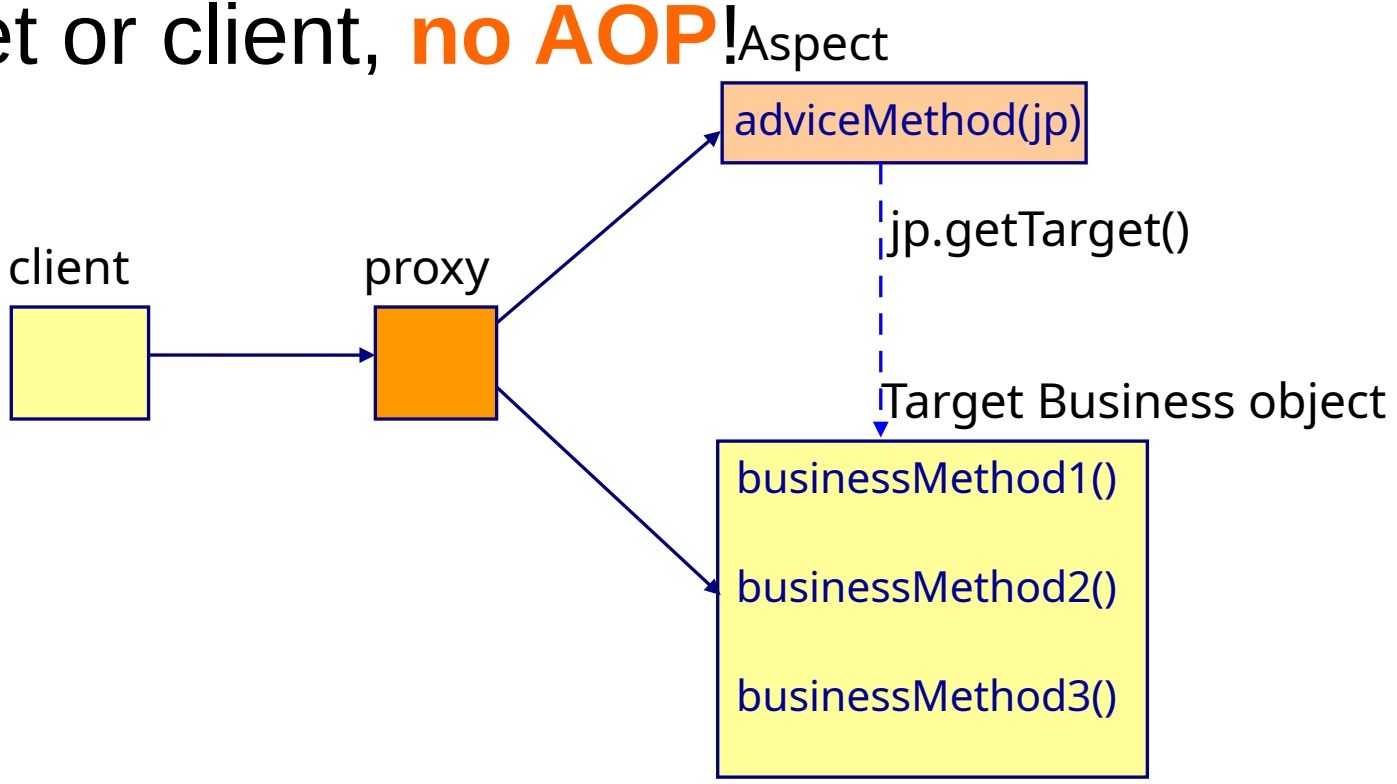
Local calls

- There is **no way** for Spring to **intercept** a local call by using a proxy



Calls to Target

- Similarly, if you invoke methods directly on Target or client, **no AOP!**



No Weaving During Startup

- During **Spring startup** there is **no AOP**
- Not during any of these activities
  - Bean creation
  - Reference injection
  - Value injection
  - Postconstruct Init method


MAHARISHI
HIGHER EDUCATION
FOR HIGHER CONSCIOUSNESS
INTERNATIONAL UNIVERSITY