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

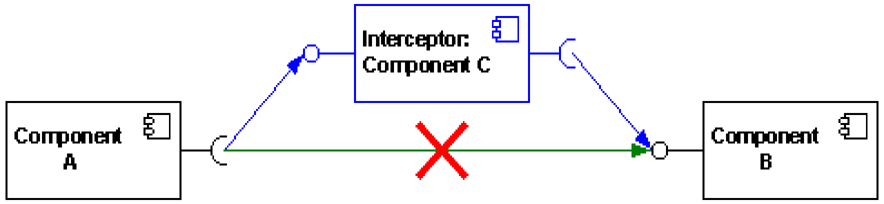
- Cross Cutting Concerns are usually scattered or tangled throughout your code

- 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**



By UlrichAAB - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=6746767

- 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
    - If the business object doesn’t implement any interface
    - Using Objenesis

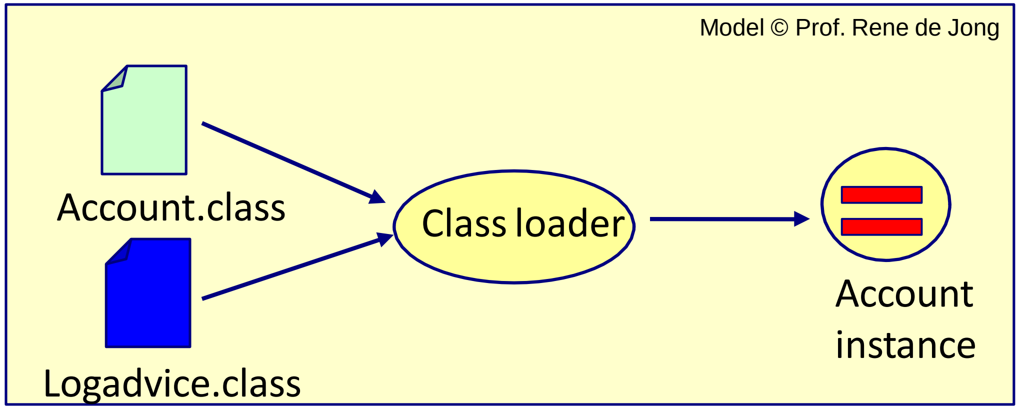
JDK > CGLIB

- 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



CS544 EA

AOP: Terminology

Advice

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

```
@Aspect
@Component
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
@Component
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
@Component
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.getSigna
            }
}
```

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

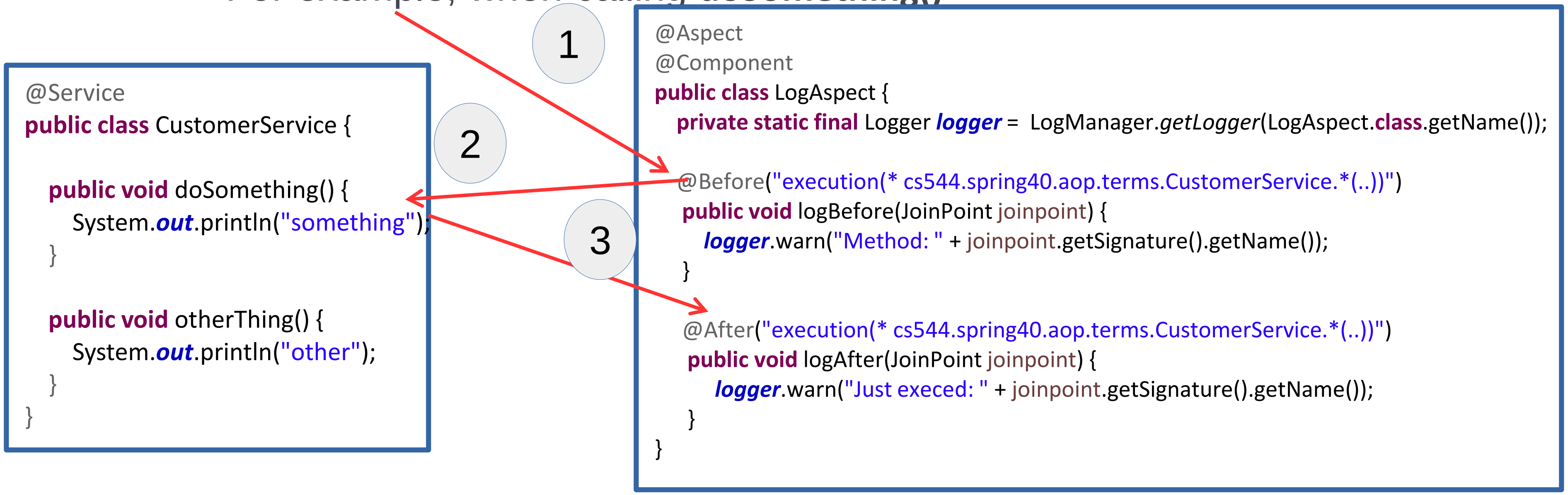
```
@Aspect
@Component
public class LogAspect {
			private static final Logger logger =  LogManager.getLogger(LogAspect.class.getName());

			@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()**

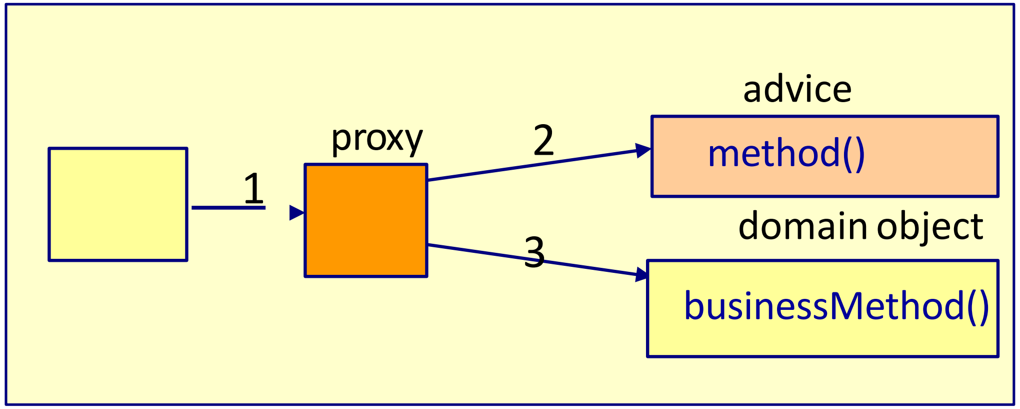


```java
@Service
public class CustomerService {

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

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

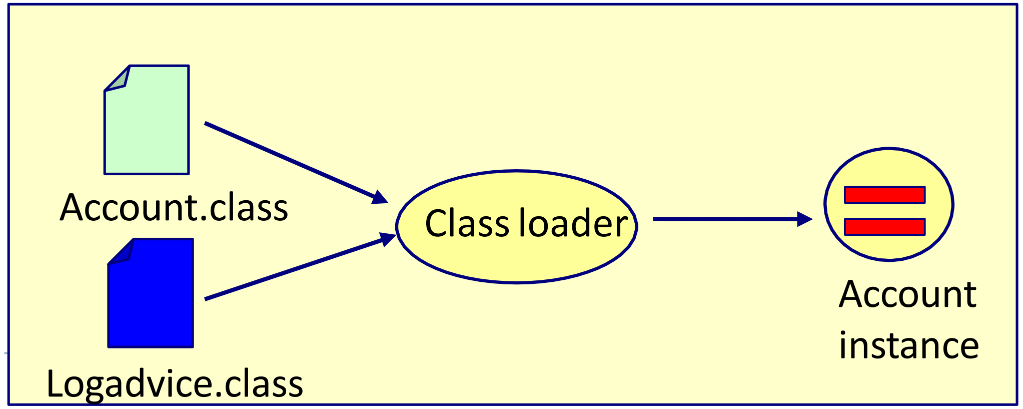
```java
@Aspect
@Component
public class LogAspect {
   private static final Logger logger =  LogManager.getLogger(LogAspect.class.getName());

   @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());
     }
}
```

Weaving

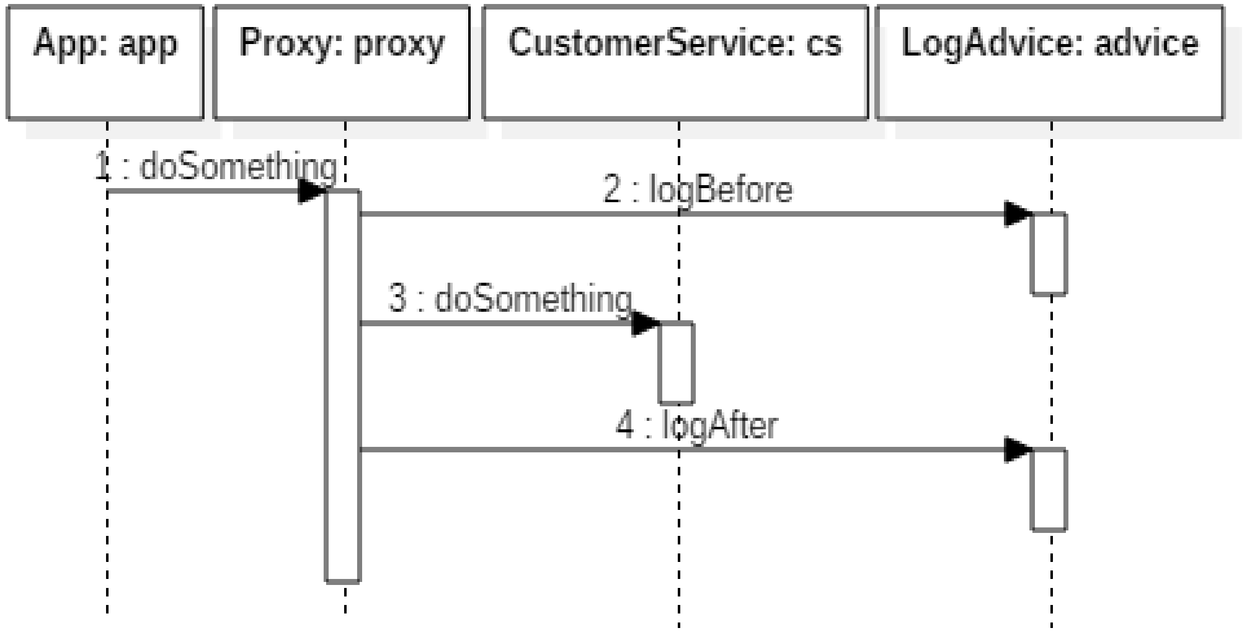
Proxy-based weaving



Bytecode weaving



Proxy Weaving Sequence Diagram



Full Example Code

```
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
@Component
public class LogAspect {
        private static Logger logger = LogManager.getLogger(LogAspect.clas

        @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 @Component to be a Bean

```
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 AspectJ annotations on its beans and create proxies for them

Full Example Code

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

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

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

   context.close();
 }
}
```

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

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

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

import org.springframework.stereotype.Service;

@Service
public class CustomerService {

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

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

XML Configuration

- Alternately XML can setup AspectJ annotations

```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"
            xsi:schemaLocation="http://www.springframework.org/schema/beans
                        http://www.springframework.org/schema/beans/spring-beans.xsd
                        http://www.springframework.org/schema/aop
                        http://www.springframework.org/schema/aop/spring-aop.xsd">

            <aop:aspectj-autoproxy />
            <bean id="customerService" class="cs544.spring40.aop.terms.CustomerService" />
            <bean id="LogAspect" class="cs544.spring40.aop.terms.LogAspect" />

</be
```

Important:
the AOP namespace

Tell spring to look for
AspectJ annotations
on its beans

LogAspect is a bean
just like everything else
(can also be injected into)

Force CGLIB Proxies

- Proxy target class (instead of from interfaces)

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

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

```
<?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"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
                http://www.springframework.org/schema/beans/spring-beans.xsd
                http://www.springframework.org/schema/aop
                http://www.springframework.org/schema/aop/spring-aop.xsd">

        <aop:aspectj-autoproxy proxy-target-class="true"/>
        <bean id="customerService" class="cs544.spring40.aop.terms.CustomerService" />
        <bean id="LogAspect" class="cs544.spring40.aop.terms.LogAspect" />
</beans>
```

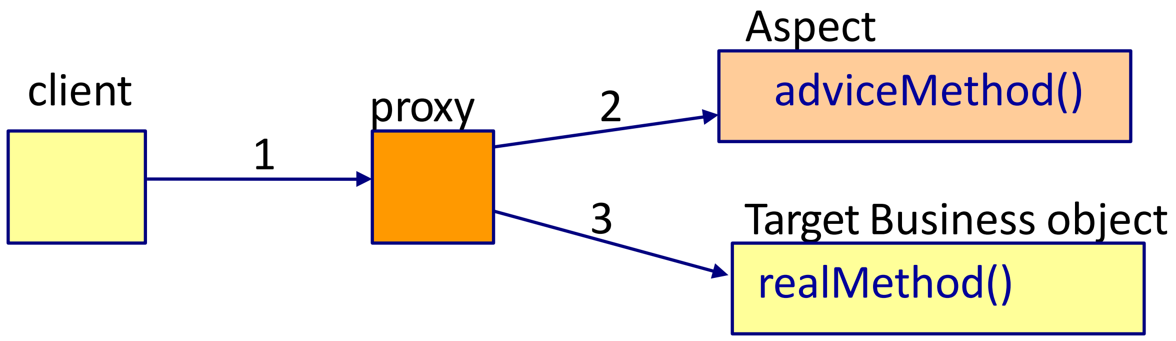
AOP: Types of Advice

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

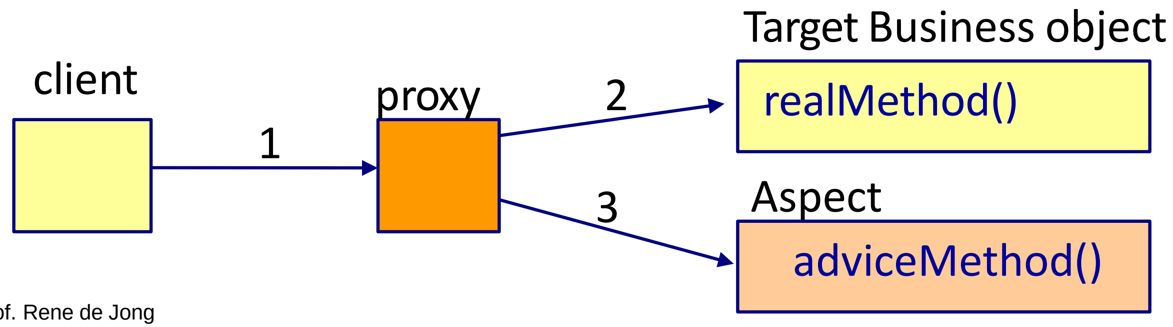
- Calls the advice method before calling the actual method



Model by Prof. Rene de Jong

@After

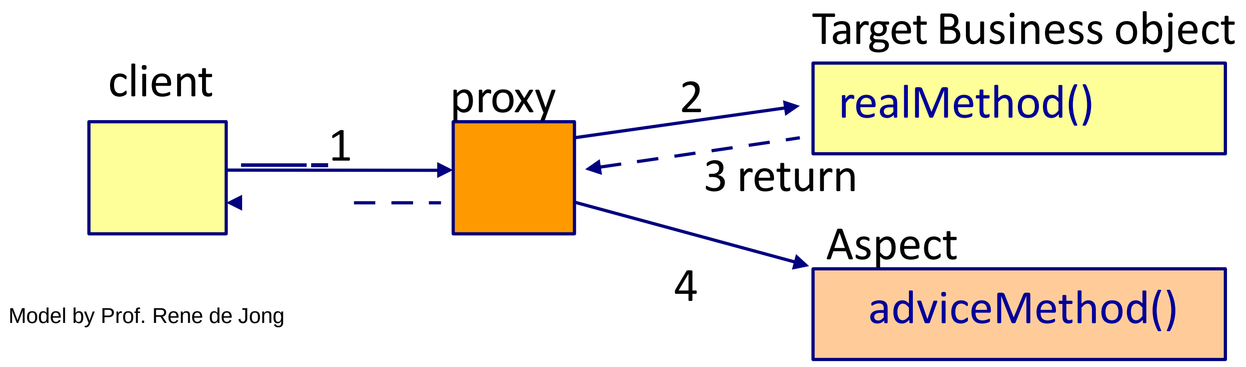
- Calls the advice method (regardless of what happens) after the real method



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

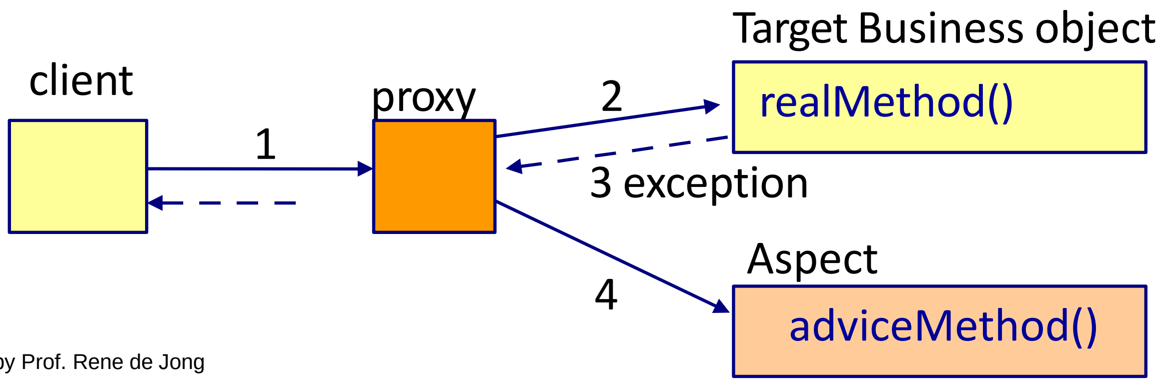
```
package cs544.spring41.aop.advices;

import org.springframework.stereotype.Service;

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

```
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

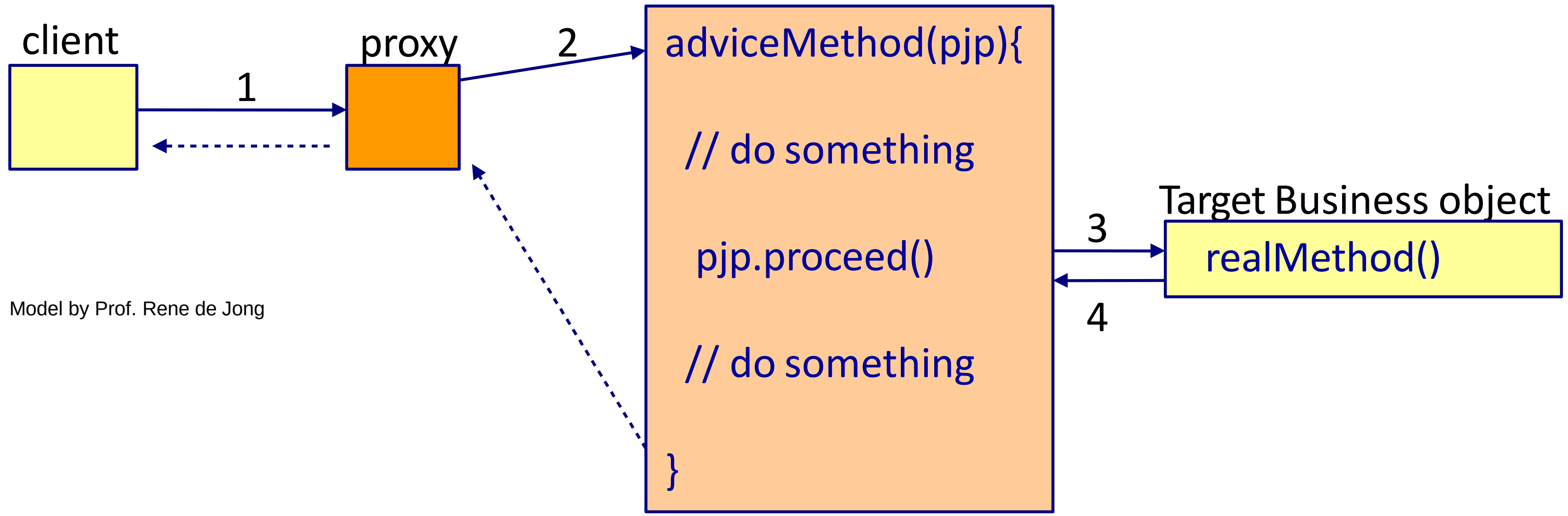
```
package cs544.spring41.aop.advices;

import org.springframework.stereotype.Service;

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

```
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";
		}
}
```

CS544 EA

AOP: JoinPoint

JoinPoint

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

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

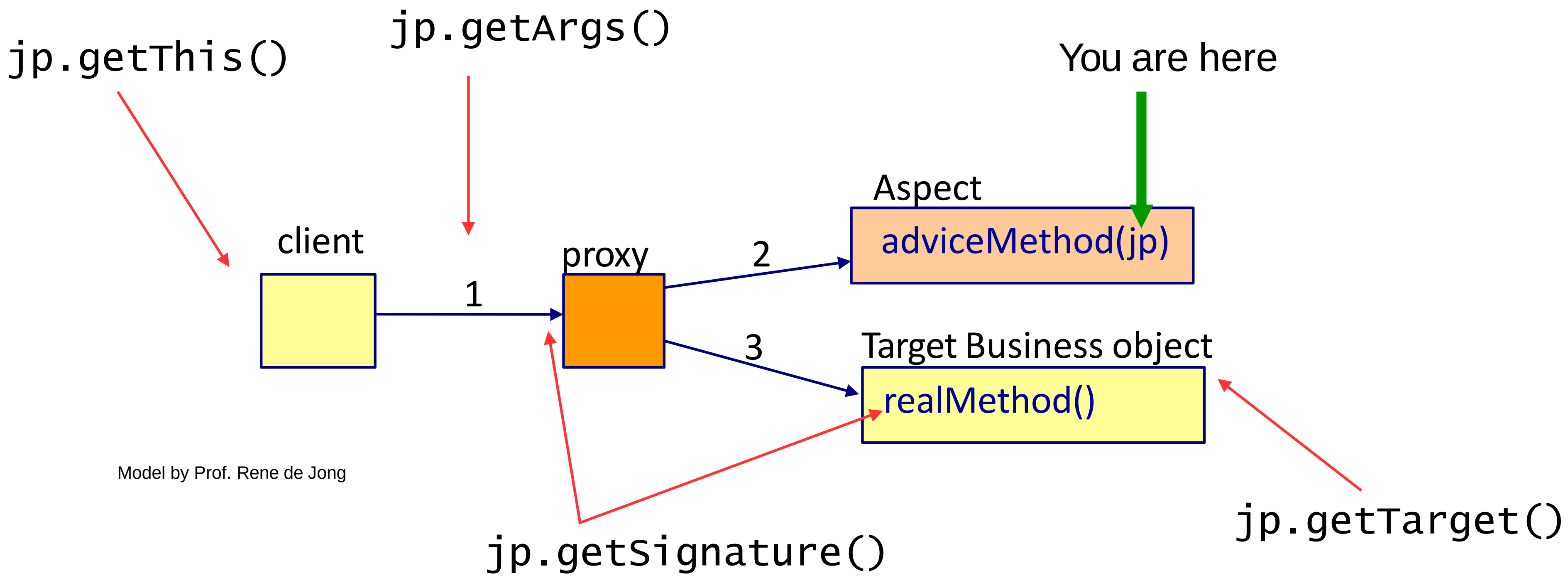
Example Code

```
@Aspect
@Component
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 call(ed)
  - Object getThis()
    - Returns the Object that calls the real method

Example

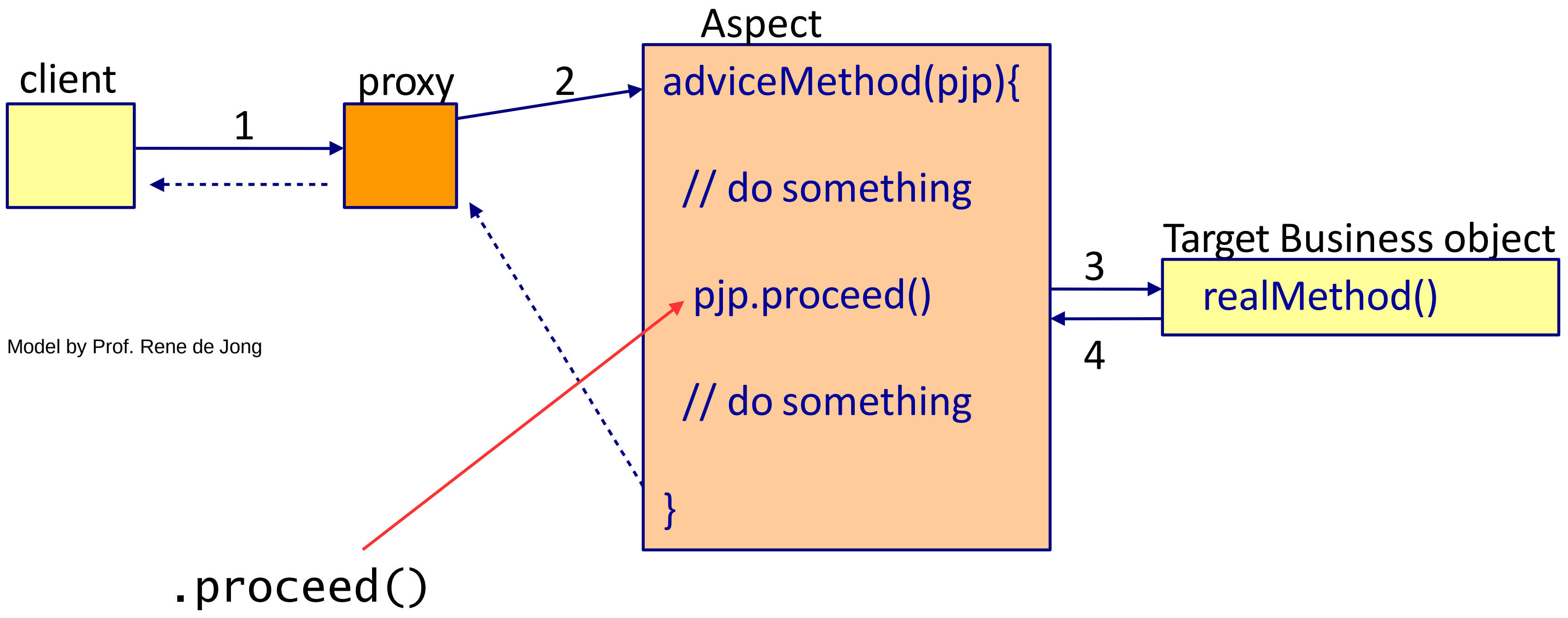

jp.getThis()
jp.getArgs()
You are here
Aspect
adviceMethod(jp)
client
proxy
1
2
3
Target Business object
realMethod()
Model by Prof. Rene de Jong
jp.getTarget()
jp.getSignature()

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, giving us (the advice) the return as an Object
  - Proceed with args allows you to **give your own version** of the args array

Proceed


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

Not Optional

- 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
  - Also has to return Object
  - And declare throws throwable (or catch it)

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;
}
```

CS544 EA

AOP: 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

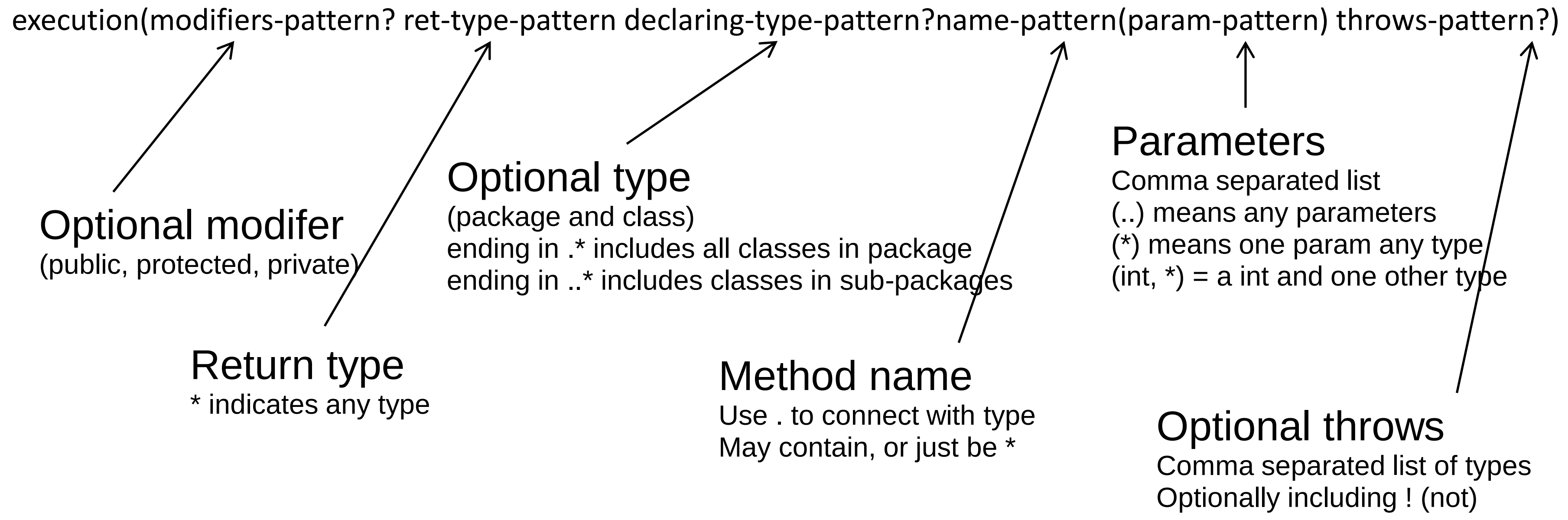
Pointcut expressions have to start with one of the following pointcut designators

- 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 (from the Spring Documentation)

- execution(public * *(..)) // any public method
- execution(* com.xyz.service.AccountService.*(..)) // any method of AccountService
- execution(* set*(..)) // any method whose name starts with set
- 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

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

@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

```
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");
        }
}
```

```
package cs544.spring42.aop.boolops;
...
@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");
                }
}
```

CS544 EA

AOP: Working with Data Inside Advice

Data and Advice Methods

- There are several ways you can receive data in an advice method
  - Through the JoinPoint (args, target, this)
  - 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: Inject DAOs to retrieve additional data from DB

```
@Component
@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
@Component
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
@Component
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);
        }
}
```

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
@Component
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
@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);
		}
}
```

Changing return value

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

```
@Aspect
@Component
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
@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());
		}
}
```

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)

```
@Aspect
@Component
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

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

import org.springframework.stereotype.Service;

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

```
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 ½

```
@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;
        }
}
```

```
@Aspect
@Component
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() and jp.getThis()

- You can ask the JoinPoint for the target object
  - Or the calling object (provided by jp.getThis)
  - Sometimes these have **useful data or DAOs**

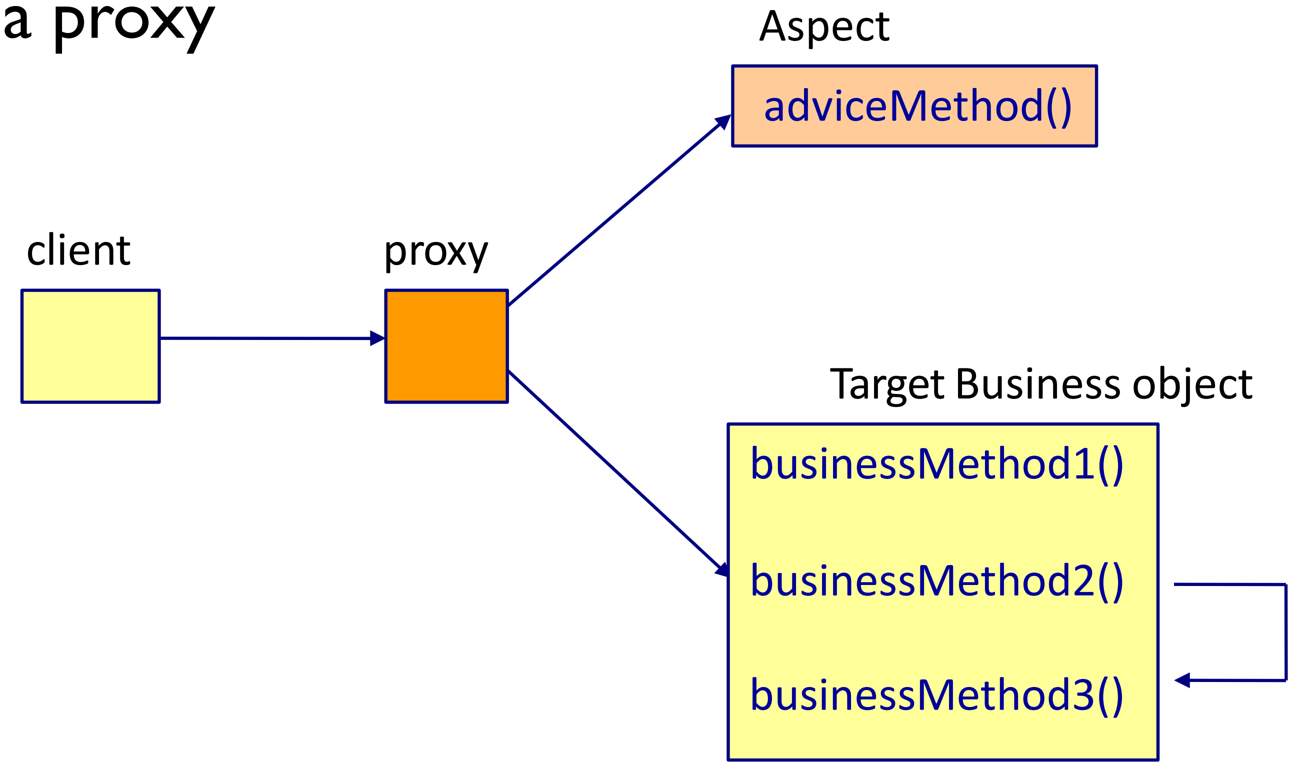
- Be aware though:
  - Calling methods on these objects will be without AOP!
  - See next section for more info

CS544 EA

AOP: Proxy Weaving Gotchas

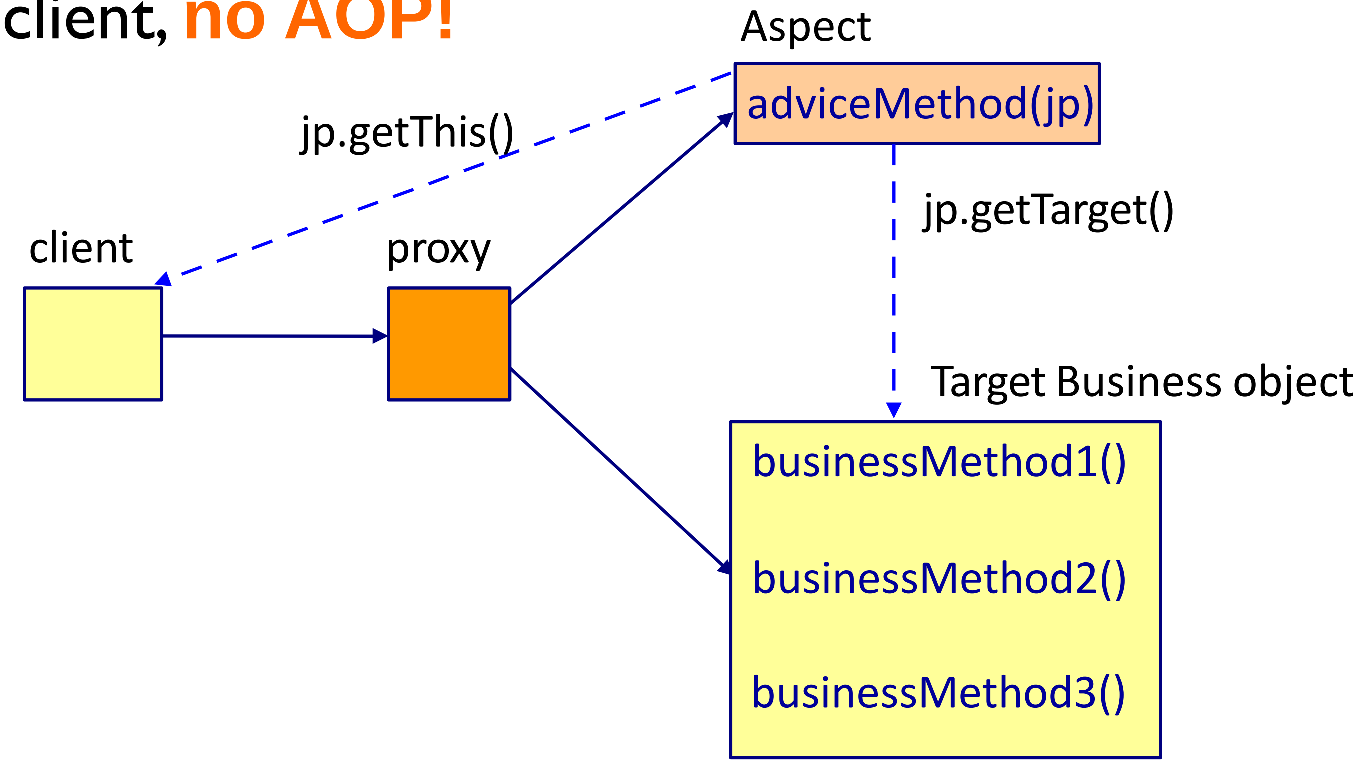
Local calls

- There is **no way** for Spring to **intercept** a local call with 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

CS544 EA

AOP: All XML

All XML

- It is possible to do AOP without annotations

```
<?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"
xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd">

  <bean id="accountService" class="cs544.spring44.aop.xml.AccountService"/>
  <bean id="traceAspect" class="cs544.spring44.aop.xml.TraceAspect"/>

  <aop:config>
    <aop:aspect id="tracebeforeAspect" ref="traceAspect">
      <aop:before method="tracebeforemethod"
        pointcut="execution(* cs544.spring44.aop.xml.AccountService.addAccount(..))" />
    </aop:aspect>

    <aop:aspect id="traceafterAspect" ref="traceAspect">
      <aop:after method="traceaftermethod"
        pointcut="execution(* cs544.spring44.aop.xml.AccountService.addAccount(..))" />
    </aop:aspect>
  </aop:config>
</beans>
```

```
package cs544.spring44.aop.xml;

import java.util.ArrayList;
import java.util.Collection;

public class AccountService implements IAccountService {
		private Collection<Account> accountList = new ArrayList<>();

		@Override
		public void addAccount(int accountNumber, Customer customer) {
			Account account = new Account(accountNumber, customer);
			accountList.add(account);
			System.out.println("in execution of method addAccount");
		}
}
```

```
package cs544.spring44.aop.xml;

import org.aspectj.lang.JoinPoint;

public class TraceAspect {

		public void tracebeforemethod(JoinPoint joinpoint) {
			System.out.println("before execution of method " + joinpoint.getSignature().getName());
		}

		public void traceaftermethod(JoinPoint joinpoint) {
			System.out.println("after execution of method " + joinpoint.getSignature().getName());
		}
}
```

In Action

```
package cs544.spring44.aop.xml;

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

public class App {
		public static void main(String[] args) {
				ConfigurableApplicationContext context;
				context = new ClassPathXmlApplicationContext("cs544/spring44/aop/xml/springconfig.xml");
				IAccountService as = context.getBean(IAccountService.class);
				as.addAccount(12345, new Customer());
				context.close();
		}
}
```

```
before execution of method addAccount
in execution of method addAccount
after execution of method addAccount
```

Spring

CS544 EA

AOP: Sequence

Sequence

- Say you add 2 before advices on one method
  - Which runs first?
- The **order of execution** is based on the order that Spring finds the advice methods
  - First the order in which Aspect classes are found
    - Order of <bean> tags in xml
    - Alphabetic Order in which component scan finds them
  - Then order of advice methods in class

```xml
<aop:aspectj-autoproxy/>
<bean id="accountService" class="accountpackage.AccountService"/>
<bean id="traceAdvice1" class="aopadvice.TraceAdvice1"/>
<bean id="traceAdvice2" class="aopadvice.TraceAdvice2"/>
```

```java
@Aspect
public class TraceAdvice1 {
  @Before("execution(* accountpackage.AccountService.addAccount(..))")
1 public void tracemethodA(JoinPoint joinpoint) {
      System.out.println("TraceAdvice1:tracemethodA");
  }
  @Before("execution(* accountpackage.AccountService.addAccount(..))")
2 public void tracemethodB(JoinPoint joinpoint) {
      System.out.println("TraceAdvice1:tracemethodB");
  }
}
```

```java
@Aspect
public class TraceAdvice2 {
  @Before("execution(* accountpackage.AccountService.addAccount(..))")
3 public void tracemethodA(JoinPoint joinpoint) {
      System.out.println("TraceAdvice2:tracemethodA");
  }
  @Before("execution(* accountpackage.AccountService.addAccount(..))")
4 public void tracemethodB(JoinPoint joinpoint) {
      System.out.println("TraceAdvice2:tracemethodB");
  }
}
```

```
   <aop:aspectj-autoproxy/>
   <bean id="accountService" class="accountpackage.AccountService"/>
   <bean id="traceAdvice2" class="aopadvice.TraceAdvice2"/>
   <bean id="traceAdvice1" class="aopadvice.TraceAdvice1"/>
```

Switching order of <bean> tags

```
  @Aspect
  public class TraceAdvice1 {
    @Before("execution(* accountpackage.AccountService.addAccount(..))")
3   public void tracemethodA(JoinPoint joinpoint) {
        System.out.println("TraceAdvice1:tracemethodA");
    }
    @Before("execution(* accountpackage.AccountService.addAccount(..))")
4   public void tracemethodB(JoinPoint joinpoint) {
        System.out.println("TraceAdvice1:tracemethodB");
    }
  }
```

```
  @Aspect
  public class TraceAdvice2 {
    @Before("execution(* accountpackage.AccountService.addAccount(..))")
1   public void tracemethodA(JoinPoint joinpoint) {
        System.out.println("TraceAdvice2:tracemethodA");
    }
    @Before("execution(* accountpackage.AccountService.addAccount(..))")
2   public void tracemethodB(JoinPoint joinpoint) {
        System.out.println("TraceAdvice2:tracemethodB");
    }
  }
```

Summary

- AOP is a way to cleanly write crosscutting concerns once and apply them many times
- There are 5 types of advice, of which the @Around advice is the most powerful
  - Allowing you not just to receive arguments, return, and exceptions, but also to alter them (pass on different ones)
- Pointcut expressions are used to connect advice to joinpoints of which execution() is the most used
- Do be aware that proxy based (interceptor) AOP does have some limitations (cannot intercept local calls etc).