Aspect Oriented Programming

CS544: Enterprise Architecture

Aspect Oriented Programming

- AOP is all about separation of concerns, where concerns that were previously tangled together are separated into different layers of code. Applying the principle that purification (cleaner, simpler code) leads to progress (easier to understand and maintain)
- In this module we will start with a basic discussion on why you may want to use AOP, followed by an overview of the different terms associated with AOP, and a small HelloWorld AOP example.
- We will then take an in depth look at the pointcut execution language, and the different types of 'Advice' that one can provide.
- After which we will finish up by looking at some of the advantages and disadvantages of AOP, and a short demonstration that AOP can also be fully configured with XML.

Aspect-Oriented Programming:

BASICS OF AOP

Crosscutting concern

- Check security for every service level method

```
public class CustomerService {

  public void getAllCustomers(){
    checkSecurity();
    ...
  }

  public void getCustomer(long customerNumber){
    checkSecurity();
    ...
  }

  public void addCustomer(long customerNumber, String firstName){
    checkSecurity();
    ...
  }

  public void removeCustomer(long customerNumber){
    checkSecurity();
    ...
  }
}
```

We have to call checkSecurity() for all methods of all service classes

Crosscutting concern

- Log every call to the database

```
public class AccountDAO {

  public void saveAccount(Account account){
     ...
     Logger.log("...");
  }

  public void updateAccount(Account account){
     ...
     Logger.log("...");
  }

  public void loadAccount(long accountNumber){
     ...
     Logger.log("...");
  }

  public void removeAccount(long accountNumber){
     ...
     Logger.log("...");
  }
}
```

We have to call Logger.log() for all methods of all DAO classes

Good programming practice principles

DRY: Don’t Repeat Yourself

- Write functionality at one place, and only at one place
- Avoid code scattering

SoC: Separation of Concern

- Separate business logic from (technical) plumbing code
- Avoid code tangling

Examples of crosscutting concerns

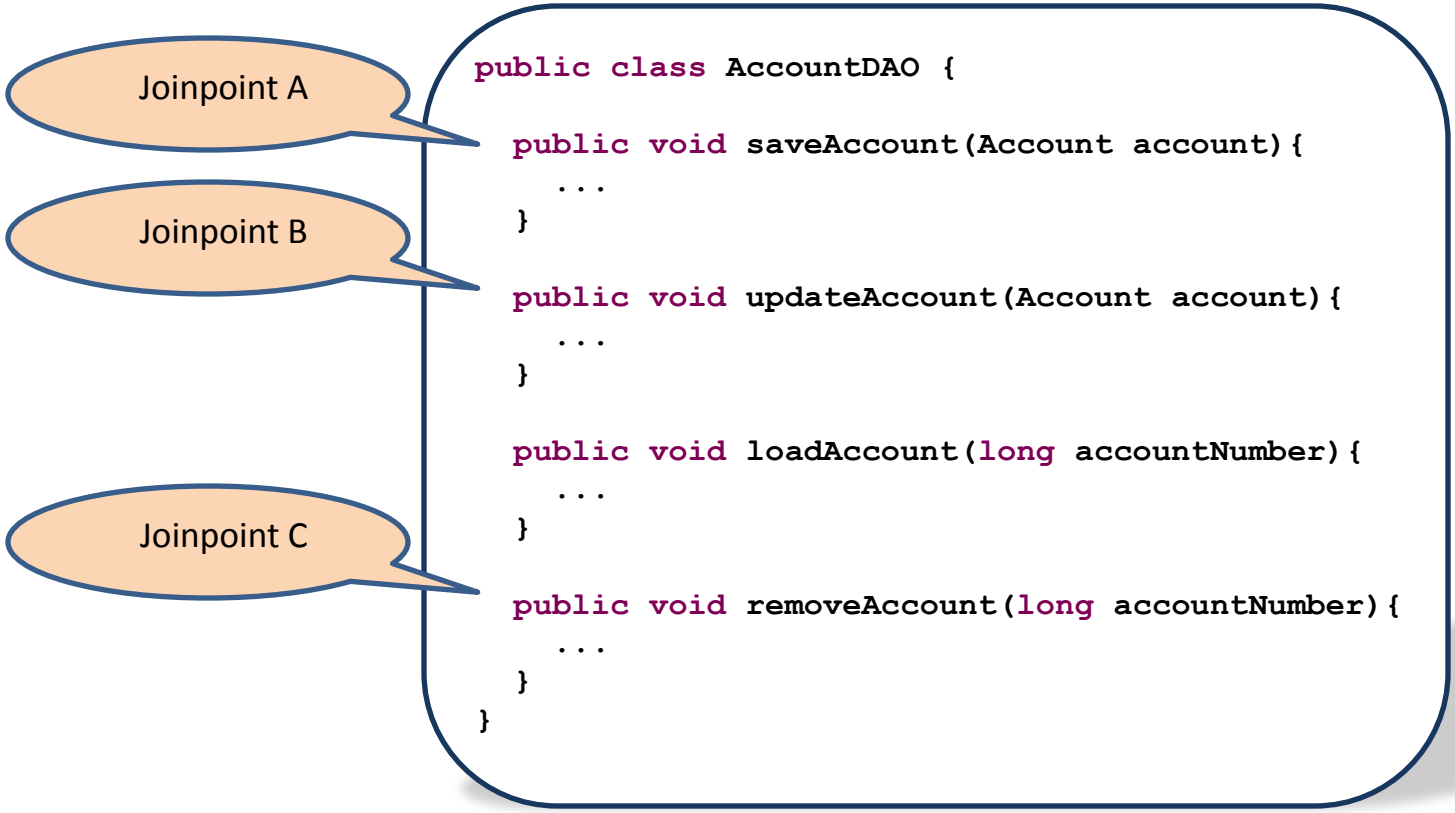
- Tracing
- Transactions
- Security
- Logging
- Exception handling
  - Retry
  - Send a SMS or an email message
- Persistency
- Send an email
- Send a JMS message
- Idempotent services
  - Check if a message is already received

AOP concepts

- Joinpoint
- Pointcut
- Aspect
- Advice
- Weaving

AOP concept: Joinpoint

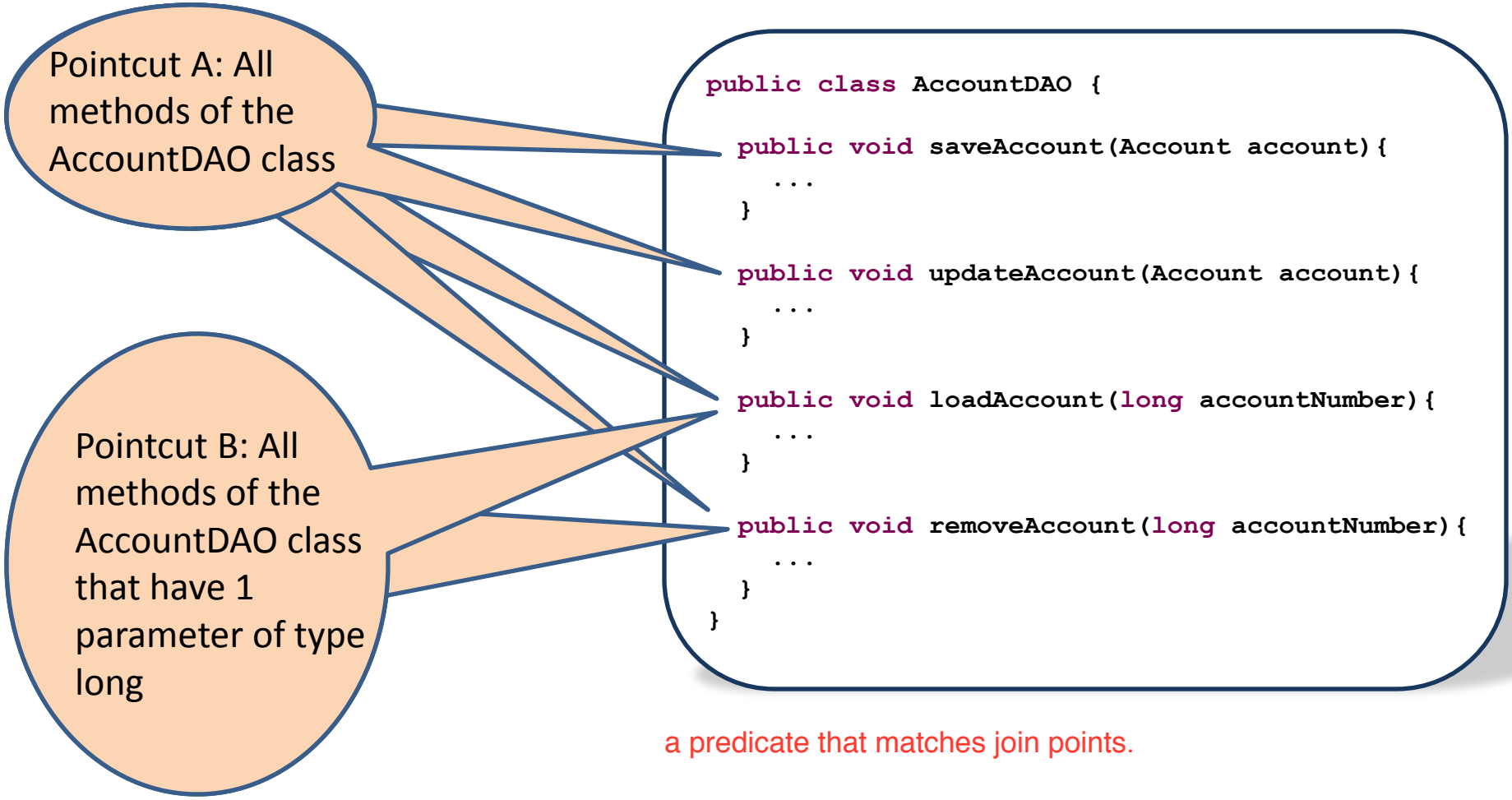
- A specific point in the code



a point during the execution of a program, such as the execution of a method or the handling of an exception. In Spring AOP, a join point always represents a method execution.

AOP concept: Pointcut

- A collection of 1 or more joinpoints



a predicate that matches join points.

AOP concept: Advice

- The implementation of the crosscutting concern

```
public class LoggingAdvice {

  public void log(){
     ...
  }
}
```

```
public class EmailAdvice {

  public void sendEmailMessage(){
     ...
  }
}
```

action taken by an aspect at a particular join point

AOP concept: Aspect

concern that cuts across multiple classes

- What crosscutting concern do I execute (=advice) at which locations in the code (=pointcut)
  - Aspect A: call the log() method of LoggingAdvice before every method call of AccountDAO
  - Aspect B: call the sendEmailMessage() method of EmailAdvice after every method call of AccountDAO that has one parameter of type long

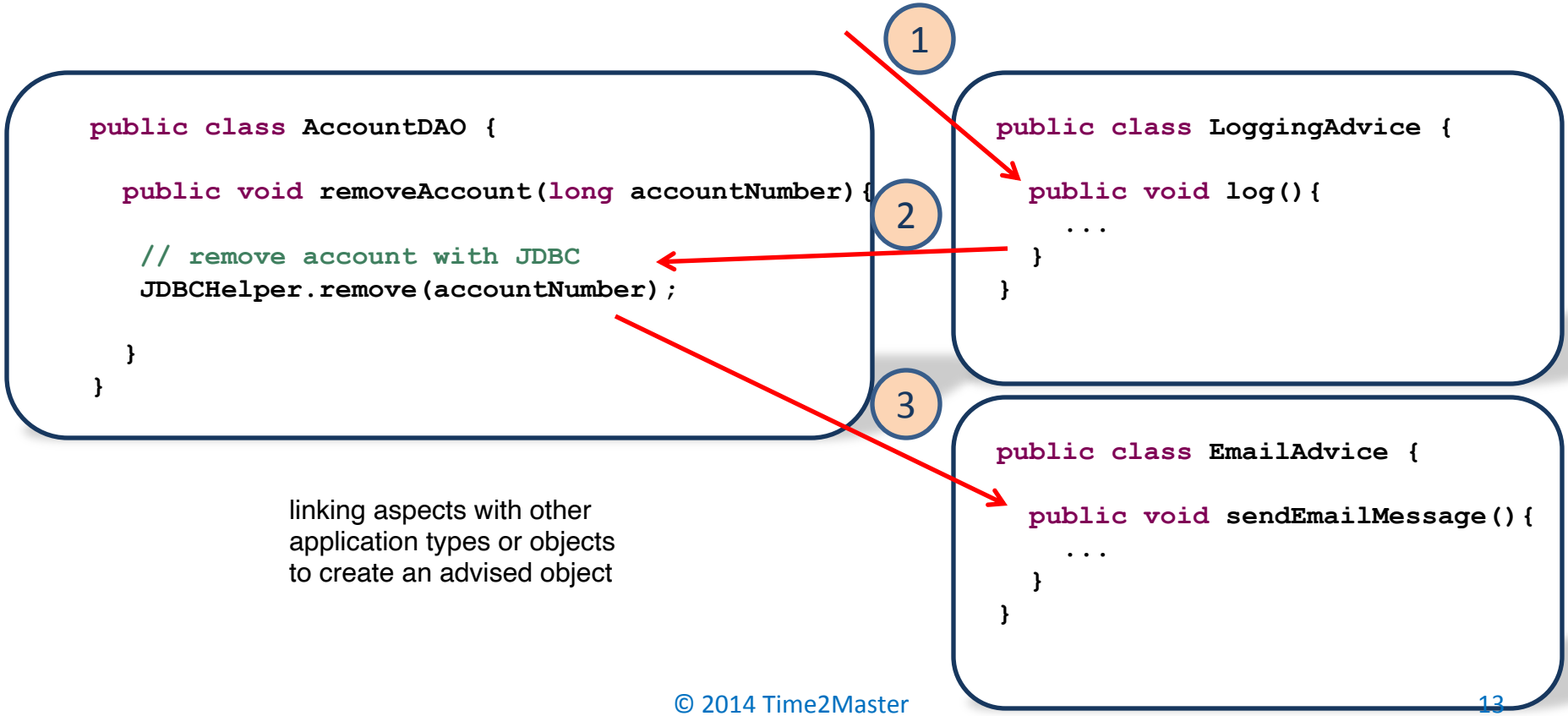
```
public class AccountDAO {

  public void saveAccount(Account account){
    ...
  }

  public void updateAccount(Account account){
    ...
  }

  public void loadAccount(long accountNumber){
    ...
  }

  public void removeAccount(long accountNumber){
    ...
  }
}
```

```
public class LoggingAdvice {

  public void log(){
    ...
  }
}
```

```
public class EmailAdvice {

  public void sendEmailMessage(){
    ...
  }
}
```

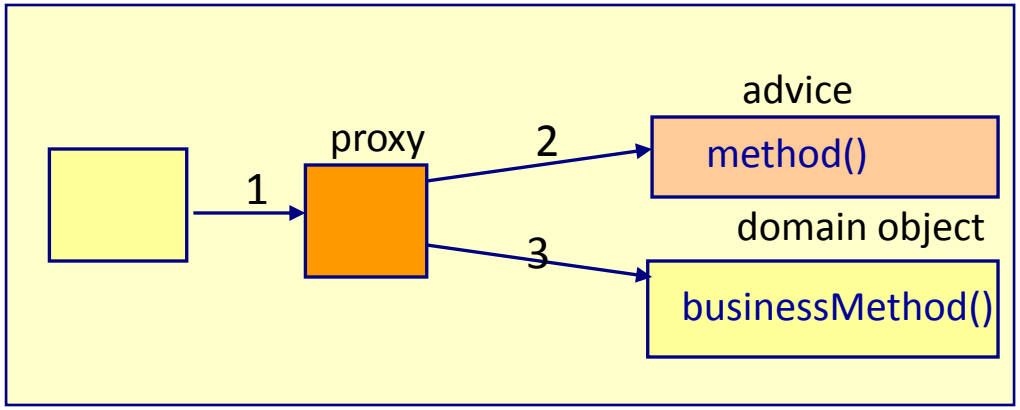
AOP concept: Weaving

- Weave the advice code together with the target code at the corresponding pointcuts such that we get the correct execution

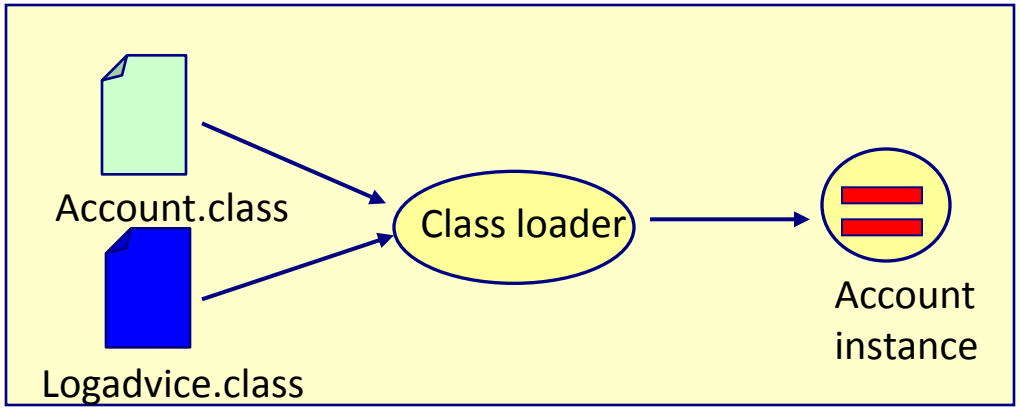


Weaving

Proxy-based weaving



Load time weaving



Helloworld AOP

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

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

The business method

The advice class

```
@Aspect
public class TraceAdvice {
  @Before("execution(* accountpackage.AccountService.*(..))")
  public void tracebeforemethod(JoinPoint joinpoint) {
     System.out.println("before execution of method "+joinpoint.getSignature().getName());
  }
  @After("execution(* accountpackage.AccountService.*(..))")
  public void traceaftermethod(JoinPoint joinpoint) {
     System.out.println("after execution of method "+joinpoint.getSignature().getName());
  }
}
```

The before advice method

The after advice method

Helloworld AOP

```
<?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-3.0.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

  <aop:aspectj-autoproxy/>
  <bean id="accountService" class="accountpackage.AccountService"/>
  <bean id="theTraceAdvice" class="aopadvice.TraceAdvice"/>
</beans>
```

The aop namespace

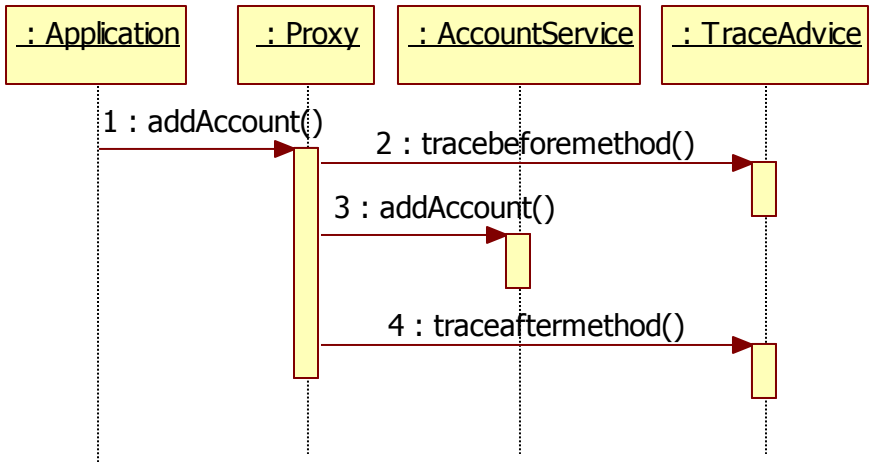
This tag tells Spring that we use annotations based AOP

The advice class needs to be in the XML configuration file

Helloworld AOP

```
public class Application {

  public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("springconfig.xml");
    IAccountService accountService = context.getBean("accountService", IAccountService.class);
    accountService.addAccount("1543", new Customer());
  }
}
```

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



Basics of AOP

- AOP allows us to enjoy greater efficiency and accomplish more by keeping our code simpler and organized.

Aspect-Oriented Programming:

POINTCUT EXECUTION LANGUAGE

Pointcut execution language

Pointcut execution language

```
@Aspect
public class TraceAdvice {
  @Before("execution(* accountpackage.AccountService.*(..))")
  public void tracebeforemethod(JoinPoint joinpoint) {
     System.out.println("before execution of method "+joinpoint.getSignature().getName());
  }
  @After("execution(* accountpackage.AccountService.*(..))")
  public void traceaftermethod(JoinPoint joinpoint) {
     System.out.println("after execution of method "+joinpoint.getSignature().getName());
  }
}
```

Pointcut execution language

```
@Before("execution(public * pkg.*.*(..))")
```

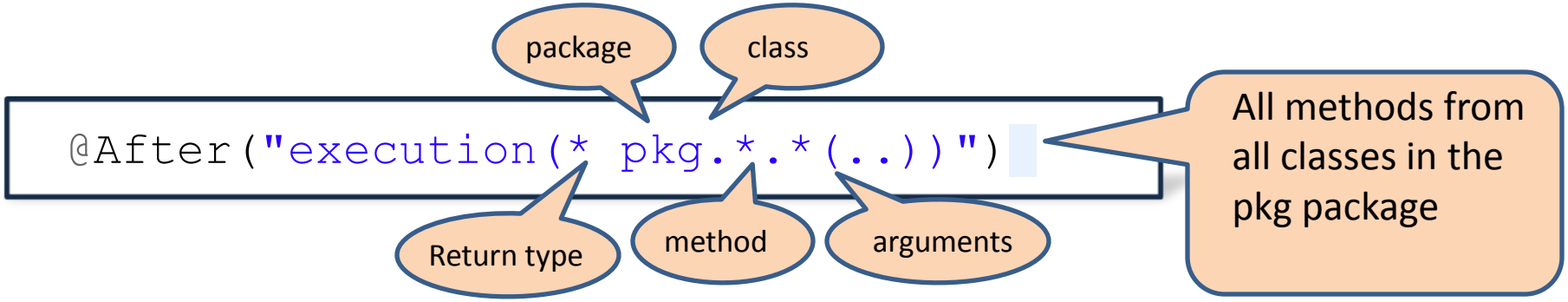
Visibility:
- Possibilities:
  - private
  - public
  - Protected
- **Optional**
- **Cannot** be *

Return type:
- The return type of the corresponding method(s)
- Not optional
- Can be *

package.class.method(args):
- Name of the package can not be *
- Name of the class can also be *
- Name of the method can also be *
- Arguments can be ..
- Not optional
- Can also be *.*(..)
- Can also be *(..)

Package not be left off or *

Pointcut execution language example



Pointcut execution language examples

```
@After("execution(public * pkg(..))")
```

All public methods

```
@After("execution(public void pkg(..))")
```

All public methods that return void

```
@After("execution(* order.*.*(..))")
```

All methods from all classes in the order package

```
@After("execution(* pkg.*.create*(..))")
```

All methods that start with create

```
@After("execution(* pkg.Customer.*(..))")
```

All methods from the Customer class

Pointcut execution language examples

```
@After("execution(* order.Customer.*(..))")
```

All methods from the Customer class in the order package

```
@After("execution(* order.Customer.getPayment(..))")
```

The getPayment () method from the Customer class in the order package

```
@After("execution(* order.Customer.getPayment(int))")
```

The getPayment () method with a parameter of type int from the Customer class in the order package

```
@After("execution(* pkg.*.*(long,String))")
```

All methods from all classes that have 2 parameters, the first of type long, and the second of type String

Pointcut Composition

- Pointcut expressions can be combined with the boolean operators:
  - **&&**, **||**, and **!**
- Their word forms are allowed as well:
  - **and**, **or**, and **not**

For more detail see the aspectj docs at:
http://www.eclipse.org/aspectj/doc/released/progguide/language-joinPoints.html

Named pointcut

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

```
@Aspect
public class CheckOrderAdvice {
  @Pointcut("execution(* *.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");
  }
}
```

Named pointcut: create a pointcut with name "checkOrder"

Use the named pointcut

Use the named pointcut

Named pointcut

```
@Aspect
public class SystemPointcuts {
  @Pointcut("execution(* pkg.OrderService.*(..))")
  public void checkOrder() {  }
}
```

Named pointcut in another class

```
@Aspect
public class CheckOrderAdvice {

  @Before ("SystemPointcuts.checkOrder()")
  public void checkOrder(JoinPoint joinpoint){
    System.out.println("check order");
  }

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

Use the named pointcut

Use the named pointcut

Pointcut designators

- execution
- Within
- target
- @annotation
- args

within

```
@After("within(order.payment.*)")
```

Any method of any class in the order.payment package

```
@After("within(order..*)")
```

Any method of any class in the order package or in a sub-package

target

```
@After("target(order.payment.ProcesPaymentImpl)")
```

Any method of class order.payment.procesPaymentImpl

```
@After("target(order.payment.ProcesPayment)")
```

Any method of the class that implements the order.payment.procesPayment interface

```
@After("target(order.payment.ProcesPayment) ||
 target(order.payment.VerifyPayment) ")
```

Any method of the class order.payment.procesPaymentImpl or order.payment.verifypaymentImpl

@annotation

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

Any method that is annotated with the @transactional annotation

args

```
@After("args(int, String, mum.edu.cs490.Person)")
```

Any method with 3 parameters with types int, java.lang.String, and mum.edu.cs490.Person

```
@After("!args(int)")
```

Any method that does not have 1 parameter of type int

Get parameters with args

```
public class Customer {
  private String name;

  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
}
```

```
@Aspect
public class TraceAdvice {
  @After("execution(* mypackage.Customer.setName(..)) && args(name)")
  public void tracemethod(JoinPoint joinpoint, String name) {
     System.out.println("method ="+joinpoint.getSignature().getName());
     System.out.println("parameter name ="+name);
  }
}
```

Add 'args' parameter

Add parameter(s) to the advice method

Get parameters example

```
public class Application {

  public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("springconfig.xml");
    Customer customer = context.getBean("customer", Customer.class);
    customer.setName("Frank Brown");
    System.out.println(customer.getName());
  }
}
```

```
public class Customer {
  private String name;

  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
}
```

```
method =setName
parameter name =Frank Brown
Frank Brown
```

```
@Aspect
public class TraceAdvice {
  @After("execution(* mypackage.Customer.setName(..)) && args(name)")
  public void tracemethod(JoinPoint joinpoint, String name) {
     System.out.println("method ="+joinpoint.getSignature().getName());
     System.out.println("parameter name ="+name);
  }
}
```

Add 'args' parameter

Add parameter(s) to the advice method

Get parameters

```
public class Customer {
  private String name;
  private int age;

  public void setNameAndAge(String name, int age){
    this.name = name;
    this.age = age;
  }
}
```

2 parameters

```
@Aspect
public class TraceAdvice {
  @Before("execution(* mypackage.Customer.setNameAndAge(..)) && args(name,age)")
  public void tracemethod(JoinPoint joinpoint, String name, int age) {
     System.out.println("method ="+joinpoint.getSignature().getName());
     System.out.println("parameter name ="+name);
     System.out.println("parameter age ="+age);
  }
}
```

Add name and age to the args parameter

Add 2 parameters to the advice method

Pointcut Execution Language

- The Pointcut Execution Language has a varied yet relatively straight forward syntax that allows you to specify onto which methods your advice should be applied.
- The biggest problem with pointcut expressions is that there is no compile time checking on them, they are nothing but a string of metadata until they are interpreted at runtime.

Aspect-Oriented Programming:

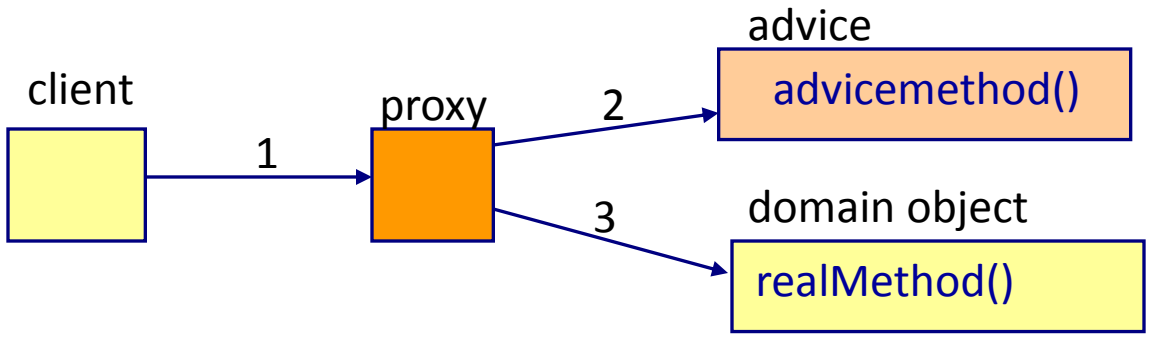
DIFFERENT TYPES OF ADVICE

Advice types

- Before
- After (finally)
- After returning
- After throwing
- Around

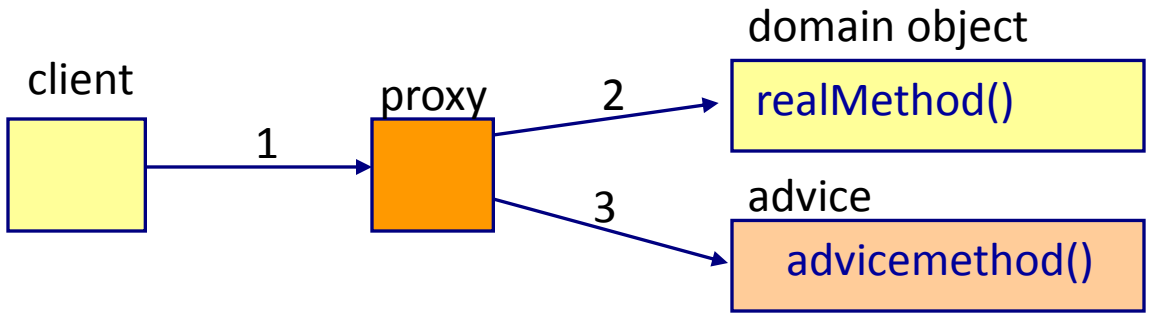
Before advice

- First call the advice method and then the real logic method



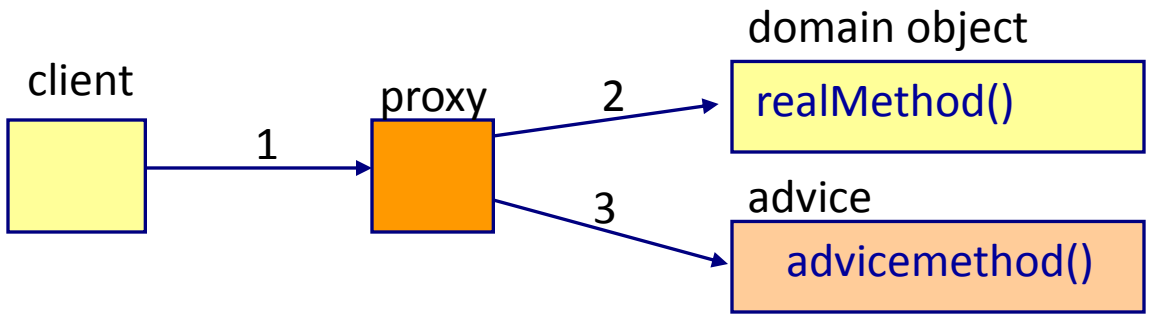
After advice

- First call the real logic method and then call the advice method (independent of how the business logic method returned: normally or with exception)



After returning advice

- First call the real logic method and when this business logic method returns normally without an exception, then call the advice method



Getting the return value

- Works only for @AfterReturning

```
public class Customer {
  private String name;

  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
}
```

getName() returns a String

```
@Aspect
public class TraceAdvice {
  @AfterReturning(pointcut="execution(* mypackage.Customer.getName(..))",returning="retValue")
  public void tracemethod(JoinPoint joinpoint, String retValue) {
    System.out.println("method ="+joinpoint.getSignature().getName());
    System.out.println("return value ="+retValue);
  }
}
```

The pointcut expression

Add 'returning' parameter

Notice the use of the JoinPoint object, more on this later

Add parameter to the advice method. The name of the parameter must be the same as the name of the returning parameter of the @AfterReturning annotation

Getting the return value example

```
public class Application {

  public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("springconfig.xml");
    Customer customer = context.getBean("customer", Customer.class);
    customer.setName("Frank Brown");
    System.out.println(customer.getName());
  }
}
```

```
public class Customer {
  private String name;

  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
}
```

```
method =getName
return value =Frank Brown
Frank Brown
```

Add 'returning' parameter

```
@Aspect
public class TraceAdvice {
  @AfterReturning(pointcut="execution(* mypackage.Customer.getName(..))",returning="retValue")
  public void tracemethod(JoinPoint joinpoint, String retValue) {
    System.out.println("method ="+joinpoint.getSignature().getName());
    System.out.println("return value ="+retValue);
  }
}
```

Add parameter to the advice method

Getting the return value

```
public class Customer {
private int age;

  public int getAge() {
    return age;
  }
  public void setAge(int age) {
    this.age = age;
  }
}
```

getAge() returns an integer

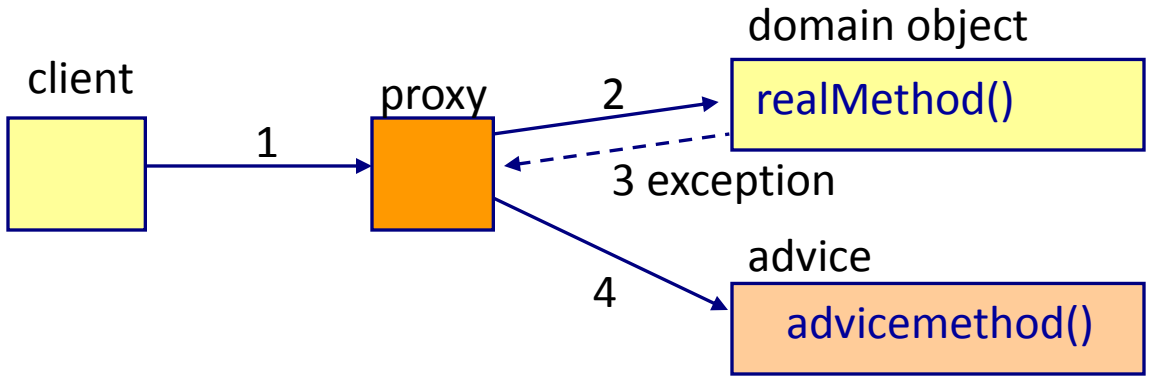
Add 'returning' parameter

```
@Aspect
public class TraceAdvice {
  @AfterReturning(pointcut="execution(* mypackage.Customer.getAge(..))",returning="retValue")
  public void tracemethod(JoinPoint joinpoint, int retValue) {
    System.out.println("method ="+joinpoint.getSignature().getName());
    System.out.println("return value ="+retValue);
  }
}
```

retValue is an int

After throwing advice

- First call the real logic method and when this business logic method throws an exception, then call the advice method



Getting the exception

- Works only for @AfterThrowing

```
public class Customer {
  public void myMethod() throws MyException{
    throw new MyException("myexception");
  }
}
```

```
public class MyException extends Exception{
  private String message;

  public MyException(String message){
    this.message=message;
  }
  public String getMessage(){
    return message;
  }
}
```

Add 'throwing' parameter

```
@Aspect
public class TraceAdvice {
  @AfterThrowing(pointcut="execution(* mypackage.Customer.myMethod(..))",throwing="exception")
  public void tracemethod(JoinPoint joinpoint, MyException exception) {
     System.out.println("method ="+joinpoint.getSignature().getName());
     System.out.println("exception message ="+exception.getMessage());
  }
}
```

Add parameter to the advice method

Getting the exception example

```
public class Application {

  public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("springconfig.xml");
    Customer customer = context.getBean("customer", Customer.class);
    try {
      customer.myMethod();
    } catch (MyException e) {
      // exception handled by advice
    }
  }
}
```

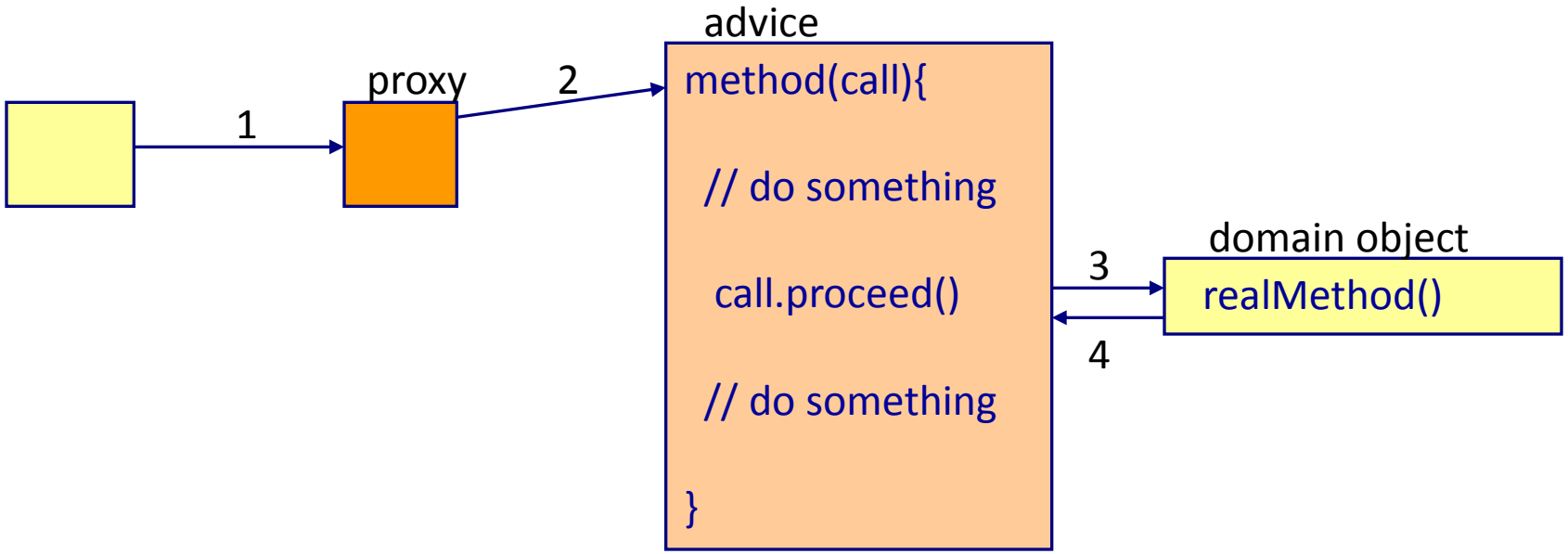
```
public class Customer {
  public void myMethod() throws MyException{
    throw new MyException("myexception");
  }
}
```

```
public class MyException extends Exception{
  private String message;

  public MyException(String message){
    this.message=message;
  }
  public String getMessage(){
    return message;
  }
}
```

```
@Aspect
public class TraceAdvice {
  @AfterThrowing(pointcut="execution(* mypackage.Customer.myMethod(..))",throwing="exception")
  public void tracemethod(JoinPoint joinpoint, MyException exception) {
     System.out.println("method ="+joinpoint.getSignature().getName());
     System.out.println("exception message ="+exception.getMessage());
  }
}
```

Around advice

- First call the advice method. The advice method calls the real logic method, and when the business logic method returns, we get back to the advice method



Around example

ProceedingJoinpoint instead of a regular JoinPoint

```
@Around("execution(* *.*.*(..))")
public Object profile (ProceedingJoinPoint call) throws Throwable{
StopWatch clock = new StopWatch("");
clock.start(call.toShortString());

Object object= call.proceed();

clock.stop();
System.out.println(clock.prettyPrint());
return object;
}
```

Create and start a stopwatch

Call the real logic method

Stop the stopwatch and print result

```
StopWatch '': running time (millis) = 1
-----------------------------------------
ms     %     Task name
-----------------------------------------
00001  100%  execution(addAccount)
```

Get parameters from A regular Joinpoint

```
public class Customer {
  private String name;

  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
}
```

```
@Aspect
public class TraceAdvice {
 @After("execution(* mypackage.Customer.setName(..))")
  public void tracemethodA(JoinPoint joinpoint) {
  Object[] args = joinpoint.getArgs();
  String name = (String)args[0];
  System.out.println("method ="+joinpoint.getSignature().getName());
  System.out.println("parameter name ="+name);
  }
}
```

Get the arguments from the joinpoint

Take the first argument

Get multiple parameters from the Joinpoint

```
public class Customer {
  private String name;
  private int age;

  public void setNameAndAge(String name, int age){
    this.name = name;
    this.age = age;
   }
}
```

2 parameters

```
@Aspect
public class TraceAdvice {
  @Before("execution(* mypackage.Customer.setNameAndAge(..))")
  public void tracemethod(JoinPoint joinpoint ) {
     Object[] args = joinpoint.getArgs();
     String name = (String)args[0];
     int age = (Integer)args[1];
     System.out.println("method ="+joinpoint.getSignature().getName());
     System.out.println("parameter name ="+name);
     System.out.println("parameter age ="+age);
  }
}
```

Changing parameters from @Around ProceedingJoinPoint (1/2)

```
public class Calculator {
   public int add(int x, int y){
   System.out.println("Calculator.add receiving x= "+x+" and y= "+y);
   return x+y;
   }
}
```

```
@Aspect
public class CalcAdvice {
  @Around("execution(* Calculator.add(..))")
  public Object changeNumbers (ProceedingJoinPoint call) throws Throwable{
    Object[] args = call.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= call.proceed(args);

    System.out.println("CalcAdvice.changeNumbers: call.proceed returns "+object);
    return 26;
  }
}
```

Change the parameters

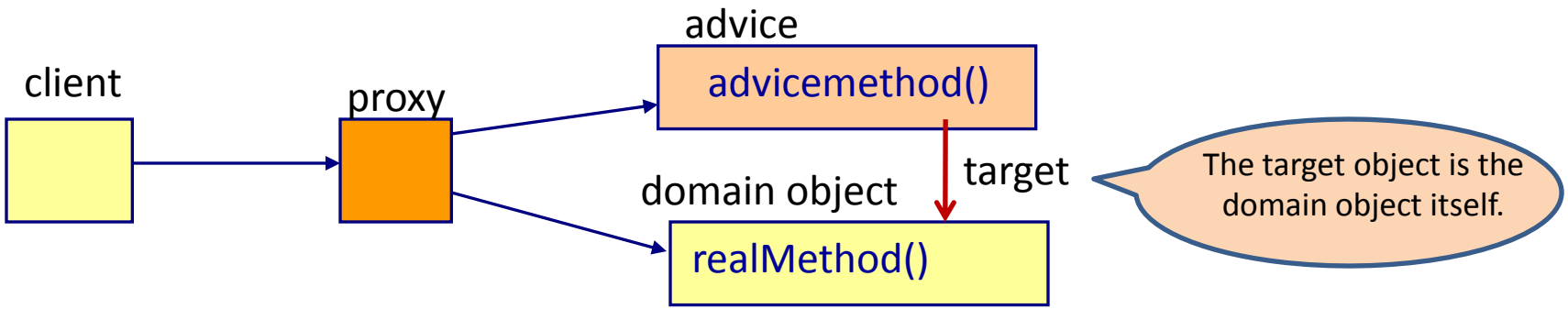
Add parameters to proceed()

Changing parameters from @Around (2/2)

```
public class CalcApplication {

  public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("springconfig.xml");
    ICalculator calculator = context.getBean("calculator", ICalculator.class);
    int result = calculator.add(3, 4);
    System.out.println("The result of 3 + 4 = "+result);
  }
}
```

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

The target class



Get the target class

```
public class Customer {
   private String name;
   private int age;

   public int getAge() {
     return age;
   }
   public void setAge(int age) {
     this.age = age;
   }
   public String getName() {
     return name;
   }
   public void setName(String name) {
     this.name = name;
   }
}
```

Get the target object from the joinpoint

```
@Aspect
public class TraceAdvice {
 @After("execution(* mypackage.Customer.setName(..))")
   public void tracemethod(JoinPoint joinpoint) {
     Customer customer = (Customer)joinpoint.getTarget();
     System.out.println("method ="+joinpoint.getSignature().getName());
     System.out.println("customer age ="+customer.getAge());
  }
}
```

Order of execution

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

First execute the advice methods of TraceAdvice1, and then TraceAdvice2

```
@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");
  }
}
```

In TraceAdvice1, first execute tracemethodA and then traceMethodB

```
@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");
  }
}
```

In TraceAdvice2, first execute tracemethodA and then traceMethodB

Order of execution

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

First execute the advice methods of TraceAdvice2, and then TraceAdvice1

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

3

4

In TraceAdvice1, first execute tracemethodA and then traceMethodB

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

1

2

In TraceAdvice2, first execute tracemethodA and then traceMethodB

Different Types of Advice

- There are 5 types of AOP advice: one before, three types of after, and one around.
- The around advice gives you the most control, but also requires a bit more programming.
- AOP is not just about what the parts are, but also about how they are related to each other.

Aspect-Oriented Programming:

PROXYING MECHANISMS

Two Ways to Create a Proxy

- JDK dynamic Proxy
  - The target has to implement an interface, proxy will implement the same interface(s)
  - Used to be preferred
- CGLIB sub class Proxy
  - Pre Spring 4, required separate cglib.jar
  - Pre Spring 4, required a default constructor
  - With Spring 4 doesn't even call constructor twice!

Proxy-target-class Property

```
<?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-3.0.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

  <aop:aspectj-autoproxy proxy-target-class="true"/>
  <bean id="accountService" class="accountpackage.AccountService"/>
  <bean id="theTraceAdvice" class="aopadvice.TraceAdvice"/>
</beans>
```

Tells Spring to use CGLib proxies

Aspect-Oriented Programming:

ADVANTAGES AND DISADVANTAGES OF AOP

Advantages of AOP

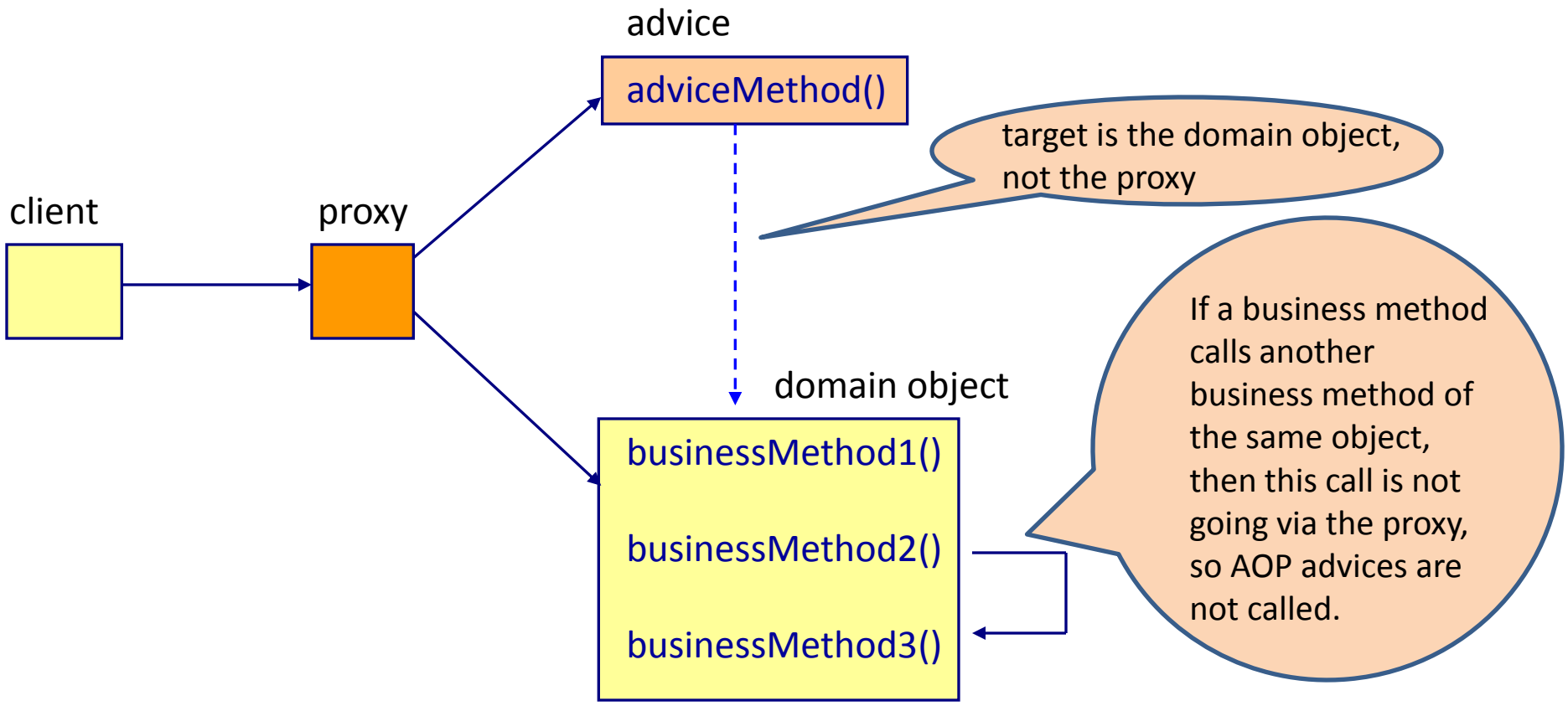
- No code tangling
  - Clean separation of business logic and plumbing code
- No code scattering
  - Same thing, just from a different perspective

Disadvantages of AOP

- You don’t have a clear overview of which code runs when
- A pointcut expression is a string that is parsed at runtime
  - No compile time checking of the pointcut expression
- You make mistakes easily
- Problems with proxy-based AOP

Be careful with AOP: always use unit testing and integration testing with AOP

Disadvantage of a proxy



Disadvantage of a proxy: example

```
public class Application {

  public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("springconfig.xml");
    Customer customer = context.getBean("customer", Customer.class);
    customer.setAge(32);
    customer.setName("Frank Brown");
  }
}
```

```
  @After("execution(* mypackage.Customer.setName(..))")
  public void tracemethodA(JoinPoint joinpoint) {
    Customer customer = (Customer)joinpoint.getTarget();
    System.out.println("method ="+joinpoint.getSignature().getName());
    System.out.println("customer age ="+customer.getAge());
  }
  @After("execution(* mypackage.Customer.getAge(..))")
  public void tracemethodB(JoinPoint joinpoint) {
    System.out.println("method ="+joinpoint.getSignature().getName());
  }
```

Within the advice method we call getAge() on the target, but advice method tracemethodB() is never called

```
method =setName
customer age =32
```

Disadvantage of a proxy: example

```
public class Application {

  public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("springconfig.xml");
    Customer customer = context.getBean("customer", Customer.class);
    customer.setNameAndAge("John Doe", 41);
  }
}
```

```
  @After("execution(* mypackage.Customer.*(..))")
  public void tracemethod(JoinPoint joinpoint) {
     System.out.println("method ="+joinpoint.getSignature().getName());
  }
```

```
public class Customer {
  private String name;
  private int age;

  public void setAge(int age) {
    this.age = age;
  }
  public void setName(String name) {
    this.name = name;
  }
  public void setNameAndAge(String name, int age){
    setName(name);
    setAge(age);
  }
}
```

```
method =setNameAndAge
```

setName() and setAge() are not called via the proxy, so tracemethod() is not called after these 2 methods

Advantages and Disadvantages

- Although AOP provides nice and clean separation of concerns, it is not without its disadvantages.
- Most importantly it's more difficult to know what runs when, pointcut expressions are not compile-time checked, and proxy based AOP has its limitations.

Aspect-Oriented Programming:

CONFIGURING AOP WITH XML

Helloworld AOP with XML

```
public class AccountService {
  Collection<Account> accountList = new ArrayList();

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

The business method

The advice class is a normal POJO

```
public class TraceAdvice {

  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());
  }
}
```

Helloworld AOP with 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:context="http://www.springframework.org/schema/context"
xmlns:aop="http://www.springframework.org/schema/aop"
xsi:schemaLocation="
     http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
     http://www.springframework.org/schema/tx
     http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
     http://www.springframework.org/schema/aop
     http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

  <bean id="accountService" class="accountpackage.AccountService"/>
  <bean id="traceAdvice" class="aopadvice.TraceAdvice"/>

  <aop:config>
    <aop:aspect id="tracebeforeAspect" ref="traceAdvice">
      <aop:before method="tracebeforemethod"
        pointcut="execution(* accountpackage.AccountService.addAccount(..))" />
    </aop:aspect>

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

The ref to the advice class

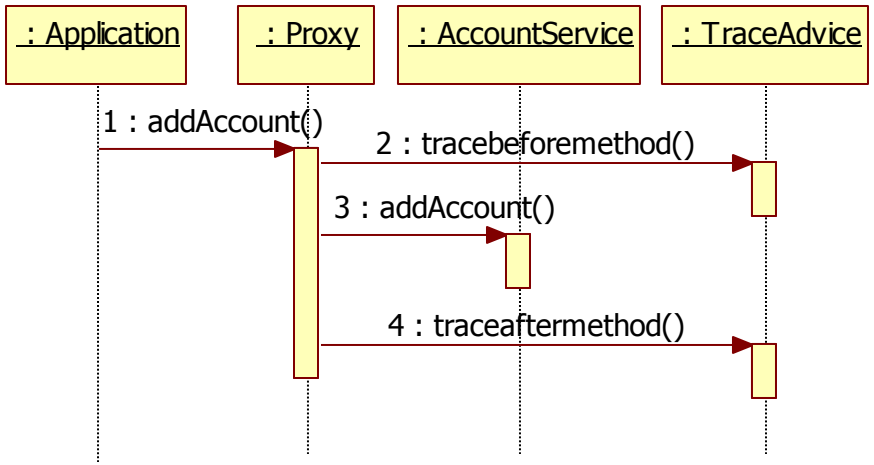
The advice method

The pointcut

Helloworld AOP with XML

```
public class Application {

  public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("springconfig.xml");
    IAccountService accountService = context.getBean("accountService", IAccountService.class);
    accountService.addAccount("1543", new Customer());
  }
}
```

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



Order of execution

```
<bean id="accountService" class="accountpackage.AccountService"/>
<bean id="traceAdvice1" class="aopadvice.TraceAdvice1"/>
<bean id="traceAdvice2" class="aopadvice.TraceAdvice2"/>

<aop:config>
  <aop:aspect id="traceAspect" ref="traceAdvice1">
1   <aop:before method="tracemethodA"
      pointcut="execution(* accountpackage.AccountService.addAccount(..))" />
2   <aop:before method="tracemethodB"
      pointcut="execution(* accountpackage.AccountService.addAccount(..))" />
  </aop:aspect>

  <aop:aspect id="traceafterAspect" ref="traceAdvice2">
3   <aop:before method="tracemethodA"
      pointcut="execution(* accountpackage.AccountService.addAccount(..))" />
4   <aop:before method="tracemethodB"
      pointcut="execution(* accountpackage.AccountService.addAccount(..))" />
  </aop:aspect>
</aop:config>
```

```
TraceAdvice1:tracemethodA
TraceAdvice1:tracemethodB
TraceAdvice2:tracemethodA
TraceAdvice2:tracemethodB
in execution of method addAccount
```

The order of the aspects and the order of the aop methods in the XML file is the order of the execution of the advice methods

Order of execution

```
<bean id="accountService" class="accountpackage.AccountService"/>
<bean id="traceAdvice1" class="aopadvice.TraceAdvice1"/>
<bean id="traceAdvice2" class="aopadvice.TraceAdvice2"/>

<aop:config>
   <aop:aspect id="traceafterAspect" ref="traceAdvice2">
1    <aop:before method="tracemethodB"
       pointcut="execution(* accountpackage.AccountService.addAccount(..))" />
2    <aop:before method="tracemethodA"
       pointcut="execution(* accountpackage.AccountService.addAccount(..))" />
   </aop:aspect>
   <aop:aspect id="traceAspect" ref="traceAdvice1">
3    <aop:before method="tracemethodB"
       pointcut="execution(* accountpackage.AccountService.addAccount(..))" />
4    <aop:before method="tracemethodA"
       pointcut="execution(* accountpackage.AccountService.addAccount(..))" />
   </aop:aspect>
</aop:config>
```

```
TraceAdvice2:tracemethodB
TraceAdvice2:tracemethodA
TraceAdvice1:tracemethodB
TraceAdvice1:tracemethodA
in execution of method addAccount
```

Getting the return value

```
public class Customer {
  private String name;

  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
}
```

getName() returns a String

```
public class TraceAdvice {
  public void tracemethod(JoinPoint joinpoint, String retValue) {
    System.out.println("method ="+joinpoint.getSignature().getName());
    System.out.println("return value ="+retValue);
  }
}
```

Add parameter to the advice method. The name of the parameter must be the same as the name of the returning parameter in the XML file

```
<bean id="customer" class="mypackage.Customer"/>
<bean id="traceAdvice" class="aopadvice.TraceAdvice"/>

<aop:config>
  <aop:aspect id="theAdvice" ref="traceAdvice">
     <aop:after-returning
        method="tracemethod"
        returning="retValue"
        pointcut="execution(* mypackage.Customer.getName(..))"/>
    </aop:aspect>
 </aop:config>
```

Add 'returning' parameter

Getting the exception

```
public class Customer {
  public void myMethod() throws MyException{
    throw new MyException("myexception");
  }
}
```

```
public class MyException extends Exception{
  private String message;

  public MyException(String message){
    this.message=message;
  }
  public String getMessage(){
    return message;
  }
}
```

```
public class TraceAdvice {

  public void tracemethod(JoinPoint joinpoint, MyException exception) {
     System.out.println("method ="+joinpoint.getSignature().getName());
     System.out.println("exception message ="+exception.getMessage());
  }
}
```

Add parameter to the advice method

```
<bean id="customer" class="mypackage.Customer"/>
<bean id="traceAdvice" class="aopadvice.TraceAdvice"/>

<aop:aspect id="theAdvice" ref="traceAdvice">
    <aop:after-throwing
       method="tracemethod"
       throwing="exception"
       pointcut="execution(* mypackage.Customer.myMethod(..))"/>
  </aop:aspect>
</aop:config>
```

Add 'throwing' parameter

Get parameters

```
public class Customer {
  private String name;

  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
}
```

```
public class TraceAdvice {

 public void tracemethod3(JoinPoint joinpoint, String name) {
     System.out.println("method ="+joinpoint.getSignature().getName());
     System.out.println("parameter name ="+name);
  }
}
```

Add parameter(s) to the advice method

```
<bean id="customer" class="mypackage.Customer"/>
<bean id="traceAdvice" class="aopadvice.TraceAdvice"/>

<aop:config>
  <aop:aspect id="theAdvice" ref="traceAdvice">
     <aop:before
        method="tracemethod"
        pointcut="execution(* mypackage.Customer.setName(..)) and args(name)"/>
  </aop:aspect>
</aop:config>
```

Add 'args' parameter

Annotations or XML

XML	Annotations
- Verbose	+ Simple
- Things are specified in 2 places, in Java and in XML	+ Everything specified in one place
+ Nice separation of concern	- Not always a nice separation of concern
+ No recompilation needed after changing the XML	- Needs recompilation after changes to the annotations
+ Works with all Java versions	- Works only with Java 1.5 and higher

Active Learning

- What is a pointcut expression?

- What is a JoinPoint object? How does it differ from a ProceedingJoinPoint?

Summary

- AOP is a technique to avoid code scattering and code tangling
- AOP can be implemented with annotations or with XML
- Spring uses proxy-based weaving by default
- Be careful with AOP, it is very powerful, but has several drawbacks

Main Point

- Aspect Oriented Programming lets us program additional logic in one place, and then declaratively apply that logic to many places.
- *Science of Consciousness*: We create harmony (single implementation), in diversity (applied to many places)