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

 3

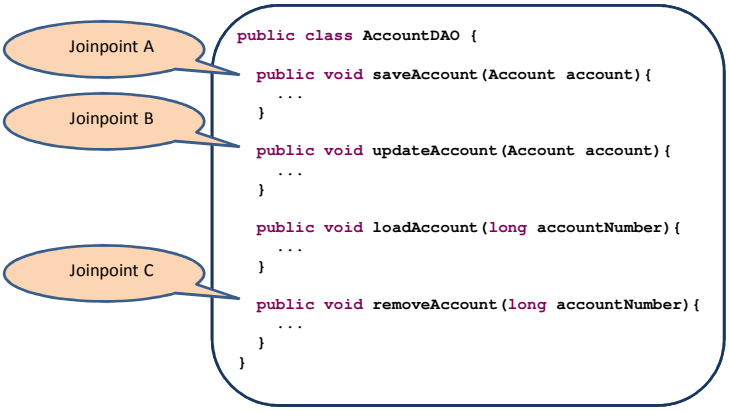
AOP concepts

- Joinpoint
- Pointcut
- Aspect
- Advice
- Weaving

 4

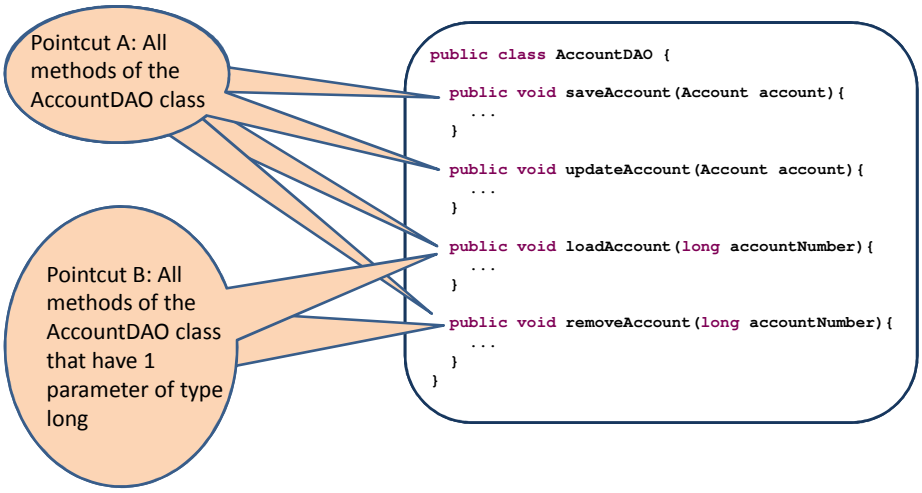
AOP concept: Joinpoint

- A specific point in the code



AOP concept: Pointcut

- A collection of 1 or more joinpoints



AOP concept: Advice

- The implementation of the crosscutting concern

```
public class LoggingAdvice {

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

```
public class EmailAdvice {

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


AOP concept: Aspect

- 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

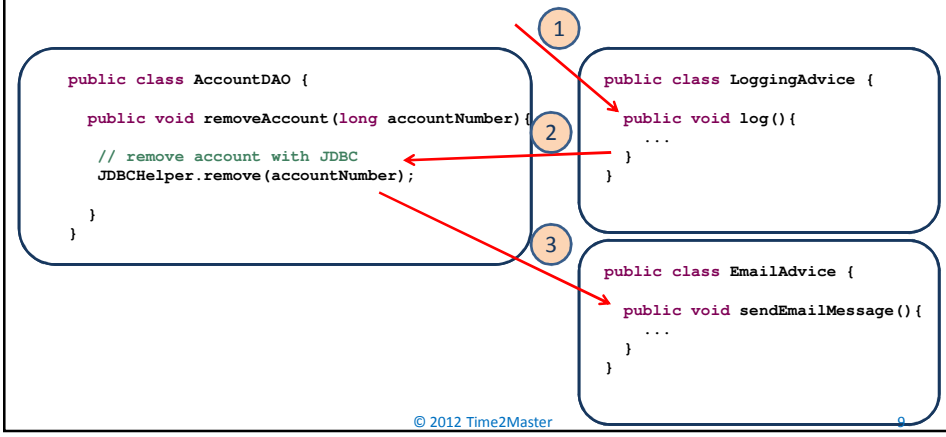
```
public class AccountDAO {

  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

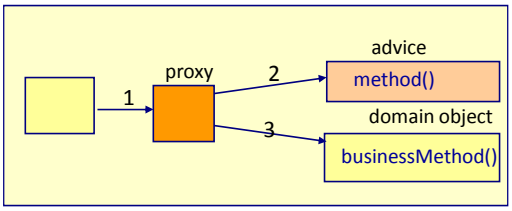
- Weave the advice code together with the target code at the corresponding pointcuts such that we get the correct execution



 9

Weaving

Proxy-based weaving



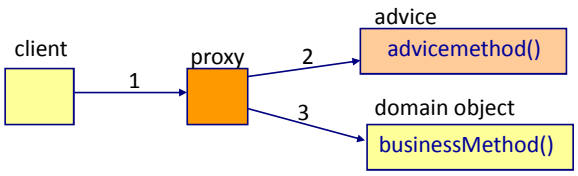
 10

Advice types

- Before
- After returning
- After throwing
- After
- Around

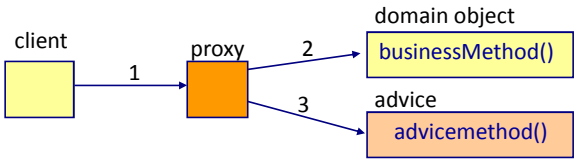
Before advice

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



After returning advice

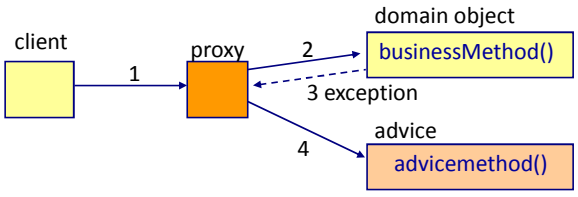
- First call the business logic method and when this business logic method returns normally without an exception, then call the advice method



 13

After throwing advice

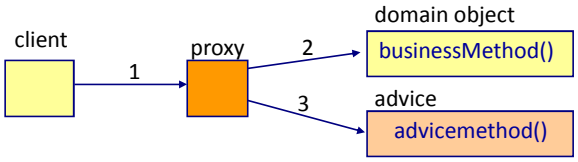
- First call the business logic method and when this business logic method throws an exception, then call the advice method



 14

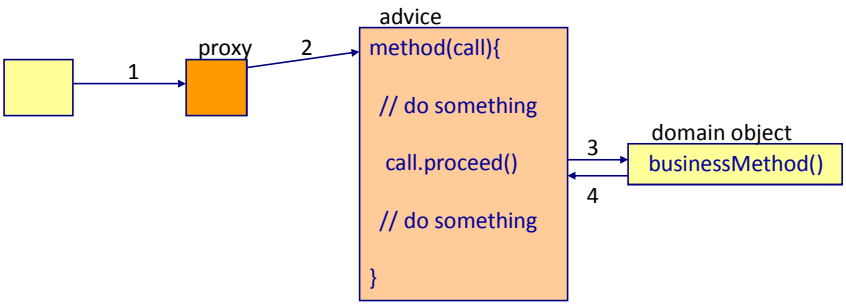
After advice

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



Around advice

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



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 with 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: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-2.5.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-2.5.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop/spring-aop.xsd">

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

The aop namespace

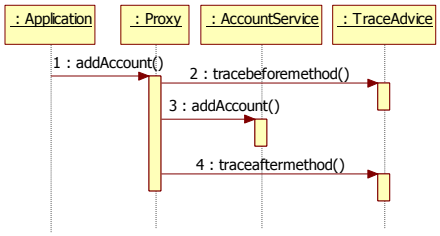
This tag tells Spring that we use annotations based AOP

The advice class needs to be in the XML configuration file

Helloworld AOP with annotations

```
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
```



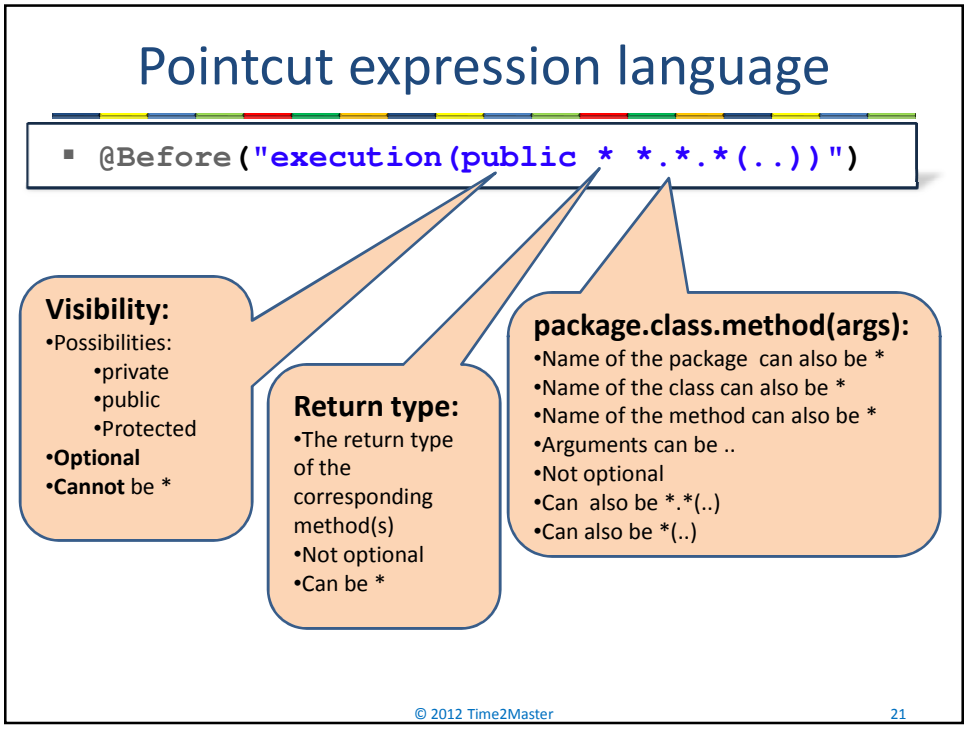
 19

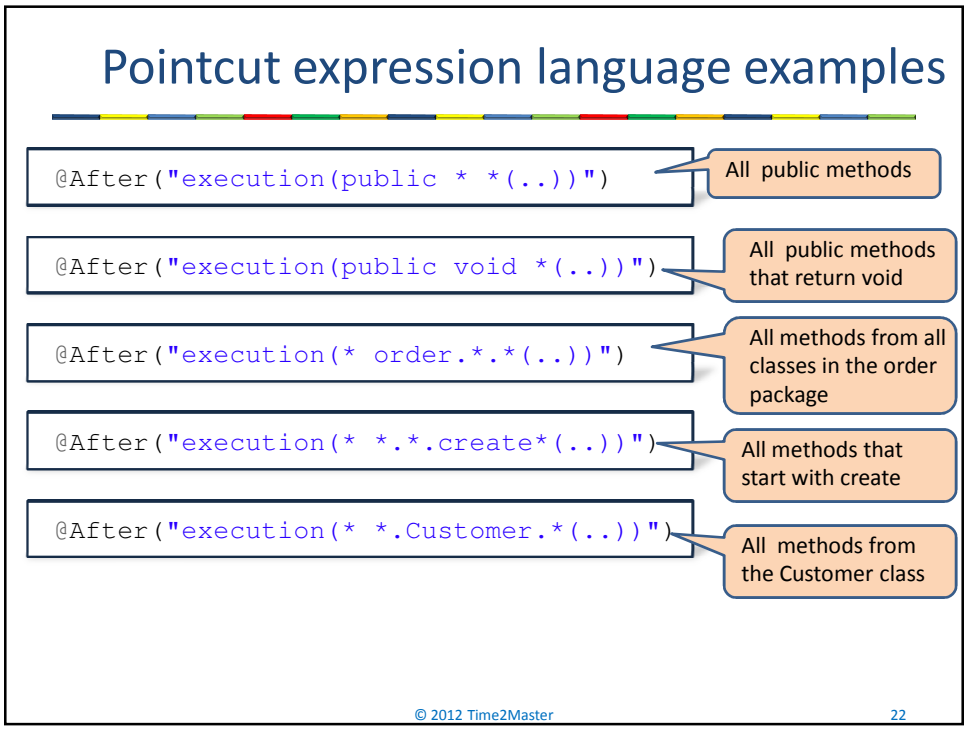
Pointcut expression language

Pointcut expression 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());
  }
}
```

 20


Pointcut expression language
@Before("execution(public * *.*.*(..))")
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 also 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 *(..)
© 2012 Time2Master
21


Pointcut expression language examples
@After("execution(public * *(..))")
All public methods
@After("execution(public void *(..))")
All public methods that return void
@After("execution(* order.*.*(..))")
All methods from all classes in the order package
@After("execution(* *.*.create*(..))")
All methods that start with create
@After("execution(* *.Customer.*(..))")
All methods from the Customer class
© 2012 Time2Master
22

Pointcut expression 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(* *.*.*(long,String))")
```

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

Around example

```
@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 business logic method

Stop the stopwatch and print result

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


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

The pointcut expression

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. The name of the parameter must be the same as the name of the returning parameter of the @AfterReturning annotation

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

Get parameters

```
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

Get parameters from the 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

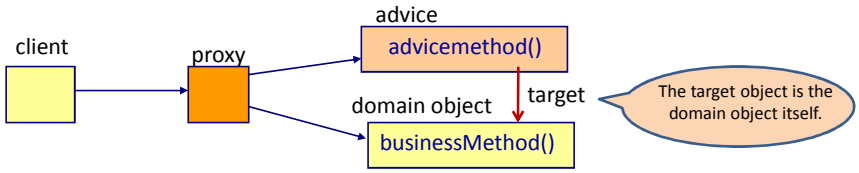
Get multiple parameters from the Joinpoint

```
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);
  }
}
```


The target class



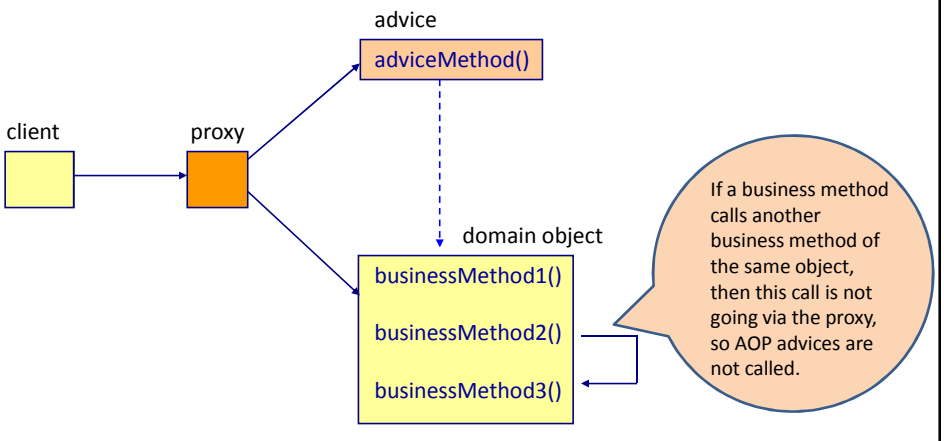
Get the target class

```
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;
  }
}
```

```
@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());
  }
}
```

Get the target object from the joinpoint

Disadvantage of a proxy



Advantages of AOP

- No code tangling
  - Clean separation of business logic and plumbing code
- No code scattering

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 can make mistakes easily
- Problems with proxy-based AOP

When do we use AOP?

- For applying generic functionality to our application
  - Logging
  - Stopwatch
  - Transactions
  - Security
- Not for specific functionality
  - Database access
  - Sending a message