



Spring Core Spring AOP

AOP :: Example

- ♦ Look at the method to get user by id:

```
public class UserService {  
    public UserDT0 getUser(Integer id) {  
        return userDAO.getUser(id);  
    }  
}
```

AOP :: Example

- ♦ Look at the method to get user by id:

```
public class UserService {  
    public UserDTO getUser(Integer id) {  
        return userDAO.getUser(id);  
    }  
}
```

- ♦ Add logging:

```
public UserDTO getUser(Integer id) {  
    log.debug("Call method getUser with id " + id);  
    UserDTO user = userDAO.getUser(id);  
    log.debug("User info is: " + user.toString());  
    return user;  
}
```

AOP :: Example

- ♦ Add exception handling:

```
public UserDTO getUser(Integer id) throws ServiceException{
    log.debug("Call method getUser with id " + id);
    UserDTO user = null;
    UserDTO user = userDao.getUser(id);
    try {
        user = userDao.getUser(id);
    } catch(SQLException e) {
        throw new ServiceException(e);
    }

    log.debug("User info is: " + user.toString());
    return user;
}
```

AOP :: Example

♦Add user rights check:

```
public UserDTO getUser(Integer id) throws ServiceException, AuthException{
    if (!SecurityContext.getUser().hasRight("getUser")) {
        throw new AuthException("Permission Denied");
    }

    log.debug("Call method getUser with id " + id);
    UserDTO user = null;
    UserDTO user = userDAO.getUser(id);

    try {
        user = userDAO.getUser(id);
    } catch (SQLException e) {
        throw new ServiceException(e);
    }

    log.debug("User info is: " + user.toString());
    return user;
}
```

AOP :: Example

♦Add results caching:

```
public UserDTO getUser(Integer id) throws ServiceException, AuthException {  
    ...  
    try {  
        if (cache.contains(cacheKey)) {  
            user = (UserDTO) cache.get(cacheKey);  
        } else {  
            user = userDao.getUser(id);  
            cache.put(cacheKey, user);  
        }  
    } catch (SQLException e) {  
        throw new ServiceException(e);  
    }  
    log.debug("User info is: " + user.toString());  
    return user;  
}
```

AOP :: Example

♦What we get:

- Large amount of the service code
- 16 lines instead of one, and the code continues to grow...

♦Types of orthogonal functionality:

- Logging
- Exception handling
- Transactions
- Caching
- User rights check
- And many others...

♦Disadvantages of the service code inside the main code:

- The code size is growing
- It's more difficult to support
- Code duplication

♦Solution: use aspects

- ⇒ Take of the orthogonal functionality to the separate classes - aspects

How aspects work

ASPECT:



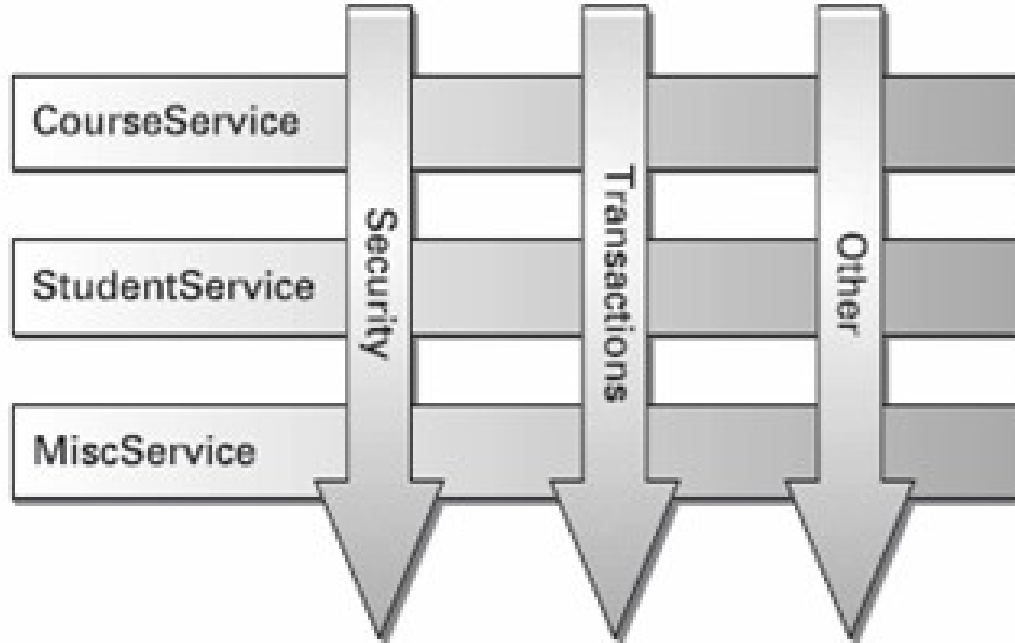
Add logging:

```
public UserDTO getUser(Integer id) {  
    log.debug("Call method getUser with id " + id);  
    UserDTO user = userDao.getUser(id);  
    log.debug("User info is: " + user.toString());  
    return user;  
}
```

Logging aspect

AOP :: Introduction

- Aspect Oriented Programming (AOP)
- AOP gives means for implementing orthogonal (crosscutting) functionality



AOP :: Introduction

- ◆ How can we implement crosscutting logic in RDBMS?

AOP :: Introduction

- ◆ Example of crosscutting logging based on RDBMS triggers:

```
/* Table level triggers */  
CREATE OR REPLACE TRIGGER DistrictUpdatedTrigger  
AFTER UPDATE ON district  
BEGIN  
    INSERT INTO info VALUES ('table "district" has changed');  
END;
```

AOP :: Logging advice example

@Aspect

```
public class LoggingAspect {  
    private final static Logger log = Logger.getLogger(LoggingAspect.class);  
  
    @Pointcut("execution(* *service.*(..))")  
    public void serviceMethod() {  
  
    }  
  
    @Around("serviceMethod()")  
    public Object logWebServiceCall(ProceedingJoinPoint thisJoinPoint) {  
        String methodName = thisJoinPoint.getSignature().getName();  
        Object[] methodArgs = thisJoinPoint.getArgs();  
        log.debug("Call method " + methodName + " with args " + methodArgs);  
        Object result = thisJoinPoint.proceed();  
        log.debug("Method " + methodName + " returns " + result);  
        return result;  
    }  
}
```

AOP :: Logging advice example

```
<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="userDao" class="UserDaoImpl"/>
    <bean name="loggingAspect" class = "LoggingAspect"/>
</beans>
```

AOP :: Logging advice example

```
public interface UserDao {  
    UserDTO getUser(int id);  
}
```

```
public class UserDaoImpl implements UserDao {
```

```
    public UserDTO getUser(int id) {  
        if (null != userDaoMap.get(id)) {  
            return userDaoMap.get(id);  
        }  
    }
```

```
    UserDTO user = new UserDTO(id);  
    userDaoMap.put(id, user);  
    return user;  
}
```

```
}
```

With the use of aspects we can automatically add:

- Logging
- Exception handling
- Transactions
- Caching
- User rights check
- And a lot more...

ex.1

AOP :: Example

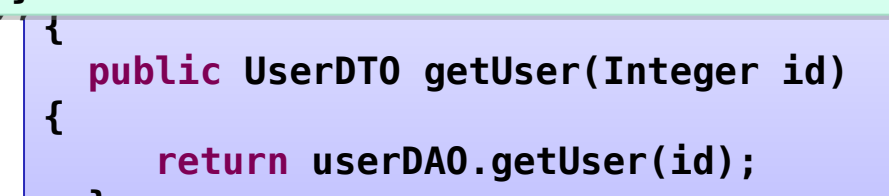
@Aspect

```
public class LoggingAspect {  
    @Pointcut("execution(* *.User(..))")  
    public void userMethod() { }
```

@Around("userMethod() ")

```
public Object log (<img alt="Diagram showing the flow of control from the @Around annotation to the log method, and from the log method to the userMethod() in the LoggingAspect class." data-bbox="23 381 501 956"/>  
    ProceedingJoinPoint thisJoinPoint) {  
        String methodName =  
            thisJoinPoint.getSignature().getName()  
        Object[] methodArgs =  
            thisJoinPoint.getArgs();  
        logger.debug("Call method " + methodName  
            + " with args " + methodArgs);  
  
        Object result = thisJoinPoint.proceed(  
            logger.debug("Method " + methodName  
                + " returns " + result);  
        return result;  
    }
```

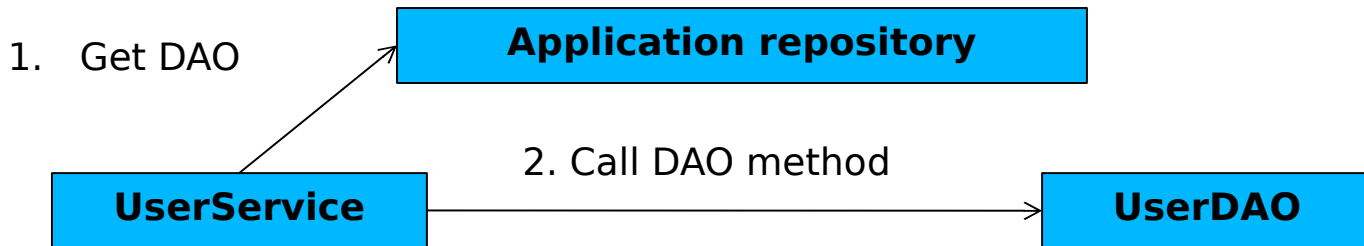
```
class UserDaoProxy implements UserDao {  
  
    public UserDTO getUser(final Integer  
id)  
    {  
        Aspect logger = new  
LoggingAspect();  
        ProceedingJoinPoint joinpoint =  
            new ProceedingJoinPoint() {  
            Object proceed() {  
                return userDao.getUser(id);  
            }  
        };  
        return logger.log(joinpoint);  
    }  
}
```



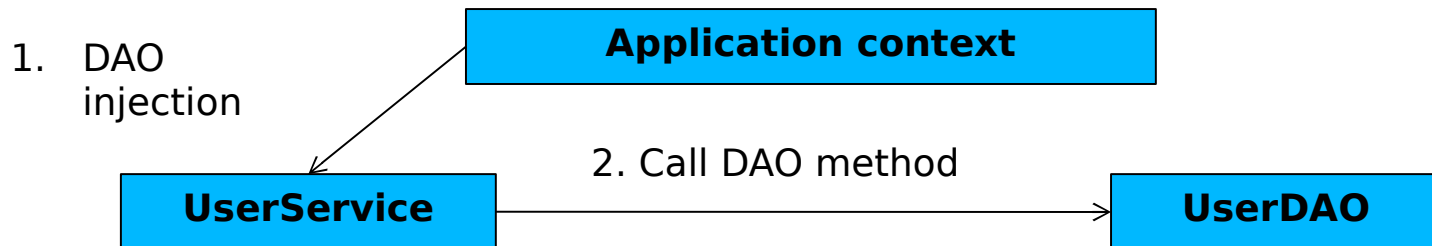
```
{  
    public UserDTO getUser(Integer id)  
    {  
        return userDao.getUser(id);  
    }  
}
```

AOP :: Introduction

Working with DAO without IoC and AOP

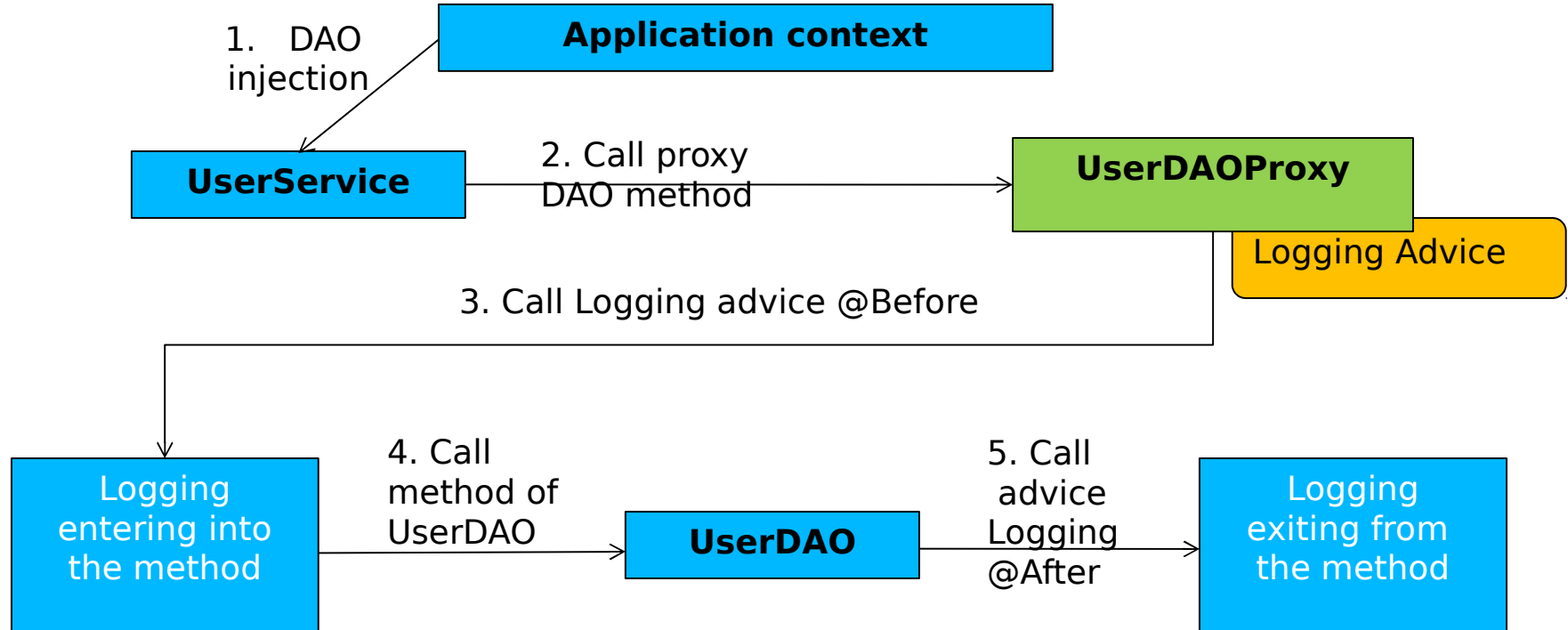


Working with DAO with IoC, but without AOP



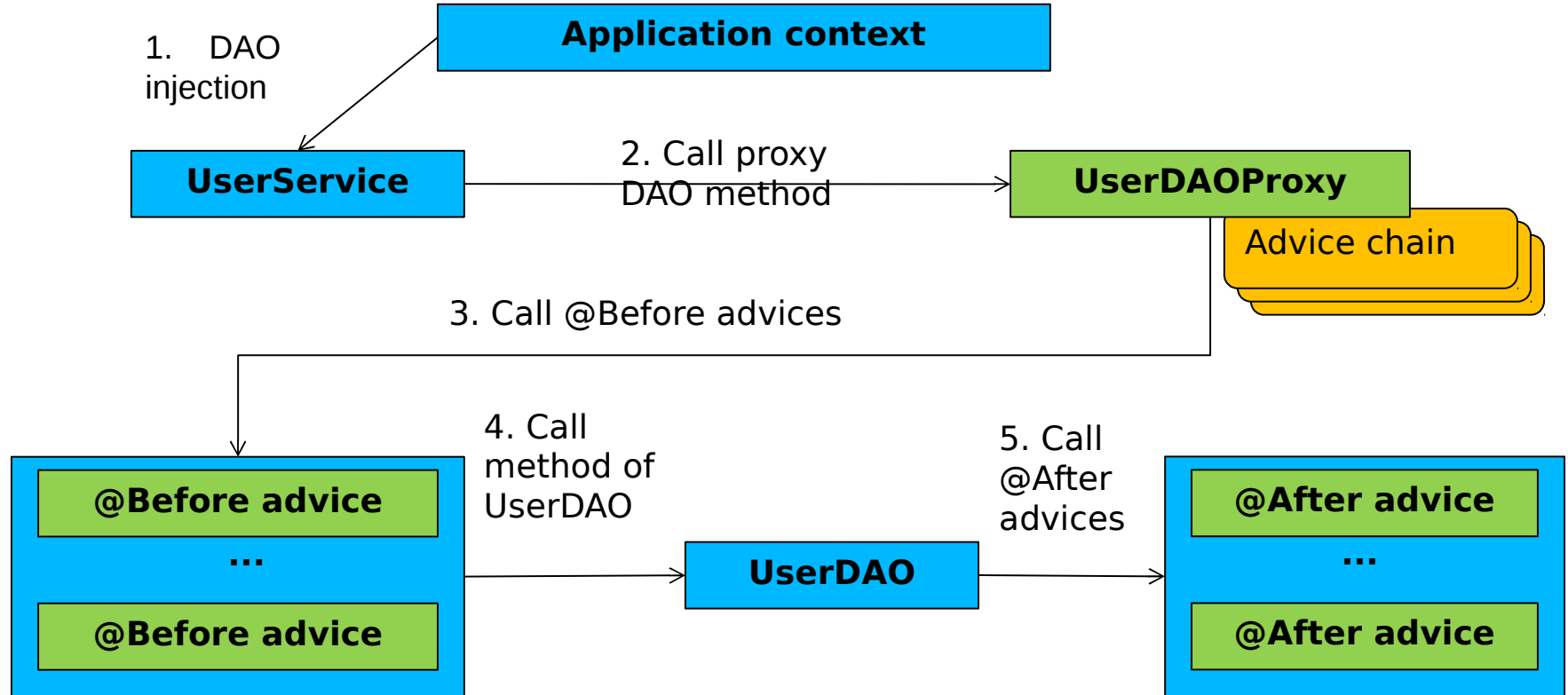
AOP :: Introduction

Working with DAO with IoC and AOP



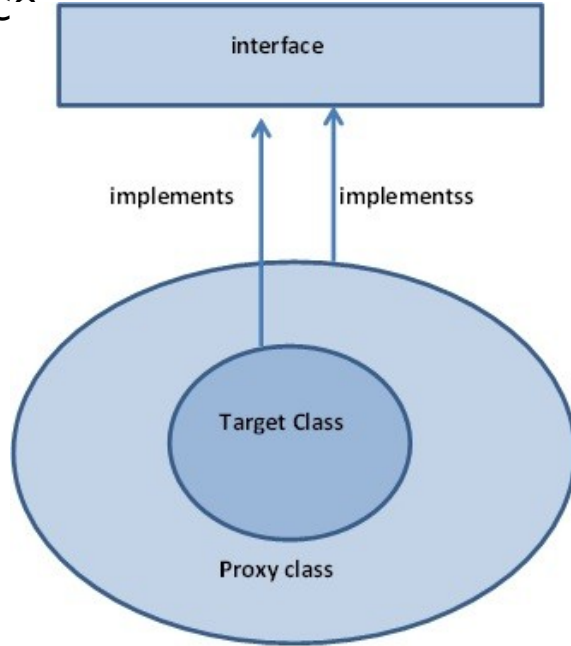
AOP :: Introduction

Working with DAO with IoC and AOP

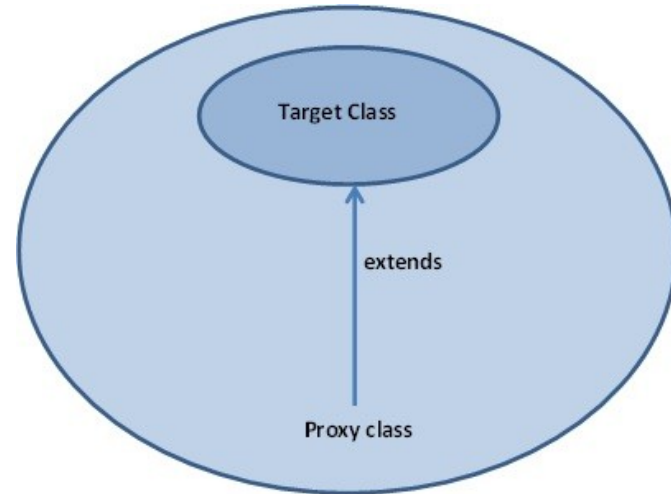


AOP :: Introduction

In Spring Framework AOP is implemented by creating proxy object for your service~

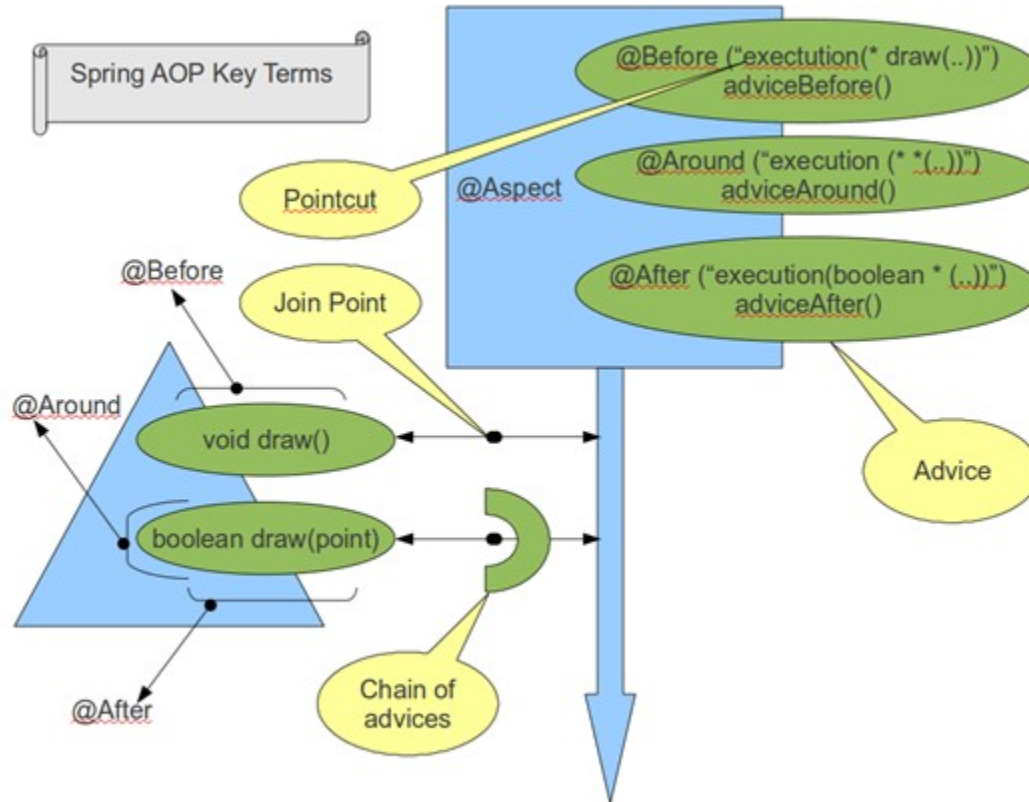


Standard mechanism of proxy
creation from JSE



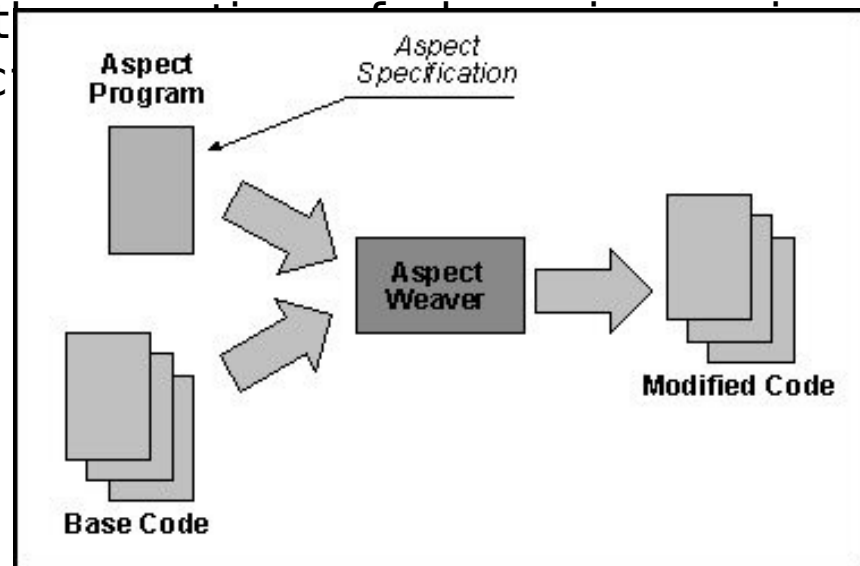
CGLIB proxy

AOP :: Key terms



AOP activation

- ♦ Weaving: applying the aspect to target object to create new proxy object;
- ♦ There are 2 additional dependencies to perform weaving:
 aspectjrt.jar
 aspectjweaver.jar
- ♦ Also you need to initiate the weaving in the configuration file: `<aop:aspect`



AOP :: Pointcut language

execution – defines the pointcut on the base of method signature

execution(@CustomAnnotation? modifiers-pattern? **ret-type-pattern**

declaring-type-pattern?.**name-pattern(param-pattern)** throws-pattern?)

? – optional parameter

declaring-type-pattern - template for the method and class name

Examples:

- **execution (* *(..))** – pointcut attaches to any method with any signature;
- **execution (int *(..))** – pointcut attaches to any method returning int;
- **execution(* com.package.subpackage.Classname.*(..))** – attaches to every method of com.package.subpackage.Classname class;

AOP :: Pointcut language

- **execution (void Test.foo(int, String))** – pointcut attaches to foo() method of Test class, taking int and String as the parameters;
- **execution (* foo.bar*.dao*.update*(..))** – pointcut attaches to any method starting from «update» in package starting from foo.bar and ending on dao;
- **bean** – attach join points to some Spring bean (or set of beans)
 - **bean(“*Bean”)** – defines the jointpoint for all beans with id ending “Bean”
- **within** – attach join point to every method of some class
 - **within(com.package.subpackage.*)** – any join point (method execution only in Spring AOP) within the package com.package.subpackage
- **this** – matches the join points (the execution of methods when using Spring AOP) where the bean reference (Spring AOP proxy) is an instance of the given type

AOP :: Pointcut

- **this(`com.package.InterfaceName`)** - define join points to all methods in classes which implement the interface `com.package.InterfaceName`
- **target** - matches the join points (the execution of methods when using Spring AOP) where the target object (application object being proxied) is an instance of the given type
 - **target(`com.package.InterfaceName`)** - defines all methods of the object which target object implements `com.package.InterfaceName`
- **args** - match the join points where the arguments are instances of the given types
 - **args(`String`)** - define methods with the single argument of `String` type
- **@annotation** - matching the join points where the subject of the join point (method being executed in Spring AOP) has the given annotation
 - **@annotation(`com.package.annotation.Annotation`)** - all the methods marked by annotation `@Annotation`
 - **@annotation(`org.springframework.stereotype.Repository`)** -

AOP :: Advice types

- ♦ @Around advice – surrounds the joinpoint
- ♦ Most powerful of all advices

```
@Around("@annotation(com.luxoft.springaop.example2.Log)")
public Object log (ProceedingJoinPoint thisJoinPoint) throws Throwable {
    String methodName = thisJoinPoint.getSignature().getName();
    Object[] methodArgs = thisJoinPoint.getArgs();
    logger.info("Call method " + methodName + " with args " +
                methodArgs);
    Object result = thisJoinPoint.proceed();
    logger.info("Method " + methodName + " returns " + result);
    return result;
}
```

ex.2

AOP use cases

- ◆ Logging-related
- ◆ Security checks
- ◆ Transaction management
- ◆ Exception handling
- ◆ User-rights check
- ◆ Profiling

AOP :: Aspects grouping

@Aspect

public class SystemArchitecture {

@Pointcut("within(com.xyz.someapp.web..*)")
public void inWebLayer() {}

@Pointcut("within(com.xyz.someapp.service..*)")
public void inServiceLayer() {}

@Pointcut("within(com.xyz.someapp.dao..*)")
public void inDataAccessLayer() {}

@Pointcut("execution(* com.xyz.someapp.dao.*.*(..))")
public void dataAccessOperation() {}

}

AOP :: Pointcut combining

Combining pointcut expressions:

```
@Pointcut("execution(public * *(..))")  
private void anyPublicOperation() {}
```

```
@Pointcut("within(com.xyz.someapp.trading..*)")  
private void inTrading() {}
```

```
@Pointcut("anyPublicOperation() && inTrading()")  
private void tradingOperation() {}
```

AOP :: Advice types

- ♦ Can decide, should we execute joinpoint or return its own result:

```
@Around("com.luxoft.example.SystemArchitecture.businessService()")
public Object accessRightsCheck(ProceedingJoinPoint pjp) throws Throwable
{

    if (currentUser.hasRights()) {
        return pjp.proceed();
    } else {
        throw new AuthorizationException();
    }

    return null;

}
```

AOP :: Use of @AfterThrowing

@Aspect

```
public class AfterThrowingExample {
```

```
    @AfterThrowing(
```

```
pointcut="com.luxoft.example.SystemArchitecture.dataAccessOperation()"
```

```
,
```

```
    throwing="ex")
```

```
    public void doRecoveryActions(DataAccessException ex) {
```

```
        // ...
```

```
    }
```

```
}
```

- ♦ There is no way to return to the calling method or continue processing on the subsequent line
- ♦ If you handle the exception here, it won't prevent it to bubble up the chain

AOP :: Advice types summary

@Before – executed before joinpoint

There's no possibility not to execute joinpoint, except for throw an exception

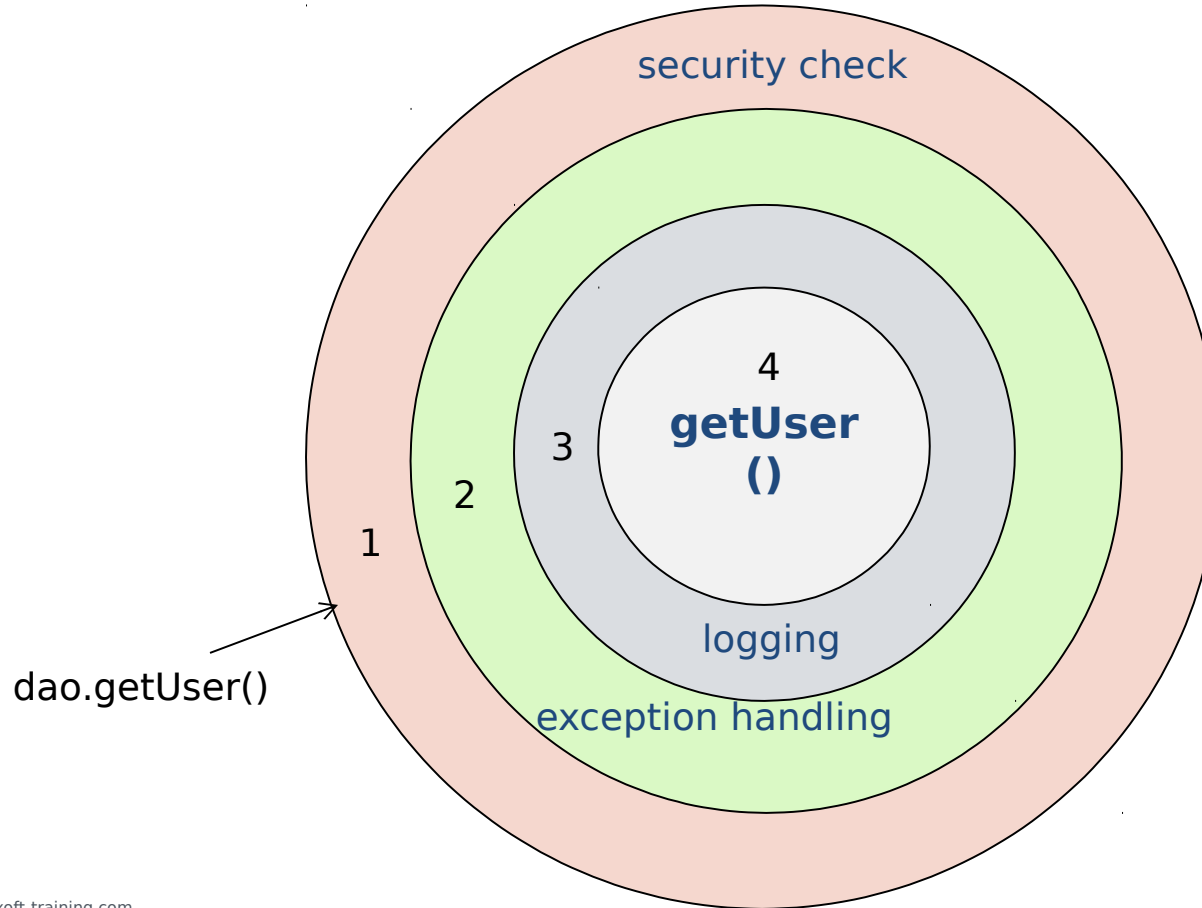
@Around – executed before and after joinpoint

@AfterReturning – after success joinpoint execution – method is finished without exception

@AfterThrowing – in case of exception in joinpoint

@After – executed after the joinpoint

AOP :: Aspects chaining



Matryoshka doll

AOP :: @Order

The order of aspects execution can be defined with use of @Order annotation:

```
@Aspect
@Order(1)
public class AspectA
{
    @Before(".....")
    public void doIt() {}
}
```

```
@Aspect
@Order(2)
public class AspectB
{
    @Before(".....")
    public void doIt() {}
}
```

The order of advices in the aspect are defined by its order in the aspect source code.

ex.4

Exercise

Lab guide:

- Exercise 5

Exercise

Lab guide:

- Exercise 6