Spring4
Spring AOP

Aspect Oriented Programming (AOP)

Aspect-Oriented Programming (AOP) complements Object-Oriented Programming (OOP) by providing another way of thinking about program structure.

The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the *aspect*.

Aspects enable the modularization of concerns such as transaction management that cut across multiple types and objects. (Such concerns are often termed *crosscutting* concerns in AOP literature.)

One of the key components of Spring is the *AOP framework*. While the Spring IoC container does not depend on AOP, meaning you do not need to use AOP if you don't want to, AOP complements Spring IoC to provide a very capable middleware solution.

AOP concepts

Aspect: Aspect – a standard code/feature that is scattered across multiple places in the application and is typically different than the actual Business Logic (for example, Transaction management). Each aspect focuses on a specific cross-cutting functionality

In Spring AOP, aspects are implemented using regular classes annotated with the @Aspect annotation (the @AspectJ style).

Join point: 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.**

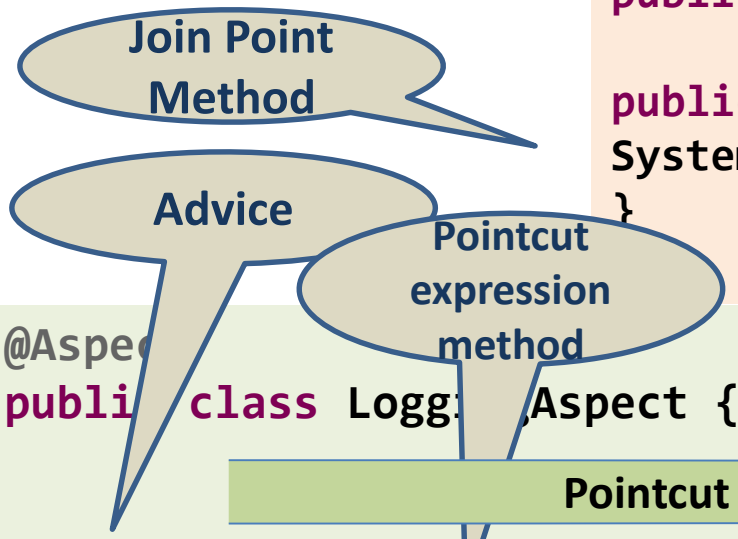
***Advice*:** action taken by an aspect at a particular join point. Different types of advice include "around," "before" and "after" advice. (Advice types are discussed in the following slides.) Many AOP frameworks, including Spring, model an advice as an *interceptor*, maintaining a chain of interceptors *around* the join point.

Pointcut : a regular expression that matches a joinpoint. Each time any join point matches a pointcut, a specified advice associated with that pointcut is executed

AOP concepts

The concept of join points as matched by pointcut expressions is central to AOP, and Spring uses the AspectJ pointcut expression language by default.

Pointcut: a predicate that matches **join points**. **Ex. addCustomer() is a join point method**. **Advice** is associated with a **pointcut expression** and runs at any join point matched by the pointcut .



Example

```
public class Customer implements ICustomer{

public void addCustomer(){
System.out.println("addCustomer() is running ");
}
```

```
@Aspec
publi  class Logg   Aspect {

@Before("execution(* com.varaunited.trg.service.ICustomer.addCustomer(..))")
public void logBeforeAdvice(JoinPoint joinPoint) {
System.out.println("logBefore() is running!");
System.out.println("hijacked : " + joinPoint.getSignature().getName());
}
}
```

Pointcut Expression

Note: execution(* com.varaunited.trg.service.ICustomer.addCustomer(..)) is Pointcut expression and logBefore() method is called as pointcut expression method or before advice method.

Understanding Pointcut Expressions

Pointcut is an expression language of Spring AOP.
The @Pointcut annotation is used to define the pointcut.

Examples

```
@Pointcut("execution(* ClassName.*(..))")
private void doSomething(){

}
```

The name of the pointcut expression is doSomething(). It will be applied on all the methods of class regardless of return type.

@Pointcut("execution(public ClassName.*(..))") : It will be applied on all the public methods of the class.

@Pointcut("execution(* ClassName.*(..))") : It will be applied on all the methods of the class.

@Pointcut("execution(public ClassName.set*(..))") : It will be applied on all the public setter methods of the class.

@Pointcut("execution(int ClassName.*(..))") : It will be applied on all the methods of the class that returns int value.

Spring AOP Pointcut Methods and Reuse

Sometimes we have to use same Pointcut expression at multiple places, we can create an empty method with @Pointcut annotation and then use it as expression in advices.

Example

```
@Aspect
public class LoggingAspect {

@Pointcut("execution(* com.varaunited.trg.service.ICustomer.addCustomer(..))")
public void addCustomerPointcut(){}

@Before("addCustomerPointcut()")
public void logBeforeAdvice(JoinPoint joinPoint) {
System.out.println("logBefore() is running!");
System.out.println("hijacked : " + joinPoint.getSignature().getName());
}

@Before("addCustomerPointcut()")
public void anotherLogBeforeAdvice(JoinPoint joinPoint) {
System.out.println("Another logBefore() is running!");
System.out.println("hijacked : " + joinPoint.getSignature().getName());
}
}
```

AOP concepts

Target Object: They are the objects on which advices are applied. Spring AOP is implemented using **runtime proxies** so this object is always a proxied object. What it means is that a subclass is created at runtime where the target method is overridden and advices are included based on their configuration.

AOP proxy: Spring AOP implementation uses JDK dynamic proxy to create the Proxy classes with target classes and advice invocations, these are called AOP proxy classes. We can also use CGLIB proxy by adding it as the dependency in the Spring AOP project.

AOP Weaving: Spring AOP makes it possible to modularize and separate logging, transaction like services and apply them declaratively to the components so that developers can focus on specific concerns.

Aspects are wired into objects in the spring XML file in the way as JavaBean. This process is known as 'Weaving'.
So it the process of **linking aspects** with targeted objects to create an advised object

Weaving

Weaving

Both AspectJ and Spring AOP uses the different type of weaving which affects their behavior regarding performance and ease of use.

AspectJ makes use of three different types of weaving:

1. **Compile-time weaving**: The AspectJ compiler takes as input both the source code of our aspect and our application and produces a woven class files as output
2. **Post-compile weaving**: This is also known as binary weaving. It is used to weave existing class files and JAR files with our aspects
3. **Load-time weaving**: This is exactly like the former binary weaving, with a difference that weaving is postponed until a class loader loads the class files to the JVM

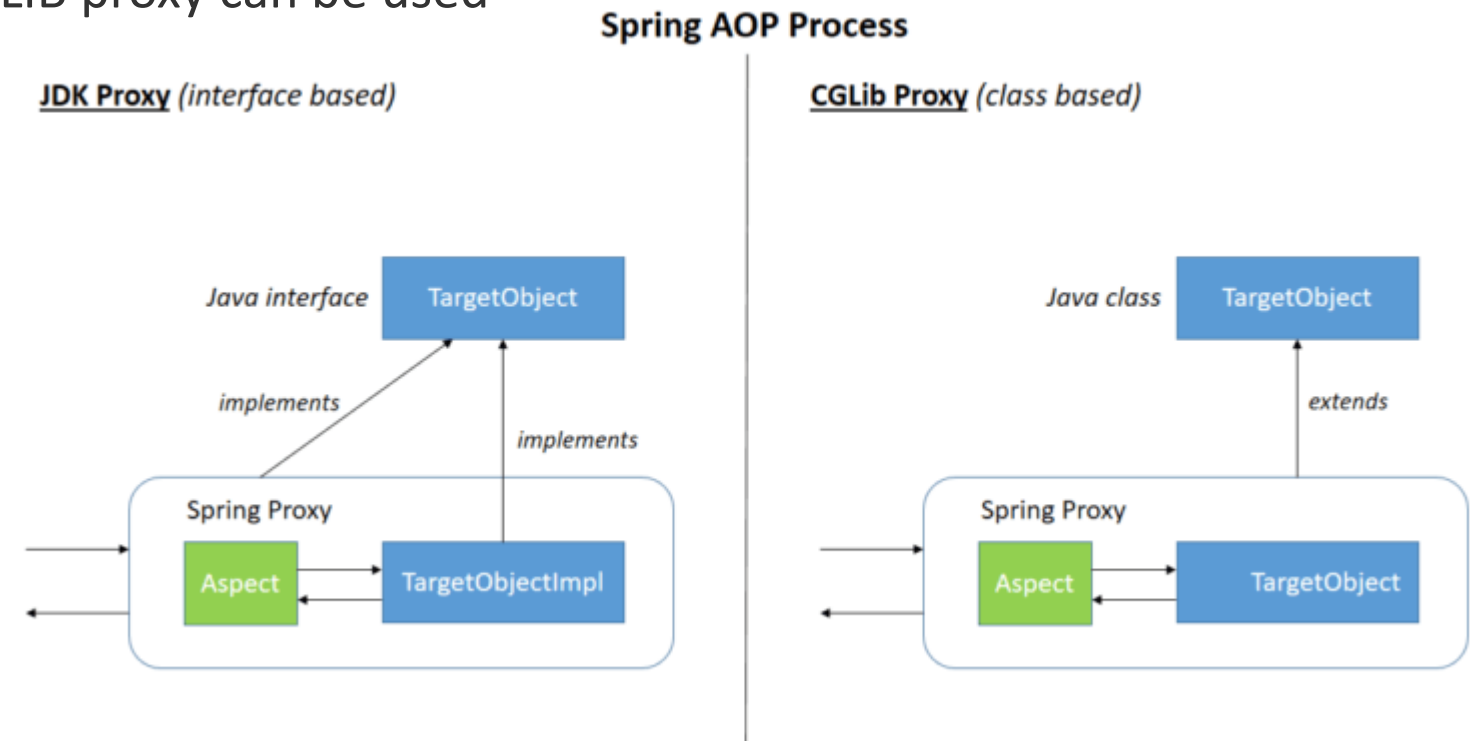
NOTE: Spring AOP makes use of runtime weaving.

With runtime weaving, the aspects are woven during the execution of the application using proxies of the targeted object – using either JDK dynamic proxy or CGLIB proxy.

Weaving

Spring AOP is a proxy-based AOP framework. This means that to implement aspects to the target objects, it'll create proxies of that object. This is achieved using either of two ways:

1. JDK dynamic proxy – the preferred way for Spring AOP. Whenever the targeted object implements even one interface, then JDK dynamic proxy will be used
2. CGLIB proxy – if the target object doesn't implement an interface, then CGLIB proxy can be used



Spring AOP Vs AspectJ

AspectJ, on the other hand, doesn't do anything at runtime as the classes are compiled directly with aspects.

And so unlike Spring AOP, it doesn't require any design patterns. To weave the aspects to the code, it introduces its compiler known as AspectJ compiler (ajc), through which we compile our program and then runs it by supplying a small (< 100K) runtime library.

Spring AOP	AspectJ
Implemented in pure Java	Implemented using extensions of Java programming language
No need for separate compilation process	Needs AspectJ compiler (ajc) unless LTW is set up
Only runtime weaving is available	Runtime weaving is not available. Supports compile-time, post-compile, and load-time Weaving
Less Powerful – only supports method level weaving	More Powerful – can weave fields, methods, constructors, static initializers, final class/methods, etc…
Can only be implemented on beans managed by Spring container	Can be implemented on all domain objects
Supports only method execution pointcuts	Support all pointcuts
Proxies are created of targeted objects, and aspects are applied on these proxies	Aspects are weaved directly into code before application is executed (before runtime)
Much slower than AspectJ	Better Performance
Easy to learn and apply	Comparatively more complicated than Spring AOP

AOP Advice Types

Based on the execution strategy of advices, they are of following types.

1.Before Advice: These advices runs before the execution of join point methods. Use **@Before** annotation to mark an advice type as Before advice.

Example

```
@Aspect
public class LoggingAspect {


@Before("execution(* com.varaunited.trg.service.ICustomer.addCustomer(..))")
public void logBefore(JoinPoint joinPoint) {

System.out.println("logBefore() is running!");
System.out.println("hijacked : " + joinPoint.getSignature().getName());
System.out.println("******");
}
}
```

AOP Advice Types

2. After (finally) Advice: An advice that gets executed after the join point method finishes executing, whether normally or by throwing an exception. We can create after advice using **@After** annotation.

3. After Returning Advice: Sometimes we want advice methods to execute only if the join point method executes normally.
Use **@AfterReturning** annotation to mark a method as after returning advice.

4. After Throwing Advice: This advice gets executed only when join point method throws exception, we can use it to rollback the transaction declaratively.
Use **@AfterThrowing** annotation for this type of advice.

AOP Advice Types

5. Around Advice: This is powerful advice that surrounds the join point method and we can also choose whether to execute the join point method or not.

We can write advice code that gets executed before and after the execution of the join point method. It is the responsibility of around advice to invoke the join point method and return values if the method is returning something.
Use **@Around** annotation to create around advice methods.

Example

```
@Around("execution(*
com.varaunited.trg.service.ICustomer.addCustomerAround(..))")
public void logAround(ProceedingJoinPoint joinPoint) throws Throwable {
System.out.println("logAround() is running!");
System.out.println("hijacked method : " + joinPoint.getSignature().getName());
System.out.println("hijacked arguments : " + Arrays.toString(joinPoint.getArgs()));
System.out.println("Around before is running!");
joinPoint.proceed(); //continue on the intercepted method
System.out.println("Around after is running!");
}
```

XML declaration

- Include the following namespaces

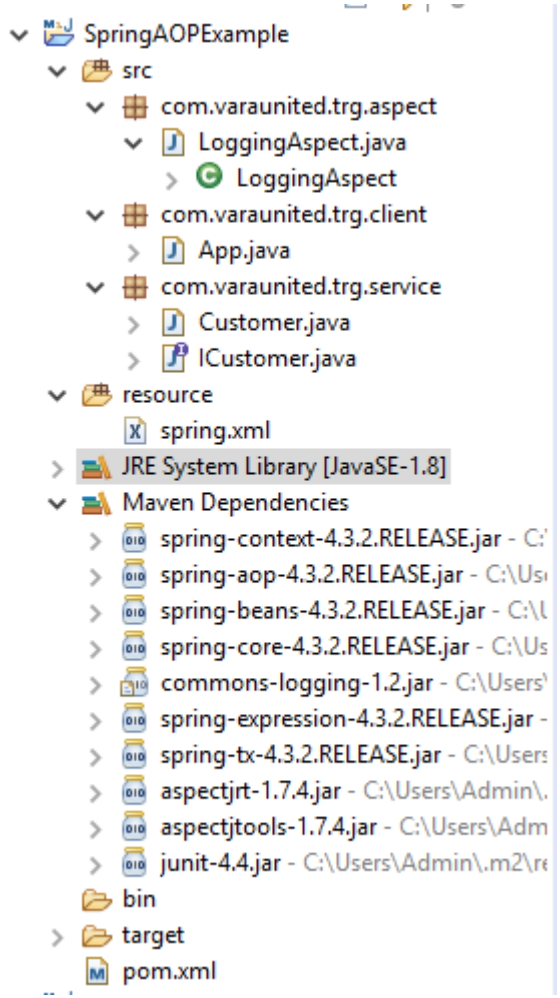
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd">

…..
</beans>
```

- Add **aop:aspectj-autoproxy** element to enable Spring AspectJ support with auto proxy at runtime

<aop:aspectj-autoproxy />

Spring project with AOP implementations

Spring provides support for using AspectJ annotations to create aspects.
All the above AOP annotations are defined in **org.aspectj.lang.annotation** package.




```
<dependencies>
<!-- Spring and Transactions -->
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-context</artifactId>
<version>4.3.2.RELEASE</version>
</dependency>
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-tx</artifactId>
<version>4.3.2.RELEASE</version>
</dependency>
<!-- AspectJ dependencies -->
<dependency>
<groupId>org.aspectj</groupId>
<artifactId>aspectjrt</artifactId>
<version>1.7.4</version>
<scope>runtime</scope>
</dependency>
<dependency>
<groupId>org.aspectj</groupId>
<artifactId>aspectjtools</artifactId>
<version>1.7.4</version>
</dependency>
</dependencies>
```

Spring project with AOP implementations

```
public interface ICustomer {
public void addCustomer();
public String addCustomerReturnValue();
public void addCustomerThrowException() throws Exception;
public void addCustomerAround(String name);}
```

```
public class Customer implements ICustomer{
public void addCustomer(){
System.out.println("addCustomer() is running ");
}

public String addCustomerReturnValue(){
System.out.println("addCustomerReturnValue() is running ");
return "Hello World!!";
}

public void addCustomerThrowException() throws Exception {
System.out.println("addCustomerThrowException() is running ");
throw new Exception("Generic Error");
}
public void addCustomerAround(String name){
System.out.println("addCustomerAround() is running, args : " + name);
}
}
```

Spring project with AOP implementations

Aspect: An aspect is a class that implements enterprise application concerns that cut across multiple classes, such as transaction management. We can use Spring AspectJ integration to define a class as Aspect using @Aspect annotation.

LoggingAspect.java

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
…….

@Aspect
public class LoggingAspect {

@Before("execution(* com.varaunited.trg.service.ICustomer.addCustomer(..))")
public void logBefore(JoinPoint joinPoint) {

    System.out.println("logBefore() is running!");
    System.out.println("hijacked : " +
    joinPoint.getSignature().getName());
    System.out.println("******");
}
```

Spring project with AOP implementations

```
@After("execution(* com.varaunited.trg.service.ICustomer.addCustomer(..))")
public void logAfter(JoinPoint joinPoint) {

    System.out.println("logAfter() is running!");
    System.out.println("hijacked : " + joinPoint.getSignature().getName());
    System.out.println("******");
}

@AfterReturning(
      pointcut = "execution(*
com.varaunited.trg.service.ICustomer.addCustomerReturnValue(..))",
      returning= "result")
public void logAfterReturning(JoinPoint joinPoint, Object result) {

    System.out.println("logAfterReturning() is running!");
    System.out.println("hijacked : " + joinPoint.getSignature().getName());
    System.out.println("Method returned value is : " + result);
    System.out.println("******");
}
```

Spring project with AOP implementations

```
@AfterThrowing(
      pointcut = "execution(*
com.varaunited.trg.service.ICustomer.addCustomerThrowException(..))",
      throwing= "error")
public void logAfterThrowing(JoinPoint joinPoint, Throwable error) {

System.out.println("logAfterThrowing() is running!");
System.out.println("hijacked : " + joinPoint.getSignature().getName());
System.out.println("Exception : " + error);
System.out.println("******");
}

@Around("execution(* com.varaunited.trg.service.ICustomer.addCustomerAround(..))")
public void logAround(ProceedingJoinPoint joinPoint) throws Throwable {
System.out.println("logAround() is running!");
System.out.println("hijacked method : " + joinPoint.getSignature().getName());
System.out.println("hijacked arguments : " +
Arrays.toString(joinPoint.getArgs()));
System.out.println("Around before is running!");
joinPoint.proceed(); //continue on the intercepted method
System.out.println("Around after is running!");
System.out.println("******");
  }}
```

Spring project with AOP implementations

spring.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 ">

<!-- Enable AspectJ style of Spring AOP -->
<aop:aspectj-autoproxy />

<bean id="customerBean" class="com.varaunited.trg.service.Customer" />

<!-- Aspect -->
<bean id="logAspect" class="com.varaunited.trg.aspect.LoggingAspect" />

</beans>
```

Spring project with AOP implementations

App.java

```
public class App {

public static void main(String[] args) {
try{
    ClassPathXmlApplicationContext context = new
    ClassPathXmlApplicationContext("spring.xml");

    ICustomer customer= (ICustomer) context.getBean("customerBean");
    customer.addCustomer();
    customer.addCustomerReturnValue();
    customer.addCustomerThrowException();
    customer.addCustomerAround("Srinivas");
    context.close();

} catch(Exception e){
//e.printStackTrace();
}
}
}
```

Spring Web Application applying AOP annotation

```
@Service
@Transactional
public class UserService implements
IUserService{

@Autowired
private IUserDAO userDAO;
…
}
```

```
@Repository
public class UserDAO implements IUserDAO{
@Autowired
private JdbcTemplate jdbcTemplate;

@Override
public User userRegistration(User user) throws
UserException {
String sql="insert into user_vara values(?,?,?,?)";
int n=jdbcTemplate.update(sql,new Object[]
{user.getUserId(),user.getUserName(),user.getEmail(),
user.getPassword()});
if(n>0) {
return user;
}else {
return null;
}}}
```

@Transactional does transaction management magic

Look at the logs carefully and notice the AOP and Proxy classes created by Spring framework.
Spring framework is using Around advice to generate a proxy class for UserDAO and only committing the transaction if the method returns successfully. If there is any exception, it is rolling back the whole transaction.

Thank You!