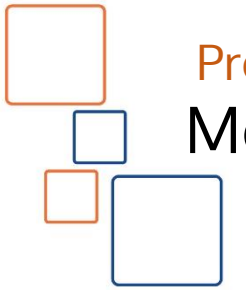
Spring Framework

THE RIGHT TECHNOLOGY STACK FOR THE JOB AT HAND



Presented By
Mohsen Diab

Java™ Education
and Technology Services

Invest In Yourself ,
Develop Your Career

Course Outline

- Lesson 1: Introduction
- Lesson 2: The Concepts of AOP
- Lesson 3: AOP family
- Lesson 4: Classic Spring proxy-based AOP
- Lesson 5: @AspectJ annotation-driven aspects
- Lesson 6: Introduction to "Aspect Introduction"
- Lesson 7: More Details about Pointcut
- Lesson 8: Declaring Aspects with XML-Based Configurations
- *** **References & Recommended Reading**

Lesson 1

Introduction

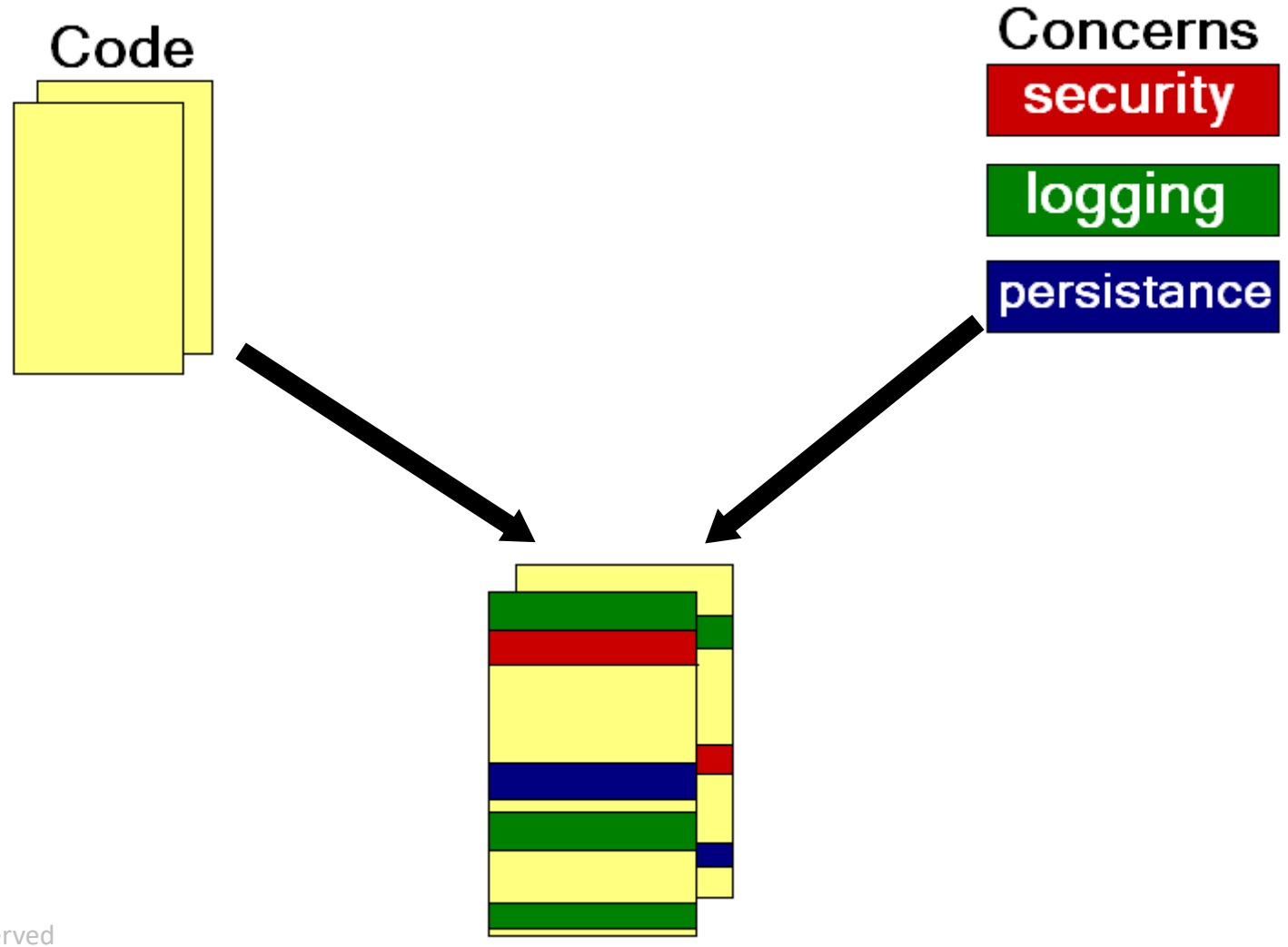
OOP Concept

- The OOP concepts allow you to write programs that feature:
  - **Modularity:**
    - The concept of class.
  - **Reusability:**
    - Class can be reusable in different places.
  - **Reliability:**
    - The data is encapsulated within objects, it can be manipulated only through the methods that define the object's interface. (Directly manipulating the data is not possible.)
  - **Extendibility:**
    - The concept of inheritance.

The limitation of OOP

- We will show that writing clear programs using only OOP is impossible in at least two cases:
  - Crosscutting functionalities,
  - Code scattering.

1. Crosscutting functionalities



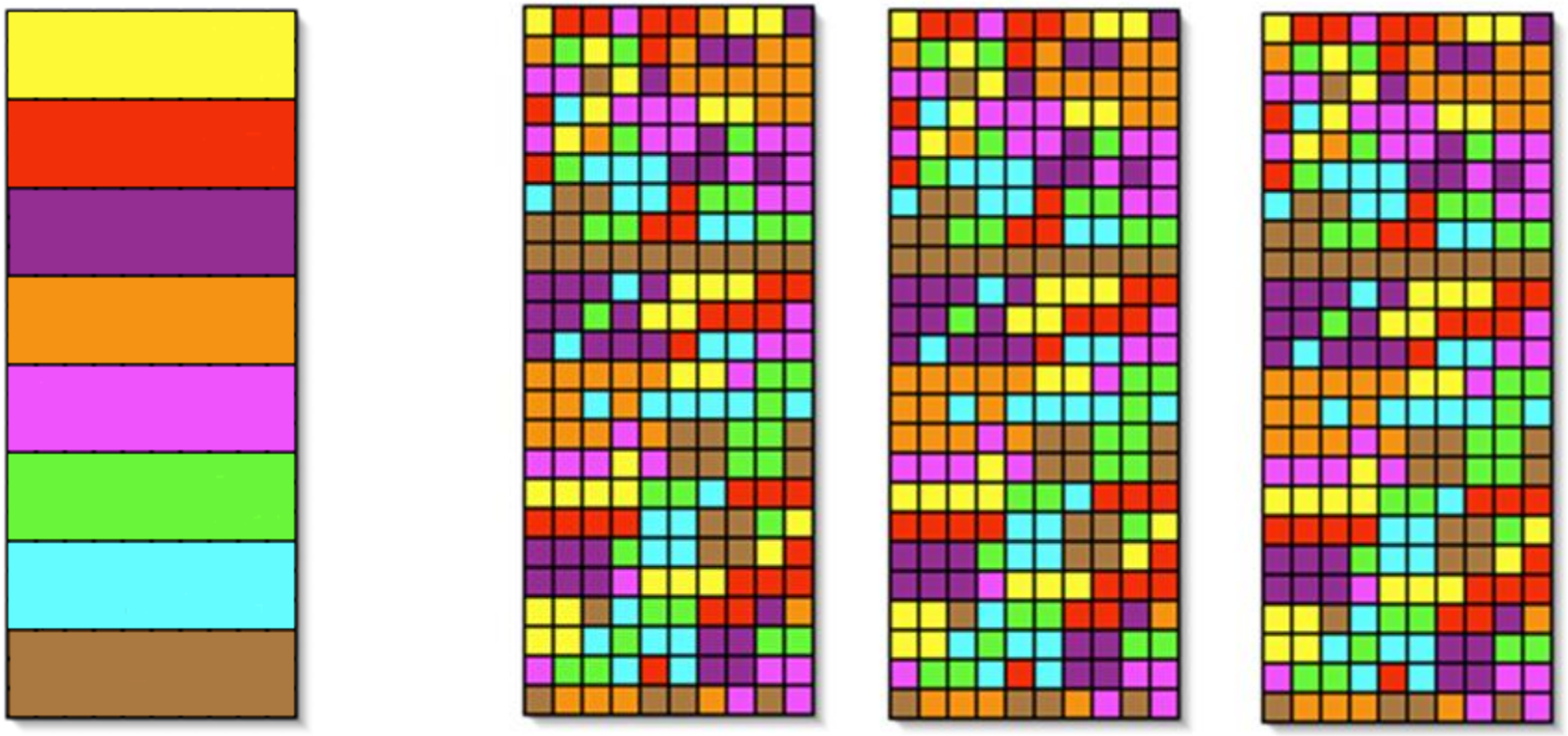
1. Crosscutting functionalities (Ex.)

- Although the classes are programmed independently of one another, they are sometimes behaviorally interdependent.
- For example:
  - A Customer object must not be deleted while an outstanding order remains unpaid.
  - To Enforce this rule, you could modify the customer-deletion method to determines whether all the orders have been paid.

1. Crosscutting functionalities (Ex.)

- This solution is deficient for several reasons:
  - Determining whether an order has been paid doesn't belong to customer management but to order management.
  - The Customer class should not need to be aware of all the data-integrity rules.
  - Once the customer class implements any functionality that is linked to a different class, customer is no longer independently reusable, in many cases.
- The customer class is not the ideal place to implement this rule.
- You might be thinking about integrating this functionality into an order class instead, but this solution is no better.
- No reason exists for the order class to allow the deletion of a customer.
- This rule is linked to neither the customers nor the orders but cuts across these two types of entities.

2. Code Scattering



2. Code Scattering (Ex.)

- In OOP, objects interact is by invoking methods.
- When you call a method, you don’t care about the service implementation. You must ensure the parameters correspond to the method’s signature.
- If you alter just the body of the method, the calling of the method will not be change.
- If you change the method’s signature, you must then modify all the calls to the method in all classes that invoke the method.

2. Code Scattering (Ex.)

- The main point is this:
  - Even though the implementation of a method is located in a single class, the calls to that method can be scattered throughout the application.
  - This phenomenon of code scattering slows down maintenance tasks and makes it difficult for object-oriented applications to adapt and evolve.

Lesson 2

The Concepts of AOP

Aspect Oriented Programming

- 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.)
- 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 in Spring Framework

- AOP is used in the Spring Framework to:
    1. Provide declarative enterprise services.
        - The most important such service is declarative transaction management.
    2. Let users implement custom aspects.
        - Complementing their use of OOP with AOP.

Aspect Oriented Programming (Ex.)

- Every new programming paradigm brings with it a set of concepts and definitions.
- This was the case for the OO approach, with the concepts of:
  - encapsulation, inheritance, and polymorphism.
- AOP concepts are not specific to any language, in the same way that the concept of OOP.
- The modularization in OOP is based on the data that are encapsulated in the classes.
- With AOP, the modularization can occur in two dimensions:
  - Base functionalities:
    - Implemented by classes.
  - Crosscutting functionalities:
    - Implemented by aspects (logging, security, …)

Code Segment without AOP

```java
public class CalculatorImpl implements Calculator {

    @Override
    public double add(double firstOperand, double secondOperand) {
        double result = firstOperand + secondOperand;
        System.out.println(firstOperand + "+" + secondOperand + "=" + result);
        return result;
    }

    @Override
    public double sub(double firstOperand, double secondOperand) {
        double result = firstOperand - secondOperand;
        System.out.println(firstOperand + "-" + secondOperand + "=" + result);
        return result;
    }
```

Base functionalities

Crosscutting functionalities

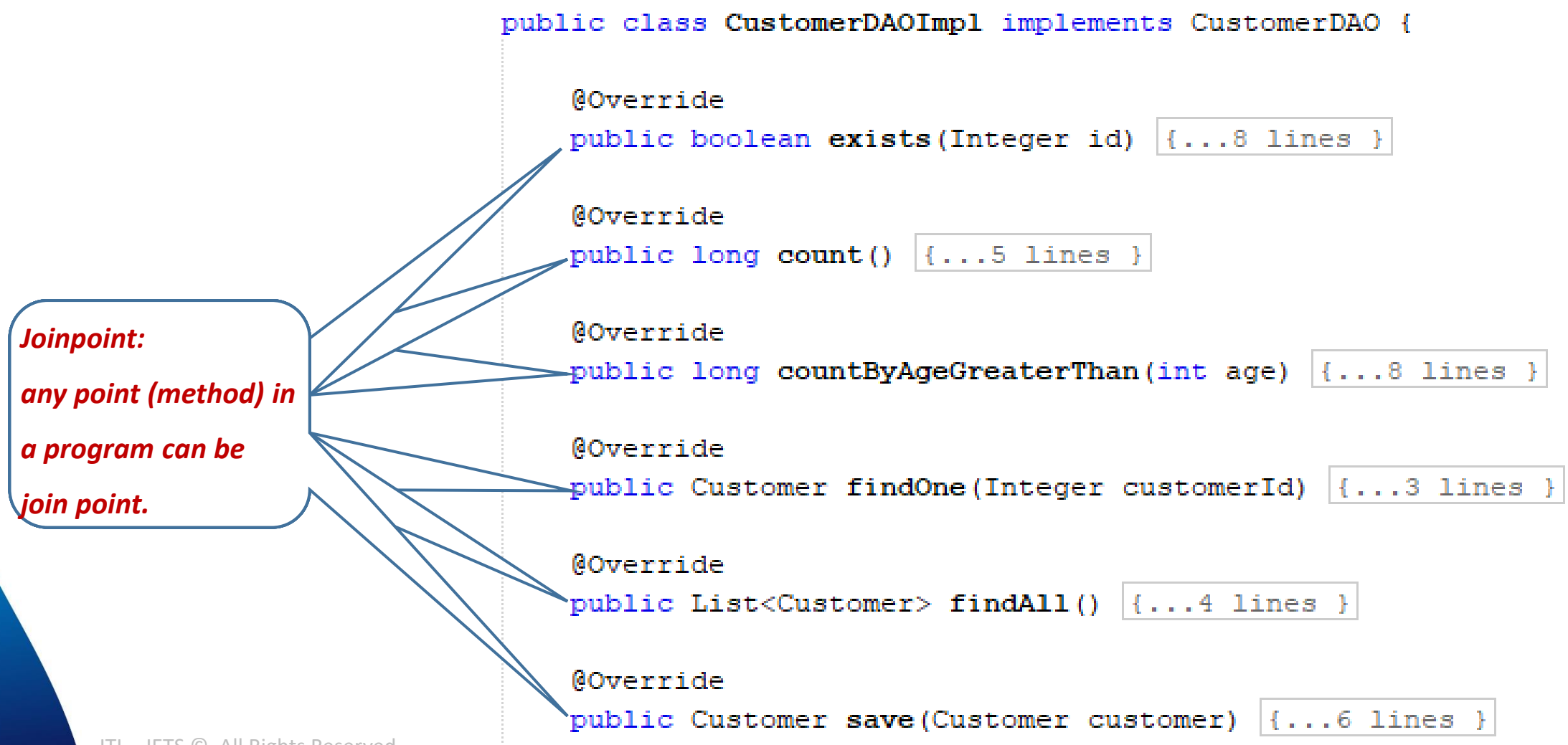
Code Segment without AOP

```
public class CustomerDAOImpl implements CustomerDAO {

    @Override
    public Customer save(Customer customer) {
        entityManager.getTransaction().begin();
        entityManager.persist(customer);
        entityManager.getTransaction().commit();
        return customer;
    }

    @Override
    public void update(Customer customer) {
        entityManager.getTransaction().begin();
        entityManager.merge(customer);
        entityManager.getTransaction().commit();
    }
```

Base functionalities

Crosscutting functionalities

AOP Concepts (Join point)



AOP Concepts (Join point) (Ex.)

Joinpoint:

any point (method) in a program can be join point.

```
public class CalculatorImpl implements Calculator {

    @Override
    public double add(double firstOperand, double secondOperand) {...4 lines }

    @Override
    public double sub(double firstOperand, double secondOperand) {...4 lines }

    @Override
    public double multi(double firstOperand, double secondOperand) {...4 lines }

    @Override
    public double div(double firstOperand, double secondOperand) {...4 lines }
}
```

AOP Concepts (Join point) (Ex.)

- Join point:
- A point during the execution of a program, such as the execution of a method or the handling of an exception.
- A point in the control flow of a program where one or several aspects apply
- Each instruction (method) of a program can be a joinpoint.
- In Spring AOP, a join point always represents a method execution.
- The task of the aspect programmer:
  - Wire between the selected joinpoints and a given aspect.

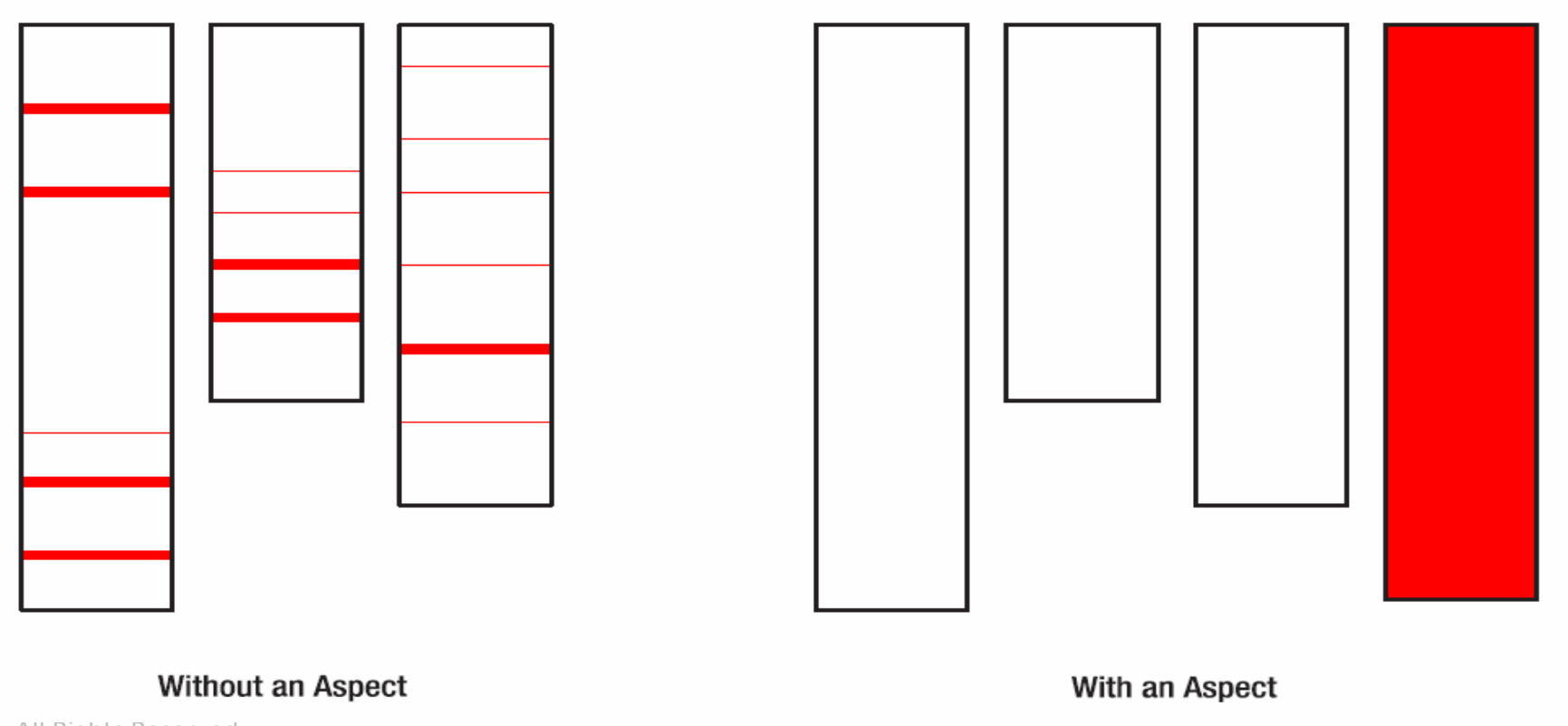
AOP Concepts (Aspect)

- Aspect:
- A modularization of a concern that cuts across multiple classes.
- is a program unit (like class) that capture the crosscutting functionalities.
- Example:
  - Transaction management is a good example of a crosscutting concern in enterprise Java applications.
- In Spring AOP, aspects are implemented by using regular classes
  - Declared and defined through schema-based approach.
  - Declared by annotated with the @Aspect annotation (the @AspectJ style).

AOP Concepts (Aspect) (Ex.)

- Aspect:

- A programming unit designed to capture a functionality that crosscuts an application.



AOP Concepts (Aspect) (Ex.)

- Aspect:
- An aspect is composed of two parts:
    1. Advice code.
    2. Pointcut.
- The Advice code:
    - Contains the code(crosscutting functionalities) to be executed.
- The Pointcut:
    - Defines the points (which joinpoints should be use this advice) in the program where this advice should be implemented.

AOP Concepts (Aspect) (Ex.)

- Pointcut:
- A pointcut defines the "where" of an aspect should be apply.
- A set of joinpoints where an aspect applies.
- Pointcuts are application depended.
- When an aspect needs to be reused for a different application , the definition of the pointcuts will need to be adapted to the locations in the new application.
- A pointcut declaration has two part:
  - name: Method Name (ex. add, log, println, save, findAll, getBuyer)
  - expression: Match with AOP pointcut expression language (AspectJ Pointcut EL) which define.

AOP Concepts (Aspect) (Ex.)

- Advice:
- The definition of the behavior of an aspect.
- The advice code defines "what" the instructions of an aspect are.
- Advice code is associated with a pointcut to implement a crosscutting functionality.
- The advice code is never called directly but is woven into the joinpoints that are specified in the associated pointcut.
- Action taken by an aspect at a particular join point.
- Many AOP frameworks, including Spring, model an advice as an interceptor and maintain a chain of interceptors around the join point.

AOP Concepts (Aspect) (Ex.)

- Advice (Types):
- Spring AOP includes the following types of advice:
- **Before advice:**
  - Runs before a join point.
  - Does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).
- **After returning advice:**
  - Runs after a join point completes normally (ex. if a method returns without throwing an exception).

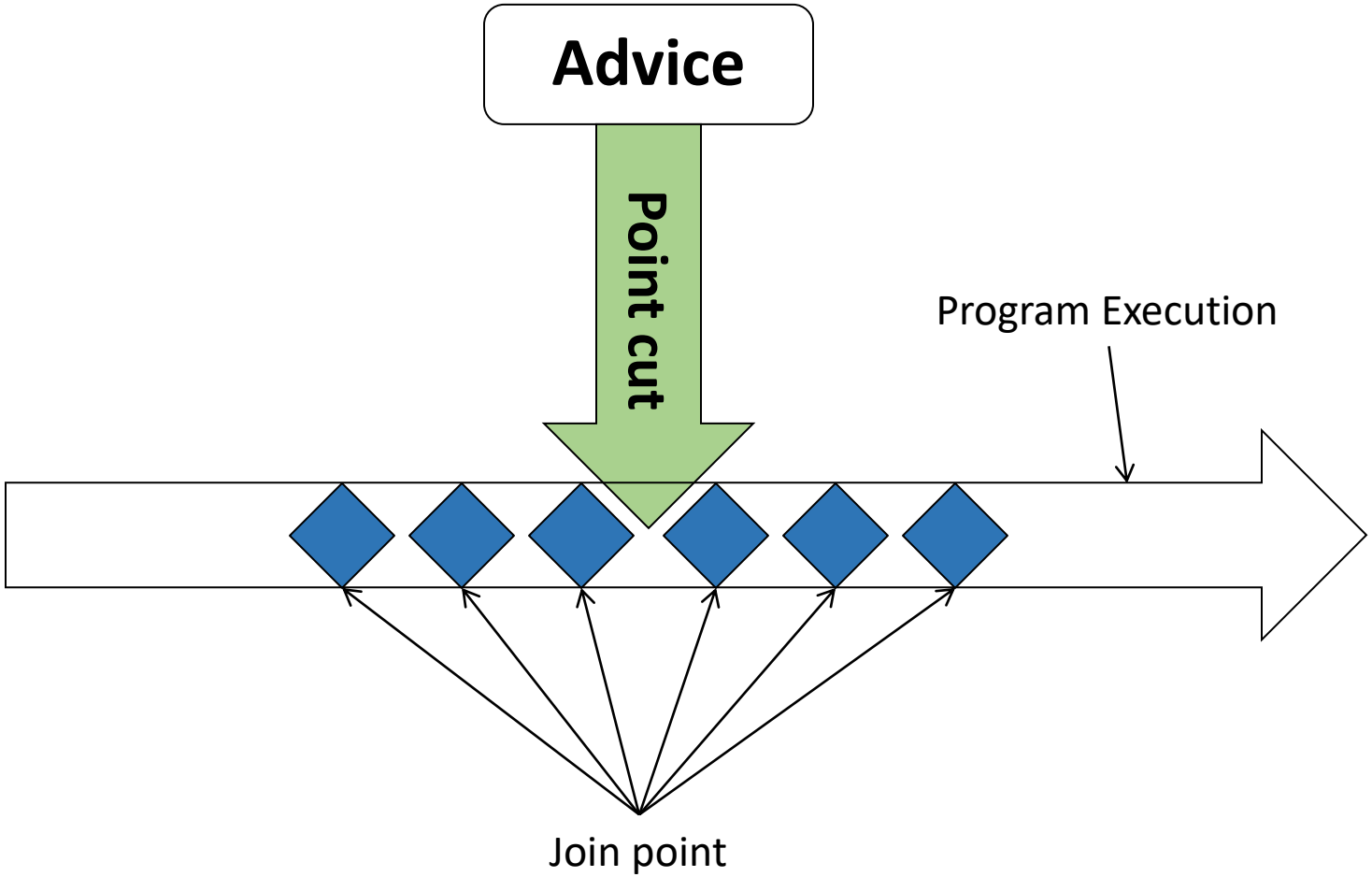
AOP Concepts (Aspect) (Ex.)

- Advice (Types):
- Spring AOP includes the following types of advice:
- **After throwing advice:**
  - Runs after a join point completes unexpectedly (ex. if a method exits by throwing an exception).
- **After (finally) advice:**
  - Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).

AOP Concepts (Aspect) (Ex.)

- Advice (Types):
- Spring AOP includes the following types of advice:
- **Around advice*: (**most powerful kind of advice**)**
  - Advice that surrounds a join point such as a method invocation.
  - Around advice can perform custom behavior before and after the method invocation.
  - It is also responsible for:
    - Choosing whether to proceed to the join point or not.
    - Shortcut the advised method execution by returning its own return value.
    - Choosing to throw an exception from the advised method.

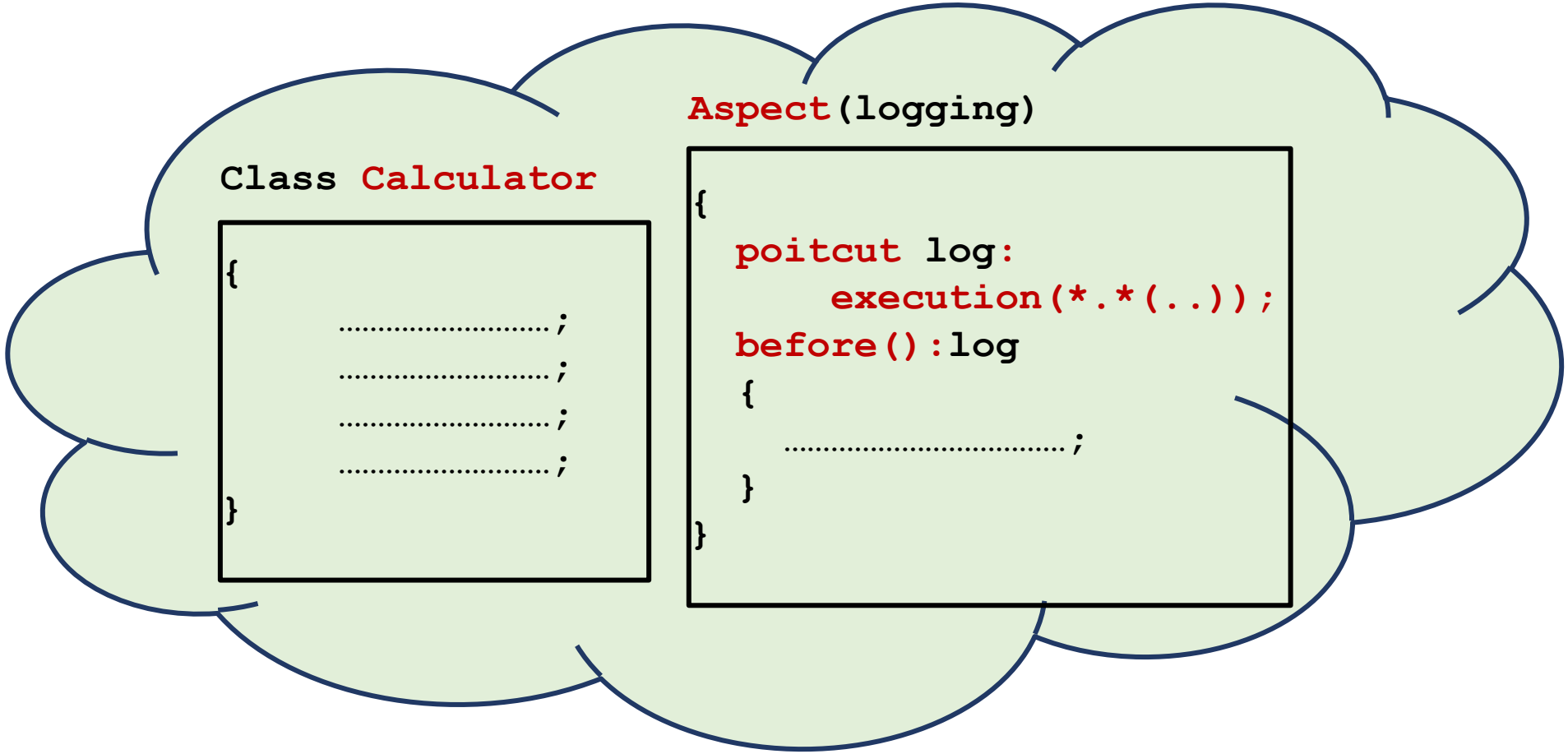
AOP Flow



AOP Concepts (Proxy)

- AOP Proxy:
- An object created by the AOP framework.
- To implement the aspect contracts (advise method executions and so on).
- In the Spring Framework, an AOP proxy is implemented:
  - Either by A JDK dynamic proxy.
  - Or a CGLIB proxy.

AOP Concepts (Weaving)



- The operation that takes the classes & aspects and inject the aspects into classes is known as aspect weaving.

AOP Concepts (Weaving) (Ex.)

- Weaving:
  - Input: The classes and aspects in the application.
  - Output: An advised Object that integrates the functionalities of these classes and the aspects.
- Linking aspects with other application types or objects to create an advised object.
- Aspect weaver:
  - A program that integrates classes and aspects.
- This can be done by AOP compiler (ex. AspectJ Compiler) at:

  * Compile time. * Load time (Runtime but in context loading) *Runtime.

- Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime.

AOP Concepts (Weaving) (Ex.)

- Compile Time Weaving (static weaving):
- With compile-time weaving, aspects are added to the application code.
- When executed, this new code does not make any distinction between the original code and the code that comes from the aspects.
- A compile-time weaver is very similar to a compiler
- Example of Compile time Weaver is AspectJ (could be compile-time/run-time weaver).
- To remove or add an aspect, a total reweaving of the application is needed.

AOP Concepts (Weaving) (Ex.)

- Compile Time Weaving (static weaving):
- The output of a compile-time weaver can be
  - Source code or bytecode.
- The advantage of Compile Time Weaving is
  - It can be easily read by a programmer.
- The disadvantage of Compile Time Weaving is
  - This code must then be compiled into bytecode, which slows down the code production chain.

AOP Concepts (Weaving) (Ex.)

- Run-Time Weaving (dynamic weaving):
- A run-time weaver is a program that executes either the application code or the aspect code, depending on the defined weaving directives.
- By adding or removing a binding, you can weave or unweave a concern while the application is running.
- The advantage of run-time weaving is
  - The distinction between application objects and aspects is clearly established.

Summary

- These concepts that were presented are independent of any implementation by a specific language or framework.
- These concepts are implemented in JDK Proxy(manually), AspectJ, JAC, JBoss AOP, and Spring.
- These concept of an aspect aims to modularize a crosscutting functionality.
- AOP is a technique that complements OOP.
- Implementing an Aspect consists of defining Advice code and Pointcuts.
- The point in the program execution where an aspect applies is called a joinpoint (Every Method).
- The advice code defines what the behavior of the crosscutting functionality, and pointcuts define where this behavior is to be applied in the application.

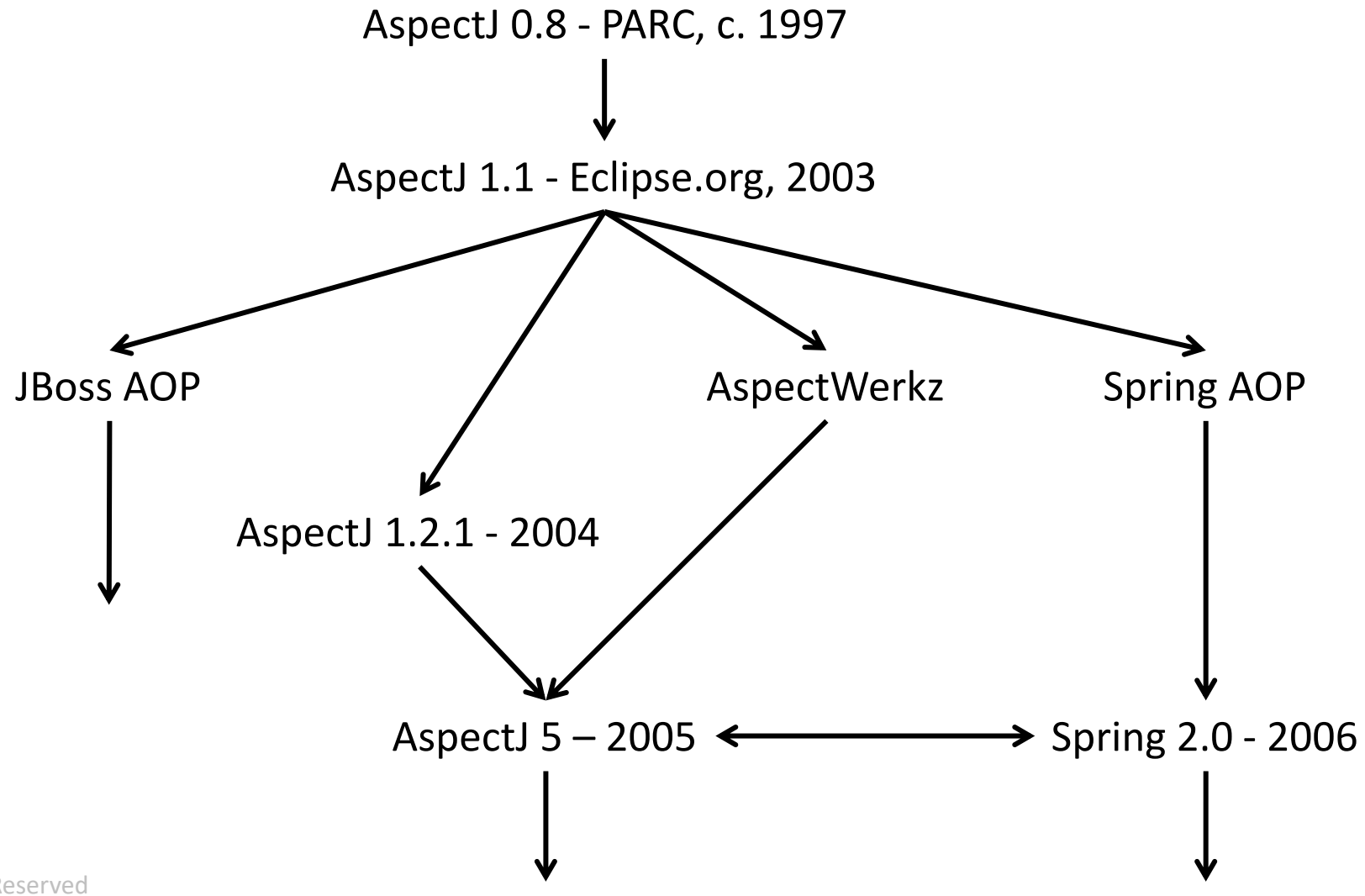
Lesson 3

AOP family

AOP family

- There are many AOP frameworks implemented for different purposes.
- The following is the most famous three open source AOP frameworks:
  - AspectJ (http://www.eclipse.org/aspectj/)
  - JBoss AOP (http://jbossaop.jboss.org/)
  - Spring AOP (http://www.springframework.org/)
- AspectJ is our hero in the Java community.
- AspectJ is the most complete AOP framework in the Java community.

AOP family (Ex.)

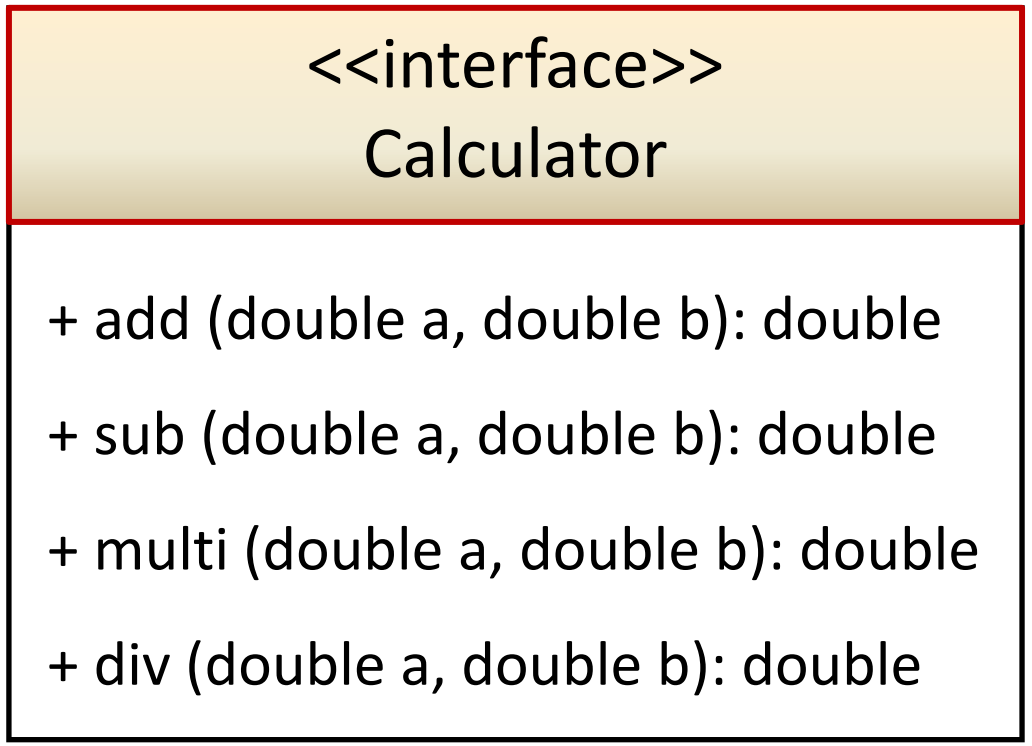


Lesson 4

Classic Spring proxy-based AOP

Java™ Education and Technology Services

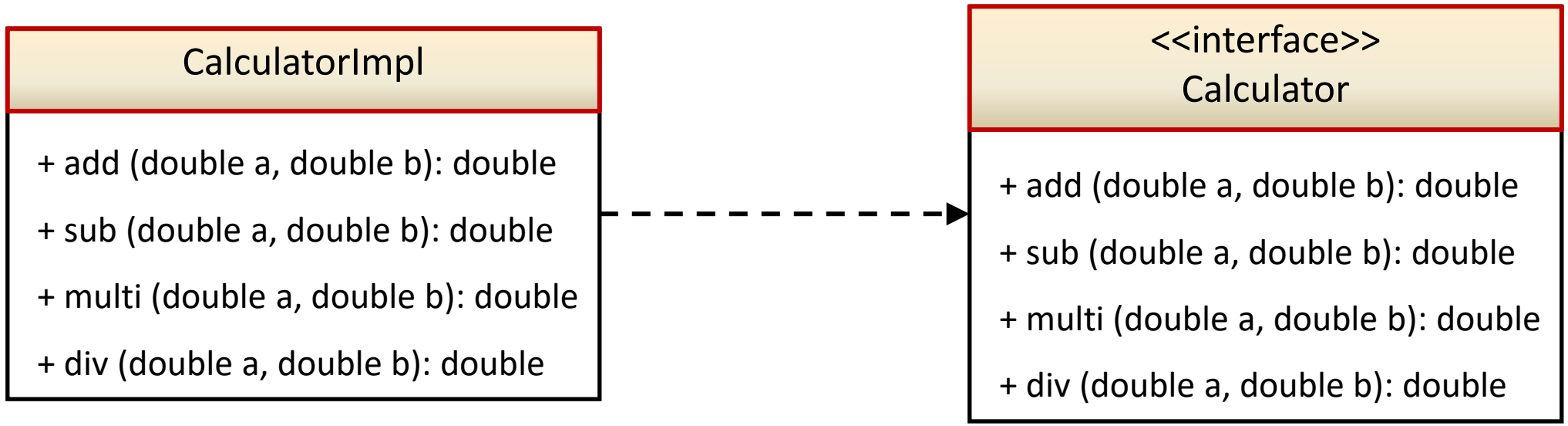
Calculator Case Service Interface (UML)



Calculator Case Service Interface

```
public interface Calculator {

    public double add(double firstOperand, double secondOperand);

    public double sub(double firstOperand, double secondOperand);

    public double multi(double firstOperand, double secondOperand);

    public double div(double firstOperand, double secondOperand);

}
```

Calculator Case Service Class (UML)



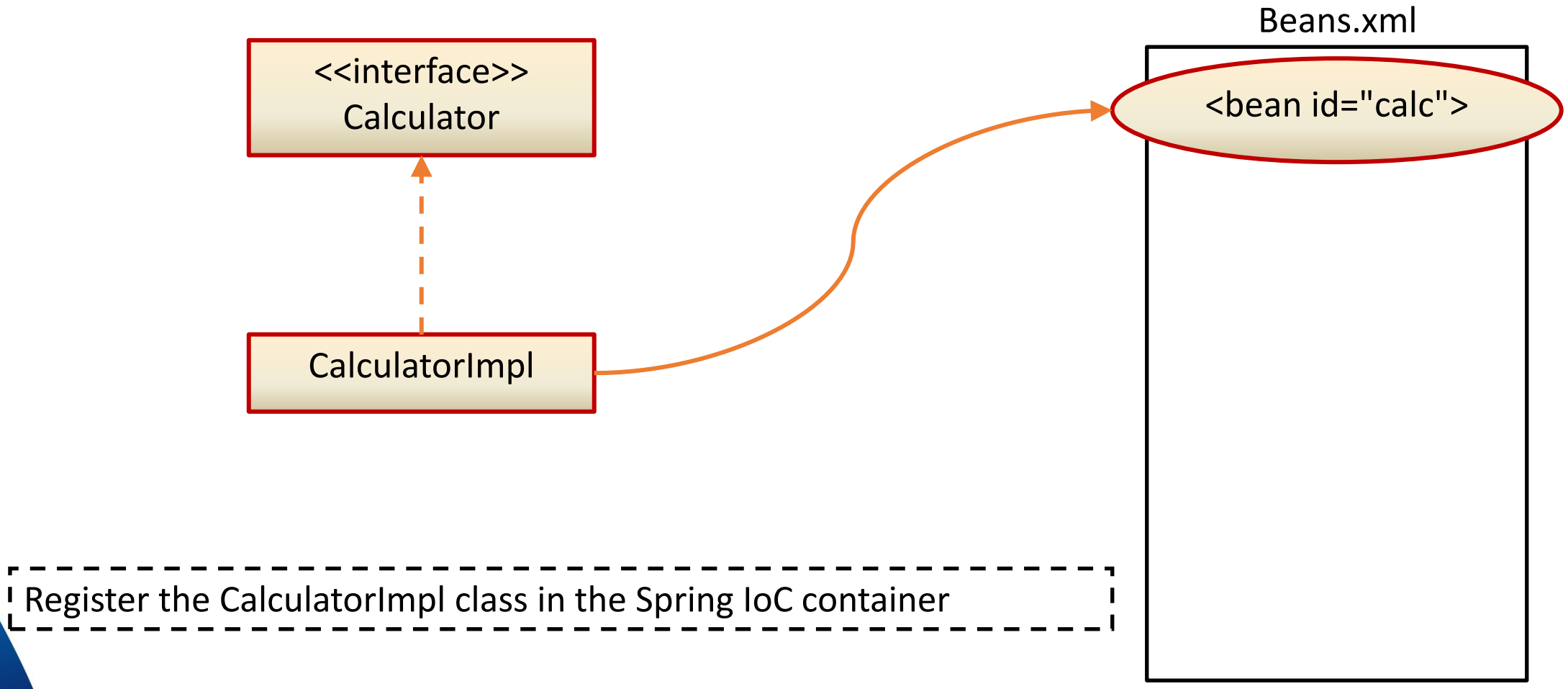
Calculator Case Service Class

```
public class CalculatorImpl implements Calculator {

    @Override
    public double add(double firstOperand, double secondOperand) {
        double result = firstOperand + secondOperand;
        System.out.println(firstOperand + "+" + secondOperand + "=" + result);
        return result;
    }

    @Override
    public double sub(double firstOperand, double secondOperand) {...5 lines }

    @Override
    public double multi(double firstOperand, double secondOperand) {...4 lines }

    @Override
    public double div(double firstOperand, double secondOperand) {...4 lines }
}
```

Calculator Case

- The aim of Spring AOP is
  - To handle crosscutting concerns for the beans declared in its IoC container.
- So, before using Spring AOP:
  - You have to migrate your calculator application to run within the Spring IoC container.

Calculator Case Workflow

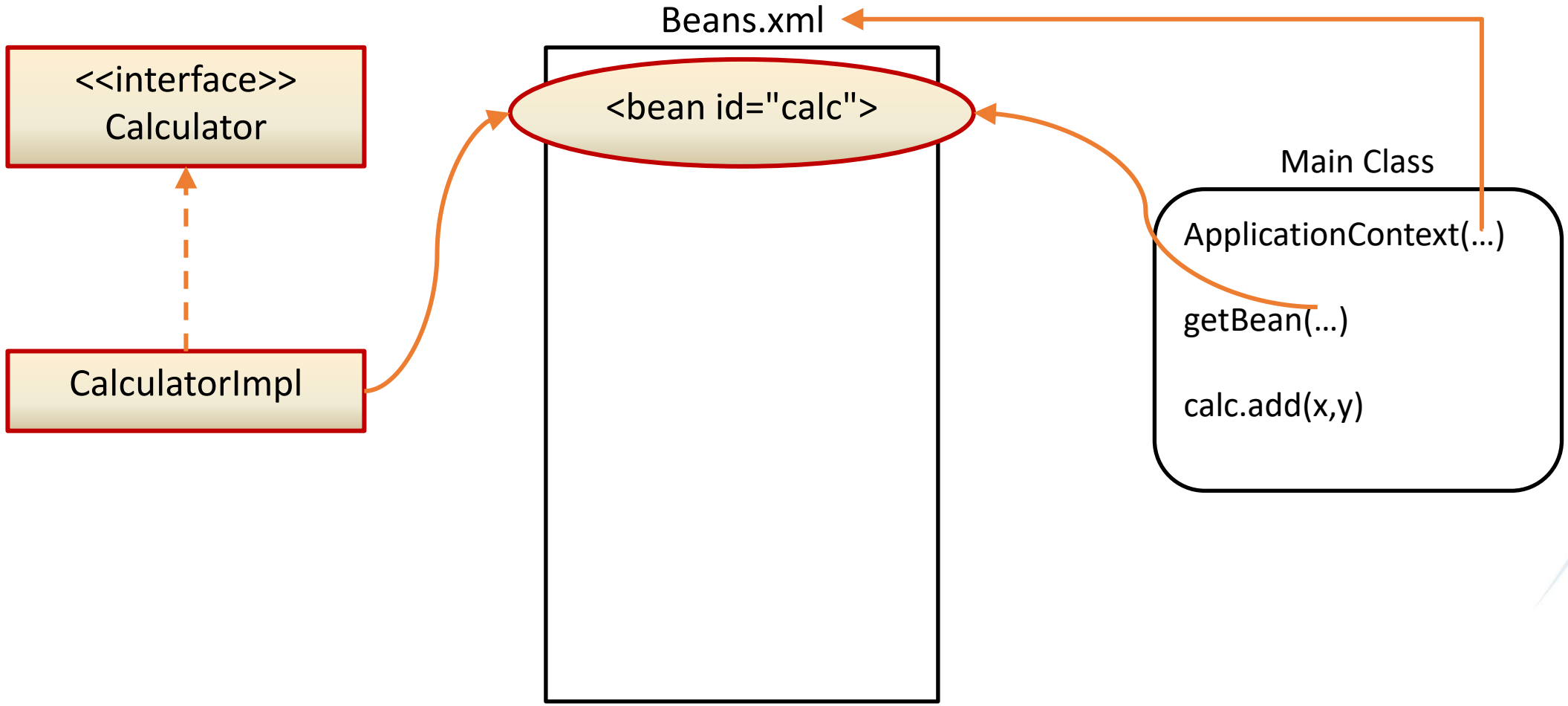


Calculator Case Workflow (Ex.)

- beans.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="calc"
          class="com.jediver.spring.service.impl.CalculatorImpl"/>
</beans>
```

Calculator Case Workflow (Ex.)



Use the CalculatorImp class in Main method

Calculator Case Workflow (Ex.)

- Main Class:

```
public class Main {

    public static void main(String[] args) {
        ApplicationContext context
                = new ClassPathXmlApplicationContext(
                        "com/jediver/spring/cfg/beans.xml");
        Calculator Calc = context.getBean("calc", Calculator.class);
        Calc.add(5, 10);
        Calc.sub(25, 8);

    }
}
```

Classic Spring Advices

- Spring AOP supports four types of advices:
  1. Before advice:
     - Before the method execution
  2. After returning advice:
     - After the method returns a result
  3. After throwing advice:
     - After the method throws an exception
  4. Around advice:
     - Around the method execution
- When using the Spring AOP, advices are written by implementing one of the advice interfaces

Classic Spring Advices (Before advice)

```
public class CalculatorBefore implements MethodBeforeAdvice {

    @Override
    public void before(Method method, Object[] args, Object target)
            throws Throwable {
        System.out.println("The method: " + method.getName()
                + ";\n" + "The arguments: " + Arrays.toString(args));
    }
}
```

Classic Spring Advices (After returning advice)

```
public class CalculatorAfterReturn implements AfterReturningAdvice {

    @Override
    public void afterReturning(Object returnValue, Method method,
            Object[] args, Object target) throws Throwable {
        System.out.println("The method: " + method.getName()
                + ";\n" + "The arguments: " + Arrays.toString(args)
                + ";\n" + "The return: " + returnValue);
    }
}
```

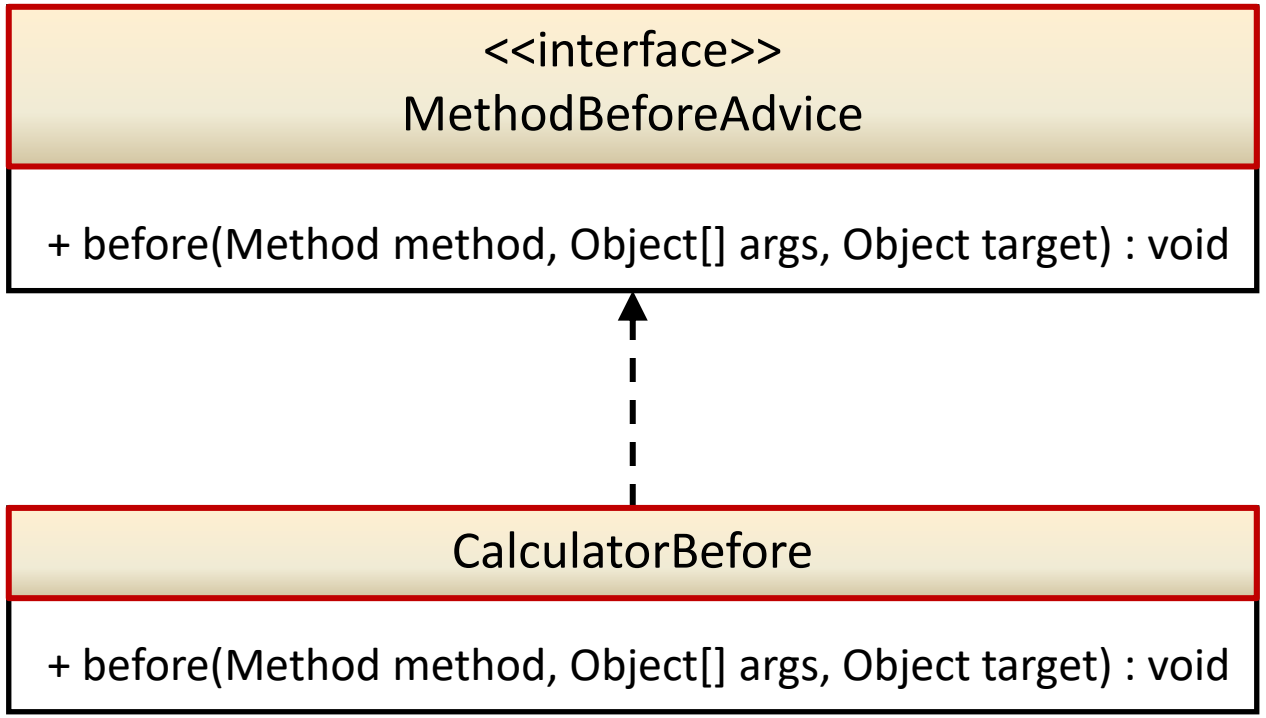
Classic Spring Advices (After throwing advice)

```
public class CalculatorAfterThrow implements ThrowsAdvice {

    public void afterThrowing(IllegalArgumentException exception)
            throws Throwable {
        System.err.println("Illegal arguments.....");
    }
}
```

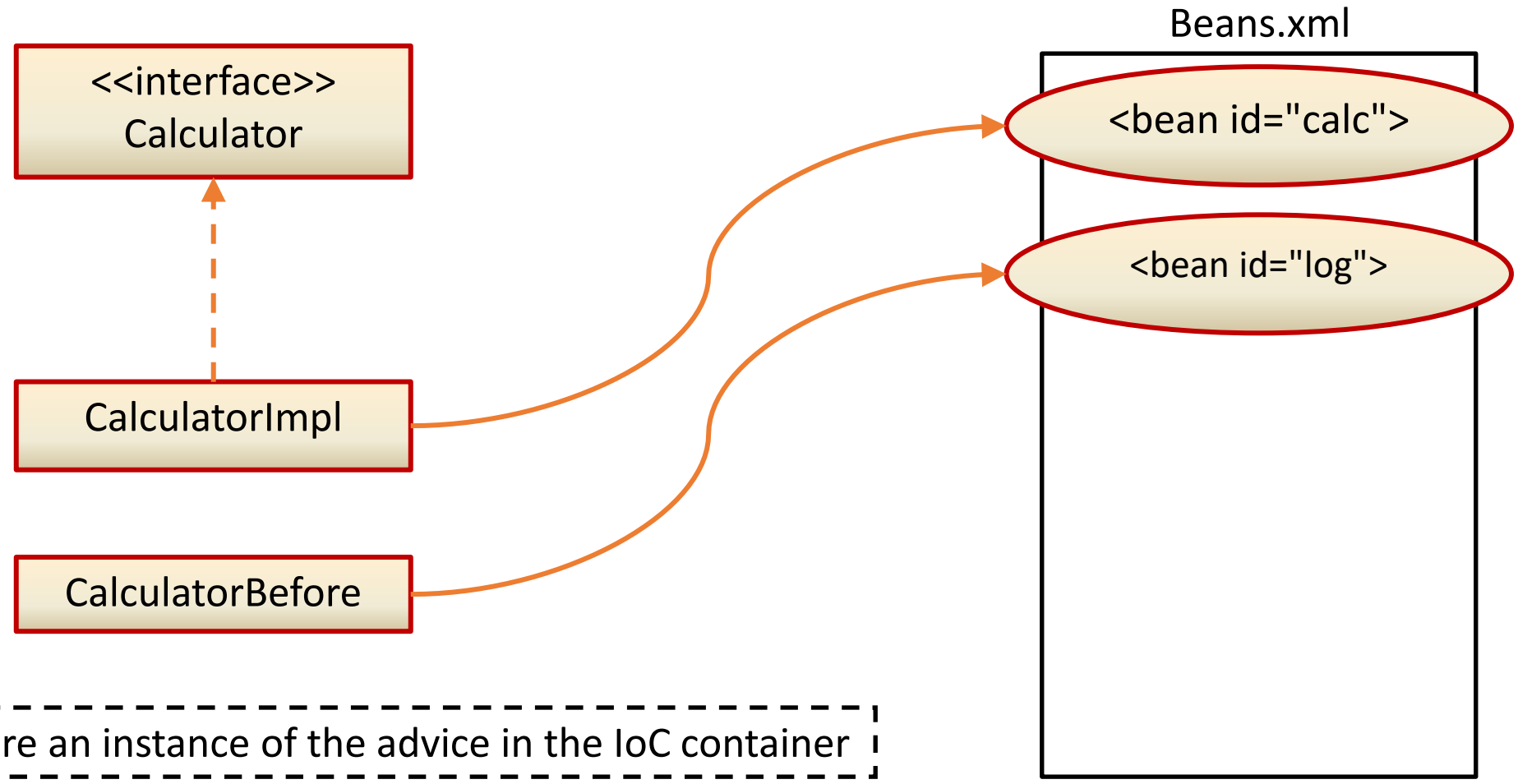
Classic Spring Advices (Around advice)

```
public class CalculatorAround implements MethodInterceptor {

    @Override
    public Object invoke(MethodInvocation mi) throws Throwable {
        System.out.println("The method: " + mi.getMethod().getName()
                + ";\n" + "The arguments: " + Arrays.toString(mi.getArguments()));
        Object result = null;
        try {
            result = mi.proceed();
        } catch (IllegalArgumentException ex) {
            ex.printStackTrace();
            throw ex;
        }
        return result;
    }
}
```

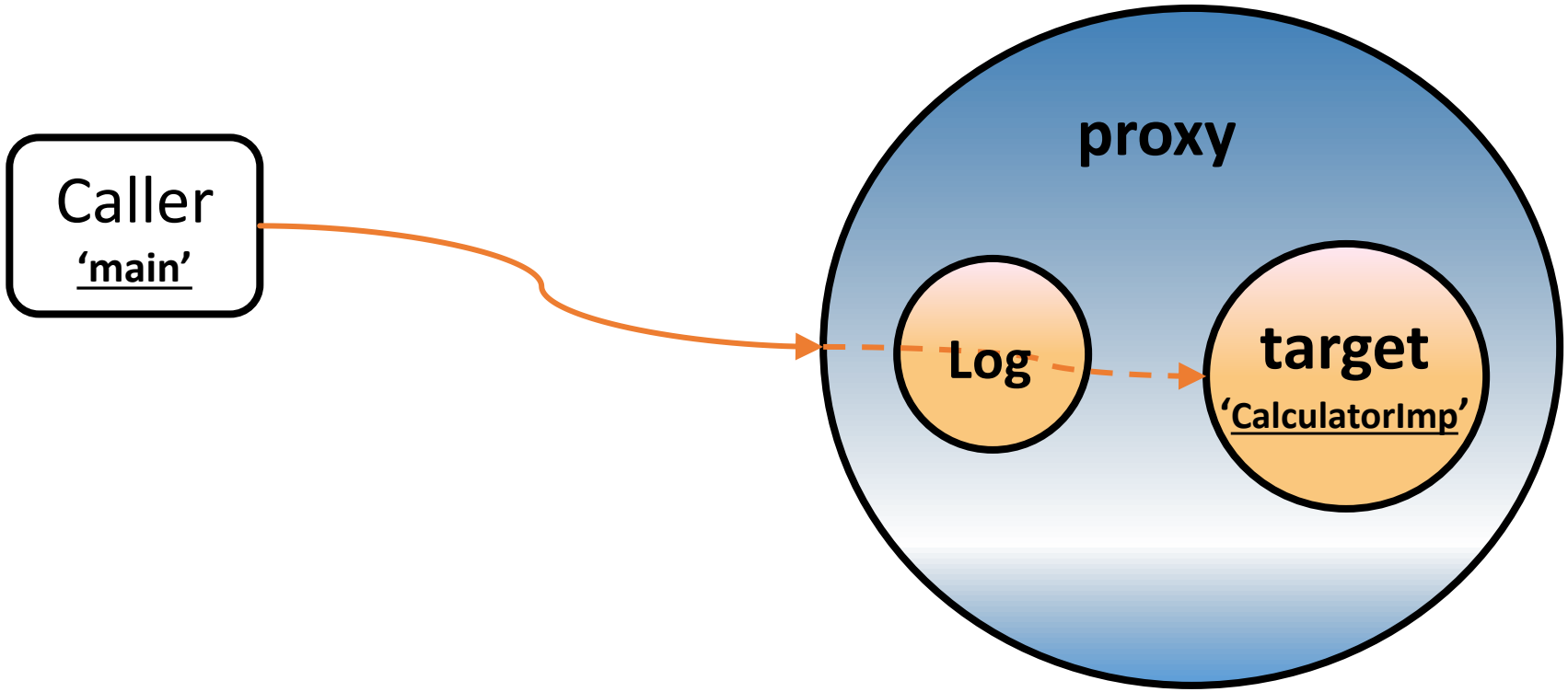
Calculator Case Service Class (UML)



Calculator Case Workflow (Ex.)



Calculator Case Workflow (Ex.)



Create proxy for the target (CalculatorImp class)

Calculator Case Workflow (Ex.)

- The next step is to apply it to the calculator beans.
- First, we have to declare an instance of this advice in the IoC container.
- Then, the most important step is to create a proxy for each of your calculator beans to apply this advice.
- Proxy creation in Spring AOP is accomplished by a factory bean called ProxyFactoryBean.

Calculator Case Workflow (Ex.)

```
<bean id="log"
      class="com.jediver.spring.service.aspect.CalculatorBefore"/>
<bean id="calculatorProxy"
      class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces">
        <list>
            <value>com.jediver.spring.service.Calculator</value>
        </list>
    </property>
    <property name="target" ref="calc"/>
    <property name="interceptorNames">
        <list>
            <value>log</value>
        </list>
    </property>
</bean>
```

Calculator Case Workflow (Ex.)

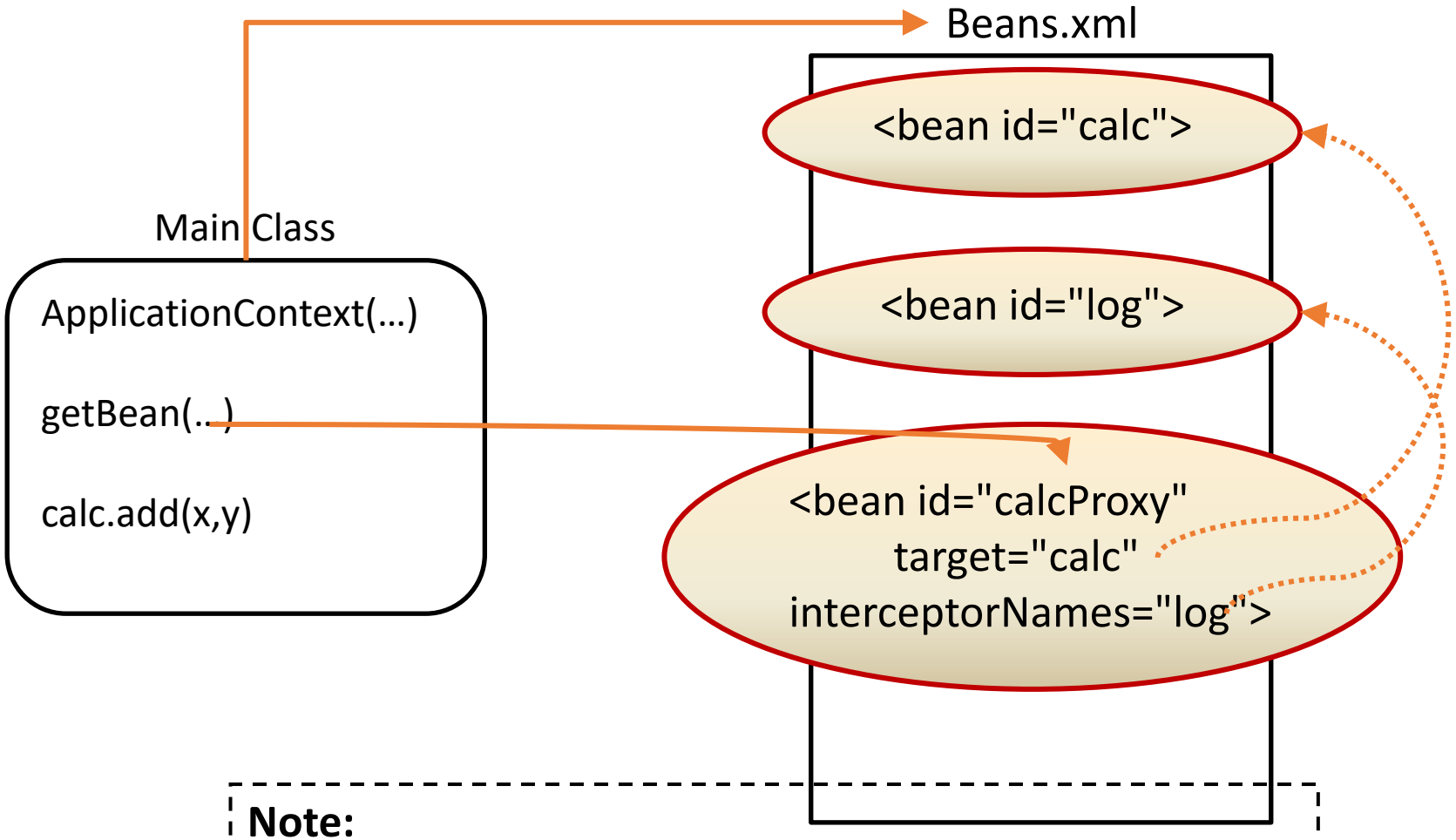
Java™ Education and Technology Services

- By default, ProxyFactoryBean can automatically detect the interfaces that the target bean implements and proxy all of them.
- So, if you want to proxy all the interfaces of a bean, you needn’t specify them explicitly.

Calculator Case Workflow (Ex.)

```
<bean id="log"
      class="com.jediver.spring.service.aspect.CalculatorBefore"/>
<bean id="calculatorProxy"
      class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces">
        <list>
            <value>com.jediver.spring.service.Calculator</value>
        </list>
    </property>
    <property name="target" ref="calc"/>
    <property name="interceptorNames">
        <list>
            <value>log</value>
        </list>
    </property>
</bean>
```

Calculator Case Workflow (Ex.)



Note:
All methods declared in the target class will be advised.

Calculator Case Workflow (Ex.)

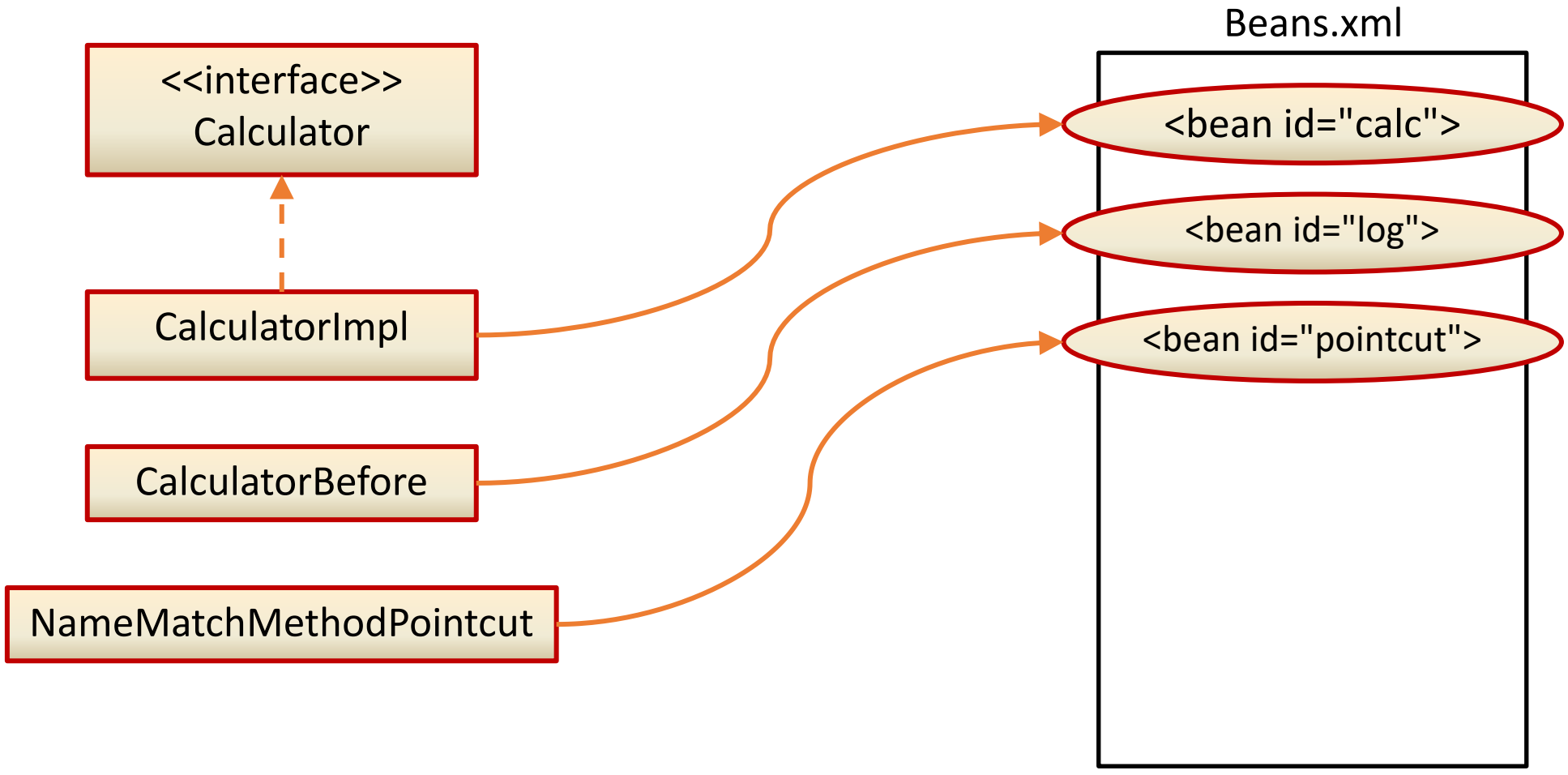
- In the Main class, you should get the proxy beans from the IoC container instead to have your logging advice applied.

```
public static void main(String[] args) {
    ApplicationContext context
            = new ClassPathXmlApplicationContext(
                    "com/jediver/spring/cfg/beans.xml");
    Calculator Calc = context.getBean("calculatorProxy", Calculator.class);
    Calc.add(5, 10);
    Calc.sub(25, 8);
}
```

Classic Spring Pointcuts

- When you specify an advice for an AOP proxy, all of the methods declared in the target class/proxy interfaces will be advised.
- But if you want to advise some of methods, you must declare pointcut.
- Spring provides a family of pointcut classes for you to match program execution points.
- You can simply declare beans of these types in your bean configuration file to define pointcuts.

Calculator Case Workflow (Ex.)



Method Name Pointcuts

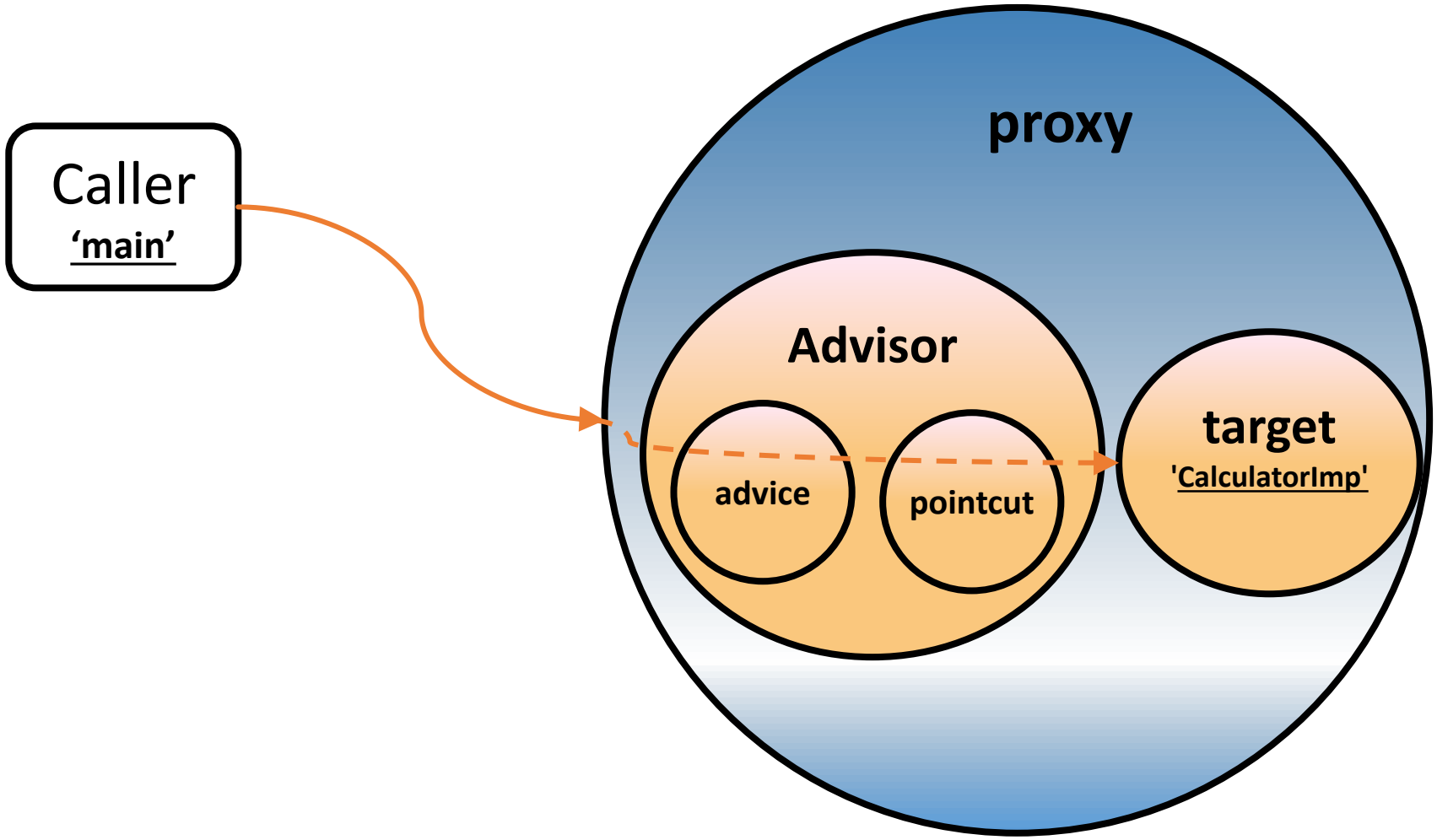
- To advise a single method only, you can use NameMatchMethodPointcut to match the method statically by its name.
- You can specify a particular method name or method name expression with wildcards in the mappedName property.

```
<bean id="methodNamePointcut"
      class="org.springframework.aop.support.NameMatchMethodPointcut">
    <property name="mappedName" value="add"/>
</bean>
```

Advisor

- A pointcut must be associated with an advice to indicate where the advice should be applied.
- Such an association is called an advisor in classic Spring AOP.
- The class DefaultPointcutAdvisor is simply for associating a pointcut and an advice.
- An advisor is applied to a proxy in the same way as an advice.

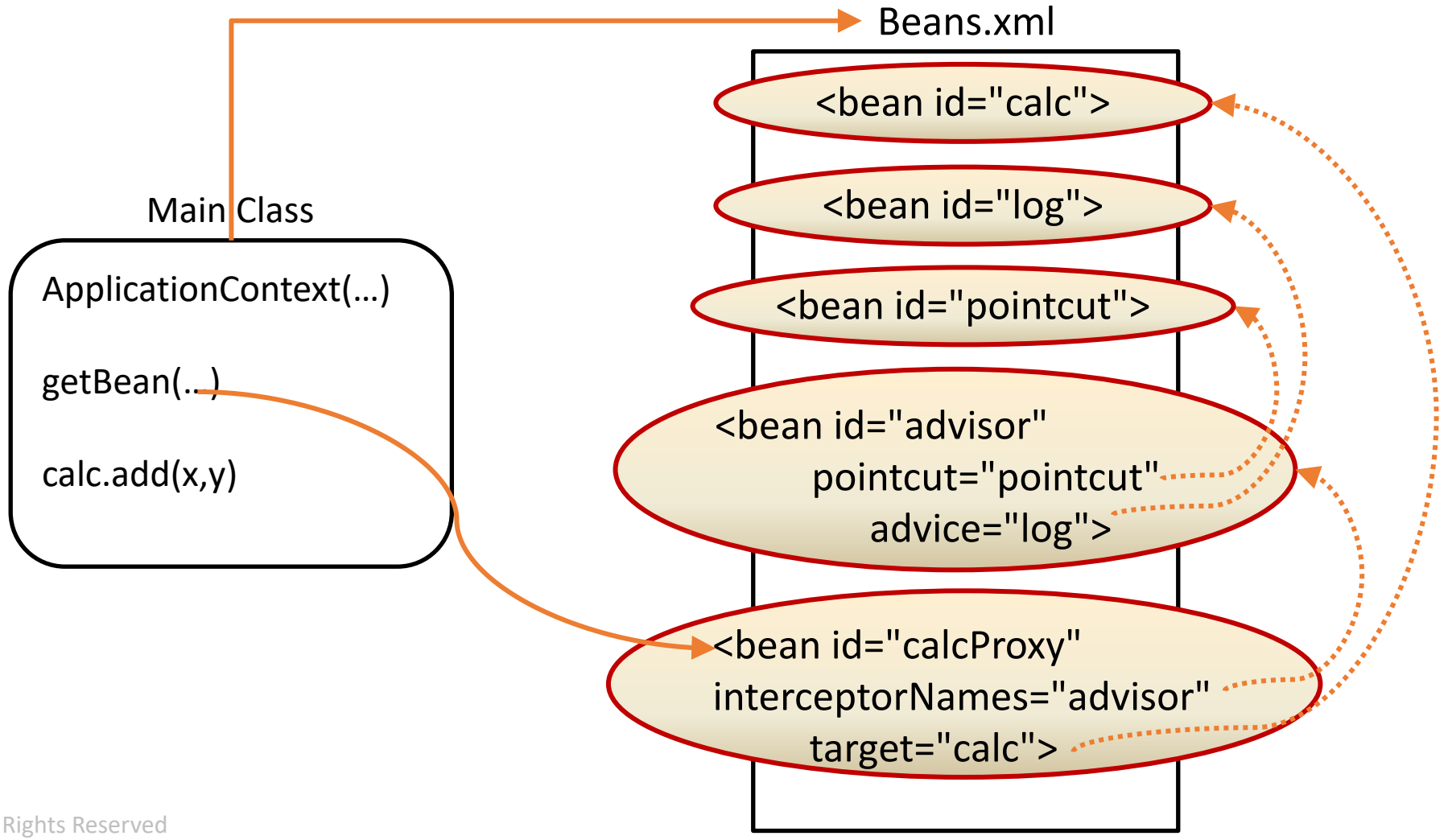
Calculator Case Workflow (Ex.)



Calculator Case Workflow (Ex.)

```
<bean id="methodNameAdvisor"
      class="org.springframework.aop.support.DefaultPointcutAdvisor">
    <property name="pointcut" ref="methodNamePointcut"/>
    <property name="advice" ref="log"/>
</bean>
<bean id="calculatorProxy"
      class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="calc"/>
    <property name="interceptorNames">
        <list>
            <value>methodNameAdvisor</value>
        </list>
    </property>
</bean>
```

Calculator Case Workflow (Ex.)



Local Cut For Multi Method

- Spring provides a convenient advisor class for you to declare an advisor in one shot. For NameMatchMethodPointcut, the advisor class is NameMatchMethodPointcutAdvisor.

```
<bean id="methodNameAdvisor"
      class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
    <property name="mappedNames">
        <list>
            <value>add</value>
            <value>sub</value>
        </list>
    </property>
    <property name="advice" ref="log"/>
</bean>
```

Regular Expression Pointcuts

- You can match methods using a regular expression.
- You can use RegexMethodPointcutAdvisor to specify one or more regular expressions.
- Ex. The following regex match the methods with the keyword multi or div in the method name:

```
<bean id="regexAdvisor"
      class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="patterns">
        <list>
            <value>.*multi.*</value>
            <value>.*div.*</value>
        </list>
    </property>
    <property name="advice" ref="log"/>
</bean>
```

Regular Expression Pointcuts

- The following define the proxy with the two advisors:

```
<bean id="calculatorProxy"
      class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="calc"/>
    <property name="interceptorNames">
        <list>
            <value>methodNameAdvisor</value>
            <value>regexAdvisor</value>
        </list>
    </property>
</bean>
```

Creating Proxies Automatically

- In Spring AOP, you need to create a proxy for each bean to be advised and link it with the target bean.
- Spring provides a facility called auto proxy creator to create proxies for your beans automatically.
- With an auto proxy creator, you no longer need to create proxies manually with ProxyFactoryBean.
- Spring has two built-in auto proxy creator implementations for you to choose from.
  - BeanNameAutoProxyCreator:
  - DefaultAdvisorAutoProxyCreator

Creating Proxies Automatically

- Spring has two built-in auto proxy creator implementations for you to choose from.
  - **BeanNameAutoProxyCreator:**
  - It requires a list of bean name expressions to be configured.
  - In each bean name expression, you can use wildcards to match a group of beans.
  - Ex. The following auto proxy creator will create proxies for the beans whose names end with Calc. Each of the proxies created will be advised by the advisors specified in the auto proxy creator.

Creating Proxies Automatically (Ex.)

```
<bean id="userCalc"
      class="com.jediver.spring.service.impl.CalculatorImpl"/>
<bean id="calculatorProxy"
      class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
    <property name="beanNames">
        <list>
            <value>*Calc</value>
        </list>
    </property>
    <property name="interceptorNames">
        <list>
            <value>methodNameAdvisor</value>
            <value>regexAdvisor</value>
        </list>
    </property>
</bean>
```

Creating Proxies Automatically (Ex.)

- In the Main class, you can simply get the beans by their original names even without knowing that they have been proxied.

```
public static void main(String[] args) {
    ApplicationContext context
            = new ClassPathXmlApplicationContext(
                    "com/jediver/spring/cfg/beans.xml");
    Calculator Calc = context.getBean("userCalc", Calculator.class);
    Calc.add(5, 10);
    Calc.sub(25, 8);
}
```

Creating Proxies Automatically

- Spring has two built-in auto proxy creator implementations for you to choose from.
  - **DefaultAdvisorAutoProxyCreator:**
  - There's nothing you have to configure for this auto proxy creator.
  - It will automatically check for each bean with each advisor declared in the IoC container.
  - If any of the beans is matched by an advisor's pointcut, DefaultAdvisorAutoProxyCreator will automatically create a proxy for it.

```
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>
```

  - However, you must use this auto proxy creator with great care, as it may advise beans that you don't expect to be advised.

Creating Proxies Automatically (Ex.)

- In the Main class, you can simply get the beans by their original names even without knowing that they have been proxied.

```
public static void main(String[] args) {
    ApplicationContext context
            = new ClassPathXmlApplicationContext(
                    "com/jediver/spring/cfg/beans.xml");
    Calculator Calc = context.getBean("userCalc", Calculator.class);
    Calc.add(5, 10);
    Calc.sub(25, 8);
}
```

Lesson 5

@AspectJ annotation-driven aspects

@AspectJ support

- @AspectJ refers to a style of declaring aspects as regular Java classes annotated with annotations.
- The @AspectJ style was introduced by the AspectJ project as part of the AspectJ 5 release.
- Spring interprets the same annotations as AspectJ 5, using a library supplied by AspectJ for pointcut parsing and matching.
- The AOP runtime is still pure Spring AOP, though, and there is no dependency on the AspectJ compiler or weaver.

Enabling @AspectJ Support

- To use @AspectJ aspects in a Spring configuration, you need to enable Spring support for configuring Spring AOP based on @AspectJ aspects and auto-proxying beans based on whether or not they are advised by those aspects.
- By auto-proxying, we mean that, Spring will automatically create proxies for any of your beans that are matched by your AspectJ aspects.
- You also need to ensure that AspectJ's aspectjweaver.jar library is on the classpath of your application (version 1.8 or later).
- This library is available in the lib directory of an AspectJ distribution or from the Maven Central repository.

Enabling @AspectJ Support (Ex.)

- First of all you must enable spring aop and enable auto-proxying, so you must import it by aop namespace:

```
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"
```

- <aspectj-autoproxy> Tag that enable spring aop and enable auto-proxying:

```
<aop:aspectj-autoproxy/>
```

Declaring an Aspect

- Starting from Spring 2.x AOP framework supports its aspects to be written as POJOs annotated with @Aspect.
- But you still have to register the aspect in the Spring IoC container by declaring them as bean definition.
- With @AspectJ support enabled, any bean defined in your application context with a class that is an @AspectJ aspect (has the @Aspect annotation) is automatically detected by Spring and used to configure Spring AOP.

Declaring an Aspect (Ex.)

- Declare the aspect class be Annotating with @Aspect:

```
@Aspect
public class CalculatorBefore {
```

- We still have to register the aspect in the Spring IoC container by declaring them as bean definition:

```
<bean id="log"
      class="com.jediver.spring.service.aspect.CalculatorBefore"/>
```

Declaring an Advice

- An advice is a simple Java method in an aspect with one of the advice annotations:
  - * @Before
  - * @Around
  - * @AfterReturning
  - * @AfterThrowing
  - * @After
- Advice is associated with a pointcut expression and runs before, after, or around method executions matched by the pointcut.
- The pointcut expression may be either a simple reference to a named pointcut or a pointcut expression declared in place.
- **NOTE:** An aspect may include one or more advices.

Declaring an Advice (Ex.) - Before

```
@Aspect
public class CalculatorBefore {

    @Before("execution(* com.jediver.spring.service..add(..))")
    public void before() {
        System.out.println("Before your service method");
    }
}
```

any modifier & any return type

any class under service package or subpackages

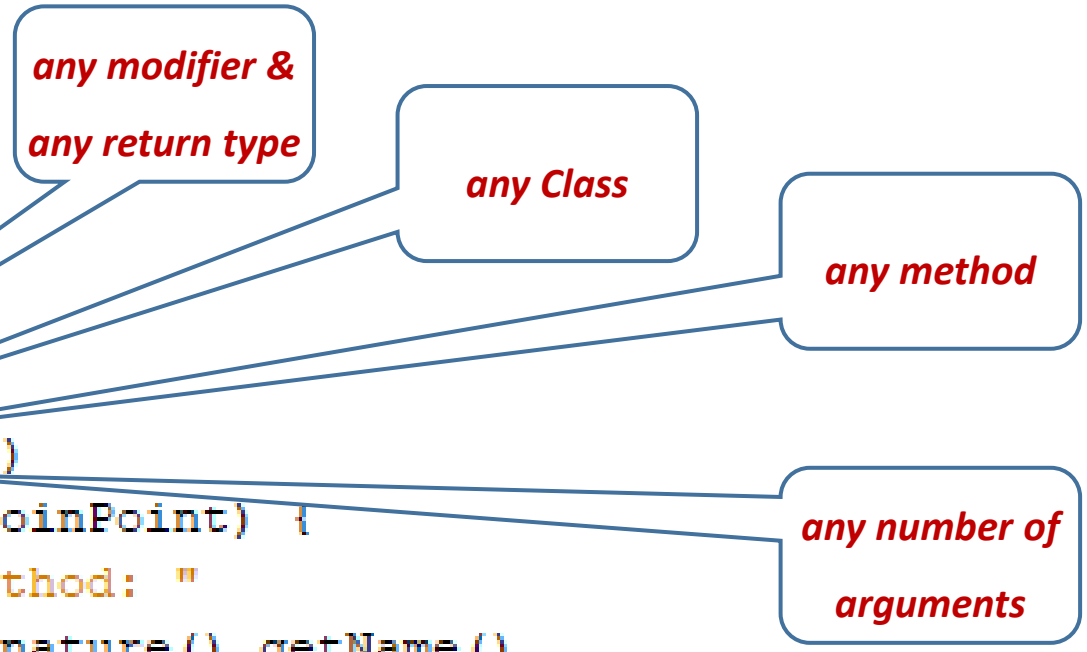
any number of arguments

Declaring an Advice (Ex.) - Before

- In an advice, to access the detail of the current join point,
- you can declare an argument of type JoinPoint in your advice method.
- Then you can get access to join point details such as the method name and argument values.

Declaring an Advice (Ex.) - Before

```
@Aspect
public class CalculatorBefore {

    @Before("execution(* *.*(..))")
    public void before(JoinPoint joinPoint) {
        System.out.println("The method: "
                + joinPoint.getSignature().getName()
                + ";\n" + "The arguments: "
                + Arrays.toString(joinPoint.getArgs()));
    }
}
```



Declaring an Advice (Ex.) - Before

```
@Aspect
public class CalculatorBefore {

    @Before("execution(* com.jediver.spring.service..add(..))")
    public void before(JoinPoint joinPoint)
            throws Throwable {
        System.out.println("The method: "
                + joinPoint.getSignature().getName()
                + ";\n" + "The arguments: "
                + Arrays.toString(joinPoint.getArgs()));
    }
}
```

any modifier & any return type

any class under service package or subpackages

any number of arguments

Declaring an Advice (Ex.) - After

- An after advice is executed after a join point finishes, whenever it returns a result or throws an exception.

```
@Aspect
public class CalculatorAfter {

    @After("execution(* *.*(..))")
    public void after(JoinPoint joinPoint) {
        System.out.println("The method: "
                + joinPoint.getSignature().getName()
                + ";\n" + "The arguments: "
                + Arrays.toString(joinPoint.getArgs()));
    }
}
```

any modifier & any return type

any Class

any method

any number of arguments

Declaring an Advice (Ex.) - AfterReturning

```
@Aspect
public class CalculatorAfterReturn {

    @AfterReturning("execution(* *.*(..))")
    public void after(JoinPoint joinPoint) {
        System.out.println("1-The method: "
                + joinPoint.getSignature().getName()
                + ";\n" + "The arguments: "
                + Arrays.toString(joinPoint.getArgs()));
    }
```

Declaring an Advice (Ex.) - AfterReturning

```
    @AfterReturning(returning = "result",
            pointcut = "execution(* com.jediver.spring.service..add(..))")
    public void after(JoinPoint joinPoint, Object result) {
        System.out.println("2-The method: "
                + joinPoint.getSignature().getName()
                + ";\n" + "The arguments: "
                + Arrays.toString(joinPoint.getArgs())
                + ";\n" + "The return: " + result);
    }
}
```

Declaring an Advice (Ex.) - AfterThrowing

```
@Aspect
public class CalculatorAfterThrow {

    @AfterThrowing("execution(* com.jediver.spring.service..add(..))")
    public void afterThrowing(JoinPoint joinPoint)
            throws Throwable {
        System.out.println("The method: "
                + joinPoint.getSignature().getName()
                + ";\n" + "The arguments: "
                + Arrays.toString(joinPoint.getArgs()));
        System.err.println("Exception.......");
    }
```

- If you are interested in one particular type of exception, you can declare it as the argument type of the exception.
- Then your advice will be executed only when exceptions of compatible type (i.e., this type and its subtypes) are thrown

Declaring an Advice (Ex.) - AfterThrowing

```
    @AfterThrowing(pointcut = "execution(* *.*(..))",
            throwing = "exception")
    public void afterThrowing(IllegalArgumentException exception)
            throws Throwable {
        System.err.println("Illegal arguments.......");
    }

    @AfterThrowing(throwing = "throwable",
            pointcut = "execution(* com.jediver.spring.service..add(..))")
    public void afterThrowing(JoinPoint joinPoint, Throwable throwable)
            throws Throwable {
        System.out.println("The method: "
                + joinPoint.getSignature().getName()
                + ";\n" + "The arguments: "
                + Arrays.toString(joinPoint.getArgs()));
        System.err.println("Illegal arguments.......");
    }
}
```

Declaring an Advice (Ex.) - Around

- It is the most powerful of all the advice types.
- You can combine all the actions of the preceding advices into one single advice.
- You can even control when, and whether, to proceed with the original join point execution.

Declaring an Advice (Ex.) - Around

```
@Aspect
public class CalculatorAround {

    @Around("execution(* com.jediver.spring.service..add(..))")
    public Object around(ProceedingJoinPoint joinPoint)
            throws Throwable {
        System.out.println("The method: "
                + joinPoint.getSignature().getName()
                + ";\n" + "The arguments: "
                + Arrays.toString(joinPoint.getArgs()));
        Object result = null;
        try {
            result = joinPoint.proceed();
        } catch (IllegalArgumentException ex) {
            ex.printStackTrace();
            throw ex;
        }
        return result;
    }
}
```

Specifying Aspect Precedence

- When there's more than one aspect applied to the same join point, the precedence of the aspects is undefined unless you have explicitly specified it.

- The precedence of aspects can be specified by using the @Order annotation.

```
@Aspect
@Order(1)
public class CalculatorBefore {
```

```
@Aspect
@Order(0)
public class CalculatorBefore1 {
```

Reusing Pointcut Definitions

- A pointcut can be declared as a simple method with the @Pointcut annotation.
- The method body of a pointcut is usually empty, as it is unreasonable to mix a pointcut definition with application logic.
- The access modifier of a pointcut method controls the visibility of this pointcut as well.
- Other advices can refer to this pointcut by the method name.

Reusing Pointcut Definitions (Ex.)

```
@Aspect
public class CalculatorAspect {

    @Pointcut("execution(* com.jediver.spring.service..add(..))")
    private void addOperation() {
    }

    @AfterReturning(returning = "result",
            pointcut = "addOperation()")
    public void after(JoinPoint joinPoint, Object result) {...7 lines }

    @Before("addOperation()")
    public void before(JoinPoint joinPoint) {...6 lines }

}
```

Reusing Pointcut Definitions

- If your pointcuts are shared between multiple aspects, it is better to centralize them in a common class. In this case, they must be declared as public.

```
@Aspect
public class CalculatorPointcut {

    @Pointcut("execution(* com.jediver.spring.service..add(..))")
    public void addOperation() {
    }
}
```

Reusing Pointcut Definitions (Ex.)

```
@Aspect
public class CalculatorBefore {

    @Before("CalculatorPointcut.addOperation()")
    public void before(JoinPoint joinPoint) {
        System.out.println("The method: "
                + joinPoint.getSignature().getName()
                + ";\n" + "The arguments: "
                + Arrays.toString(joinPoint.getArgs()));
    }
}
```

- When you refer to this pointcut, you have to include the class name as well.

```
@Aspect
public class CalculatorAfterReturn {

    @Before("CalculatorPointcut.addOperation()")
    public void after(JoinPoint joinPoint) {
        System.out.println("1-The method: "
                + joinPoint.getSignature().getName()
                + ";\n" + "The arguments: "
                + Arrays.toString(joinPoint.getArgs()));
    }
}
```

Lesson 6

Introduction to "Aspect Introduction"

Introducing Behaviors to Beans

- Introduction is a special type of advice in AOP.
- It allows your objects to implement an interface dynamically by providing an implementation class for that interface.
- It seems as if your objects had extended the implementation class at runtime.
- You are able to introduce multiple interfaces with multiple implementation classes to your objects at the same time.
- This can achieve the same effect as multiple inheritance.

Introducing Behaviors to Beans Example

- You have two interfaces:
  - MaxCalculator to define the max() operation.
  - MinCalculator to define the min() operation.
- Then you have an implementation for each interface.
- With introduction, you can make CalculatorImp dynamically implement both
  - The MaxCalculator interface by using the implementation class MaxCalculatorImp.
  - The MinCalculator interface by using the implementation class MinCalculatorImp.
- The brilliant idea behind introduction is that you needn't modify the CalculatorImp class to introduce new methods.
- That means you can introduce methods to your existing classes even without source code available.

Introducing Behaviors to Beans Example (Ex.)

```
public interface MaxCalculator {

    public double max(double firstOperand, double secondOperand);
}
```

```
public class MaxCalculatorImpl implements MaxCalculator {

    @Override
    public double max(double firstOperand, double secondOperand) {
        double result = (firstOperand >= secondOperand)
                ? firstOperand : secondOperand;
        System.out.println("The Max is:" + result);
        return result;
    }
}
```

Introducing Behaviors to Beans Example (Ex.)

```
public interface MinCalculator {

    public double min(double firstOperand, double secondOperand);
}
```

```
public class MinCalculatorImpl implements MinCalculator {

    @Override
    public double min(double firstOperand, double secondOperand) {
        double result = (firstOperand <= secondOperand)
                ? firstOperand : secondOperand;
        System.out.println("The Max is:" + result);
        return result;
    }
}
```

Introducing Behaviors to Beans Example

- Introductions, like advices, must be declared within an aspect.
- You can declare an introduction by annotating an arbitrary field with the @DeclareParents.

```
@Aspect
public class CalculatorAspect {

    @DeclareParents(defaultImpl = MaxCalculatorImpl.class,
            value = "com.jediver.spring.service.impl.CalculatorImpl")
    public MaxCalculator maxCalculator;
    @DeclareParents(defaultImpl = MinCalculatorImpl.class,
            value = "com.jediver.spring.service.impl.CalculatorImpl")
    public MinCalculator minCalculator;
}
```

Introducing Behaviors to Beans Example

- The last step, don't forget to declare an instance of this aspect in the application context and your bean without define definitions for MaxCalculatorImpl and MinCalculatorImpl.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop/spring-aop.xsd">
    <aop:aspectj-autoproxy/>
    <bean id="calc"
          class="com.jediver.spring.service.impl.CalculatorImpl"/>
    <bean id="aspect"
          class="com.jediver.spring.service.aspect.CalculatorAspect"/>
</beans>
```

Introducing Behaviors to Beans Example

- Finally your MainClass:

```
public static void main(String[] args) {
    ApplicationContext context
            = new ClassPathXmlApplicationContext(
                    "com/jediver/spring/cfg/beans.xml");
    Calculator calculator = context.getBean("calc", Calculator.class);
    calculator.add(5, 10);
    calculator.sub(25, 8);
    MaxCalculator maxCalculator = (MaxCalculator) calculator;
    maxCalculator.max(4, 5);
    MinCalculator minCalculator = (MinCalculator) calculator;
    minCalculator.min(8, 6);

}
```

Lesson 7

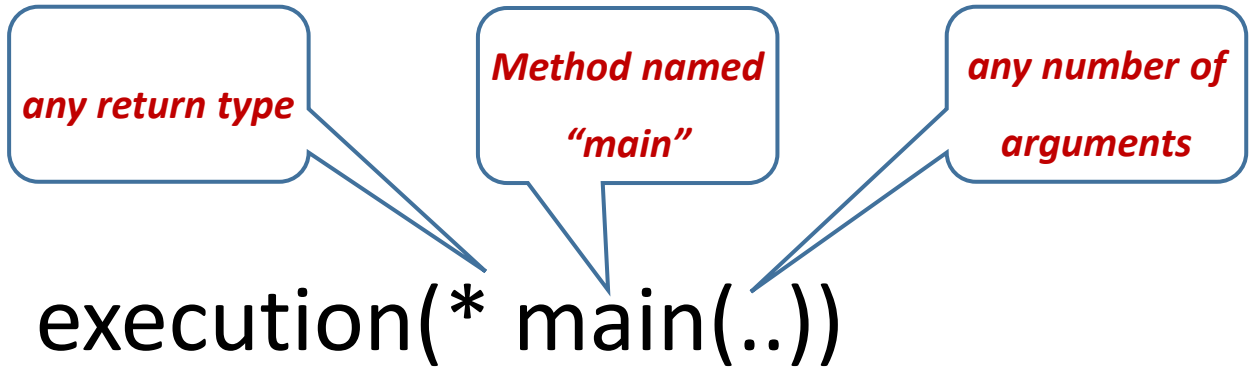
More Details about Pointcut

Method Execution Join Points

- A method execution join point occurs whenever a method is executed.
- Lots of information can be used for matching these join points:
  - Method name
  - Parameter types
  - Return type
  - Declared exceptions
  - Declaring type
  - Modifiers

Method Execution Join Points

- execution:
- The execution pointcut designator matches method and constructor execution join points based on the signature of the method/constructor.
- At a minimum you must specify:
  - A return type pattern
  - A name pattern
  - A parameter list pattern
- Everything else is optional
- Example:



Method Execution Join Points

- In a name pattern:
  - You may use * to match any number of characters
  - No other wildcards are allowed
  - Ex. (get*)
- In return type pattern:
  - The most basic type pattern is *
  - It matches any type
- In parameter list pattern:
  - The most basic parameter list pattern is '..'
  - It matches zero or more parameters of any type.

Simple type matching

- A type name matches that type
  - execution(void main(..))
  - execution(* main(String[]))
- If a type is not imported, you must fully qualify the name
  - execution(* *.*(java.io.Serializable))
- "imports" only make sense in the context of a source code compilation unit.
- Pointcut expressions in annotations and in Spring XML files must always use fully qualified type names

Matching modifiers

- Just use the modifiers as you would in the corresponding member declaration
  - execution(public static void main(..))
- You can also use negation
  - execution(!private * *.*(..))
- The absence of any modifiers at the start of a signature pattern does not mean "default visibility" to match members with default visibility use.
  - !private !protected !public …

Parameter matching Examples

- … (..)
- any parameters
- … (*)
- a single parameter of any type
- … (*,..)
- at least one parameter
- … (*,..,String,*)
- at least three parameters, the penultimate must be a String (!)

Advanced type matching

- You can use the * wildcard in type name matching
  - *Dao
- Use ".*" postfix to match any type in a given package
  - java.io.*
- Use "..*" postfix to match any type in a given package or 'sub-package'
  - com.jediver.myapp..*

Advanced type matching (Ex.)

- The "+" postfix on the end of a type pattern means (or any subtype of a type matched by that pattern)
  - Number+
    - Number, Integer, Float, Double,…
- Any type pattern can be negated with '!'
  - !(Number+)
- In signature matching, a type matches exactly that type, and that type only
  - This is matching based on what is declared in the signature: 'Number' does not match Float

Advanced type matching (Ex.)

- You can even combine type patterns if you need too
  - (com.jediver.myapp..*+ && !Serializable+)
  - A type in com.jediver.myapp or sub-package (or a sub-type of any of them), that is not serializable
  - (int || Integer)
  - Either the primitive type int, or the wrapper

Matching constructor execution

- Constructors have no return type in their signature
- So you don't specify a return type pattern
- Use the method name 'new'
  - execution(new(..))
- matches any constructor execution
- can still use parameter, modifier, throws patterns

Matching constructor execution (Ex.)

- But what type are we constructing???
- To match based on this information, we need to use a declaring type pattern
- just prefix the method name with the (optional) declaring type pattern to further narrow matching
  - execution(Account.new(..))
  - execution(Serializable+.new(..))

Declaring type for method execution

- We can of course specify a declaring type pattern when matching method executions too
  - execution(public * org.xyz.myapp..*.*(..))
  - execution(* Account.get*(..))

Method execution in overriding

- execution(* Account.toString())
- Means an execution join point that has a signature "Account.toString()";
- execution(* SubAccount.toString())
- A join point has multiple signatures

1. String SubAccount.toString()
2. String Account.toString()
3. String Object.toString()

- If no Accounts override toString() this will not match Object.toString().

Lesson 8

Declaring Aspects with XML-Based Configurations

Declaring Aspects with XML-Based Configurations

- If your JVM version is 1.4 or below, or you don’t want your application to have a dependency on AspectJ, or you want to override the annotations.
- You can declare aspects with XML file.
- <aop:aspectj-autoproxy/>
- This element should be deleted so that Spring AOP will ignore the AspectJ annotations.

Declaring Aspects

Java™ Education and Technology Services

- All the Spring AOP configurations must be defined inside the <aop:config> element.
- For each aspect:
  - you create an <aop:aspect> element to refer to a backing bean instance for the concrete aspect implementation.
- So, your aspect beans must have an identifier for the <aop:aspect> elements to refer to.

Declaring Aspects (Ex.)

```java
public class CalculatorBefore {

    public void before() {
        System.out.println("Before your Service Method");
    }
}
```

```java
public class CalculatorBefore {

    public void before(JoinPoint joinPoint) {
        System.out.println("The method: "
                + joinPoint.getSignature().getName()
                + ";\n" + "The arguments: "
                + Arrays.toString(joinPoint.getArgs()));
    }
}
```

Declaring Pointcuts

- A pointcut may be defined either
  - under the <aop:aspect> element:
  - The pointcut will be visible to the aspect only.
  - under the <aop:config> element:
  - It will be a global pointcut, which is visible to all the aspects.

Declaring Aspect Advices

- Declared by

  * <aop:before>
  * <aop:around>
  * <aop:after>
  * <aop:after-returning>
  * <aop:after-throwing>

- An advice element requires:
  - Either a pointcut-ref attribute to refer to a pointcut
  - Or a pointcut attribute to embed a pointcut expression directly.
- The method attribute specifies the name of the advice method in the aspect class.

Beans Definitions XML

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop/spring-aop.xsd">
    <bean id="calc"
          class="com.jediver.spring.service.impl.CalculatorImpl"/>
    <bean id="aspect"
          class="com.jediver.spring.service.aspect.CalculatorBefore"/>
```

Beans Definitions XML (Ex.)

```
    <aop:config>
        <aop:aspect id="aspect1" ref="aspect" >
            <!-- @Before -->
            <aop:pointcut id="pointCutBefore"
                          expression="execution(* com.jediver.spring.service..add(..))" />
            <aop:before method="before"
                        pointcut-ref="pointCutBefore" />
            <aop:around>
        </aop:aspect>
    </aop:config>
</beans>
```

References & Recommended Reading

References & Recommended Reading

- Spring Framework Documentation Version 6.0.4
- Spring in Action 5th Edition
- Cloud Native Java
- Learning Spring Boot 2.0
- Spring 5 Recipes: A Problem-Solution Approach