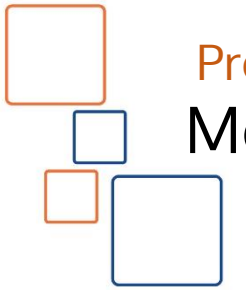
Spring Framework

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

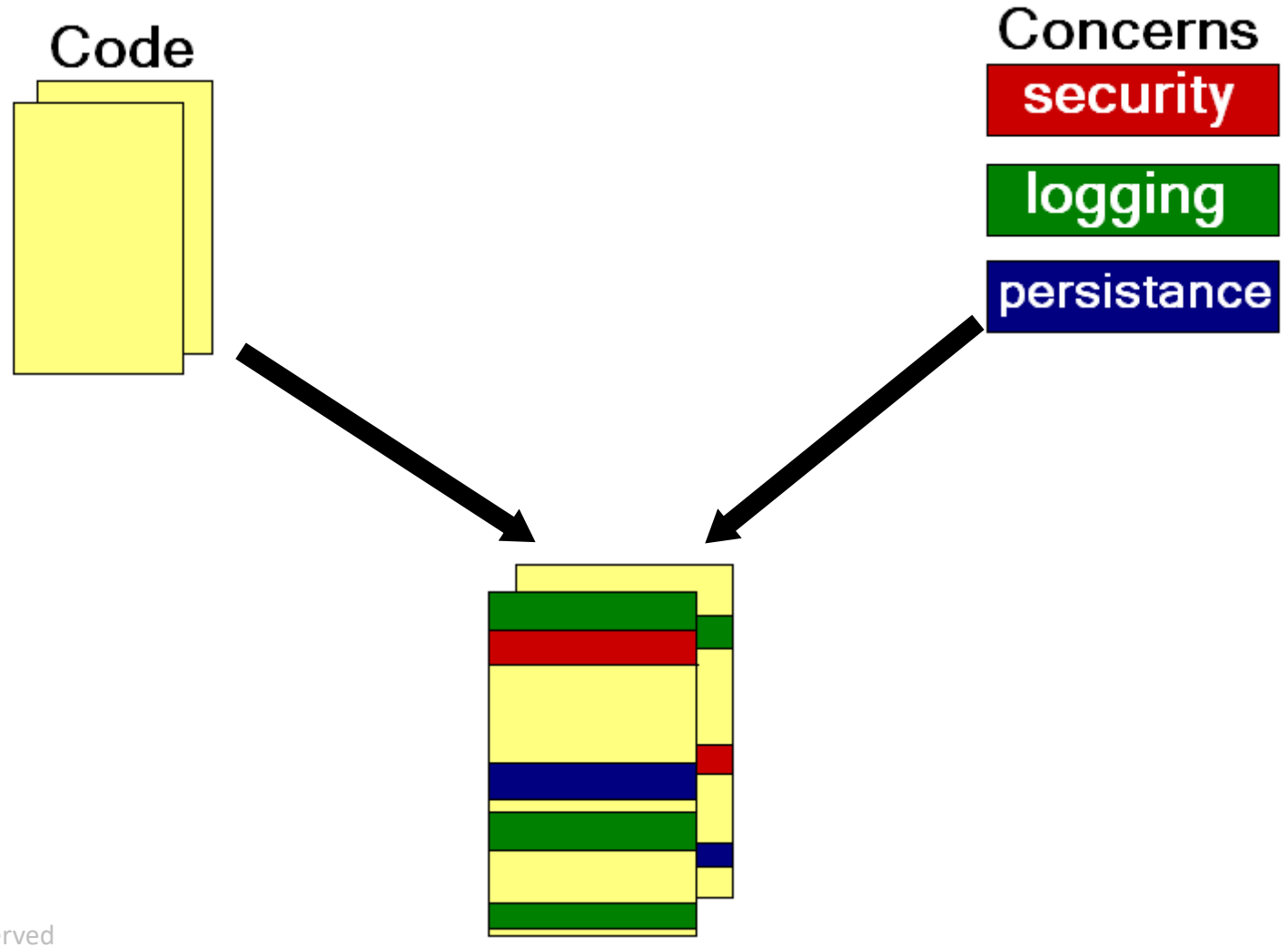
OOP Concept

- 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



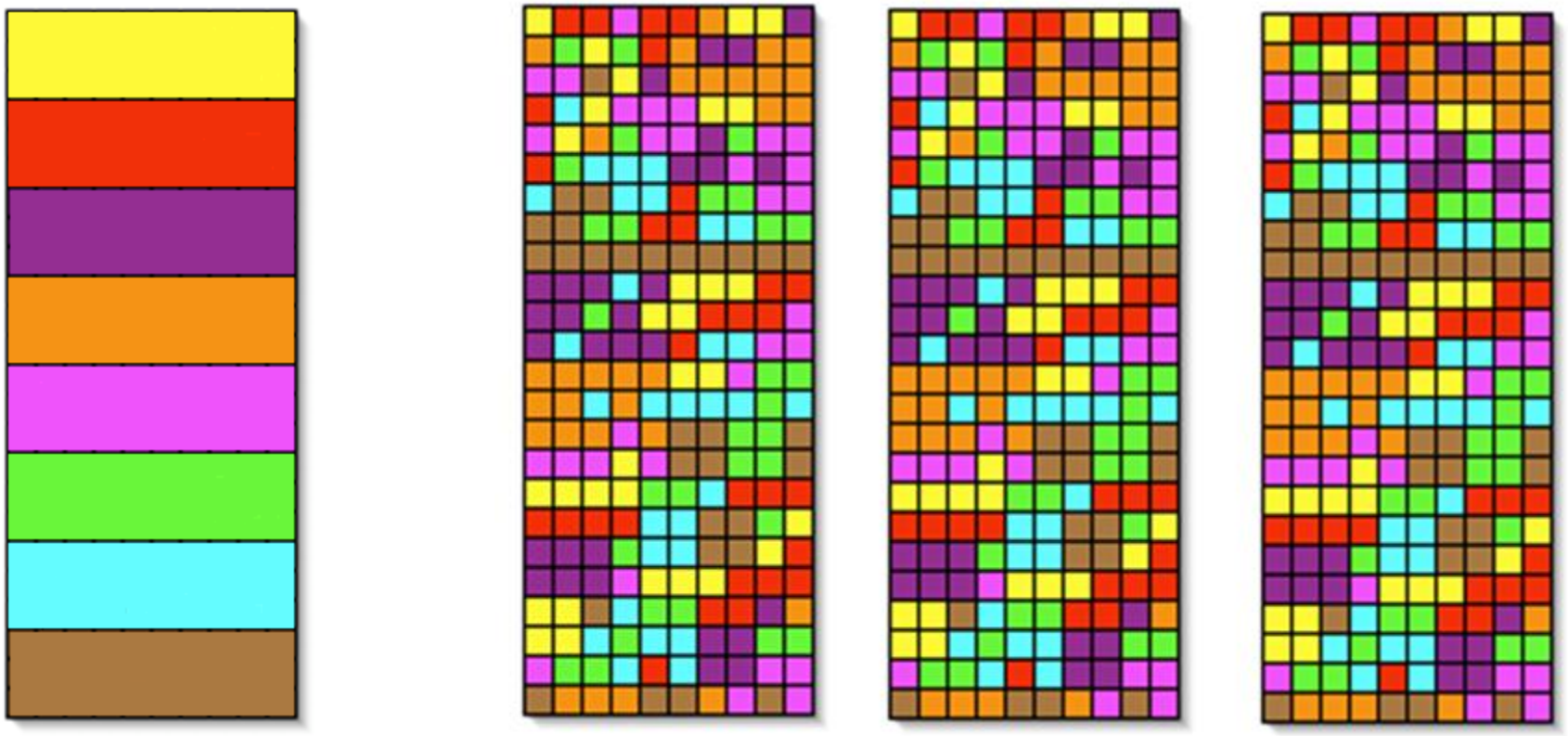
1. Crosscutting functionalities (Ex.)

- 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

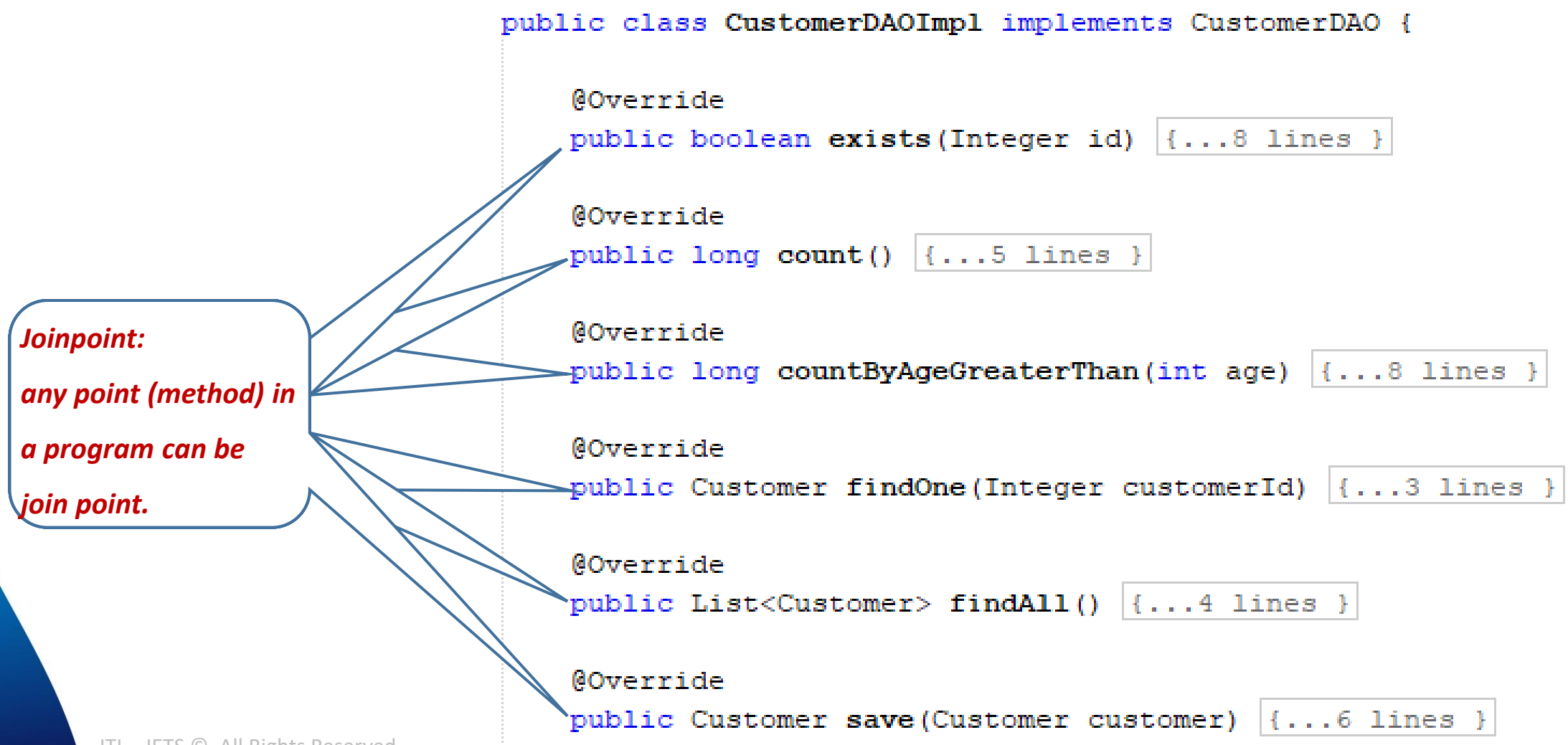
Code Segment without AOP

```
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.

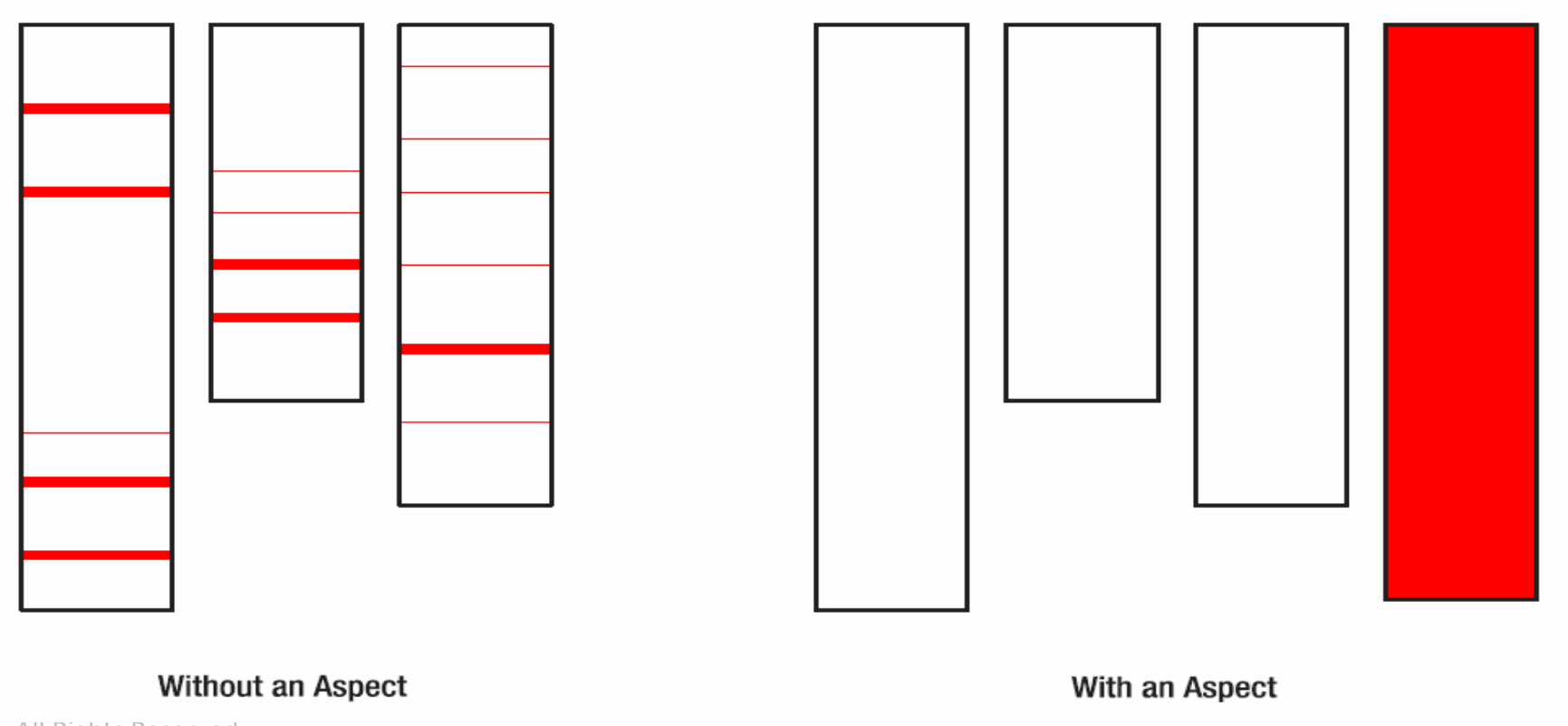
AOP Concepts (Aspect)

- 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).

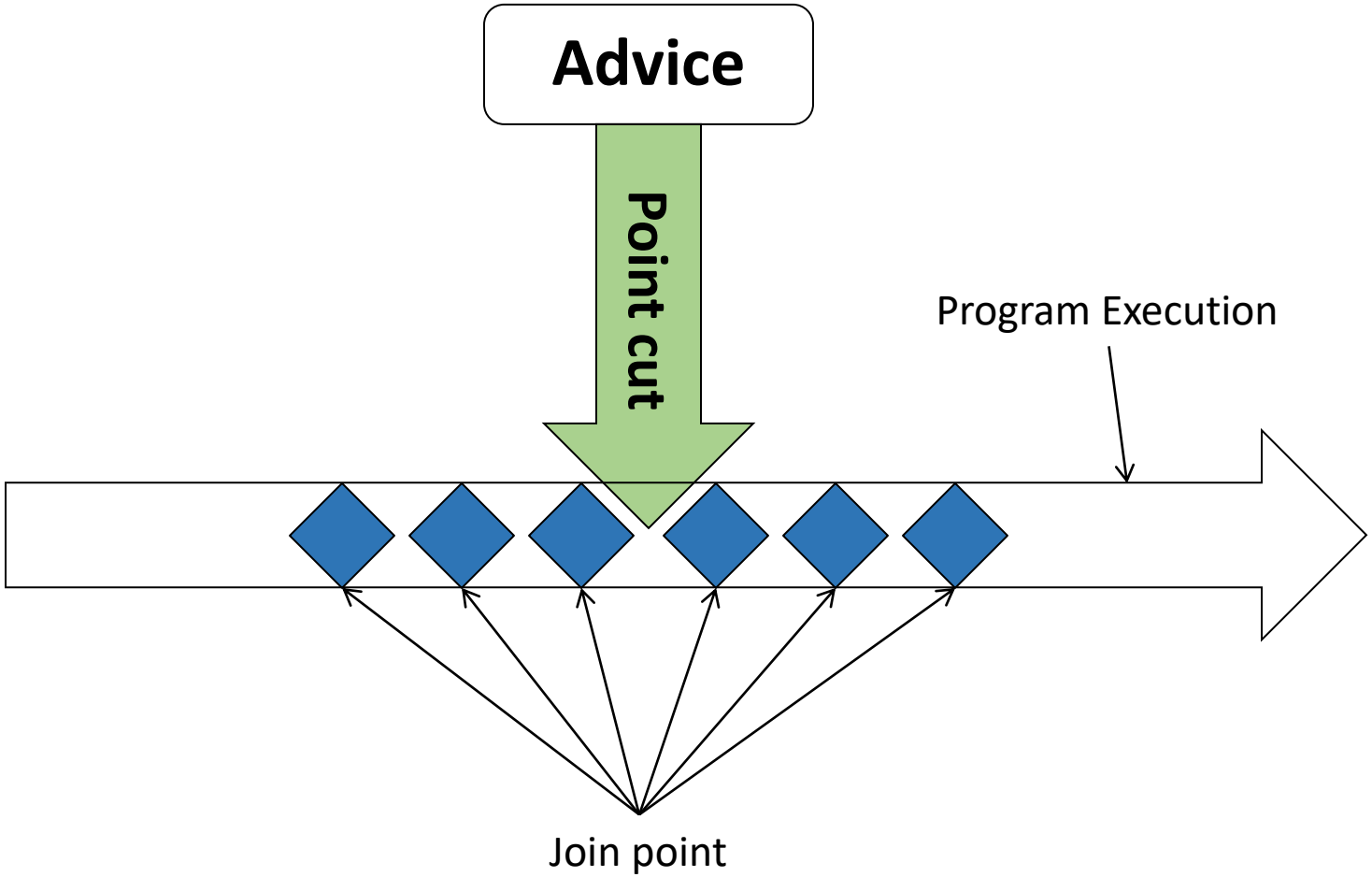
AOP Concepts (Aspect) (Ex.)

- 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.

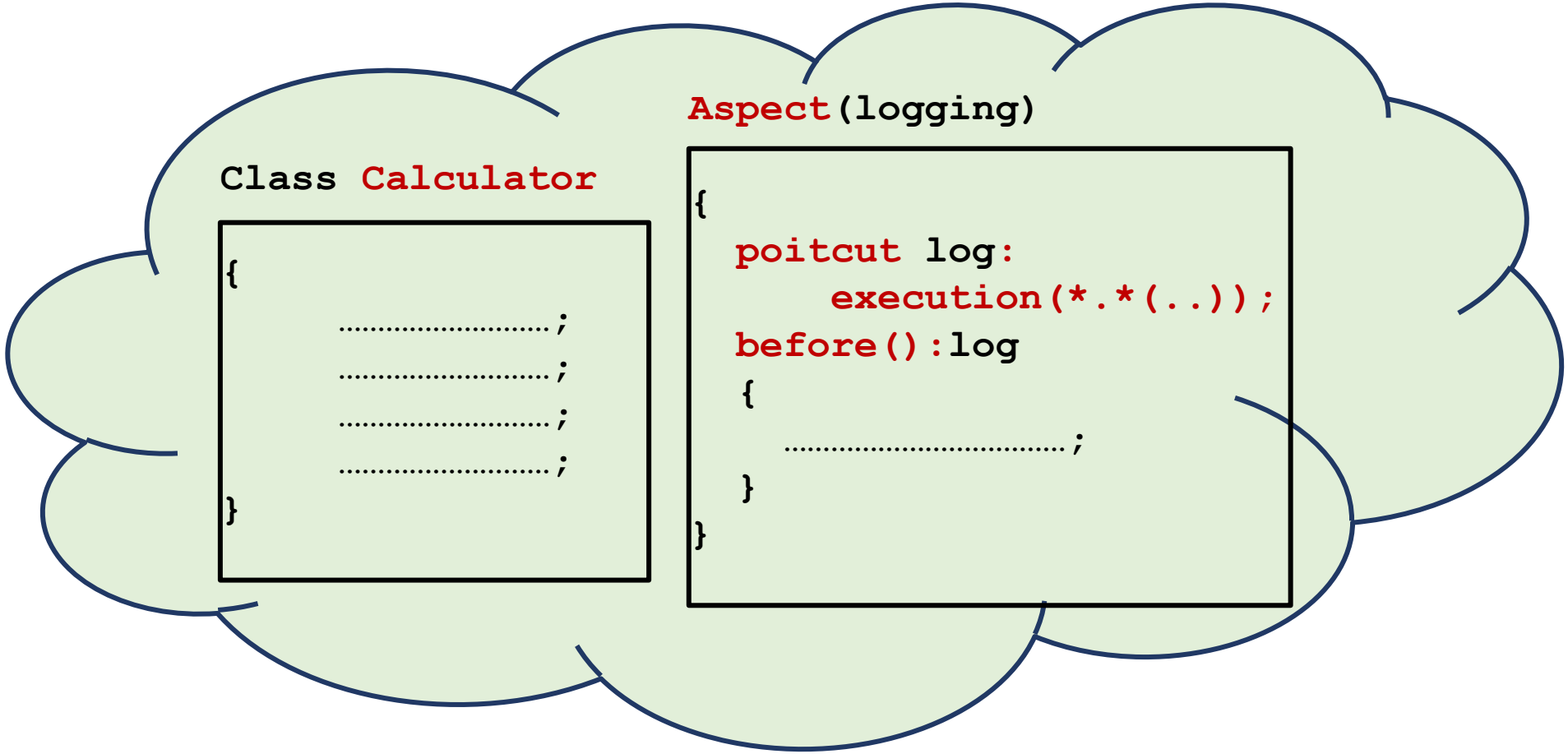
AOP Flow



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.

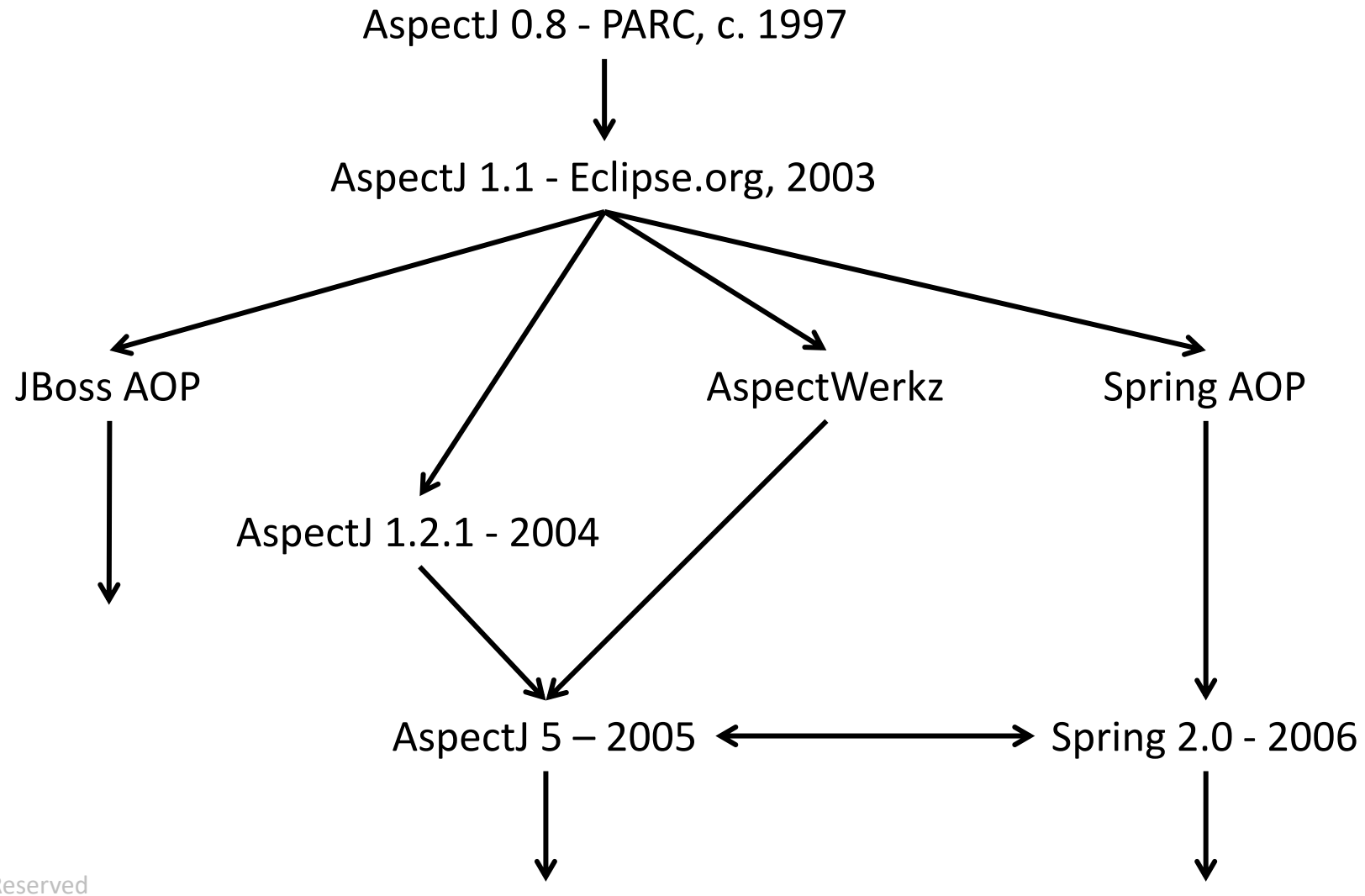
Lesson 3

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.)

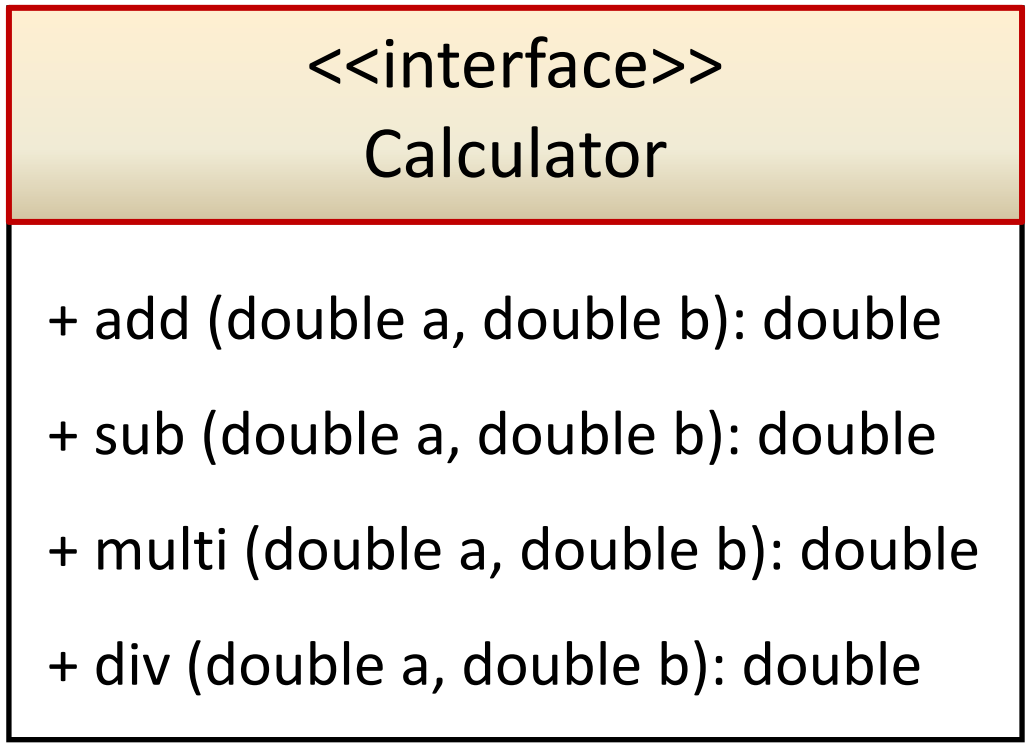


Lesson 4

Classic Spring proxy-based AOP

Java™ Education and Technology Services

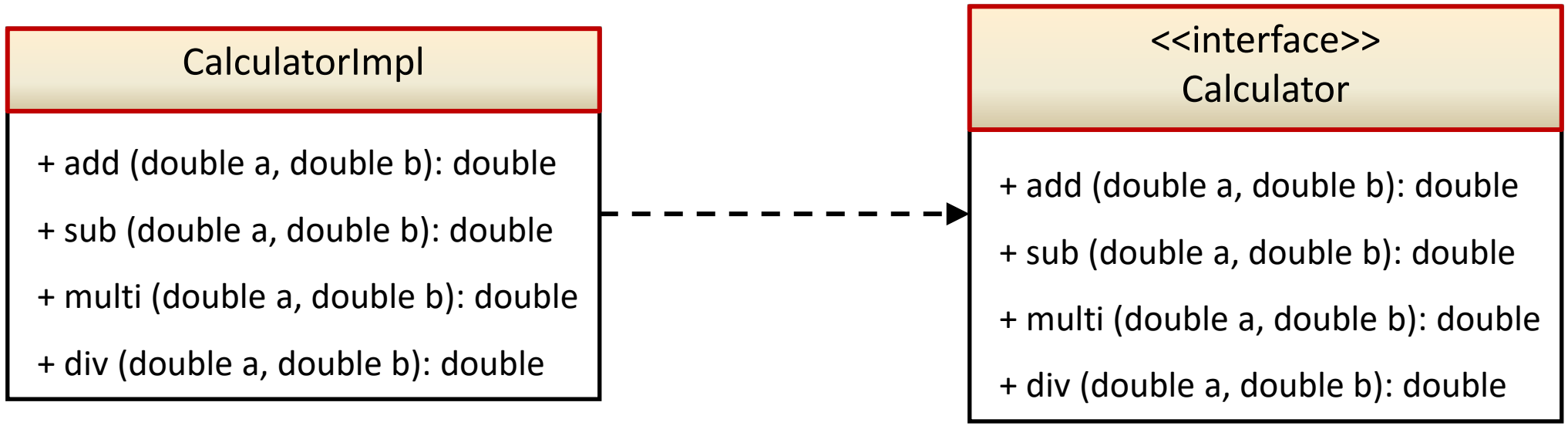
Calculator Case Service Interface (UML)



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)



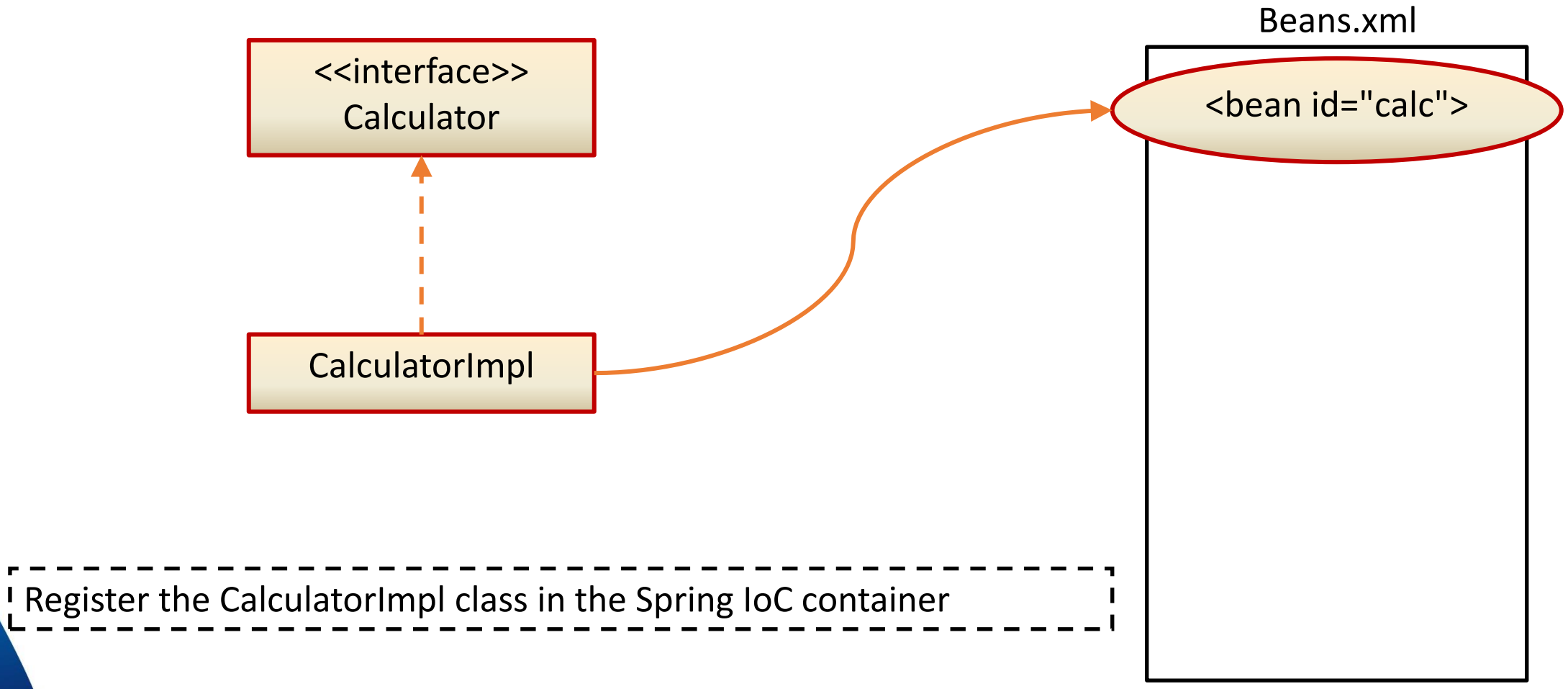
Calculator Case Service Class

```
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

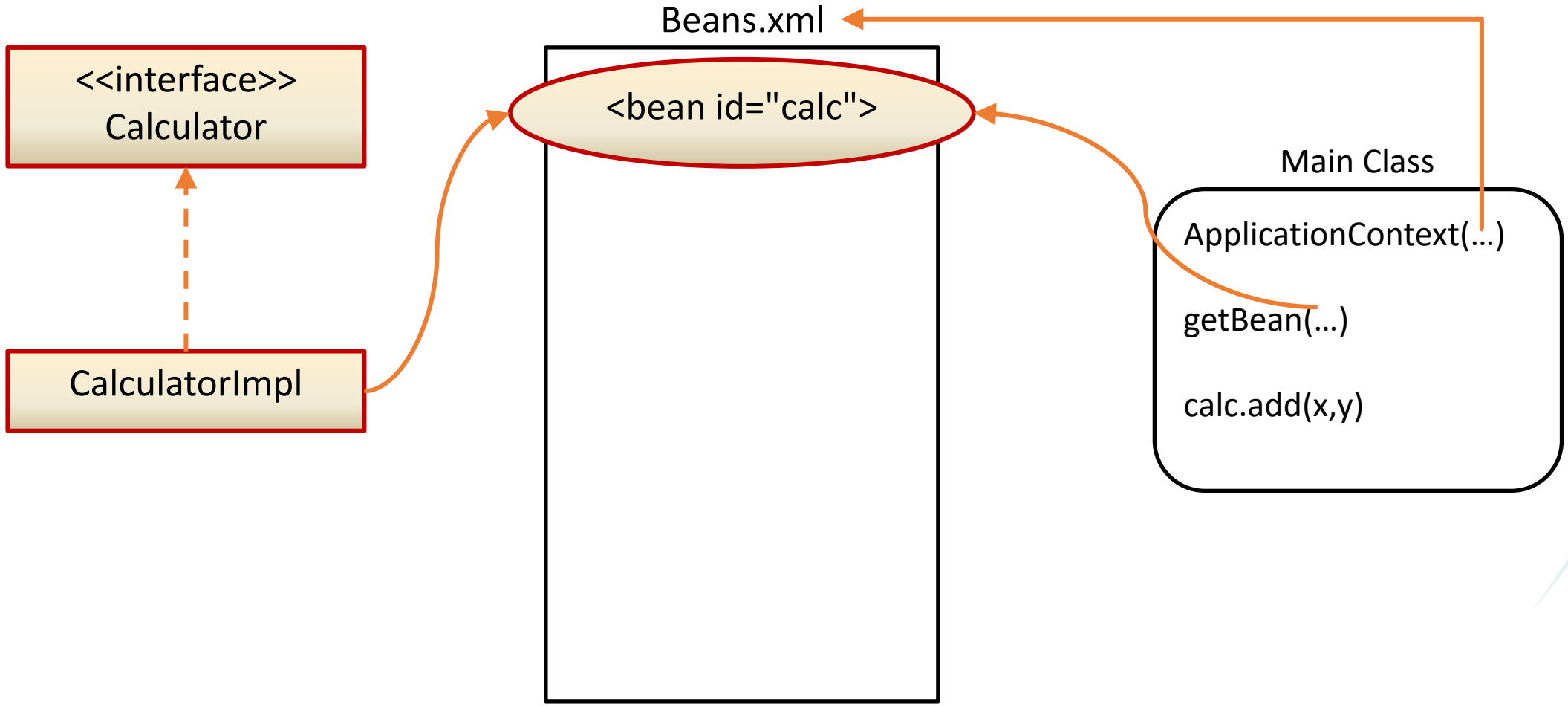


Calculator Case Workflow (Ex.)

- 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);
    }
}
```

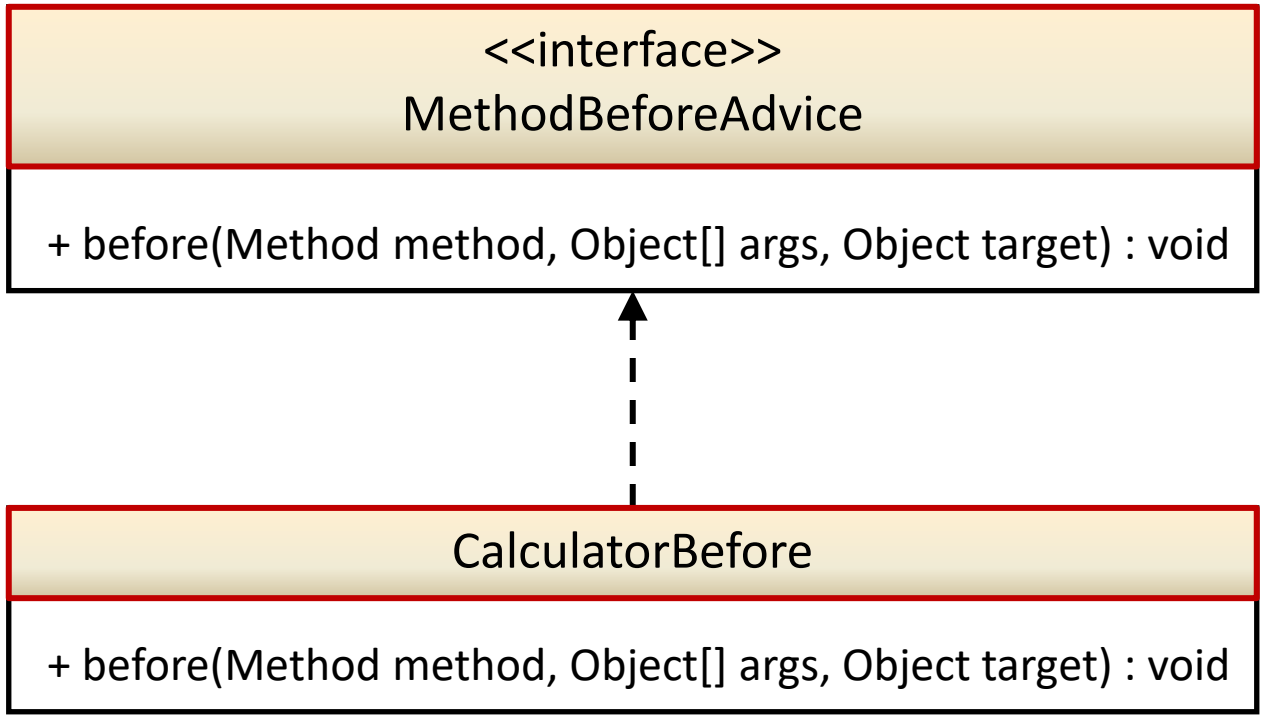
Classic Spring Advices (After throwing advice)

```
public class CalculatorAfterThrow implements ThrowsAdvice {

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

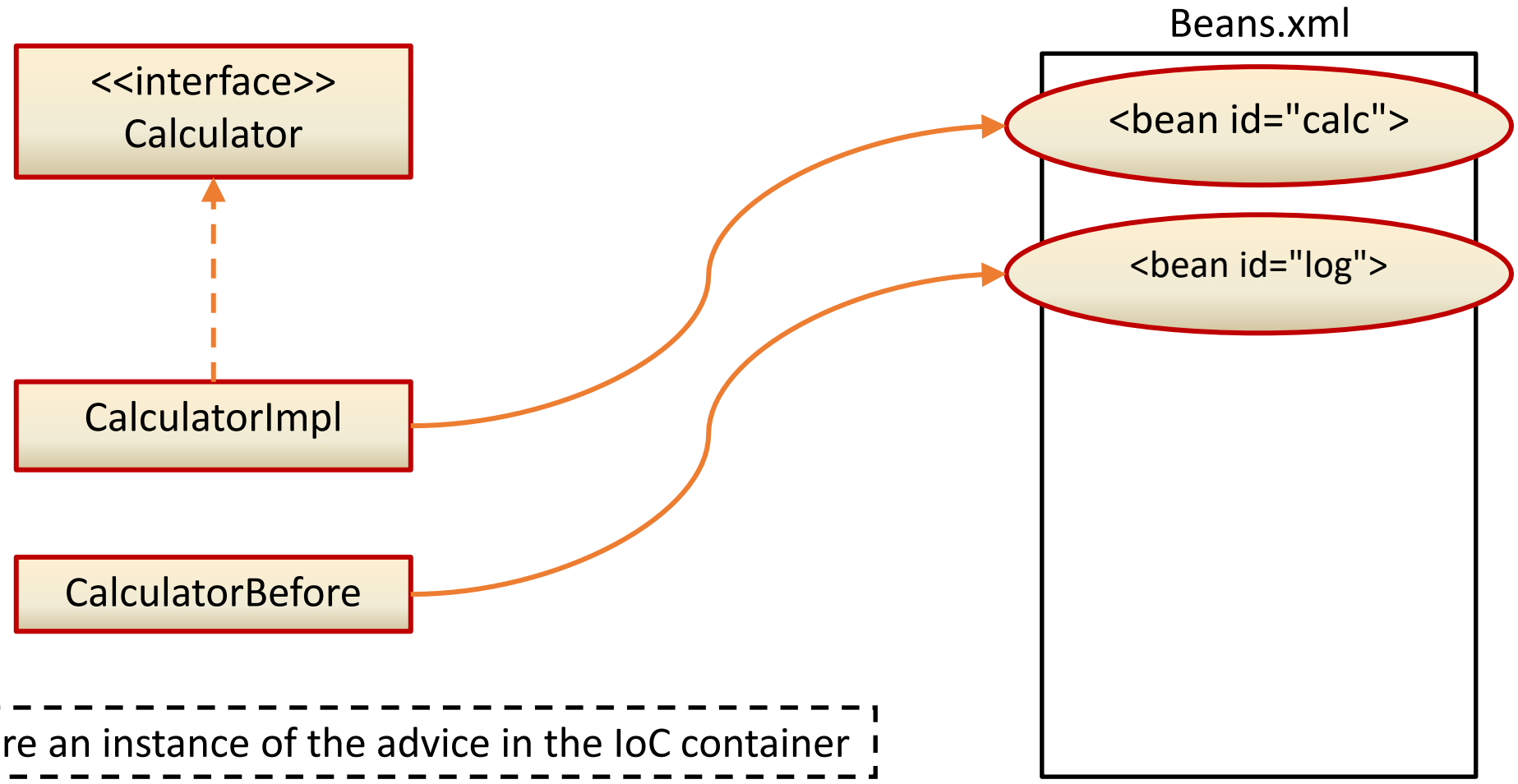
Classic Spring Advices (Around advice)

```
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;
    }
}
```

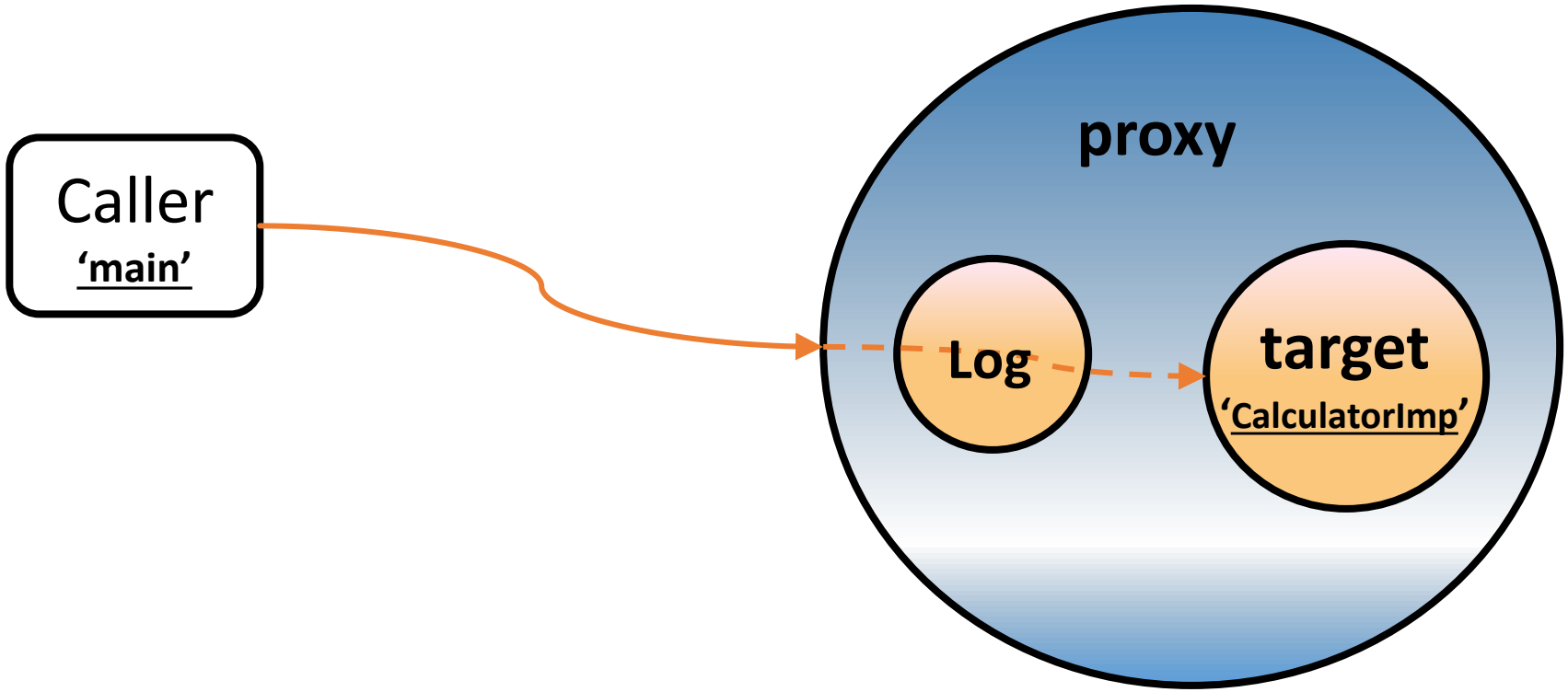
Calculator Case Service Class (UML)



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.)

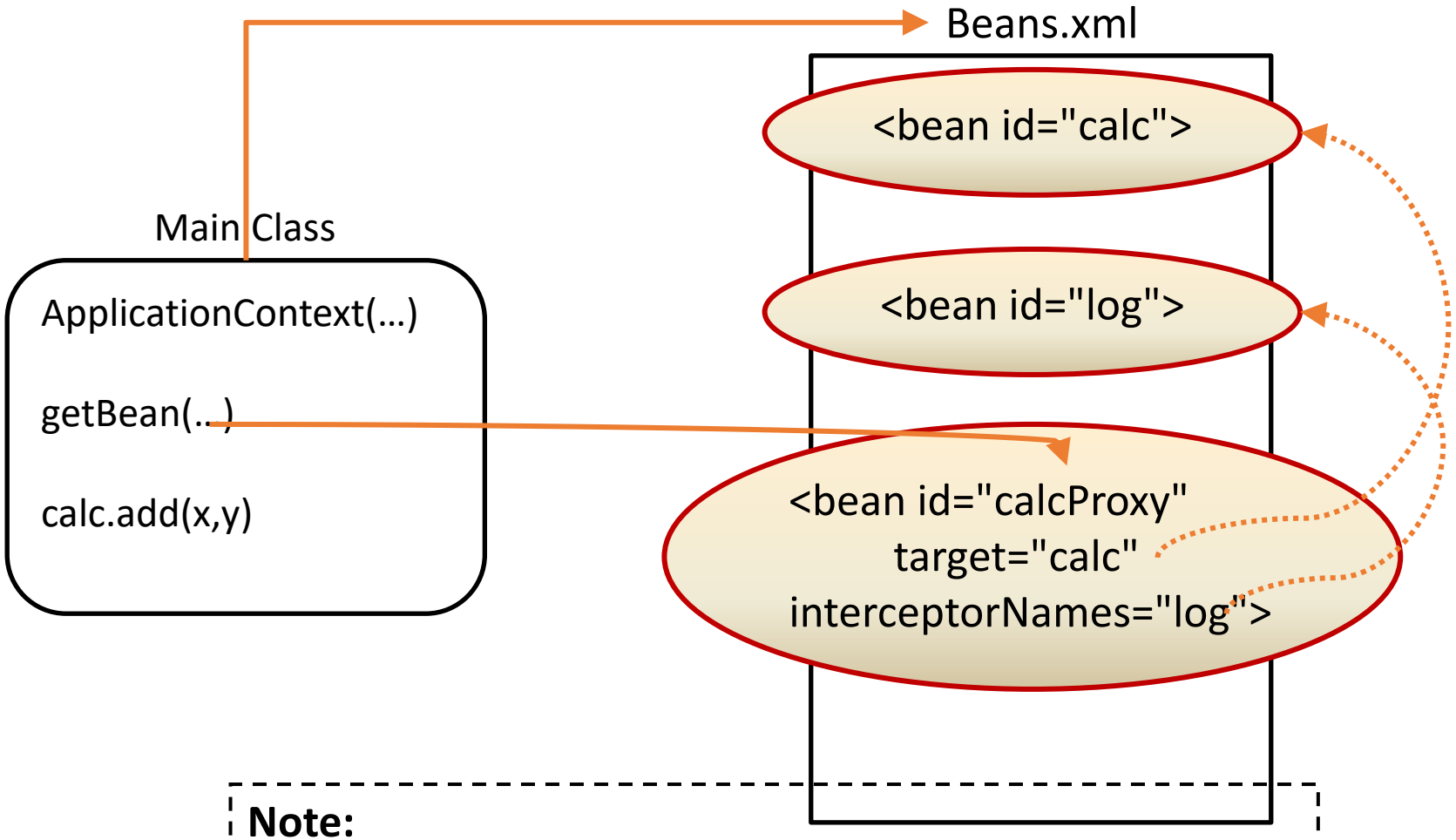
Java™ Education and Technology Services

- 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.)

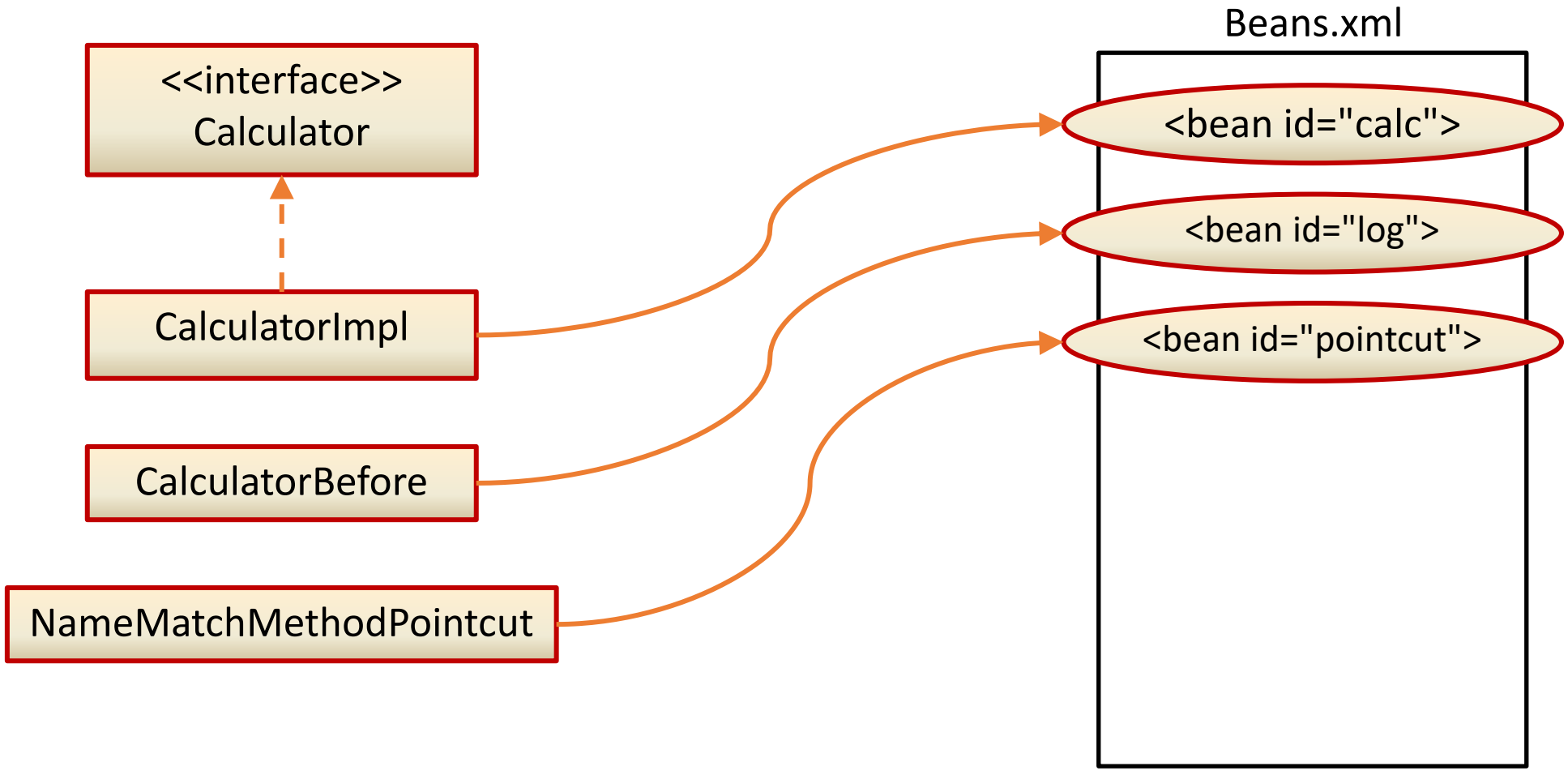
- In the Main class, you should get the proxy beans from the IoC container instead to have your logging advice applied.

```
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

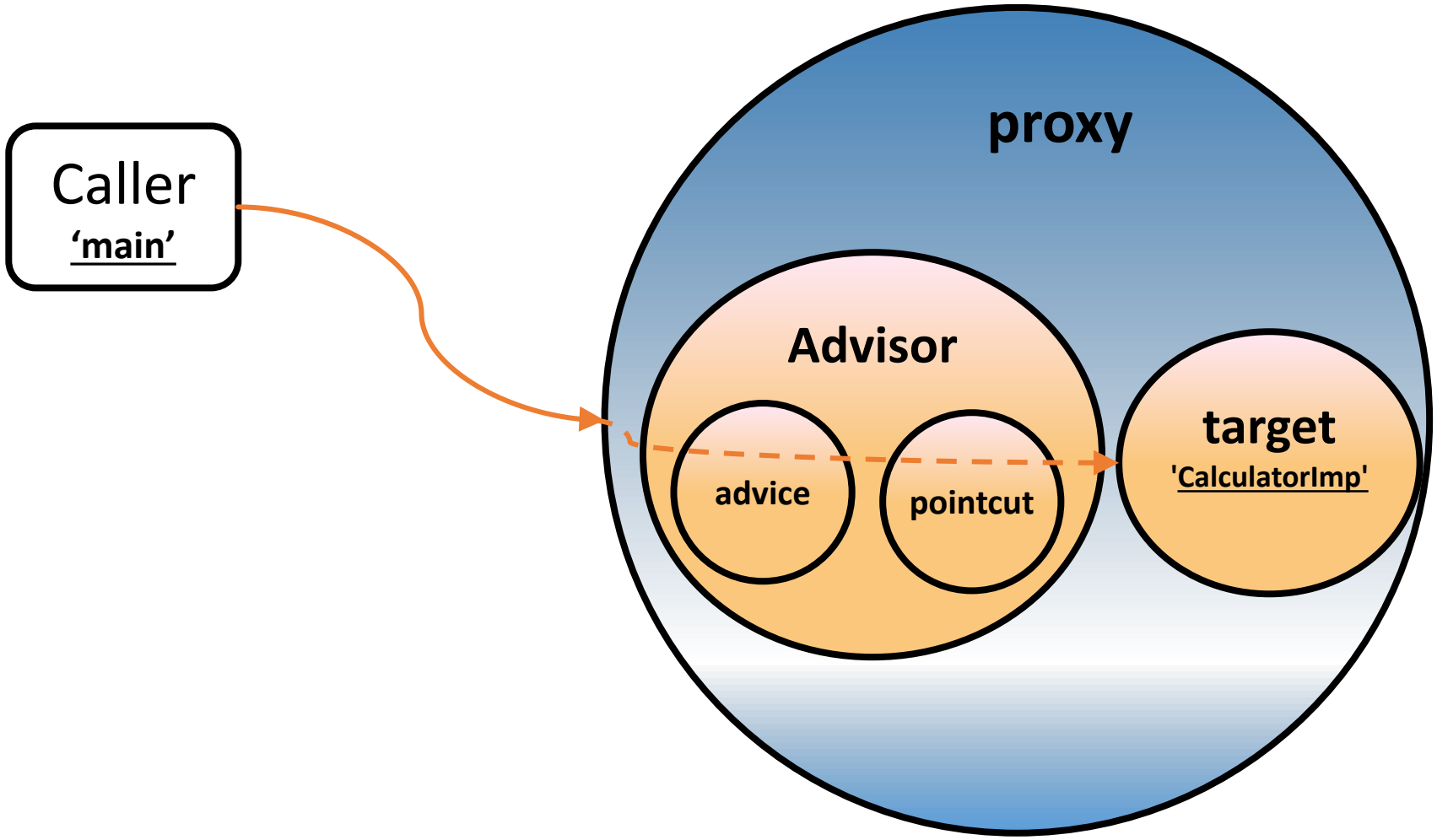
- To advise a single method only, you can use NameMatchMethodPointcut to match the method statically by its name.
- You can specify a particular method name or method name expression with wildcards in the mappedName property.

```
<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.

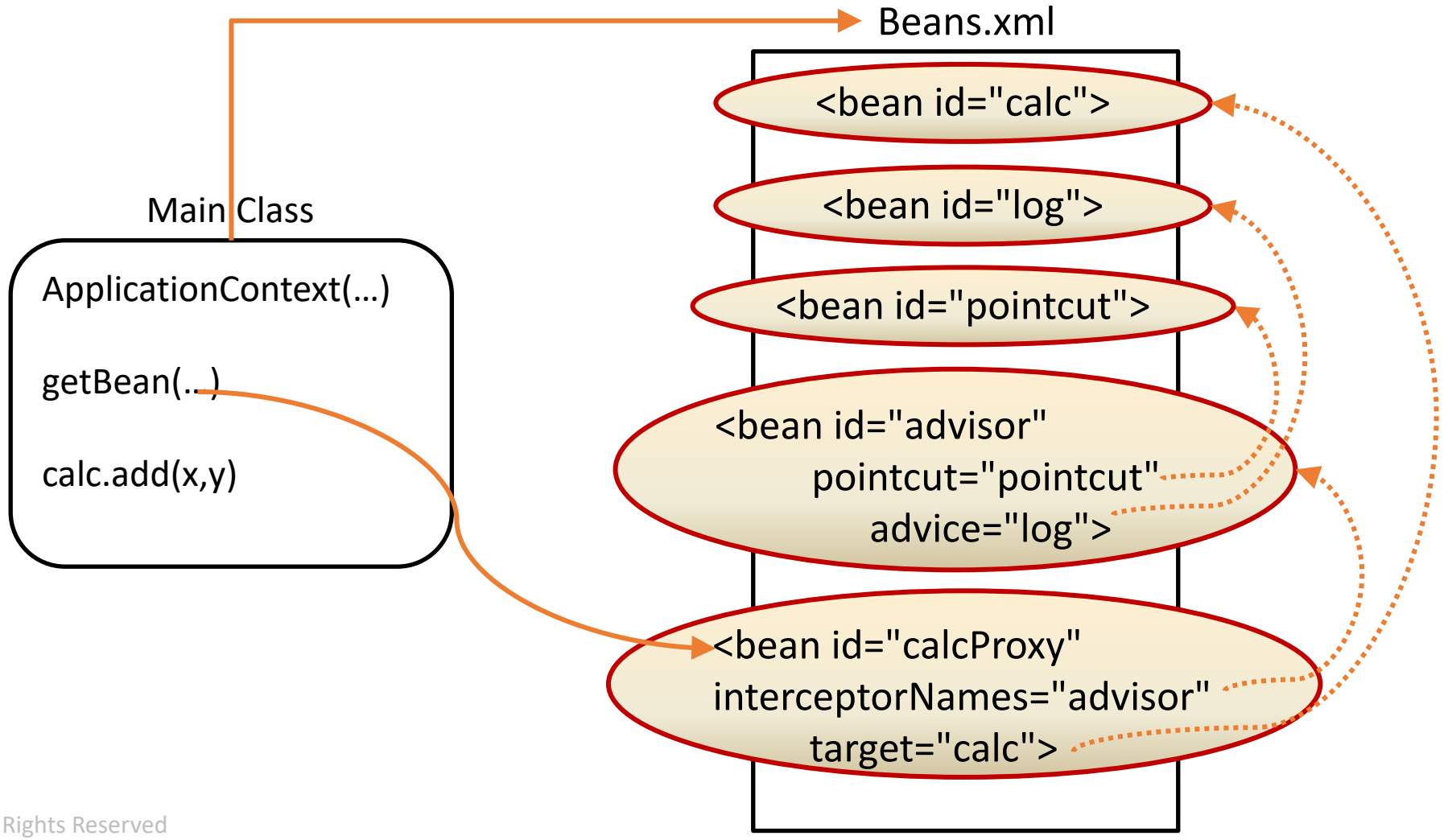
Calculator Case Workflow (Ex.)



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);
}
```