Information systems modeling

Tomasz Kubik

Aspect-oriented programming, AOP

Systems are composed of several components, each responsible for a specific piece of functionality. But often these components also carry additional responsibilities beyond their core functionality.

System services such as ***logging, transaction management, and security*** *often find their way into components whose core responsibilities is something else.* ***These system services are commonly referred to as cross-cutting concerns because they tend to cut across multiple components in a system****.*

AOP is a technique that promotes separation of concerns in a software system.

[based on *Spring in Action*]

Working with AOP

Standard way:

```
public void someMethod() {
    System.out.println("Entering method");
    // do something
    System.out.println("Leaving method");
}
```

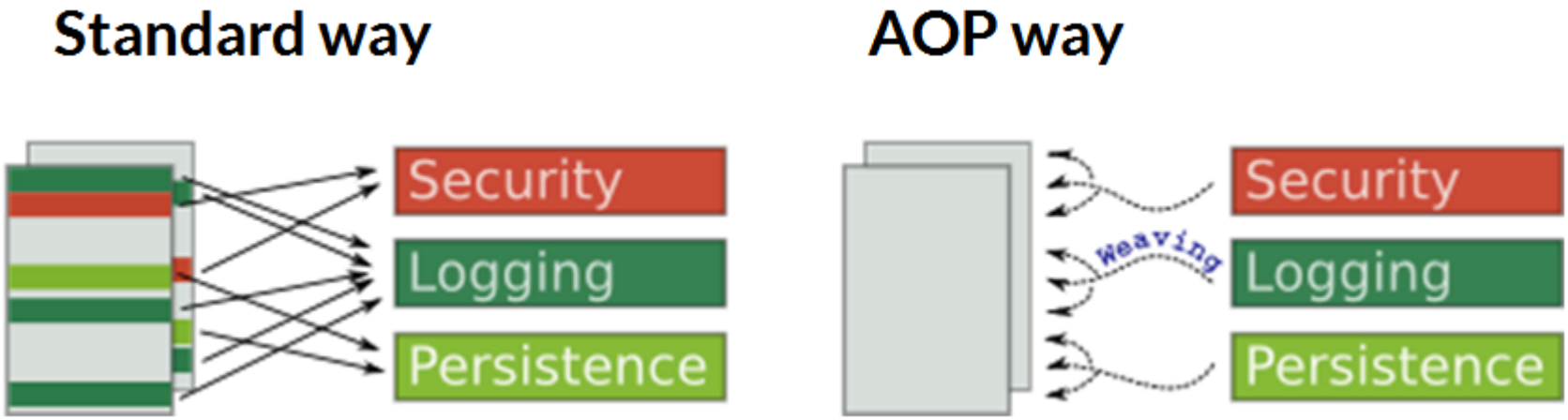
It would be nice to have a simple method without surrounding println:

```
public void someMethod() {
    // do something
}
```

and a method that will run everytime `someMethod()` is executed:

```
public void aroundSomeMethod(final ProceedingJoinPoint thisJoinPoint)
throws Throwable {
    System.out.println("Entering method");
    thisJoinPoint.proceed();
    System.out.println("Leaving method");
}
```

Working with AOP



http://www.christianschenk.org/blog/aop-with-aspectj/

AOP in brief

- It can help to modularize application for functionality that spans across multiple boundaries
- It encapsulates features and follows *Single Responsiblity* by moving cross-cutting concerns (logging, error handling, etc.) out of the main components
- When used appropriately AOP can lead to higher levels of maintainability and extensibility of software over time
- There are usually two ways of accomplishing AOP:
  - injecting code *automagically* by a preprocessor before/after a method,
  - attaching proxy classes that intercept a method call and can then execute things before/after a method call.

- In practice didn't become as useful as originally expected. It works well for injecting code modifications, like monitoring, debugging, and logging logic. However, other mechanisms were found to be "good enough" for addressing cross-cutting concerns.

https://stackoverflow.com/questions/242177/what-is-aspect-oriented-programming

https://deanwampler.github.io/aspectprogramming/

AOP vocabulary

- Aspect:
  - a modularization of a concern that cuts across multiple classes
  - there can be one or more aspects in an application
  - in Spring AOP aspects are implemented using regular classes (the schema-based approach) or regular classes annotated with the @Aspect annotation (the @AspectJ style).
- Join point:
  - a point where an aspect can be plugged in (i.e. a constructor's invocation, a method's execution or an exception management)
  - in Spring AOP a join point always represents a method execution
- Advice:
  - the action to be performed in the joinpoint
  - the types of advice include "around," "before" and "after" advice
  - many AOP frameworks, including Spring, model an advice as an interceptor, maintaining a chain of interceptors around the join point.
- Pointcut:
  - contains an expression to locate the joinpoint to which the advice will be applied
  - matching join points by pointcut expressions is a central concept to AOP,
  - Spring uses the AspectJ pointcut expression language by default.

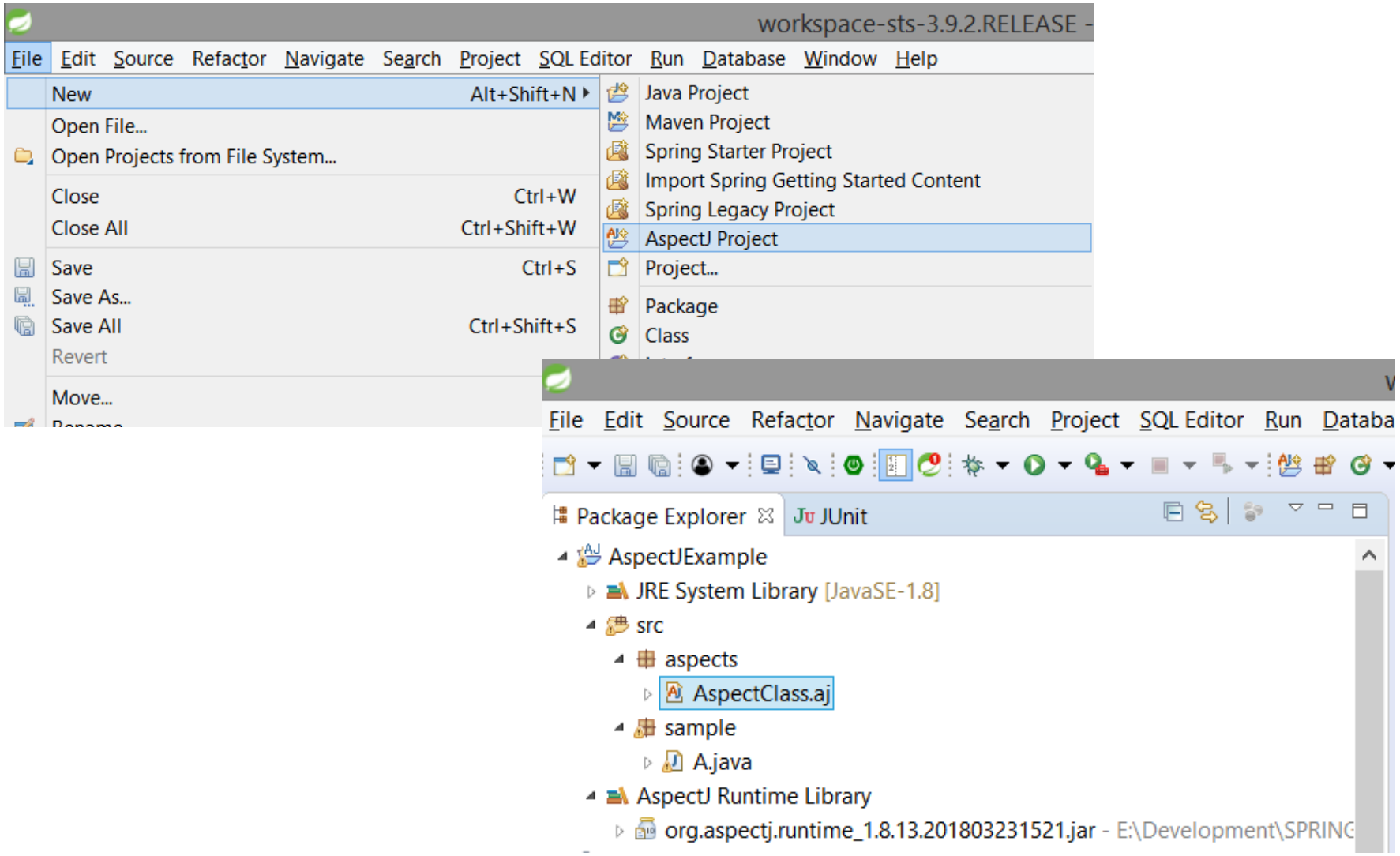
AOP vocabulary

- Introduction
  - used to declare additional methods and attributes for a particular type
  - Spring AOP allows introducing new interfaces (and a corresponding implementation) to any advised object.
  - known as inter-type declaration in the AspectJ community.
- Target object:
  - object being advised by one or more aspects (advised object).
  - Spring AOP is implemented using runtime proxies therefore this object will always be a proxied object.
- AOP proxy:
  - an object created by the AOP framework in order to implement the aspect contracts (advise method executions and so on).
  - In the Spring Framework, an AOP proxy will be a JDK dynamic proxy or a CGLIB proxy.
- Weaving:
  - linking aspects with other application types or objects to create an advised object
  - can be done at compile time (using the AspectJ compiler, for example), load time, or at runtime.
  - Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime.

Types of advices

- before
  - Run advice before the a method execution.
- after
  - Run advice after the method execution, regardless of its outcome.
- after-returning
  - Run advice after the a method execution only if method completes successfully.
- after-throwing
  - Run advice after the a method execution only if method exits by throwing an exception.
- around
  - Run advice before and after the advised method is invoked.

AspectJ – project in STS



AspecJ – example (aspect, pointcut, advice)

AspectJExample ▸ src ▸ aspects ▸ AspectClass ▸ before() : void

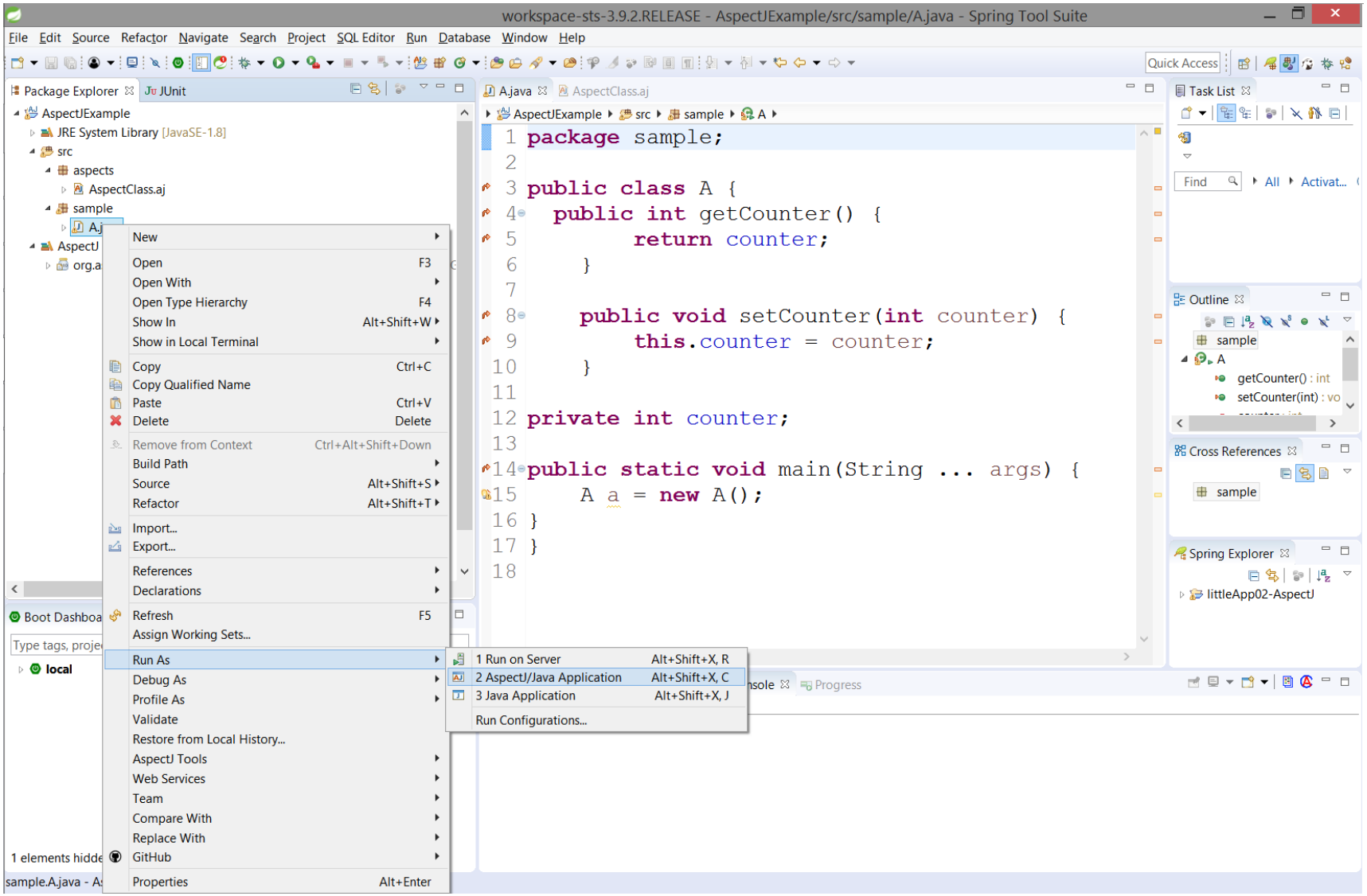
```
1  package aspects;
2  import sample.A;
3  public aspect AspectClass {
4
5      pointcut myClass(): within(A);
6
7      before(): myClass() {
8          System.out.println("" +
9            thisJoinPointStaticPart.getSignature());
10     }
11 }
```

AspectJ – example (advised class)

AspectJExample ▸ src ▸ sample ▸ A ▸

```
 1 package sample;
 2
 3 public class A {
 4   public int getCounter() {
 5         return counter;
 6     }
 7
 8     public void setCounter(int counter) {
 9         this.counter = counter;
10     }
11
12 private int counter;
13
14 public static void main(String ... args) {
15     A a = new A();
16 }
17 }
```

AspectJ – example (running app)



Declaring Aspect by annotation (@AspectJ)

```
package org.xyz;

import org.aspectj.lang.annotation.Aspect;

@Aspect
public class AspectModule {
...
}
```

https://www.tutorialspoint.com/spring/aspectj_based_aop_appoach.htm

Declaring pointcut by annotation (somwhere inside Aspect)

```
import org.aspectj.lang.annotation.Pointcut;

@Pointcut("execution(*
com.xyz.myapp.service.*.*(..))") // expression
private void businessService() {}  // signature
```

This pointcut was named '`businessService`' and will match the execution of every method available in the classes under the package `com.xyz.myapp.service`

https://www.tutorialspoint.com/spring/aspectj_based_aop_appoach.htm

Declaring advices (inside Aspect)

```
@Before("businessService()")
public void doBeforeTask(){
    ...
}

@After("businessService()")
public void doAfterTask(){
    ...
}
```

Assuming that a pointcut signature method businessService() have been already defined

```
@AfterReturning(pointcut = "businessService()", returning = "retVal")
public void doAfterReturnningTask(Object retVal) {
    // you can intercept retVal here.
    ...
}

@AfterThrowing(pointcut = "businessService()", throwing = "ex")
public void doAfterThrowingTask(Exception ex) {
   // you can intercept thrown exception here.
   ...
}

@Around("businessService()")
public void doAroundTask(){
    ...
}
```

https://www.tutorialspoint.com/spring/aspectj_based_aop_appoach.htm

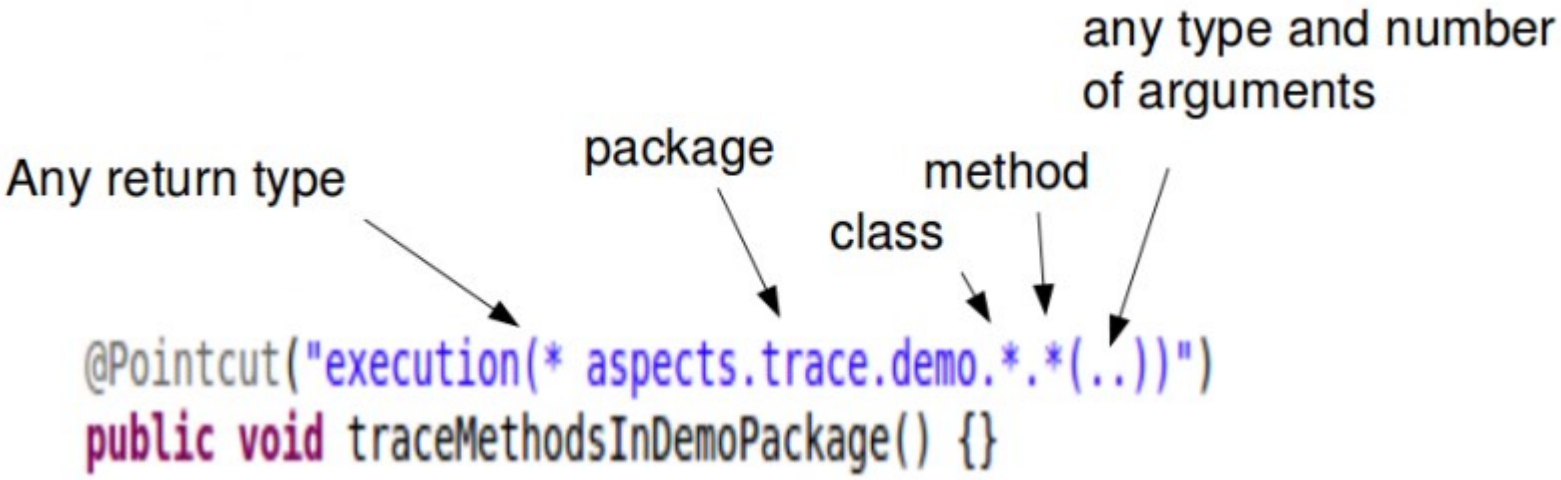
Declaring inline pointcuts

```
@Before("execution(* com.xyz.myapp.service.*.*(..))")
public doBeforeTask(){
   ...
}
```

An inline pointcut can be defined for any of the advices.

https://www.tutorialspoint.com/spring/aspectj_based_aop_appoach.htm

Pointcut definition



https://blog.espenberntsen.net/2010/03/20/aspectj-cheat-sheet/

Pointcut designators

- A method pointcut:

```
@Pointcut("[method designator](* aspects.trace.demo.*.*(..))")
public void traceMethodsInDemoPackage() {}
```

  - `call` – The pointcut will find all methods that calls a method in the demo package.
  - `execution` – The pointcut will find all methods in the demo package.
  - `withincode` – All the statements inside the methods in the demo package.

- A type pointcut:

```
@Pointcut("[type designator](*..*Test)")
public void inTestClass() {}
```

  - `within` – all statements inside the a class that ends with Test.

- A field pointcut:

```
@Pointcut("[field designator](private
org.springframework.jdbc.core.JdbcTemplate " +
        "integration.db.*.jdbcTemplate)")
public void jdbcTemplateGetField() {}
```

  - `get` – all reads to jdbcTemplate fields of type JdbcTemplate in the integration.db package. Includes all methods on this field if it's an object.
  - `set` – when you set the jdbcTemplate field of type JdbcTemplate in the integration.db package to a new value.

Spring AOP supported Pointcut Designators

- *execution* - for matching method execution join points, this is the primary pointcut designator you will use when working with Spring AOP
- *within* - limits matching to join points within certain types (simply the execution of a method declared within a matching type when using Spring AOP)
- *this* - limits matching to join points (the execution of methods when using Spring AOP) where the bean reference (Spring AOP proxy) is an instance of the given type
- *target* - limits matching to join points (the execution of methods when using Spring AOP) where the target object (application object being proxied) is an instance of the given type
- *args* - limits matching to join points (the execution of methods when using Spring AOP) where the arguments are instances of the given types
- *@target* - limits matching to join points (the execution of methods when using Spring AOP) where the class of the executing object has an annotation of the given type
- *@args* - limits matching to join points (the execution of methods when using Spring AOP) where the runtime type of the actual arguments passed have annotations of the given type(s)
- *@within* - limits matching to join points within types that have the given annotation (the execution of methods declared in types with the given annotation when using Spring AOP)
- *@annotation* - limits matching to join points where the subject of the join point (method being executed in Spring AOP) has the given annotation

https://docs.spring.io/spring/docs/4.3.14.RELEASE/spring-framework-reference/html/aop.html

Spring AOP vs AspectJ

summary of supported joinpoints:

Joinpoint	Spring AOP Supported	AspectJ Supported
Method Call	No	Yes
Method Execution	Yes	Yes
Constructor Call	No	Yes
Constructor Execution	No	Yes
Static initializer execution	No	Yes
Object initialization	No	Yes
Field reference	No	Yes
Field assignment	No	Yes
Handler execution	No	Yes
Advice execution	No	Yes

It’s also worth noting that in Spring AOP, aspects aren’t applied to the method called within the same class.

http://www.baeldung.com/spring-aop-vs-aspectj

Spring AOP vs AspectJ (summary)

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

http://www.baeldung.com/spring-aop-vs-aspectj

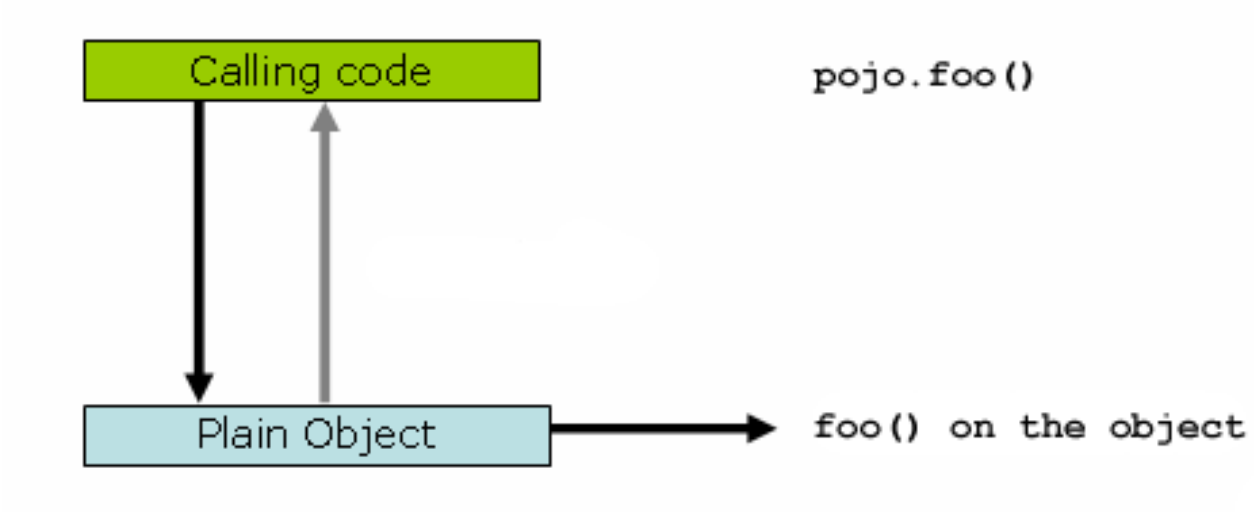
Remarks

- The full AspectJ pointcut language supports additional pointcut designators that are not supported in Spring. These are:
  - `call, get, set, preinitialization, staticinitialization, initialization, handler, adviceexecution, withincode, cflow, cflowbelow, if, @this, @withincode`
- Because Spring AOP limits matching to **only method execution join points**, the discussion of the pointcut designators above gives a narrower definition than you will find in the AspectJ programming guide.
- In addition, AspectJ itself has type-based semantics and at an execution join point both `this` and `target` refer to the same object - the object executing the method. **Spring AOP is a proxy-based system** and differentiates between the proxy object itself (bound to `this`) and the target object behind the proxy (bound to `target`).

https://docs.spring.io/spring/docs/4.3.14.RELEASE/spring-framework-reference/html/aop.html

Understanding AOP proxies

```
public class SimplePojo implements Pojo {

    public void foo() {
        // this next method invocation is a direct call on the 'this' reference
        this.bar();
    }

    public void bar() {
        // some logic...
    }
}
```

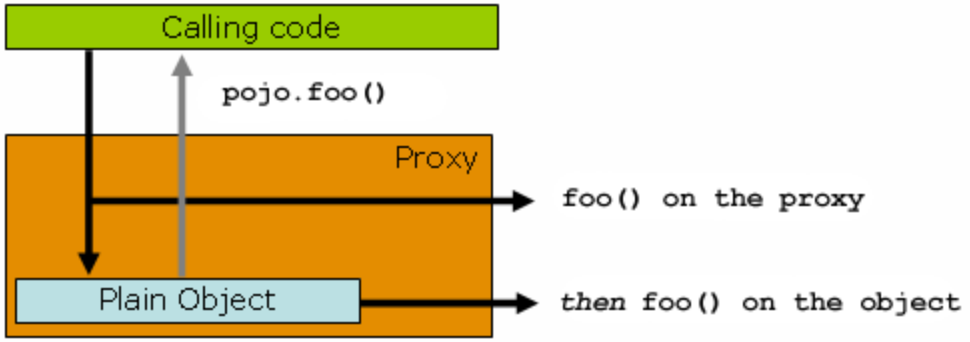



```
public class Main {

    public static void main(String[] args) {

        Pojo pojo = new SimplePojo();

        // this is a direct method call on the 'pojo' reference
        pojo.foo();
    }
}
```

https://docs.spring.io/spring/docs/3.0.0.M4/reference/html/ch07s06.html

Understanding AOP proxies

```
public class Main {

   public static void main(String[] args) {

      ProxyFactory factory = new ProxyFactory(new SimplePojo());
      factory.addInterface(Pojo.class);
      factory.addAdvice(new RetryAdvice());

      Pojo pojo = (Pojo) factory.getProxy();

      // this is a method call on the proxy!
      pojo.foo();
   }
}
```



https://docs.spring.io/spring/docs/3.0.0.M4/reference/html/ch07s06.html

Aspect vs AspectJ

AspectAnnotated.java

```java
package aspects;
import org.aspectj.lang.JoinPoint;

@Aspect
public class AspectAnnotated {

    @Pointcut(" execution (* models.A.get*(..))")
    public void pointcutGet() {
    }

    @Before("pointcutGet()")
    public void adviceBefore(JoinPoint thisJoinPoint){
        System.out.println("Before get function " +
                thisJoinPoint.getSignature().toString() +
                "; joinPoint kind: " +
                thisJoinPoint.getKind());
    }

    @AfterReturning(pointcut="pointcutGet()", returning ="counter")
    public void adviceAfter(JoinPoint thisJoinPoint, int counter){
        System.out.println("After get function " +
                thisJoinPoint.getSignature().toString() +
                "; joinPoint kind: " +
                thisJoinPoint.getKind() +
                "; counter as was get: " +
                counter);
    }
}
```

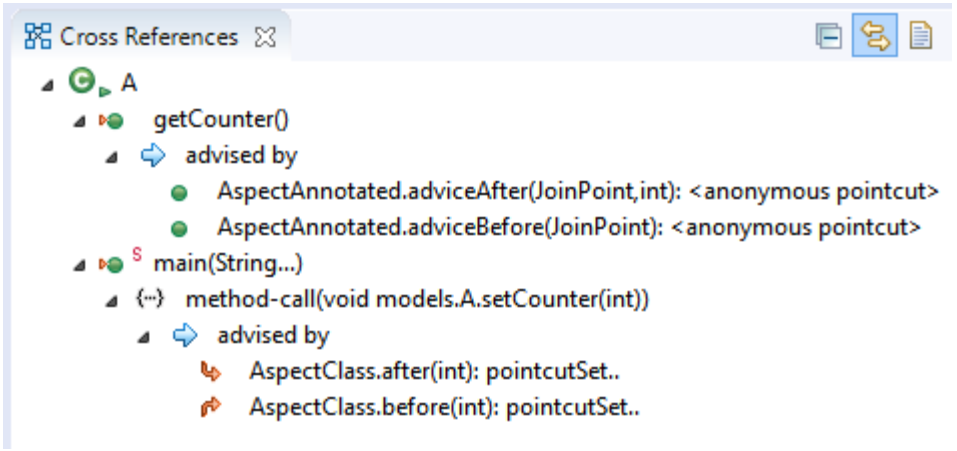
AspectClass.aj

```
package aspects;

public aspect AspectClass {

    pointcut pointcutSet(int counter):  call(* models.A.set*(..))
    && args(counter);

    before(int counter): pointcutSet(counter) {
        System.out.println("Before set function: " +
          thisJoinPointStaticPart.getSignature() +
          "; joinPoint kind: " +
          thisJoinPointStaticPart.getKind() +
          " counter to be set: " +
          counter);
    }

    after(int counter): pointcutSet(counter) {
        System.out.println("After set function: " +
                  thisJoinPointStaticPart.getSignature() +
                  "; joinPoint kind: " +
                  thisJoinPointStaticPart.getKind());
    }
}
```

AJDT support in eclipse

AspectAnnotated.java | AspectClass.aj | A.java

```
1  package models;
2
3  public class A {
4
5      public int getCounter() {
6          return counter;
7      }
8
9      public void setCounter(int counter) {
10         this.counter = counter;
11     }
12
13     private int counter = 0;
14
15     public static void main(String... args) {
16         A a = new A();
17         a.setCounter(10);
18         int counter = a.getCounter();
19         System.out.println(counter);
20     }
21 }
```



Resources

- AspectJ Development Tools plugin to eclipse

  https://download.eclipse.org/tools/ajdt/aspectj/update/?d
  https://download.eclipse.org/tools/ajdt/48/dev/update/

  https://www.eclipse.org/aspectj/doc/next/progguide/starting-production.html
- Tutorials

  https://dzone.com/articles/implementing-aop-with-spring-boot-and-aspectj
  https://www.tutorialspoint.com/spring/aop_with_spring.htm
  http://www.springboottutorial.com/spring-boot-and-aop-with-spring-boot-starter-aop
  https://marcin-chwedczuk.github.io/overview-of-spring-annotation-driven-aop
  http://data.christianschenk.org/logging-with-aspectj/xref/index.html
  http://www.baeldung.com/aspectj
- Explanation of pointcuts and aspects syntax

  https://docs.spring.io/spring/docs/5.0.x/spring-framework-reference/core.html#aop
- Explanation of weaving

  https://www.credera.com/blog/technology-insights/open-source-technology-insights/aspect-oriented-programming-in-spring-boot-part-3-setting-up-aspectj-load-time-weaving/
- Example of AspectJ+Spring Boot with weaving

  https://github.com/dsyer/spring-boot-aspectj
- Differences AOP vs AspectJ

  http://www.baeldung.com/spring-aop-vs-aspectj
  http://perfspy.blogspot.com/2013/09/differences-between-aspectj-call-and.html
- Dependency injection

  https://docs.jboss.org/weld/reference/latest/en-US/html/injection.html
  http://buraktas.com/java-cdi-dependency-injection-example/