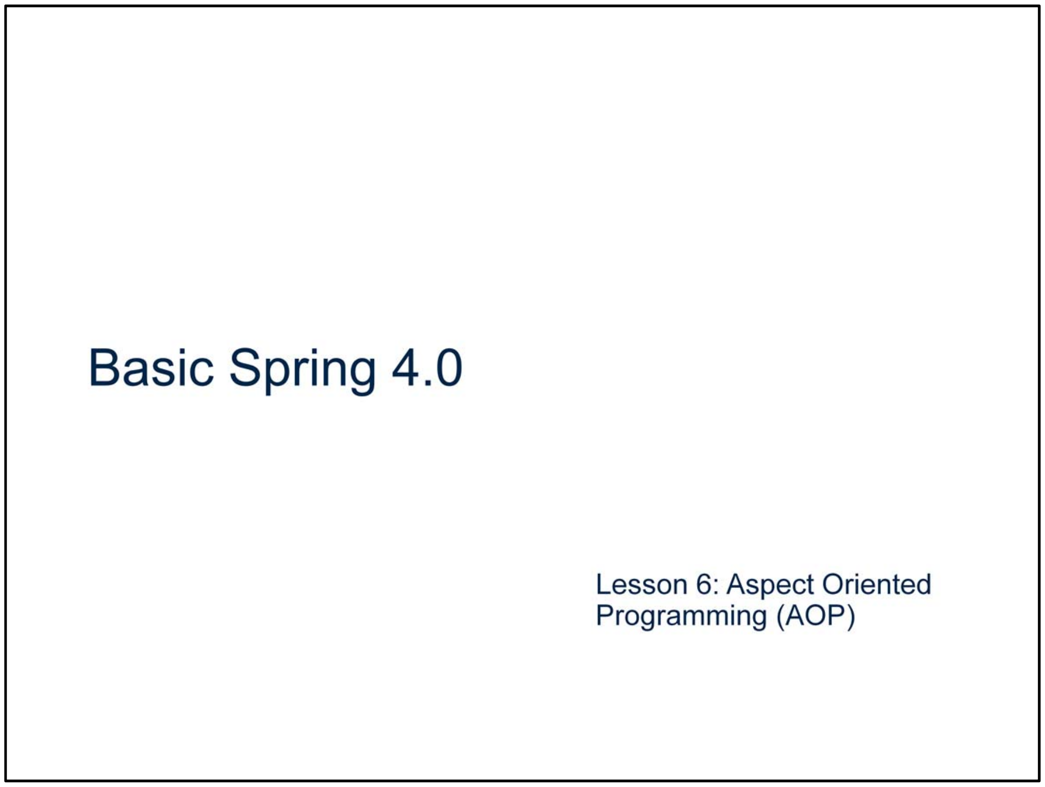
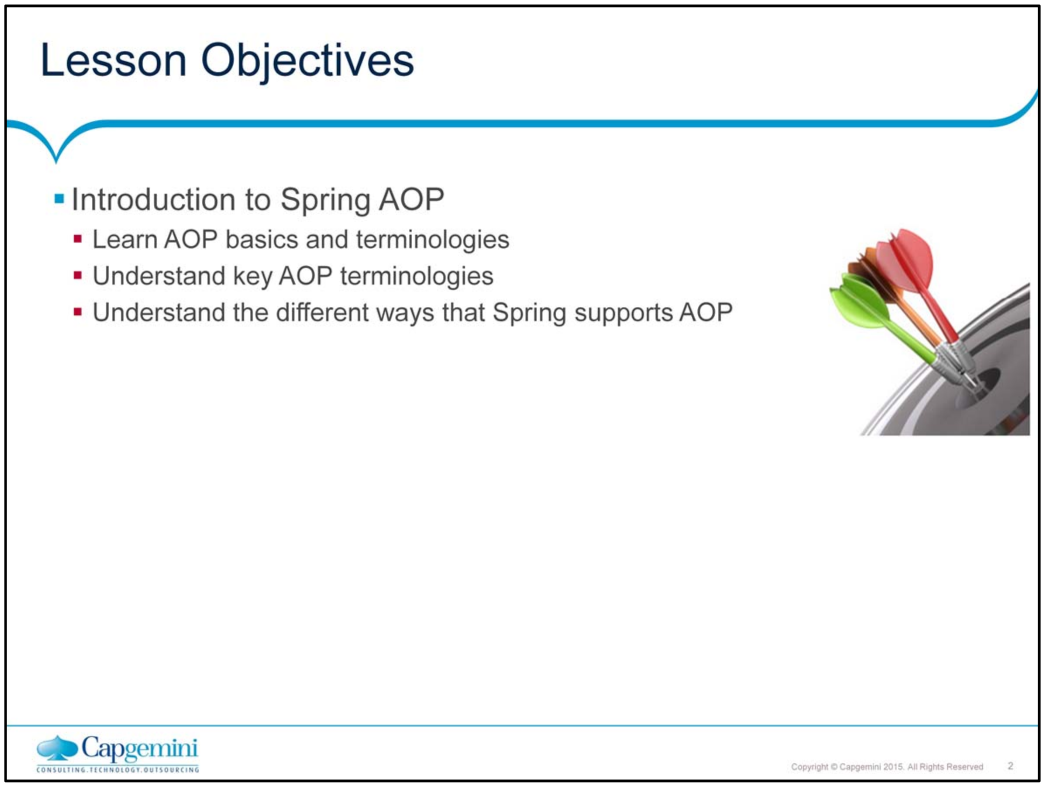


Basic Spring 4.0

Lesson 6: Aspect Oriented Programming (AOP)



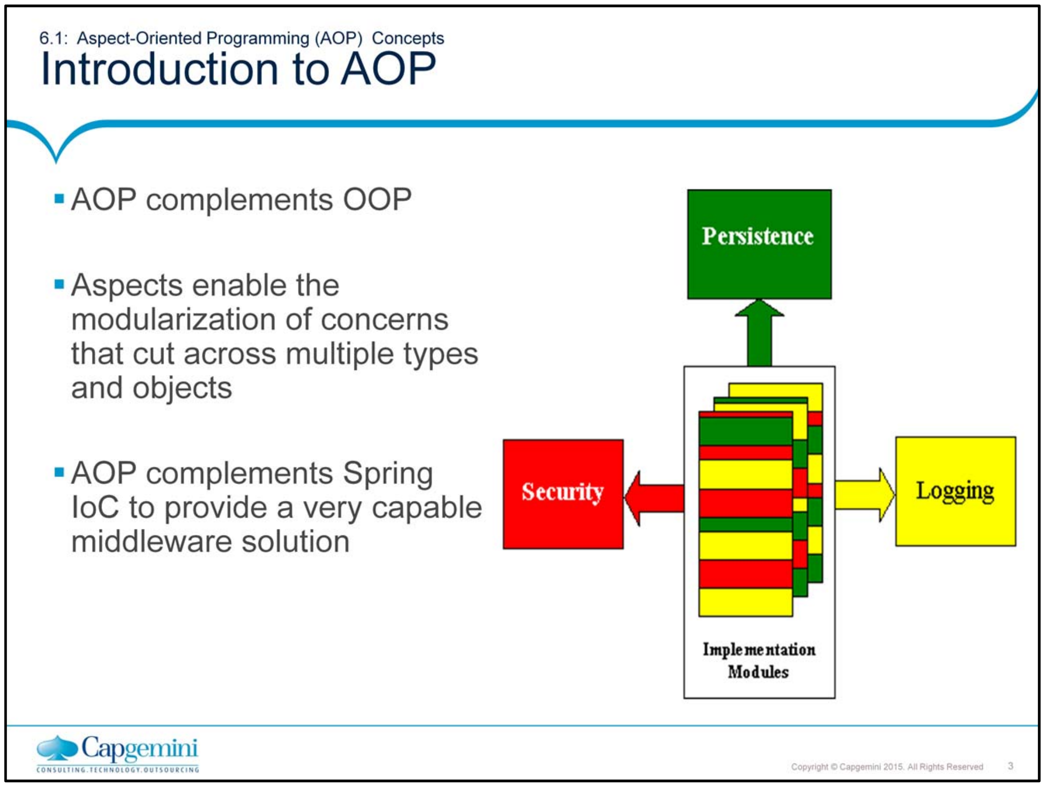
Lesson Objectives

- Introduction to Spring AOP
  - Learn AOP basics and terminologies
  - Understand key AOP terminologies
  - Understand the different ways that Spring supports AOP

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Topics to be covered:

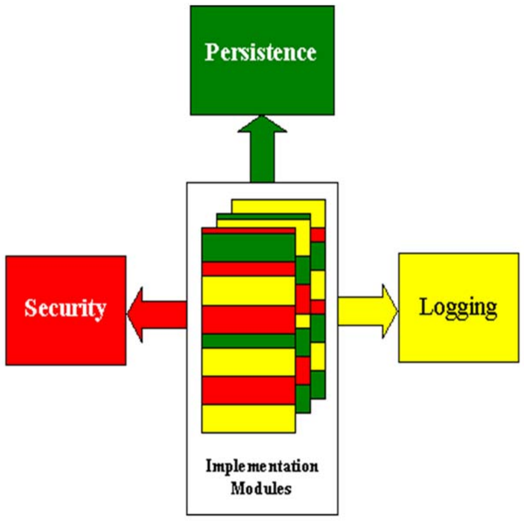
Introduction to AOP
AOP concepts
AOP support in Spring using @AspectJ support
AOP support in Spring using Schema-based AOP support



6.1: Aspect-Oriented Programming (AOP) Concepts

Introduction to AOP

- AOP complements OOP
- Aspects enable the modularization of concerns that cut across multiple types and objects
- AOP complements Spring IoC to provide a very capable middleware solution

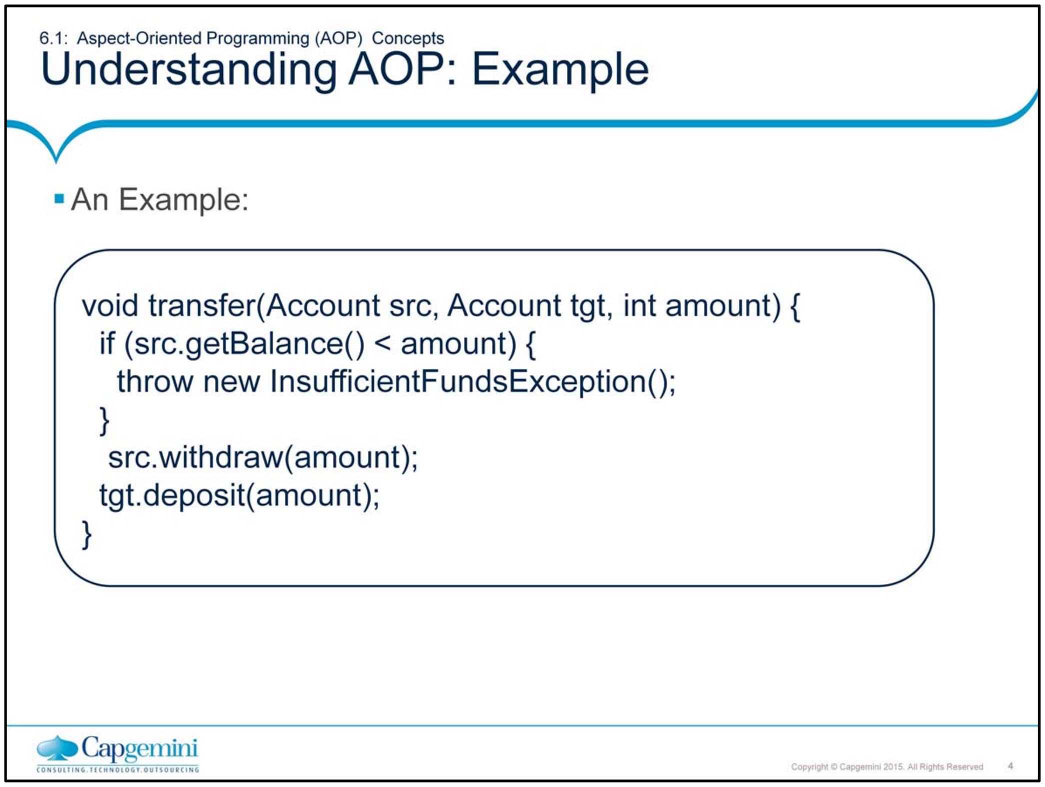


Aspect-Oriented Programming (AOP) complements Object-Oriented Programming (OOP) by providing another way of thinking about program structure. OOP decomposes applications into a hierarchy of objects; AOP decomposes prg's into aspects and concerns. This allows modularization of concerns such as transaction management that would otherwise cut across multiple objects. Such concerns are often called cross-cutting concerns. 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.).

There are two distinct types of AOP: static and dynamic. In static AOP, like AspectJ's AOP (http://eclipse.org/aspectj/), the crosscutting logic is applied to your code at compile time and you cannot change it without modifying the code and recompiling. With dynamic AOP like Spring's AOP, crosscutting logic is applied dynamically at run time. This allows you to make changes in the distribution of cross-cutting without recompiling the application.

The AOP framework is one of the key components of Spring. 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.



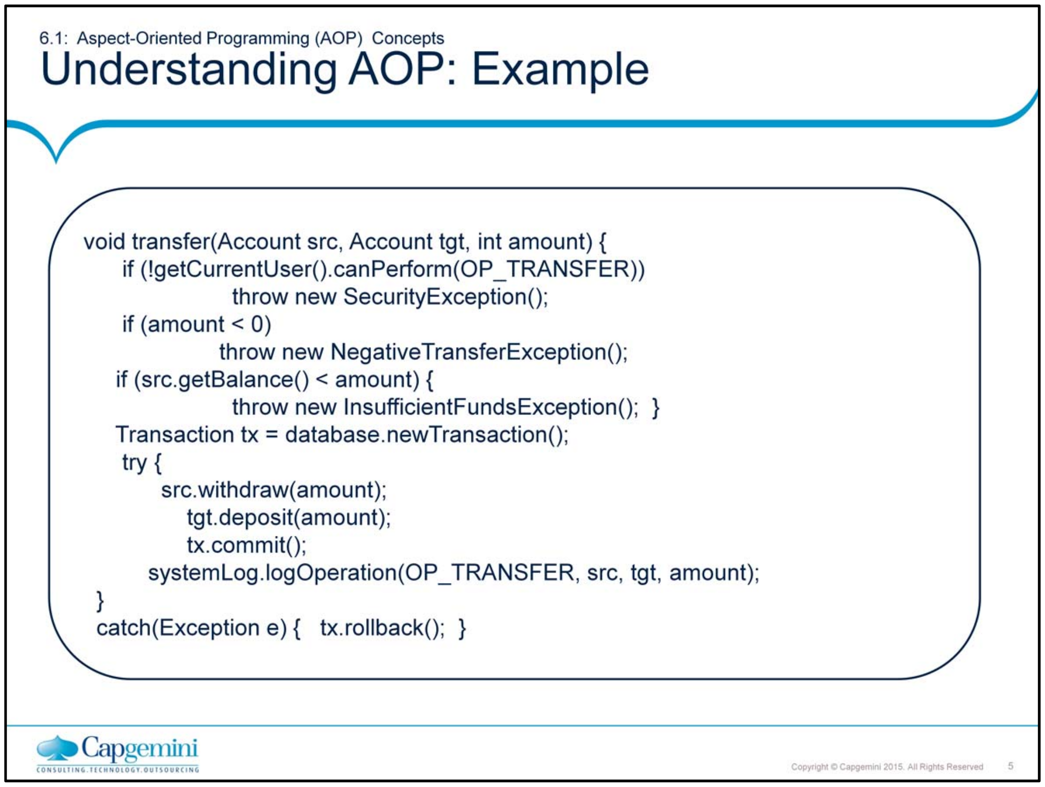
6.1: Aspect-Oriented Programming (AOP) Concepts

Understanding AOP: Example

- An Example:

```
void transfer(Account src, Account tgt, int amount) {
  if (src.getBalance() < amount) {
    throw new InsufficientFundsException();
  }
   src.withdraw(amount);
  tgt.deposit(amount);
}
```

AOP ensures that enterprise service code does not pollute application objects.
For example, consider a banking application with a conceptually simple method for transferring an amount from one account to another as seen above.
The transfer method simply checks if sufficient funds are available in the source account and if ‘yes’, performs the transfer.
However, in a real-world banking application, this transfer method seems far from adequate. We must include security checks to verify that the current user has the authorization to perform this operation. We must enclose the operation in a database transaction to prevent accidental data loss. We must log the operation to the system log, and so on.



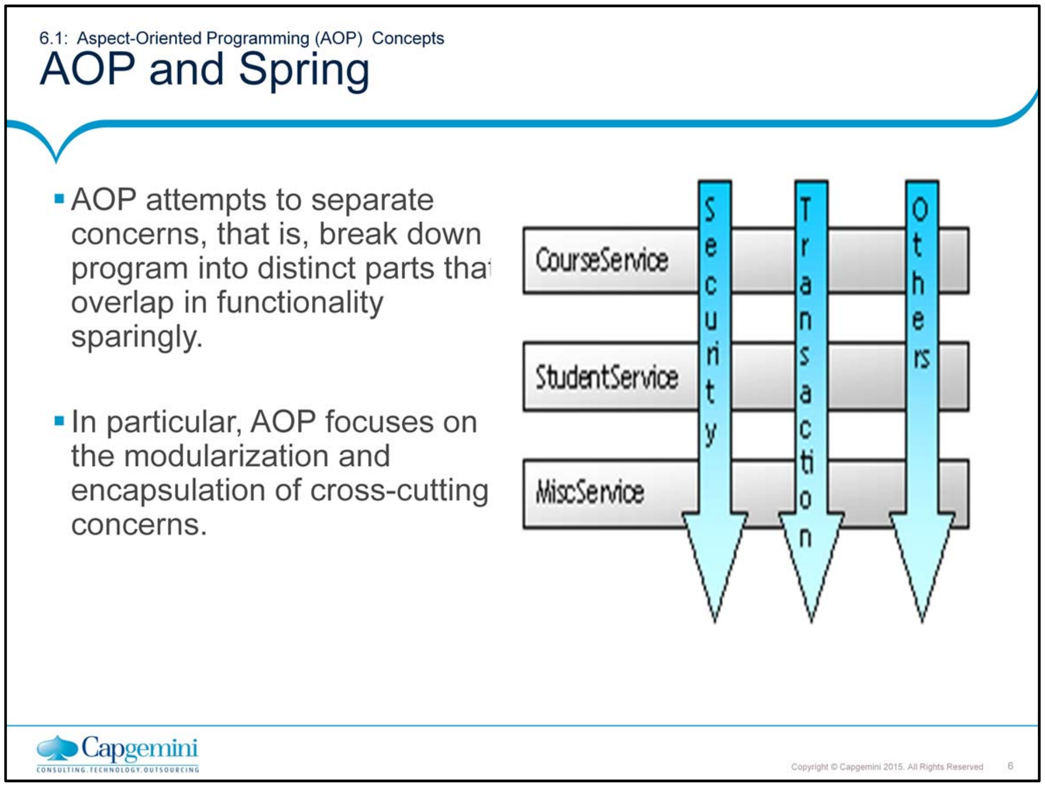
6.1: Aspect-Oriented Programming (AOP) Concepts

Understanding AOP: Example

```
void transfer(Account src, Account tgt, int amount) {
     if (!getCurrentUser().canPerform(OP_TRANSFER))
                    throw new SecurityException();
     if (amount < 0)
                  throw new NegativeTransferException();
    if (src.getBalance() < amount) {
                    throw new InsufficientFundsException();  }
    Transaction tx = database.newTransaction();
     try {
           src.withdraw(amount);
               tgt.deposit(amount);
               tx.commit();
         systemLog.logOperation(OP_TRANSFER, src, tgt, amount);
 }
 catch(Exception e) {   tx.rollback();  }
```

A very simplified version with all those new concerns would look somewhat like the code seen above.
We have now included a security check to verify that the current user has the authorization to perform this operation. We have also enclosed the operation in a database transaction in order to prevent accidental data loss.
The code has lost its elegance and simplicity because the various new concerns have become tangled with the basic functionality. The transactions, security, logging, etc. all exemplify cross-cutting concerns.
Therefore, we find that unlike the core concerns of the system, the cross-cutting concerns do not get properly encapsulated in their own modules. This increases the system complexity and makes maintenance considerably more difficult.
AOP attempts to solve this problem by allowing the programmer to develop cross-cutting concerns as full stand-alone modules called aspects.

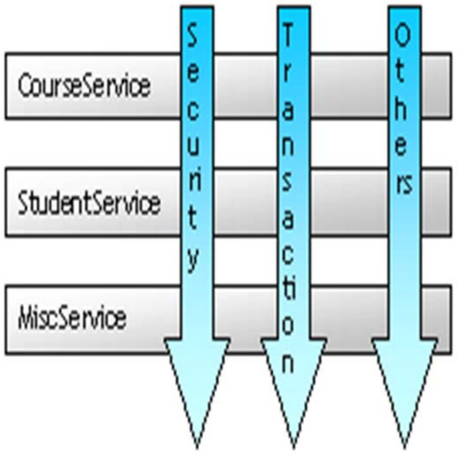
What is Aspect ? It is modularization of a concern. For example, if I have a method to which I want to apply transaction, then applying transaction is a concern. If I modularize then I can use this concern (aspect) to many beans and methods. These services can then be applied declaratively to the components.
Aspect-Oriented Programming has at its core the enabling of a better separation of concerns, by allowing the programmer to create cross-cutting concerns as first-class program modules.



6.1: Aspect-Oriented Programming (AOP) Concepts

AOP and Spring

- AOP attempts to separate concerns, that is, break down program into distinct parts that overlap in functionality sparingly.
- In particular, AOP focuses on the modularization and encapsulation of cross-cutting concerns.

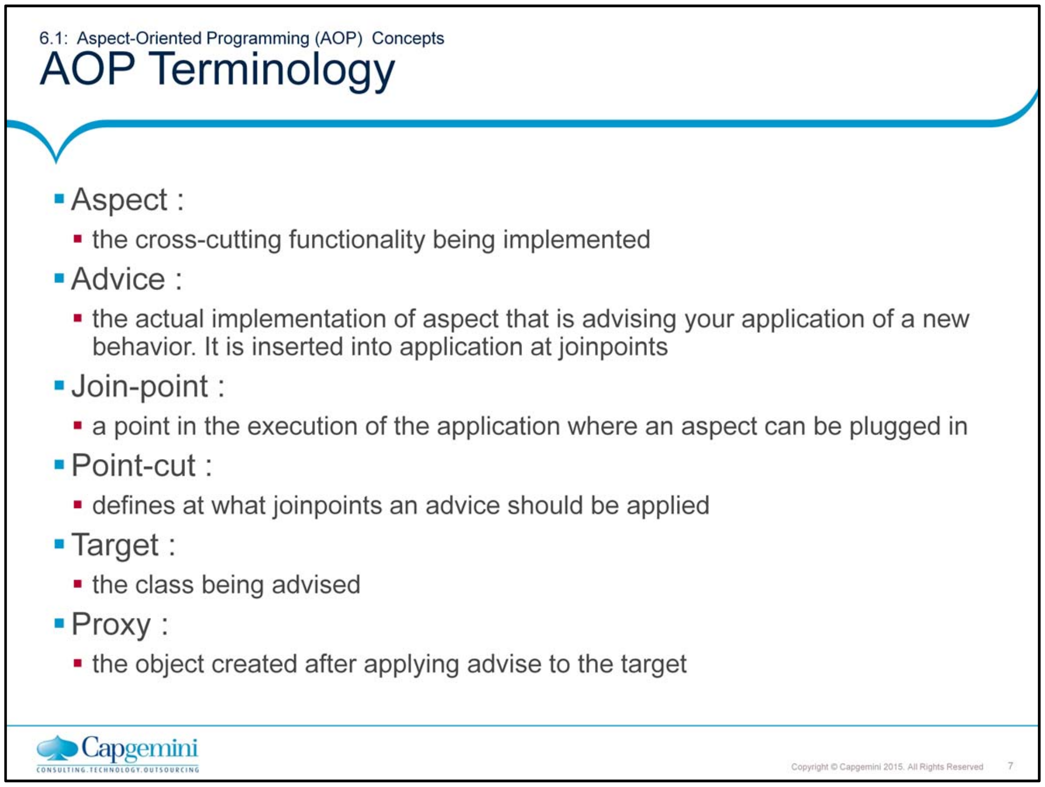


Capgemini

Aspect Oriented programming is often defined as a programming technique that promotes separation of concerns within a software system. Systems are composed of several components, each responsible for a specific piece of functionality. Often, however, these components also carry additional responsibility beyond their core functionality. System services like logging, transaction management and security often find their way into components whose core responsibility 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 makes it possible to modularize services and then apply them declaratively to the components that they should affect. This results in components that are more cohesive and that focus on their own specific concerns, completely ignorant of any system services that may be involved.

The above figure represents a typical application that is broken down into modules. Each module's main concern is to provide services for its particular domain. However, each of these modules also requires similar support functionalities such as security, logging and transaction management. AOP presents an alternative that can be cleaner in many circumstances. With AOP, you can still define the common functionality in one place, but you can declaratively define how and where this functionality is applied without having to modify the class to which you are applying the new feature. Cross-cutting concerns can now be modularized into special objects called aspects.



6.1: Aspect-Oriented Programming (AOP) Concepts

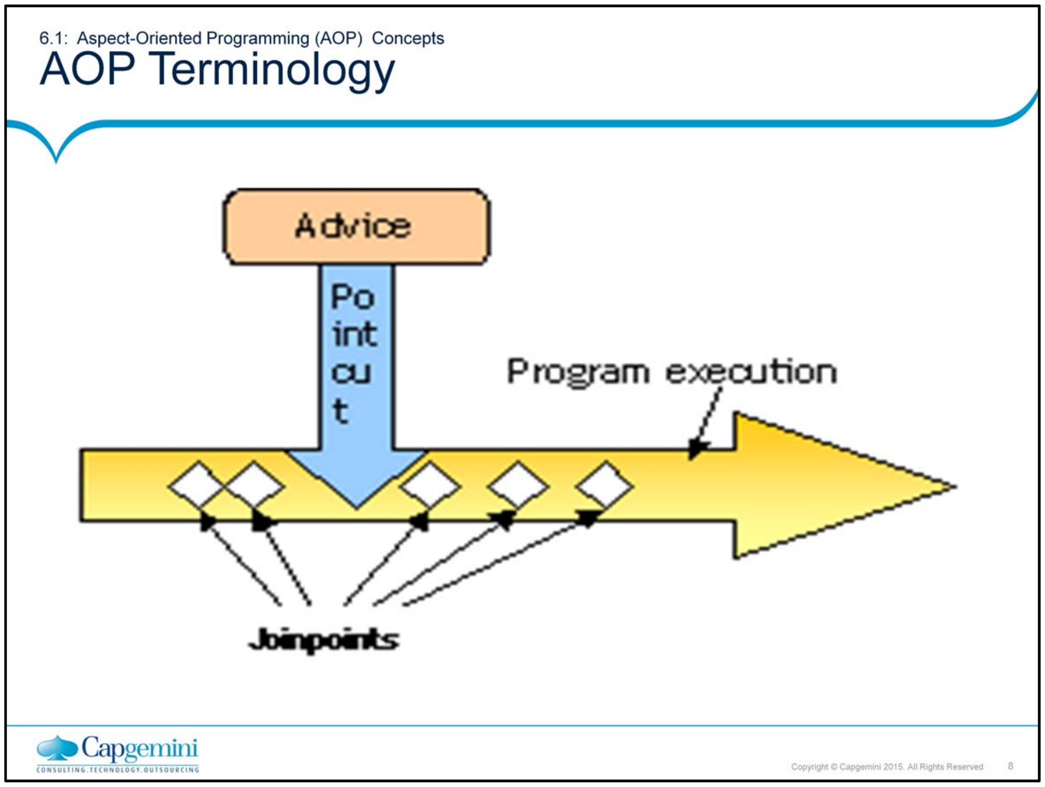
AOP Terminology

- Aspect :
  - the cross-cutting functionality being implemented
- Advice :
  - the actual implementation of aspect that is advising your application of a new behavior. It is inserted into application at joinpoints
- Join-point :
  - a point in the execution of the application where an aspect can be plugged in
- Point-cut :
  - defines at what joinpoints an advice should be applied
- Target :
  - the class being advised
- Proxy :
  - the object created after applying advise to the target

AOP Concepts:
Aspect : an aspect is the cross-cutting functionality you are implementing, that is, the area of your application you are modularizing. A simple example is logging. You can create a logging aspect and apply it throughout your application using AOP.
An aspect is also a combination of advice and pointcuts. This combination results in a definition of the logic that should be included in the application and where it should execute.

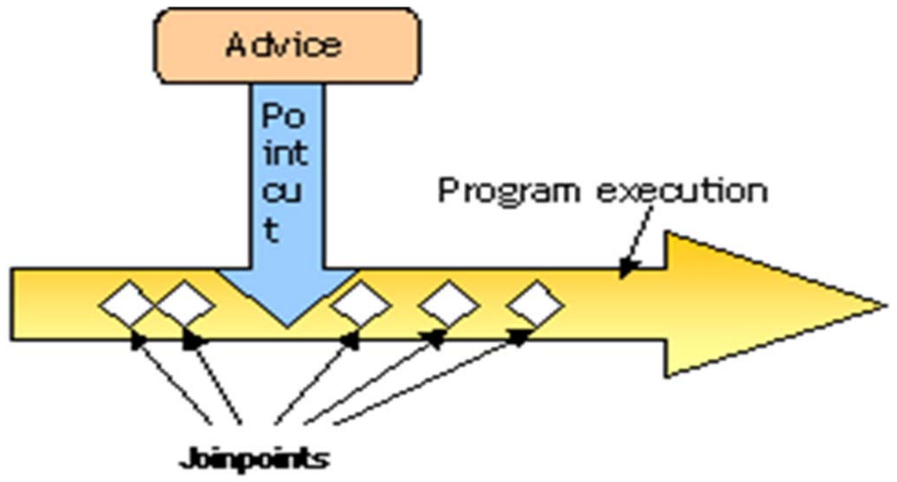
Joinpoint: this is a point in the execution of the application where an aspect can be plugged in. This point could be a method being called, an exception being thrown or even a field being modified. These are points where your aspect's code can be inserted into the normal flow of your application to add new behavior.

Advice: this is the actual implementation of aspect, that is, the code that is executed at a particular joinpoint. It is advising your application of new behavior. There are many types of advices that we shall be seeing later.



6.1: Aspect-Oriented Programming (AOP) Concepts

AOP Terminology



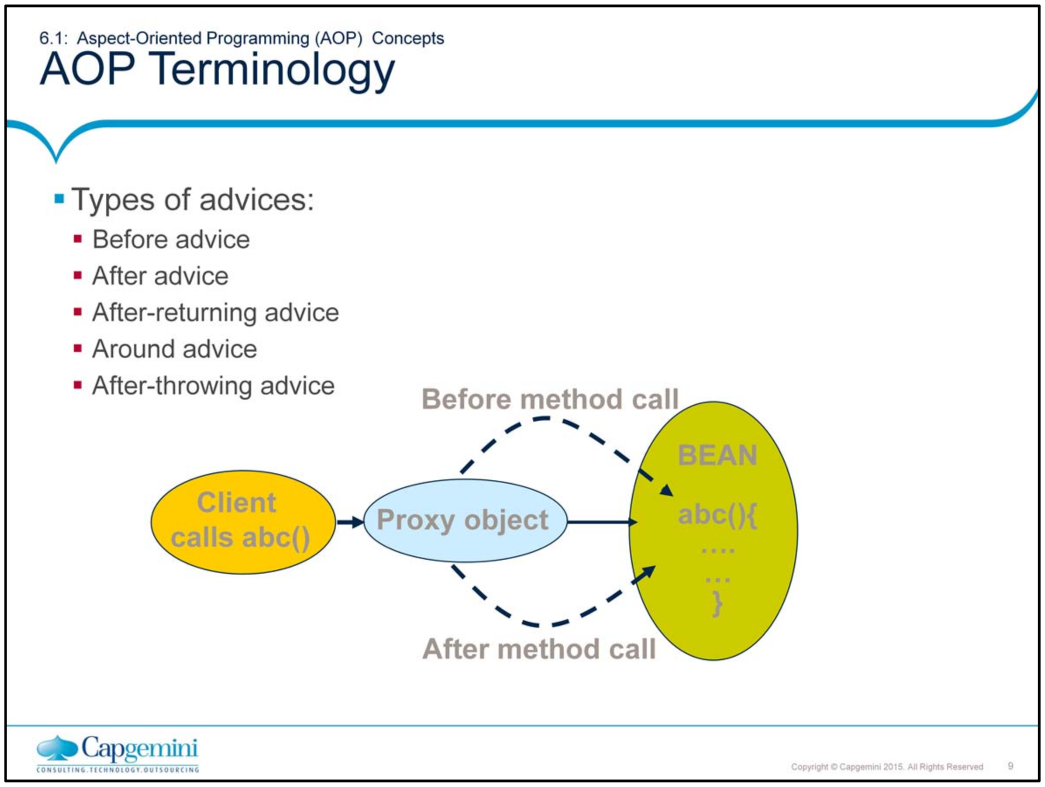
Point-cut: defines at what joinpoints an advice should be applied. Advice can be applied at any joinpoint supported by the AOP framework. By creating pointcuts, you gain fine-grained control over how you apply advice to the components in your application. A typical joinpoint is a method invocation. A typical pointcut is the collection of all method invocations in a particular class. You may not want to apply all aspects at all join-points. Point-cuts allow you to specify where you want advice to be applied.

Target: is the class being advised. Without AOP, this class would have to contain its primary logic plus the logic for any cross-cutting concerns. With AOP, this class is free to focus on its primary concern, oblivious to any advise being applied.

Proxy : is the object created after applying advise to the target. As far as client objects are concerned, the target object (pre-AOP) and proxy object (post-AOP) are the same.

Weaving : is the process of actually inserting aspects into the application code at the appropriate point.

In the above figure, the advice contains the cross-cutting behavior that needs to be applied. The join-points are well-defined points within the execution flow of the application that are candidates to have advice applied. The point-cut defines at what join-points that advice is applied.



6.1: Aspect-Oriented Programming (AOP) Concepts

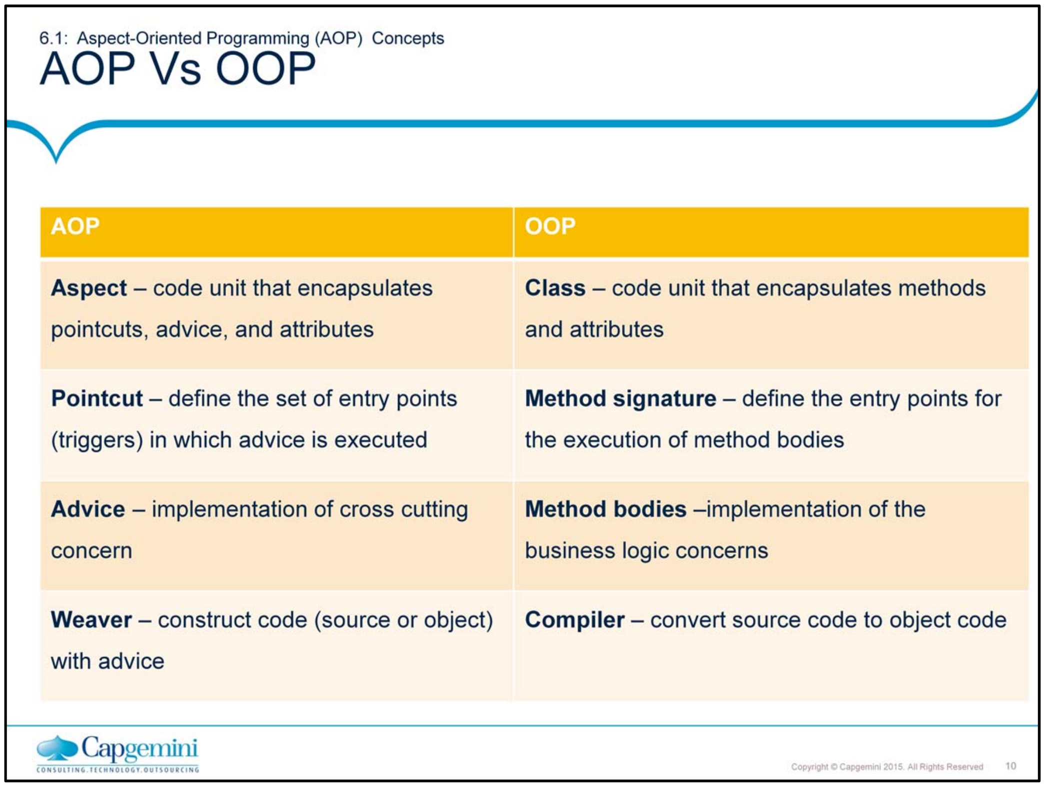
AOP Terminology

- Types of advices:
  - Before advice
  - After advice
  - After-returning advice
  - Around advice
  - After-throwing advice

Before method call
Client calls abc()
Proxy object
BEAN
abc(){
….
…
}
After method call

More on Advice: In the figure above, the client calls a method abc() on target object. But proxy object intercepts the call. The core goal of a proxy is to intercept method invocations and where necessary, execute chains of advice that apply to a particular method.
Spring’s join-point model is built around method interception. This means that the Spring advice will be woven into the application at different points around a method’s invocation. Since there are several points during the execution of a method, there are several advice types:

Before—The advice functionality takes place before the advised method is invoked.
After—The advice functionality takes place after the advised method completes, regardless of the outcome.
After-returning—The advice functionality takes place after the advised method successfully completes.
After-throwing—The advice functionality takes place after the advised method throws an exception.
Around—The advice wraps the advised method, providing some functionality before and after the advised method is invoked.

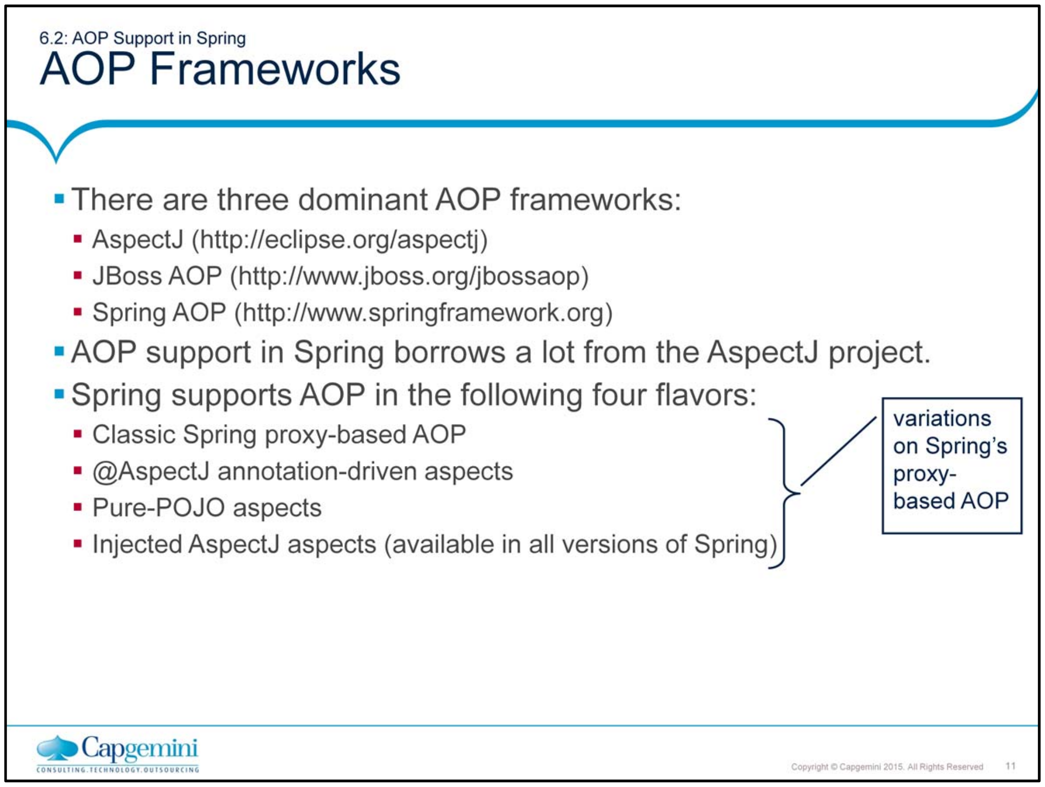


6.1: Aspect-Oriented Programming (AOP) Concepts

AOP Vs OOP

AOP	OOP
Aspect – code unit that encapsulates pointcuts, advice, and attributes	**Class** – code unit that encapsulates methods and attributes
Pointcut – define the set of entry points (triggers) in which advice is executed	**Method signature** – define the entry points for the execution of method bodies
Advice – implementation of cross cutting concern	**Method bodies** –implementation of the business logic concerns
Weaver – construct code (source or object) with advice	**Compiler** – convert source code to object code

10



6.2: AOP Support in Spring

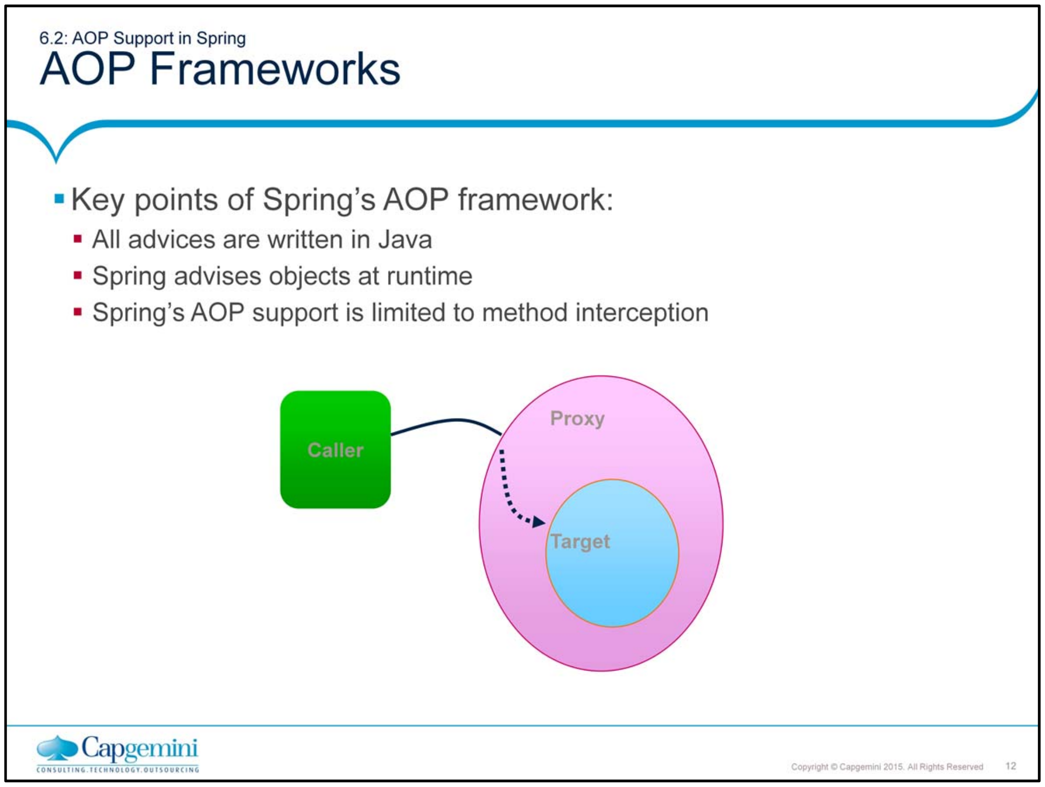
AOP Frameworks

- There are three dominant AOP frameworks:
  - AspectJ (http://eclipse.org/aspectj)
  - JBoss AOP (http://www.jboss.org/jbossaop)
  - Spring AOP (http://www.springframework.org)
- AOP support in Spring borrows a lot from the AspectJ project.
- Spring supports AOP in the following four flavors:
  - Classic Spring proxy-based AOP
  - @AspectJ annotation-driven aspects
  - Pure-POJO aspects
  - Injected AspectJ aspects (available in all versions of Spring)

variations on Spring's proxy-based AOP

AOP frameworks differ from one another. Advice can be applied at the field modification level in some frameworks, whereas others only expose the join points related to method invocations. They may also differ in how and when they weave the aspects. In any case, ability to create pointcuts that define the join points at which aspects should be woven is what makes it an AOP framework.
The AOP frameworks have undergone some housecleaning in the last few years, resulting in some frameworks merging and others going extinct. Today, we have three dominant AOP frameworks (listed above)
We shall focus on Spring AOP, this being a Spring course. But, the AOP support in Spring borrows a lot from the AspectJ project. Thus, Spring's support for AOP comes in four flavors (listed above). The first three are limited to method invocation. If you want more, consider implementing aspects in AspectJ. You can take advantage of Spring DI to inject Spring beans into AspectJ aspects.

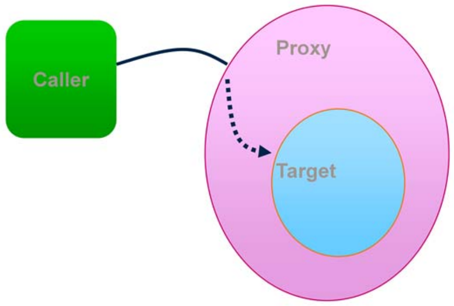
Today, Springs Classic Spring proxy-based AOP is no longer very popular, since Spring supports much cleaner and easier ways to work with aspects. Compared to simple declarative AOP and annotation-based AOP, Spring's classic AOP seems bulky and complex. Working directly with ProxyFactory- Bean can be wearying. Hence we will not cover this method in this course. However, you may refer to Appendix-D to understand the classis AOP API's better.



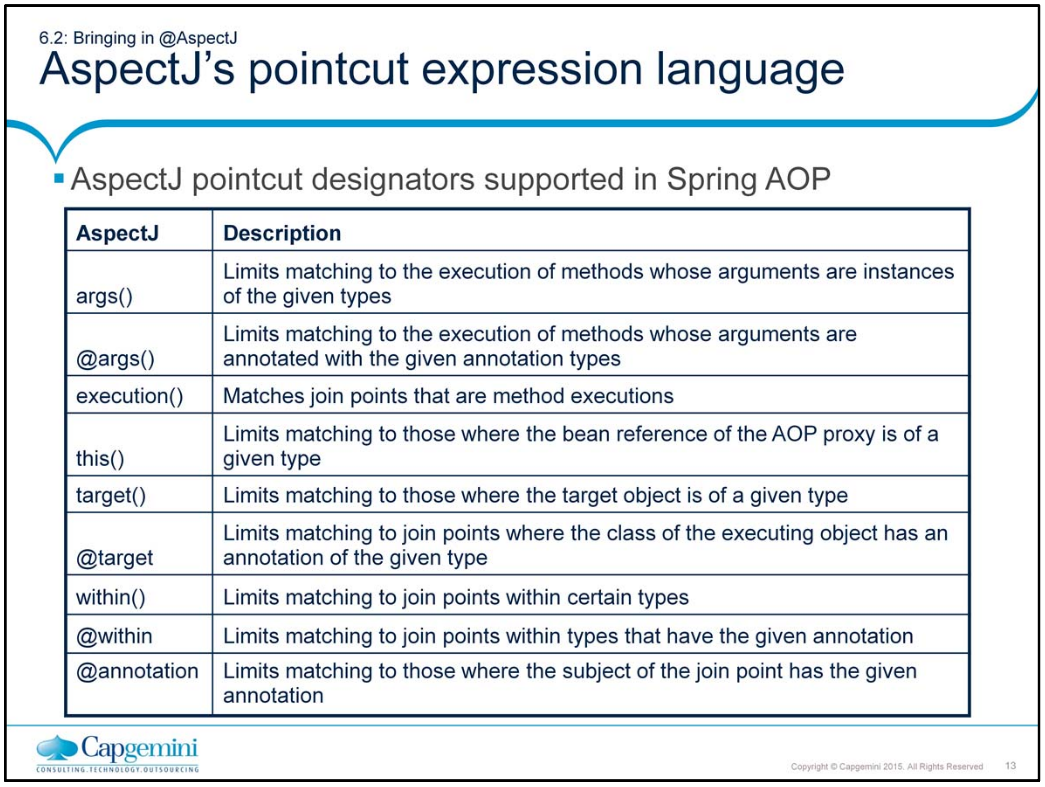
6.2: AOP Support in Spring

AOP Frameworks

- Key points of Spring's AOP framework:
  - All advices are written in Java
  - Spring advises objects at runtime
  - Spring's AOP support is limited to method interception



All advices are written in Java
Spring advises objects at runtime: Aspects are woven into Spring-managed beans at runtime by wrapping them with a proxy class. See the figure above. The proxy class wraps around the target bean, intercepting advised method calls and forwarding those calls to the target bean. Spring doesn't create a proxied object until that proxied bean is needed by the application.
Spring's AOP support is limited to method interception : Some other AOP frameworks, such as AspectJ and JBoss, provide field and constructor join points in addition to method pointcuts. Since Spring does not support field pointcuts, you cannot create very fine-grained advice, such as intercepting updates to an object's field. And without constructor pointcuts, there is no way to apply advice when a bean is instantiated. Method interception is suitable for most needs.



6.2: Bringing in @AspectJ

AspectJ's pointcut expression language

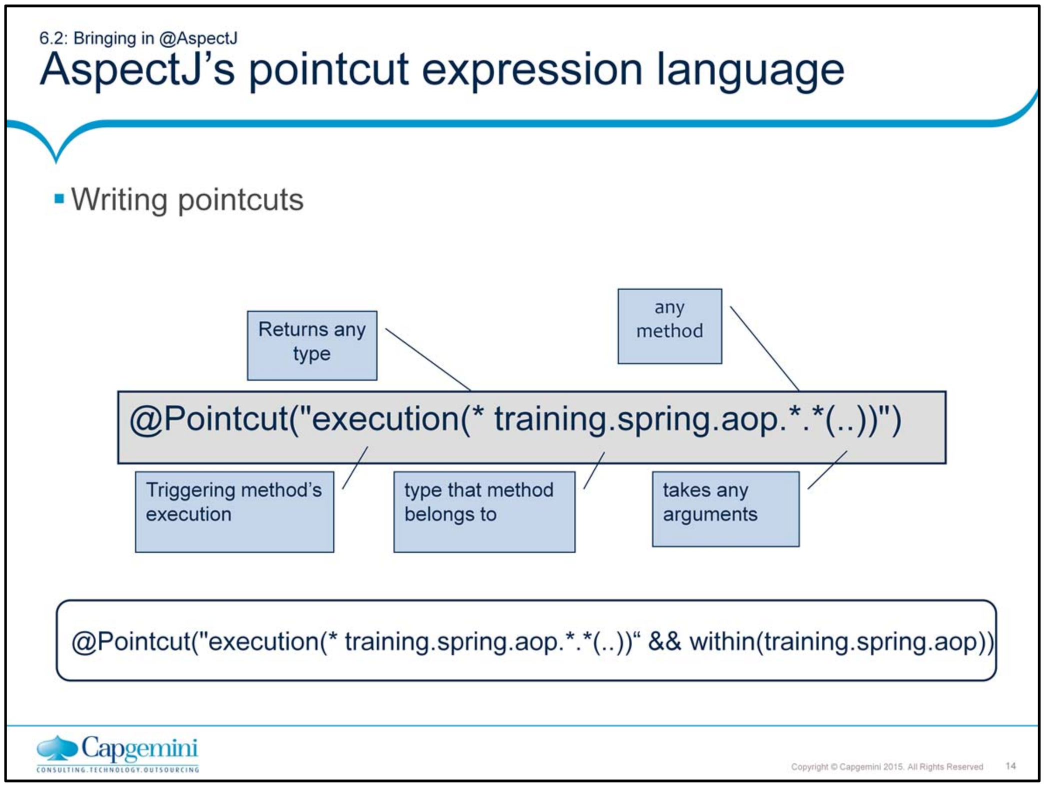
- AspectJ pointcut designators supported in Spring AOP

AspectJ	Description
args()	Limits matching to the execution of methods whose arguments are instances of the given types
@args()	Limits matching to the execution of methods whose arguments are annotated with the given annotation types
execution()	Matches join points that are method executions
this()	Limits matching to those where the bean reference of the AOP proxy is of a given type
target()	Limits matching to those where the target object is of a given type
@target	Limits matching to join points where the class of the executing object has an annotation of the given type
within()	Limits matching to join points within certain types
@within	Limits matching to join points within types that have the given annotation
@annotation	Limits matching to those where the subject of the join point has the given annotation

Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING

Pointcuts are used to pinpoint where an aspect's advice should be applied. Pointcuts are one of the most fundamental elements of an aspect. In Spring AOP, pointcuts are defined using AspectJ's pointcut expression language.

Note : Spring only supports a subset of the pointcut designators available in AspectJ. The table above lists out the AspectJ pointcut designators that are supported in Spring AOP. All of these limit Join point matches to a particular scenario. Only the execution designator actually performs matches. The other designators are used to limit those matches.



6.2: Bringing in @AspectJ

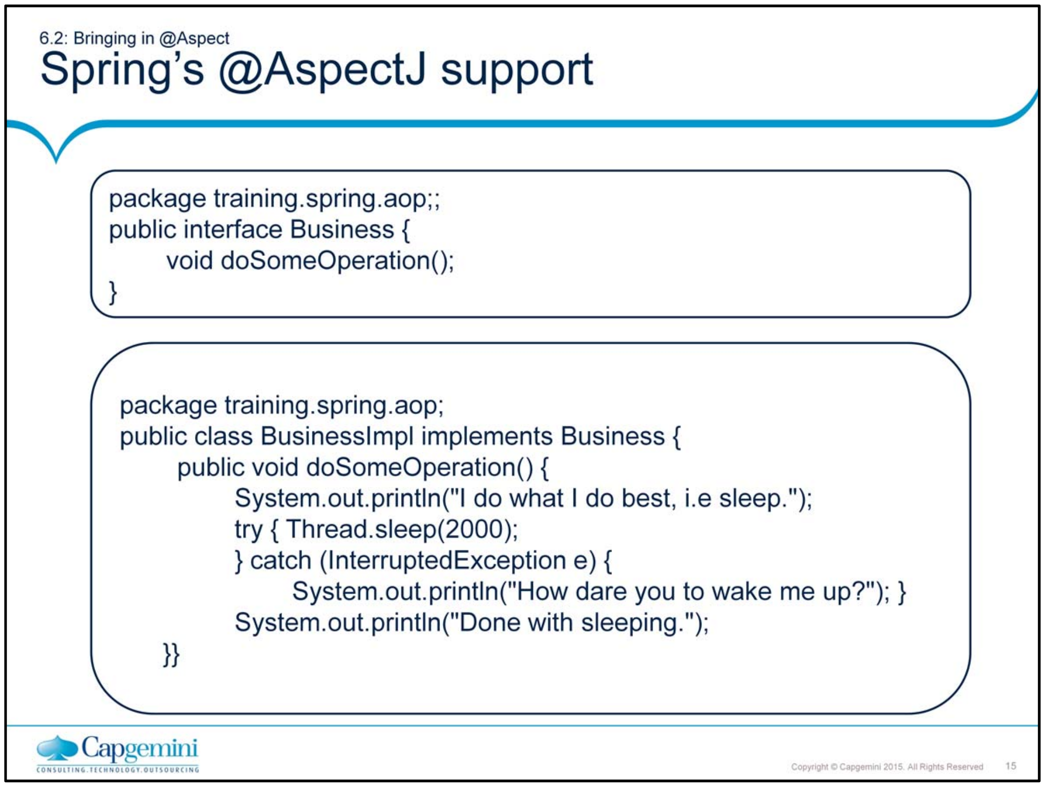
AspectJ’s pointcut expression language

- Writing pointcuts

```
@Pointcut("execution(* training.spring.aop.*.*(..))")
```

```
@Pointcut("execution(* training.spring.aop.*.*(..))“ && within(training.spring.aop))
```

Writing Pointcuts: The pointcut expression shown above is used to apply advice whenever any method of any class in the training.spring.aop package is executed. The method specification starts with an asterisk, which indicates that we do not care what type the method returns. We need to specify the fully qualified class name and the name of the method we want to select. In the method’s parameter list, the double-dot (..) indicates that the pointcut should select any method, no matter what the argument list is.
To confine the reach of that pointcut to only the training.spring.aop package, use the within() designator as seen in example above.


6.2: Bringing in @Aspect

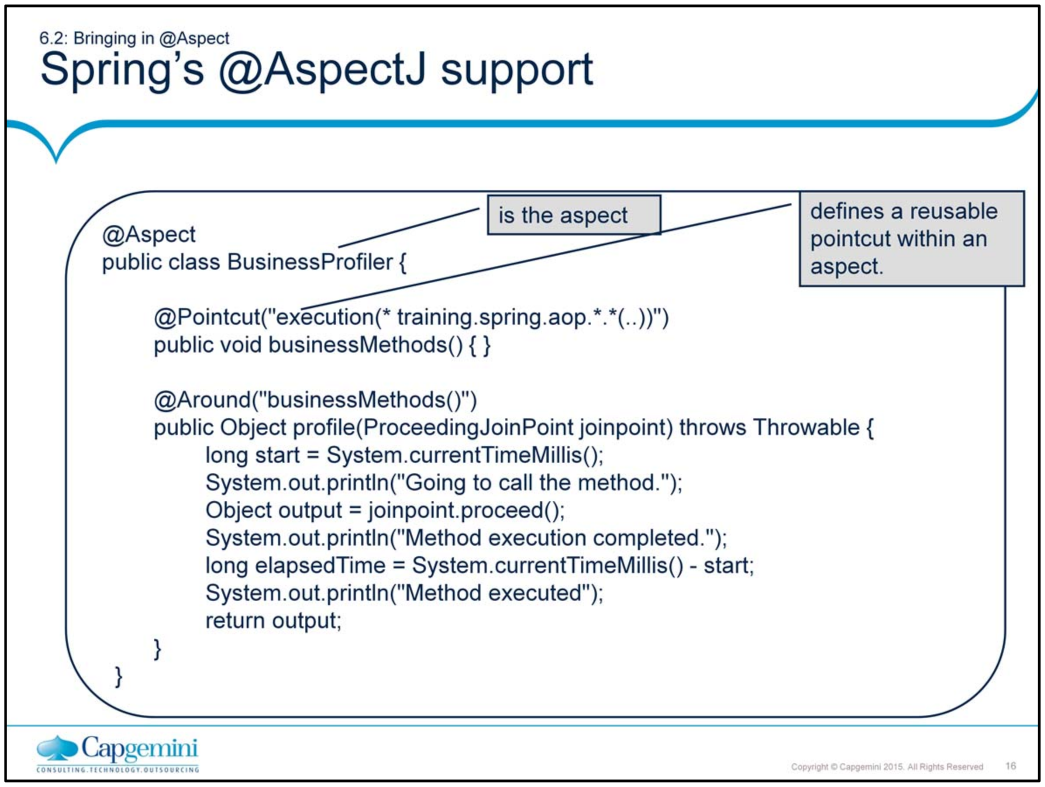
Spring's @AspectJ support

```
package training.spring.aop;;
public interface Business {
        void doSomeOperation();
}
```

```
package training.spring.aop;
public class BusinessImpl implements Business {
        public void doSomeOperation() {
                System.out.println("I do what I do best, i.e sleep.");
                try { Thread.sleep(2000);
                } catch (InterruptedException e) {
                        System.out.println("How dare you to wake me up?"); }
                System.out.println("Done with sleeping.");
        }}
```

@AspectJ refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations. When the @AspectJ support is enabled, any bean in your container that has the @Aspect annotation will be automatically detected by Spring and used to configure Spring AOP.
To use @AspectJ aspects in a Spring configuration you need to enable Spring support, and autoproxying beans based on whether or not they are advised by those aspects. By autoproxying we mean that if Spring determines that a bean is advised by one or more aspects, it will automatically generate a proxy for that bean to intercept method invocations and ensure that advice is executed as needed.
The @AspectJ support is enabled by including the following element inside your spring configuration:
<aop:aspectj-autoproxy/>
For this, you will need to bring in the necessary schema support. This is seen in the XML that is part of the subsequent demo.
Aspects (classes annotated with @Aspect) may have methods and fields just like any other class. They may also contain pointcut and advice.

The code above shows the BusinessImpl class that contains business logic. We shall add Spring AOP to profile our business methods.



6.2: Bringing in @Aspect

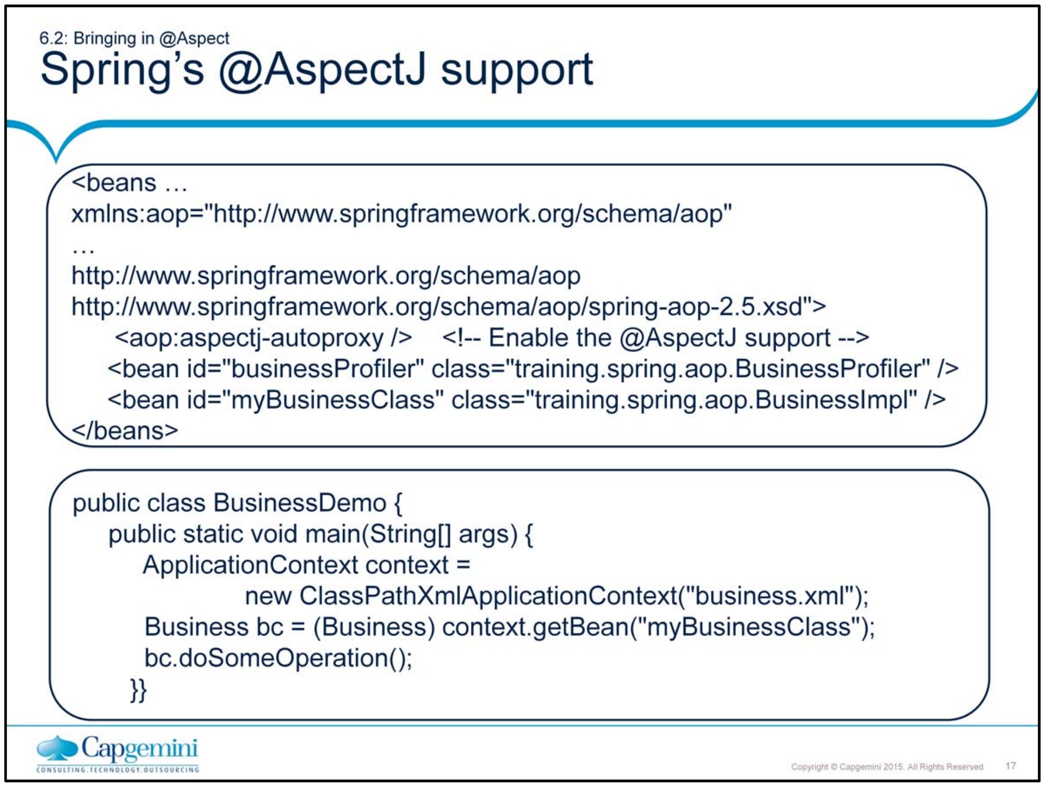
Spring's @AspectJ support

```
@Aspect
public class BusinessProfiler {

        @Pointcut("execution(* training.spring.aop.*.*(..))")
        public void businessMethods() { }

        @Around("businessMethods()")
        public Object profile(ProceedingJoinPoint joinpoint) throws Throwable {
                long start = System.currentTimeMillis();
                System.out.println("Going to call the method.");
                Object output = joinpoint.proceed();
                System.out.println("Method execution completed.");
                long elapsedTime = System.currentTimeMillis() - start;
                System.out.println("Method executed");
                return output;
        }
 }
```

is the aspect

defines a reusable pointcut within an aspect.

See the code above.
Using @AspectJ annotation, we have declared that this class is an Aspect. This will profile our business method
Using @Pointcut annotation, we have defined a pointcut that will match the execution of all public method inside training.spring.aop package.
Using @Around annotation, we have defined a Around advice which will be invoked before and after our business method. The pjp.proceed() method calls our business method from @Around advice.



6.2: Bringing in @Aspect

Spring’s @AspectJ support

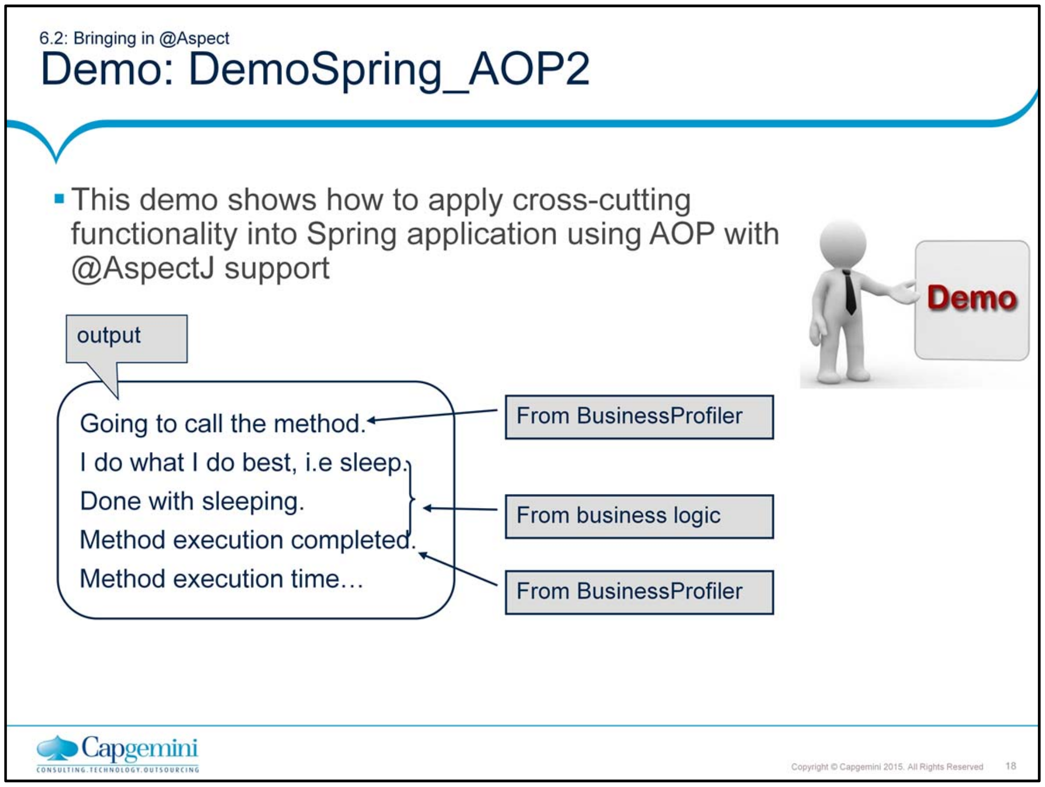
```
<beans …
xmlns:aop="http://www.springframework.org/schema/aop"
…
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">
     <aop:aspectj-autoproxy />    <!-- Enable the @AspectJ support -->
    <bean id="businessProfiler" class="training.spring.aop.BusinessProfiler" />
    <bean id="myBusinessClass" class="training.spring.aop.BusinessImpl" />
</beans>
```

```
public class BusinessDemo {
    public static void main(String[] args) {
        ApplicationContext context =
                    new ClassPathXmlApplicationContext("business.xml");
        Business bc = (Business) context.getBean("myBusinessClass");
        bc.doSomeOperation();
      }}
```

Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING

In the Spring configuration file above, we have:
Added the necessary AOP schemas.
Enabled the @AspectJ support to our application using <aop:aspectj-autoproxy />
Defined two normal Spring beans – one for our Business class and the other for Business Profiler (i.e. our aspect).

<aop:aspectj-autoproxy/> will automatically generate proxy beans whose methods match the pointcuts defined with @Pointcut annotations in @Aspect-annotated beans. To use the <aop:aspectj-autoproxy> configuration element, you’ll need to remember to include the aop namespace in your Spring configuration file.



6.2: Bringing in @Aspect

Demo: DemoSpring_AOP2

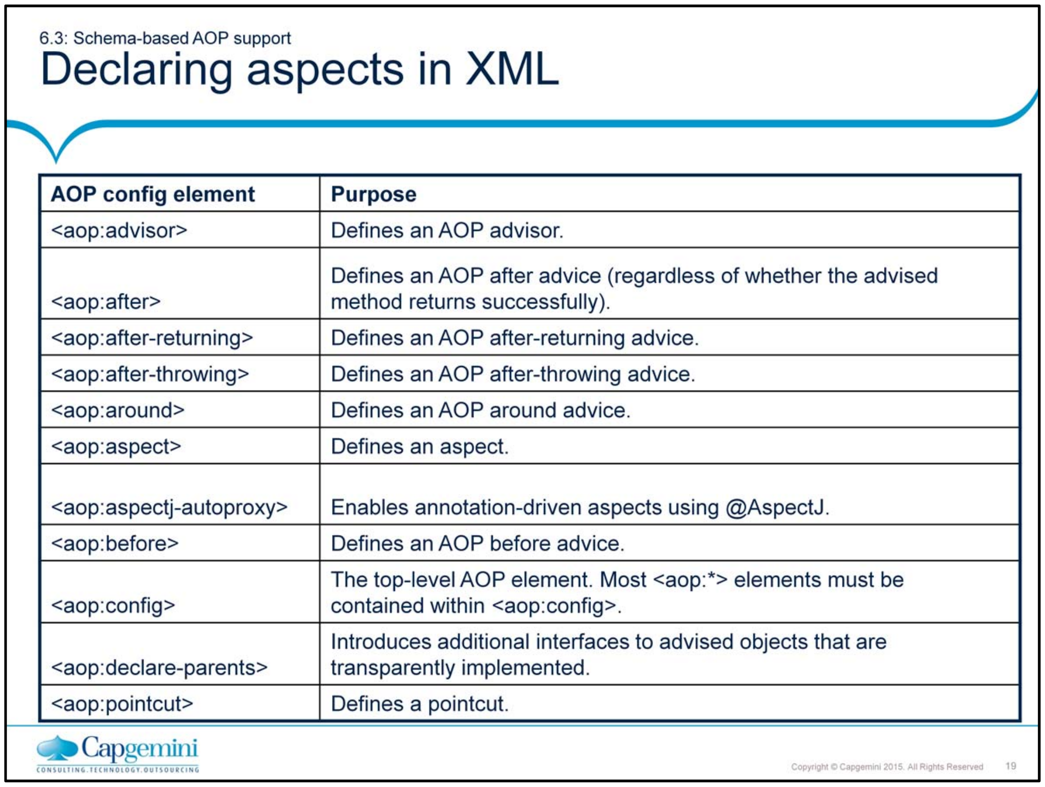
- This demo shows how to apply cross-cutting functionality into Spring application using AOP with @AspectJ support

output

```
Going to call the method.
I do what I do best, i.e sleep.
Done with sleeping.
Method execution completed.
Method execution time…
```

From BusinessProfiler
From business logic
From BusinessProfiler
Demo

Please refer to demo, DemoSpring_AOP2.
Package -> com.igate.aop
Execute the BusinessDemo.java class

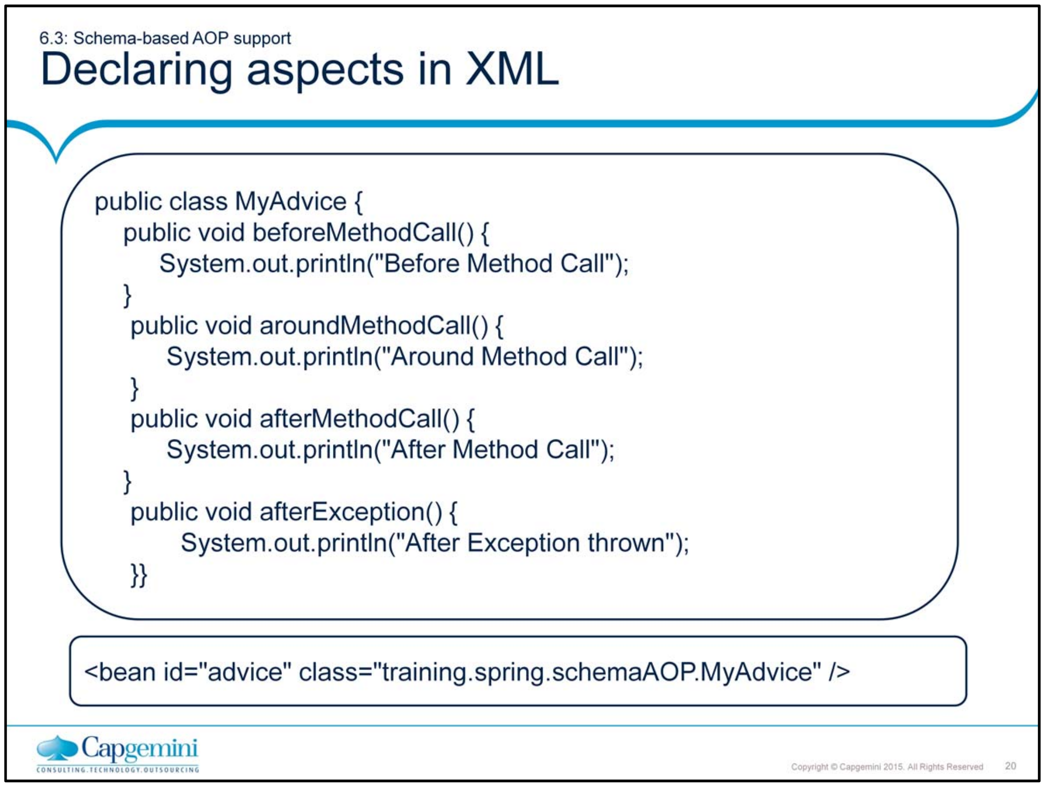


6.3: Schema-based AOP support

Declaring aspects in XML

AOP config element	Purpose
<aop:advisor>	Defines an AOP advisor.
<aop:after>	Defines an AOP after advice (regardless of whether the advised method returns successfully).
<aop:after-returning>	Defines an AOP after-returning advice.
<aop:after-throwing>	Defines an AOP after-throwing advice.
<aop:around>	Defines an AOP around advice.
<aop:aspect>	Defines an aspect.
<aop:aspectj-autoproxy>	Enables annotation-driven aspects using @AspectJ.
<aop:before>	Defines an AOP before advice.
<aop:config>	The top-level AOP element. Most <aop:*> elements must be contained within <aop:config>.
<aop:declare-parents>	Introduces additional interfaces to advised objects that are transparently implemented.
<aop:pointcut>	Defines a pointcut.

From version 2.0 onwards, Spring offers support for defining aspects using the new "aop" namespace tags. The pointcut expressions and advice kinds supported are similar to @AspectJ style.
To use the aop namespace tags, you need to import the spring-aop schema. This is seen in the XML that is part of the subsequent demo.
Within your Spring configurations, all aspect and advisor elements must be placed within an <aop:config> element (multiple <aop:config> element can exist). An <aop:config> element can contain pointcut, advisor, and aspect elements, in that order!



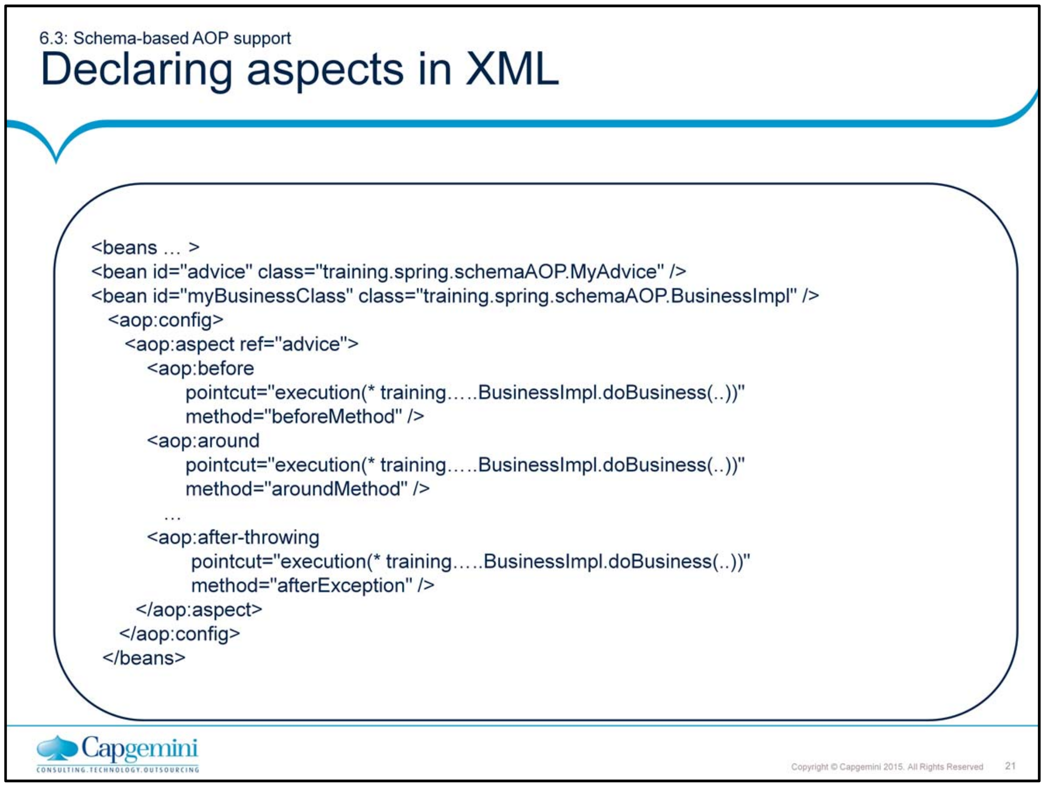
6.3: Schema-based AOP support

Declaring aspects in XML

```
public class MyAdvice {
    public void beforeMethodCall() {
        System.out.println("Before Method Call");
    }
    public void aroundMethodCall() {
        System.out.println("Around Method Call");
    }
    public void afterMethodCall() {
        System.out.println("After Method Call");
    }
    public void afterException() {
          System.out.println("After Exception thrown");
    }}
```

```
<bean id="advice" class="training.spring.schemaAOP.MyAdvice" />
```

Let us see an example of using Spring’s aop configuration namespace. The code listing above is very straightforward and simply creates functions of an advice. It’s a POJO with a handful of methods. Thus we can register it as a bean in the Spring application context like any other class. See above for how.
The MyAdvice class seems an Aspect but needs Spring’s AOP magic to become one.



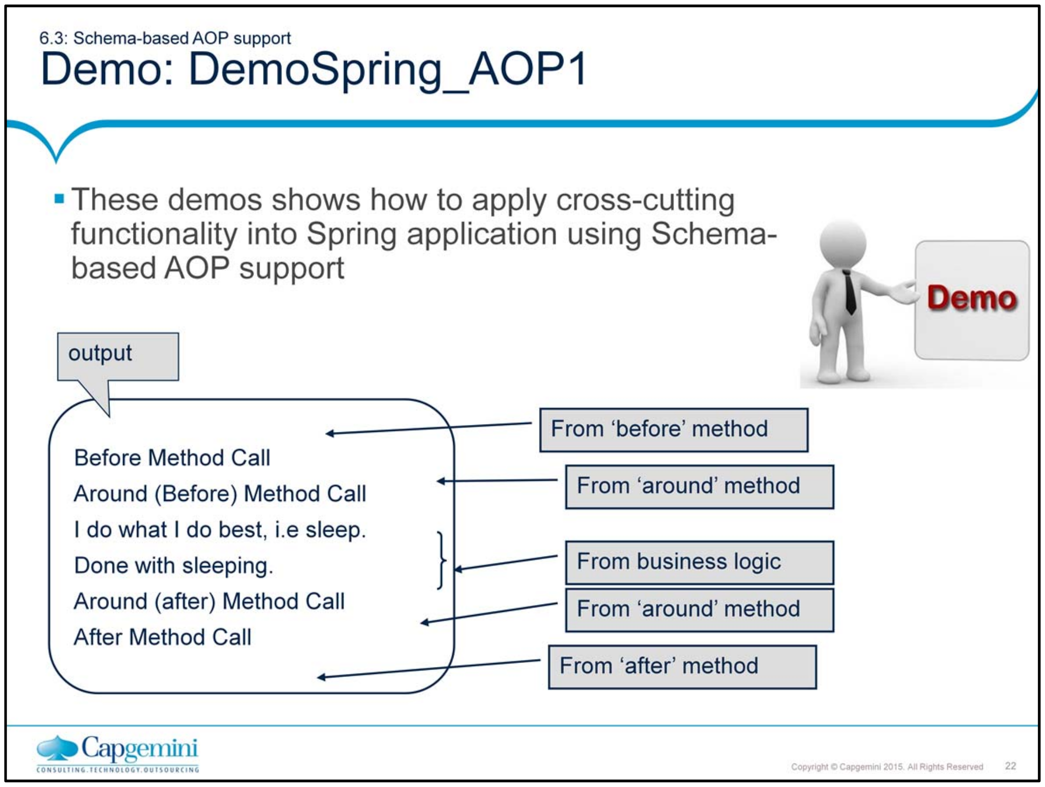
6.3: Schema-based AOP support

Declaring aspects in XML

```
<beans … >
<bean id="advice" class="training.spring.schemaAOP.MyAdvice" />
<bean id="myBusinessClass" class="training.spring.schemaAOP.BusinessImpl" />
  <aop:config>
    <aop:aspect ref="advice">
       <aop:before
            pointcut="execution(* training…..BusinessImpl.doBusiness(..))"
            method="beforeMethod" />
       <aop:around
            pointcut="execution(* training…..BusinessImpl.doBusiness(..))"
            method="aroundMethod" />
         …
       <aop:after-throwing
             pointcut="execution(* training…..BusinessImpl.doBusiness(..))"
             method="afterException" />
     </aop:aspect>
    </aop:config>
  </beans>
```

Using the Spring’s AOP configuration elements, as shown in the listing above, we have converted the MyAdvice POJO into an aspect!
Notice that most of the configuration elements are used within the context of the <aop:config> element. Within <aop:config> you may declare one or more advisors, aspects, or pointcuts. In our example, we have declared a single aspect using the <aop:aspect> element. The ref attribute references the POJO bean (MyAdvice in this case) that will be used to supply the functionality of the aspect ie methods called by any advice in the aspect.
The aspect has several bits of advice. The <aop:before> element calls aspect method (beforeMethod()) before the methods matching the pointcut are executed.
The <aop:after-throwing> element defines an after-throwing advice to call the afterException() if any exceptions are thrown.
In all advice elements, the pointcut attribute defines the pointcut where the advice will be applied. Notice that the value of the pointcut attribute is the same for all of the advice elements. That’s because all of the advice is being applied to the same pointcut.
The pointcut expression is repeated throughout the code. To avoid this, you can use <aop:pointcut> element to define a named pointcut that can be used by all of the advice elements.

```
<aop:aspect ref=“advice">
    <aop:pointcut id=“pt" expression=
         "execution(* training.spring.schemaAOP.BusinessImpl.doBusiness())“ />
<aop:before pointcut-ref=“pt“ method=“beforeMethod" />
```



6.3: Schema-based AOP support

Demo: DemoSpring_AOP1

- These demos shows how to apply cross-cutting functionality into Spring application using Schema-based AOP support

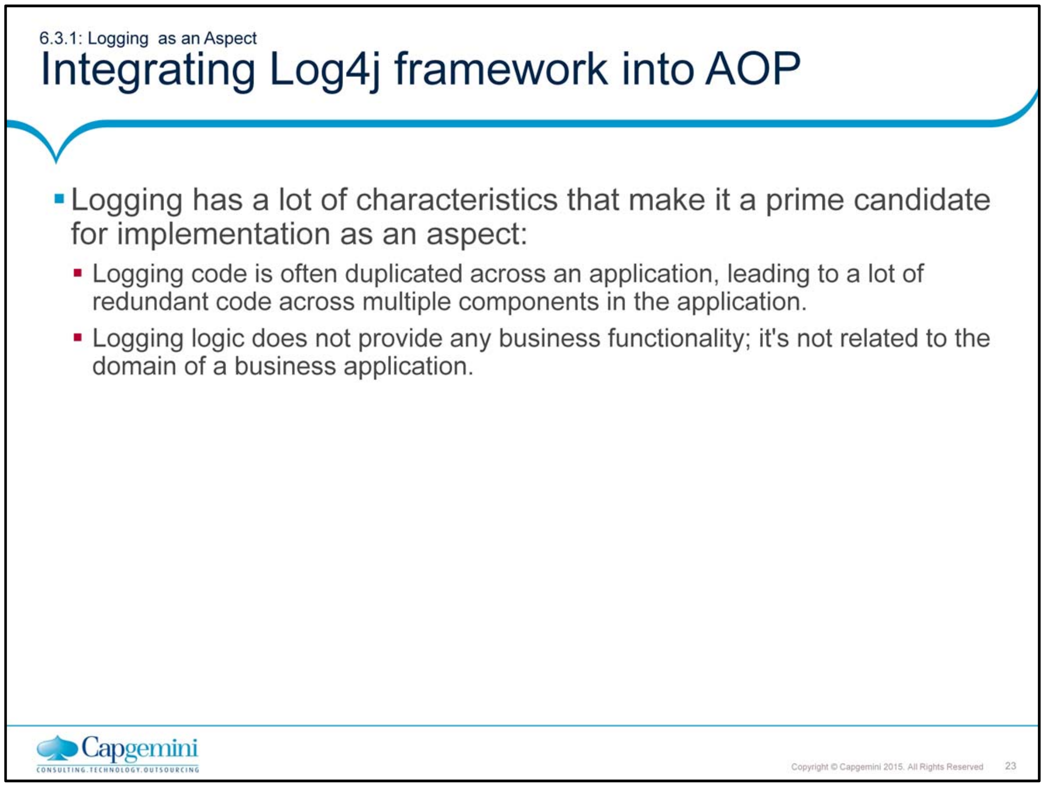
output

```
Before Method Call
Around (Before) Method Call
I do what I do best, i.e sleep.
Done with sleeping.
Around (after) Method Call
After Method Call
```

From 'before' method
From 'around' method
From business logic
From 'around' method
From 'after' method
Demo

Please refer to demo, DemoSpring_AOP1 -> com.igate.aop
Execute the BusinessDemo.java class

We have modified the aroundMethod() method in MyAdvice.java a bit to add the proceed() method. The ProceedingJoinPoint object allows us to invoke the advised method from within our advice. The advice method will do everything it needs to do and, when it's ready to pass control to the advised method, it'll call ProceedingJoinPoint's proceed() method. If you don't include the call to proceed() method, your advice will block access to the advised method.



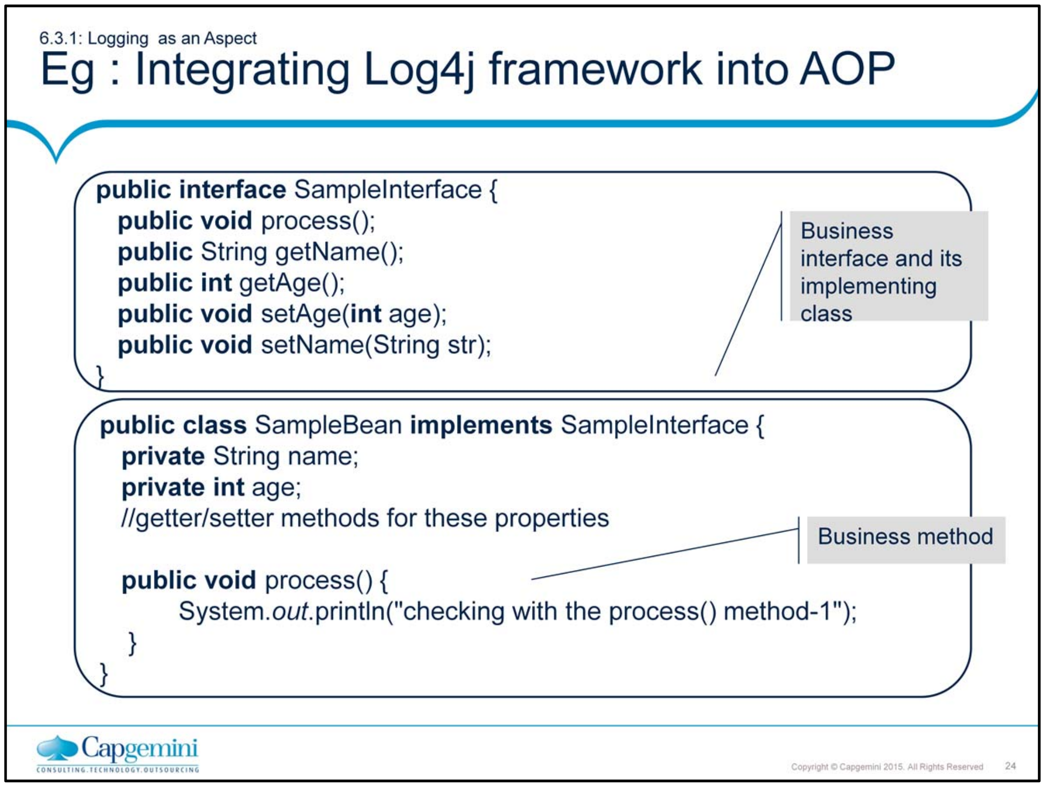
6.3.1: Logging as an Aspect

Integrating Log4j framework into AOP

- Logging has a lot of characteristics that make it a prime candidate for implementation as an aspect:
  - Logging code is often duplicated across an application, leading to a lot of redundant code across multiple components in the application.
  - Logging logic does not provide any business functionality; it's not related to the domain of a business application.

Logging has a lot of characteristics that make it a prime candidate for implementation as an aspect. The following are two of the notable characteristics:
Logging code is often duplicated across an application, leading to a lot of redundant code across multiple components in the application. Even if the logging logic is abstracted to a separate module so that different components have a single method call, that single method call is duplicated in multiple places.
Logging logic does not provide any business functionality; it's not related to the domain of a business application.
Spreading the logging across multiple components introduces some complexity to the code. Business objects not only perform business tasks, they also take care of logging functionality (and other such crosscutting concerns, such security, transaction, caching, etc.).
AOP makes it possible to modularize these crosscutting services and then apply them declaratively to the components that they should affect.

Let us see a simple example. We shall use a around advice AOP advice to intercept a method in the business class and log the operation at DEBUG level.



6.3.1: Logging as an Aspect

Eg : Integrating Log4j framework into AOP

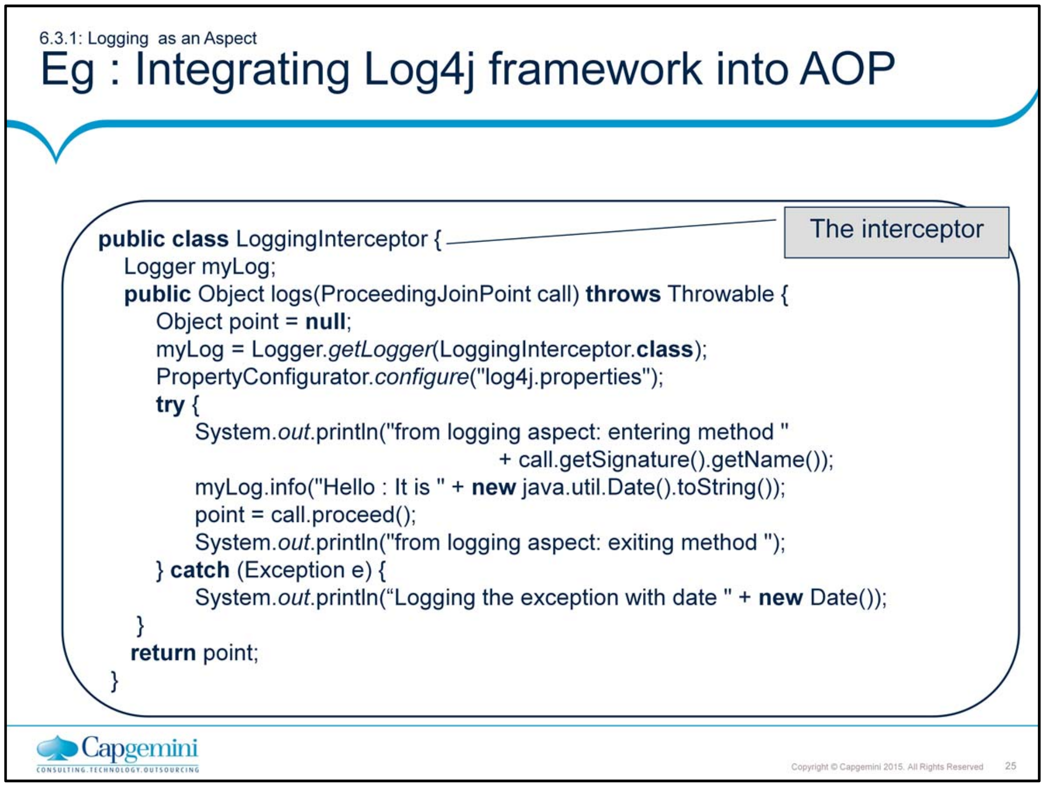
```
public interface SampleInterface {
  public void process();
  public String getName();
  public int getAge();
  public void setAge(int age);
  public void setName(String str);
}
```

Business interface and its implementing class

```
public class SampleBean implements SampleInterface {
  private String name;
  private int age;
  //getter/setter methods for these properties

  public void process() {
        System.out.println("checking with the process() method-1");
  }
}
```

Business method

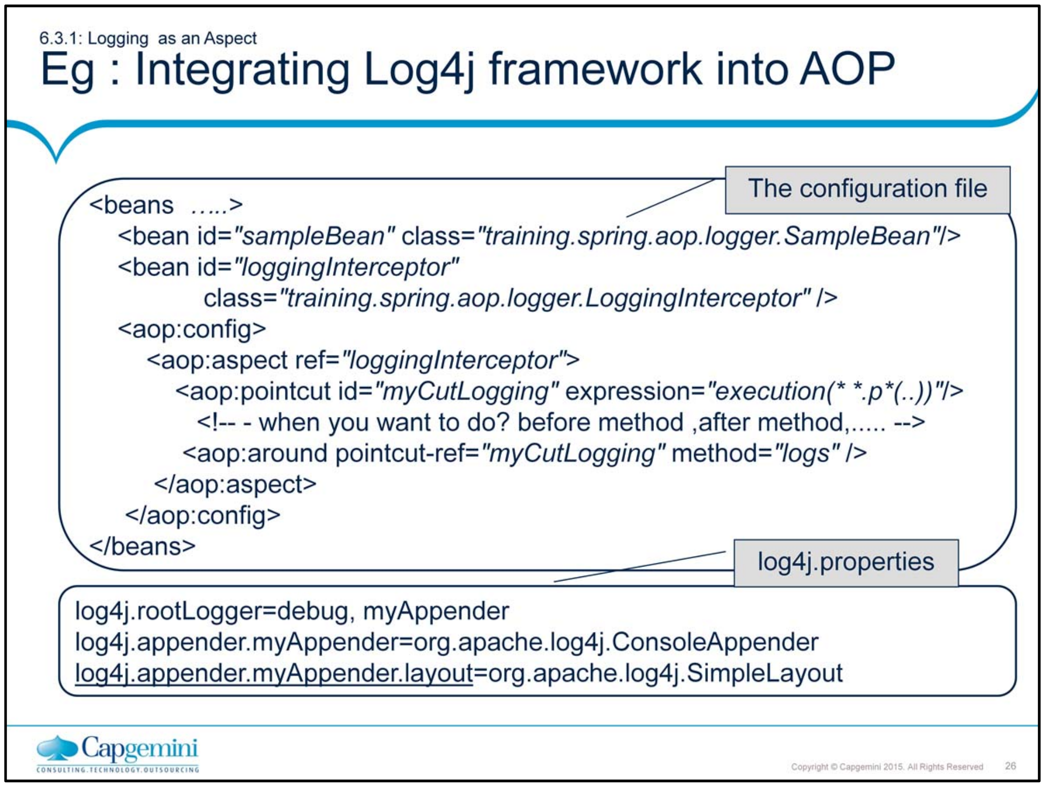
Copyright © Capgemini 2015. All Rights Reserved 24



6.3.1: Logging as an Aspect

Eg : Integrating Log4j framework into AOP

```
public class LoggingInterceptor {
   Logger myLog;
   public Object logs(ProceedingJoinPoint call) throws Throwable {
        Object point = null;
        myLog = Logger.getLogger(LoggingInterceptor.class);
        PropertyConfigurator.configure("log4j.properties");
        try {
             System.out.println("from logging aspect: entering method "
                                          + call.getSignature().getName());
             myLog.info("Hello : It is " + new java.util.Date().toString());
             point = call.proceed();
             System.out.println("from logging aspect: exiting method ");
        } catch (Exception e) {
             System.out.println("Logging the exception with date " + new Date());
    }
    return point;
 }
```



6.3.1: Logging as an Aspect

Eg : Integrating Log4j framework into AOP

The configuration file

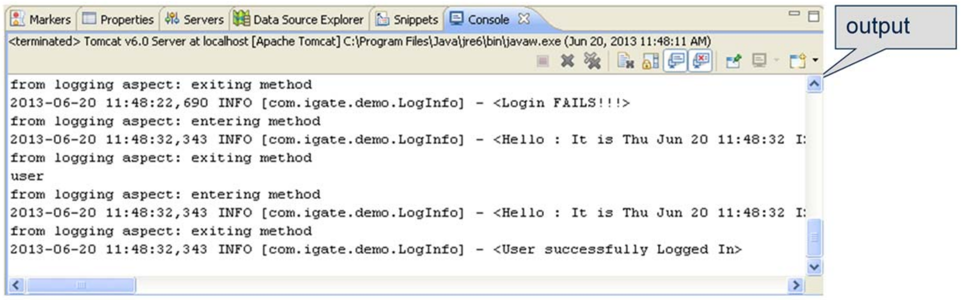
```
<beans  …..>
   <bean id="sampleBean" class="training.spring.aop.logger.SampleBean"/>
   <bean id="loggingInterceptor"
            class="training.spring.aop.logger.LoggingInterceptor" />
   <aop:config>
      <aop:aspect ref="loggingInterceptor">
         <aop:pointcut id="myCutLogging" expression="execution(* *.p*(..))"/>
           <!-- - when you want to do? before method ,after method,..... -->
          <aop:around pointcut-ref="myCutLogging" method="logs" />
       </aop:aspect>
    </aop:config>
</beans>
```

log4j.properties

```
log4j.rootLogger=debug, myAppender
log4j.appender.myAppender=org.apache.log4j.ConsoleAppender
log4j.appender.myAppender.layout=org.apache.log4j.SimpleLayout
```

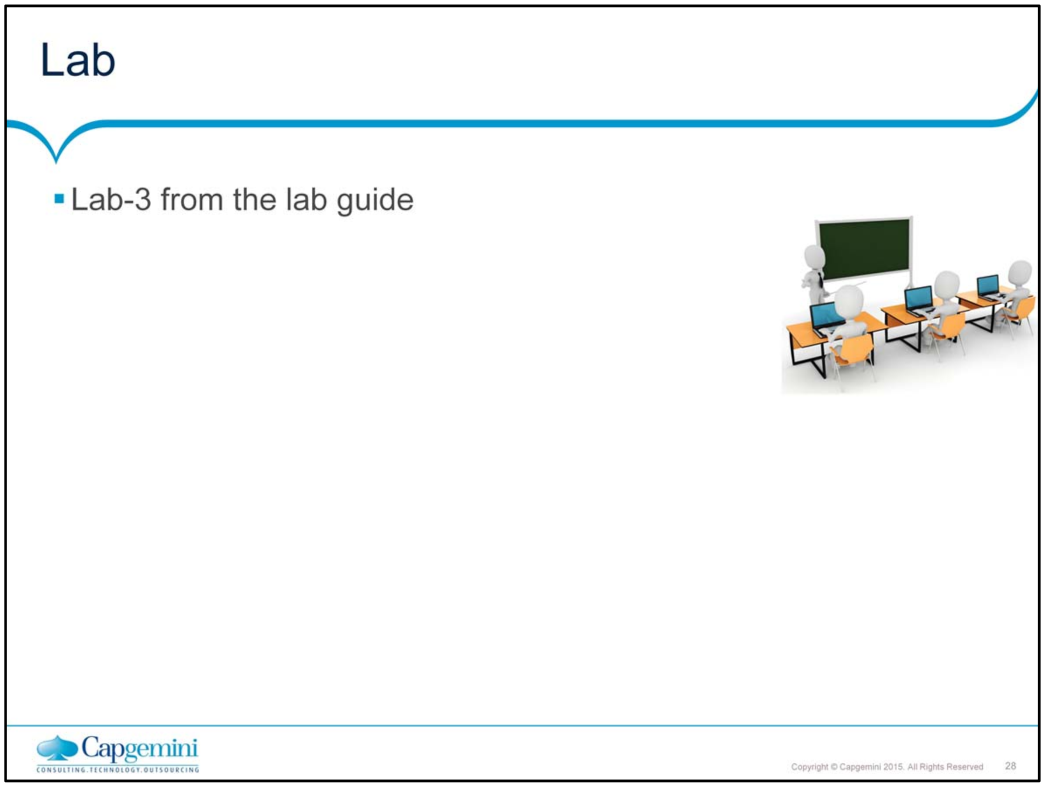
Demo: DemoMVC_AOP

- This demo shows how to integrate the Log4j logging framework with AOP using MVC based an application



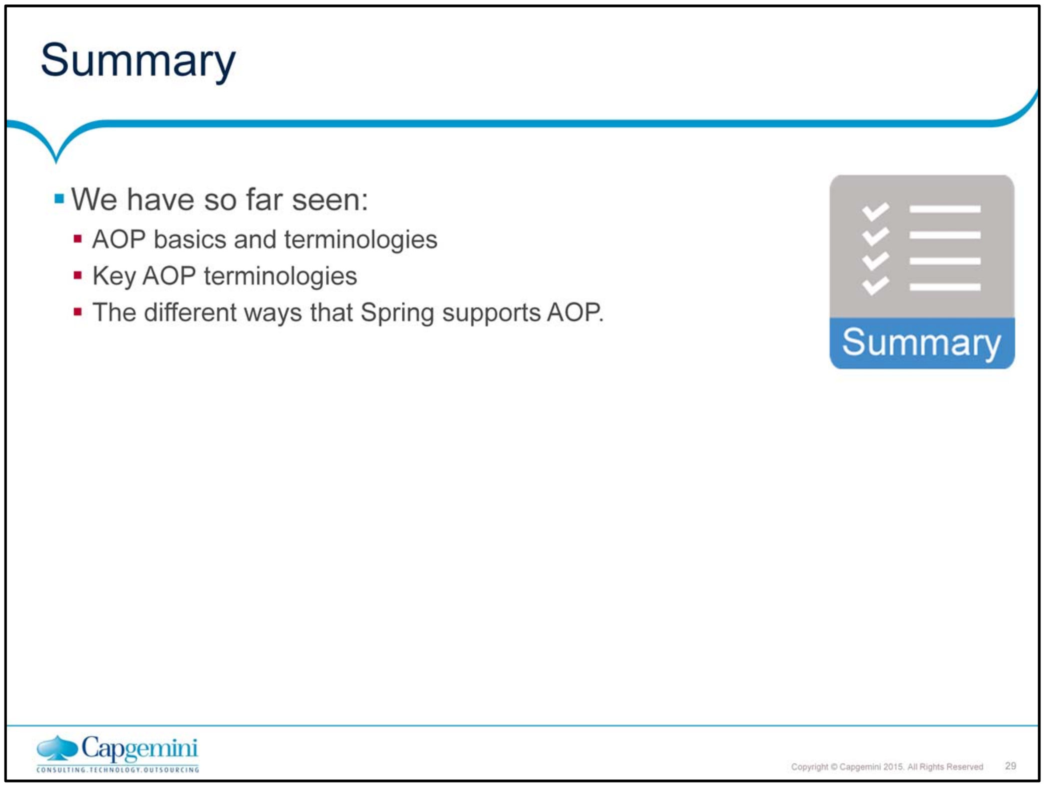
```
<terminated> Tomcat v6.0 Server at localhost [Apache Tomcat] C:\Program Files\Java\jre6\bin\javaw.exe (Jun 20, 2013 11:48:11 AM)
from logging aspect: exiting method
2013-06-20 11:48:22,690 INFO [com.igate.demo.LogInfo] - <Login FAILS!!!>
from logging aspect: entering method
2013-06-20 11:48:32,343 INFO [com.igate.demo.LogInfo] - <Hello : It is Thu Jun 20 11:48:32 I:
from logging aspect: exiting method
user
from logging aspect: entering method
2013-06-20 11:48:32,343 INFO [com.igate.demo.LogInfo] - <Hello : It is Thu Jun 20 11:48:32 I:
from logging aspect: exiting method
2013-06-20 11:48:32,343 INFO [com.igate.demo.LogInfo] - <User successfully Logged In>
```

Please refer to demo, DemoMVC_AOP



Lab

- Lab-3 from the lab guide



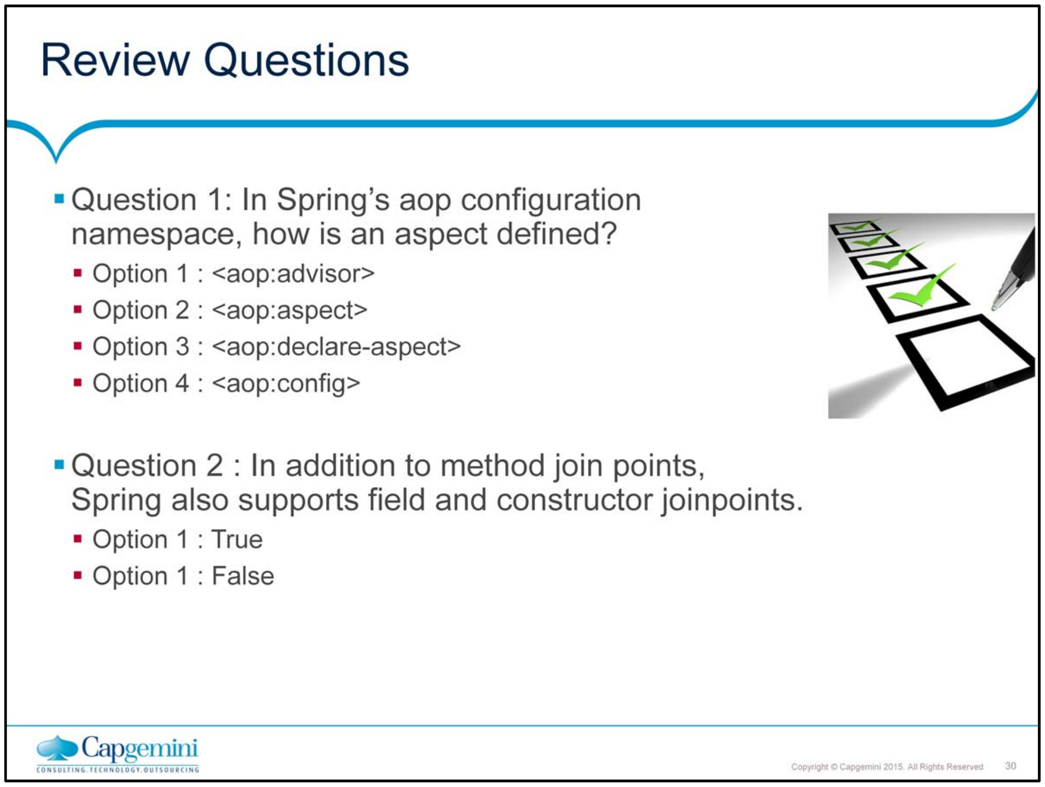
Summary

- We have so far seen:
  - AOP basics and terminologies
  - Key AOP terminologies
  - The different ways that Spring supports AOP.

AOP is a powerful complement to object-oriented programming. With aspects, you can now group application behavior that was once spread throughout your applications into reusable modules. You can then declaratively or programmatically define exactly where and how this behavior is applied. This reduces code duplication and lets your classes focus on their main functionality. Thus, the AOP technique helps you add design and run-time behavior to an object model in a non-obtrusive manner by using static and dynamic crosscutting.

The first thing that comes to mind when someone mentions AOP is logging, followed by transaction management. However, these are just special applications of AOP. AOP can be used for performance monitoring, call auditing, caching and error recovery too. More advanced uses of AOP include compile-time checks of architecture standards. For example, you can write an aspect that will enforce you to call only certain methods from certain classes.

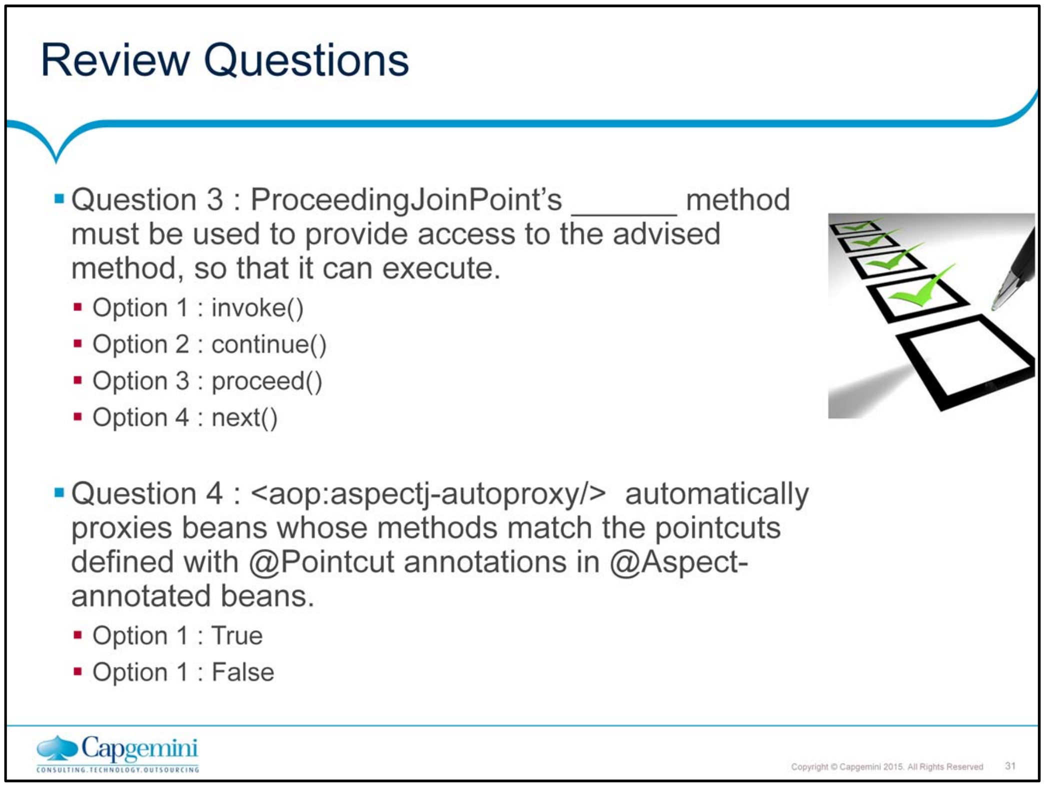
Today, Spring AOP offers fine grained object advising capabilities using AspectJ support.



Review Questions

- Question 1: In Spring's aop configuration namespace, how is an aspect defined?
  - Option 1 : <aop:advisor>
  - Option 2 : <aop:aspect>
  - Option 3 : <aop:declare-aspect>
  - Option 4 : <aop:config>

- Question 2 : In addition to method join points, Spring also supports field and constructor joinpoints.
  - Option 1 : True
  - Option 1 : False



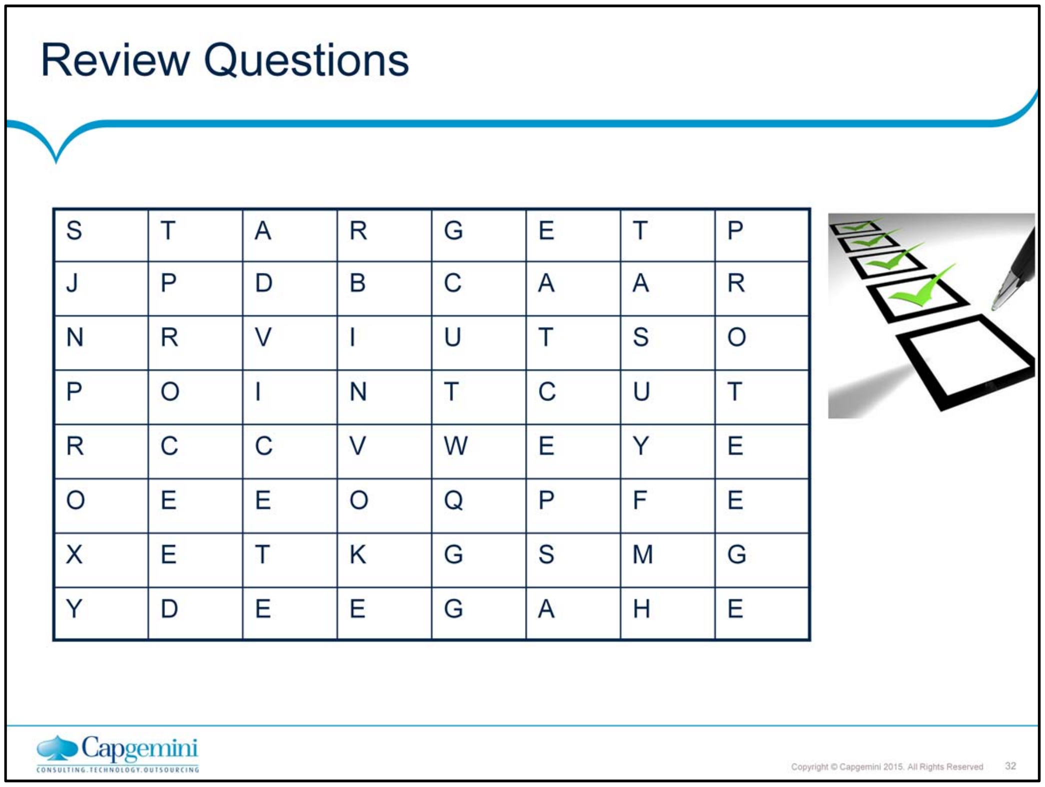
Review Questions

- Question 3 : ProceedingJoinPoint's ______ method must be used to provide access to the advised method, so that it can execute.
  - Option 1 : invoke()
  - Option 2 : continue()
  - Option 3 : proceed()
  - Option 4 : next()

- Question 4 : <aop:aspectj-autoproxy/> automatically proxies beans whose methods match the pointcuts defined with @Pointcut annotations in @Aspect-annotated beans.
  - Option 1 : True
  - Option 1 : False

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

31



Review Questions

S	T	A	R	G	E	T	P
J	P	D	B	C	A	A	R
N	R	V	I	U	T	S	O
P	O	I	N	T	C	U	T
R	C	C	V	W	E	Y	E
O	E	E	O	Q	P	F	E
X	E	T	K	G	S	M	G
Y	D	E	E	G	A	H	E

There are some AOP terms hidden in the above table. The words may be across, down or even upside down.
the cross-cutting functionality being implemented
the actual implementation of aspect that’s advising your application of new behavior
defines at what joinpoints, an advice should be applied
the class being advised
the object created after applying advise to the target
A method of the MethodInterceptor interface
Helps to go to the next interceptor in the chain