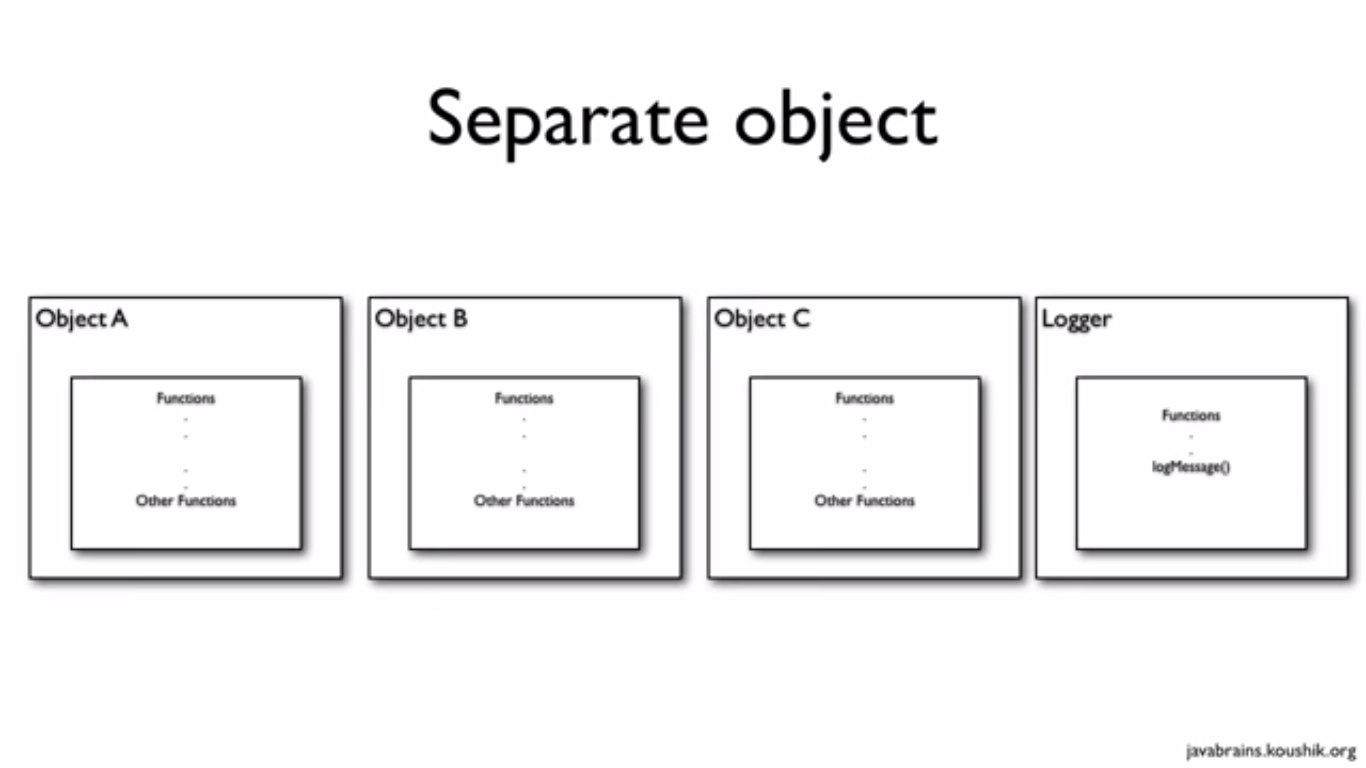
**Spring Notes**:

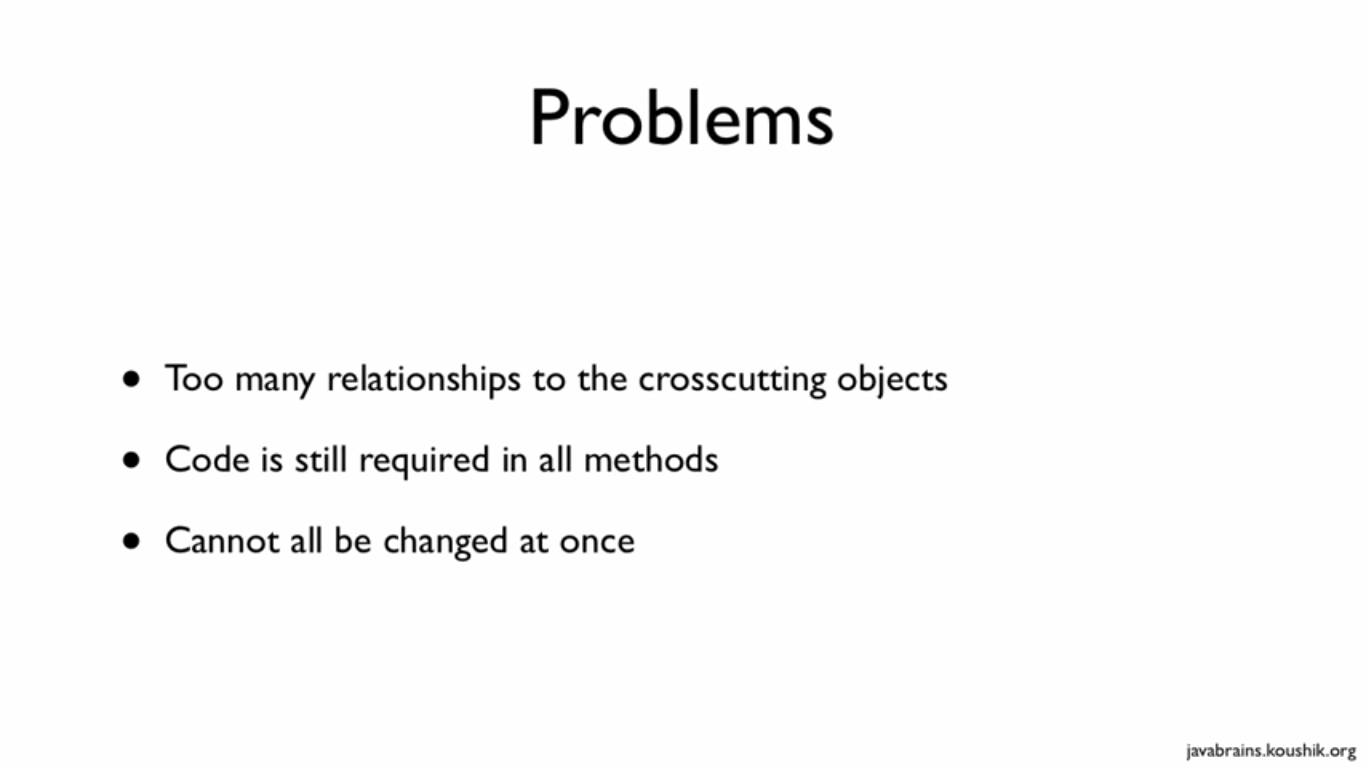
1. **Functional Programing**

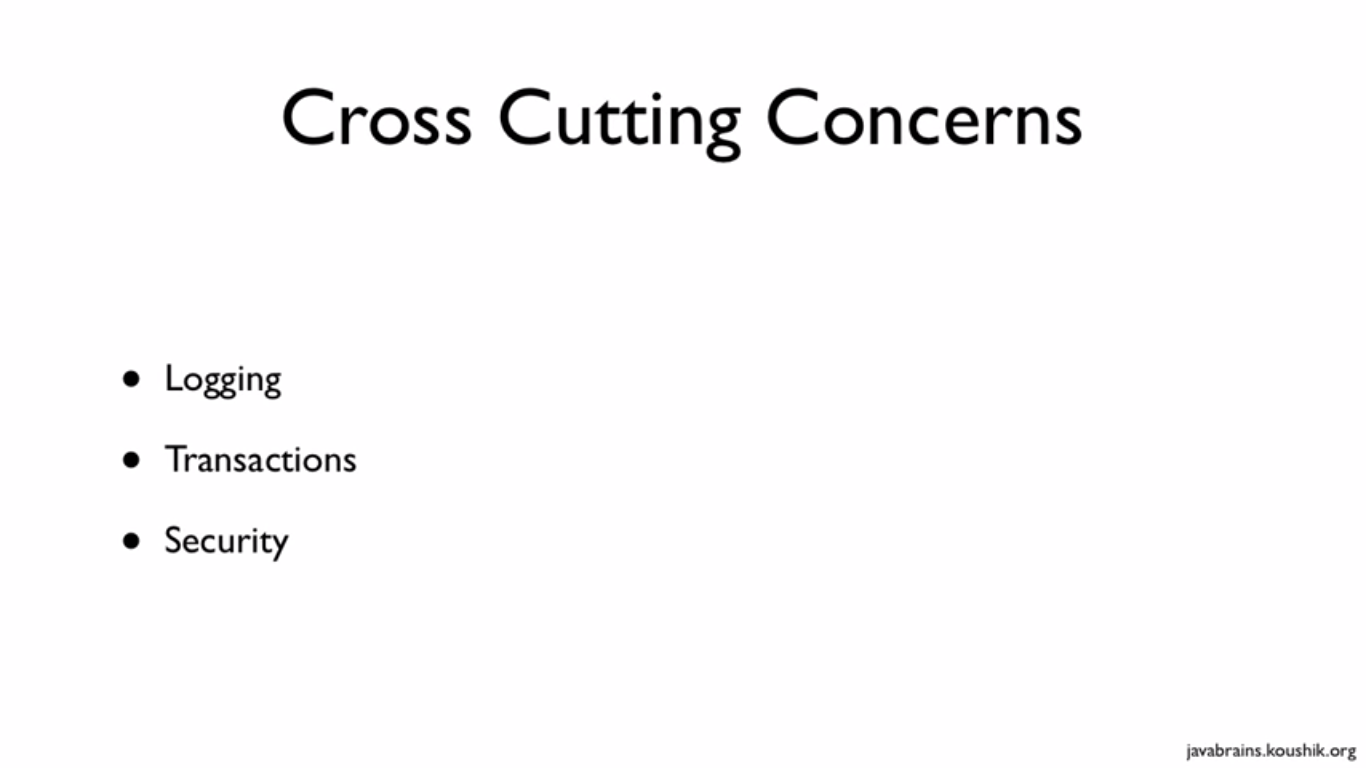
Fun A 🡪 Fun B – Fun C

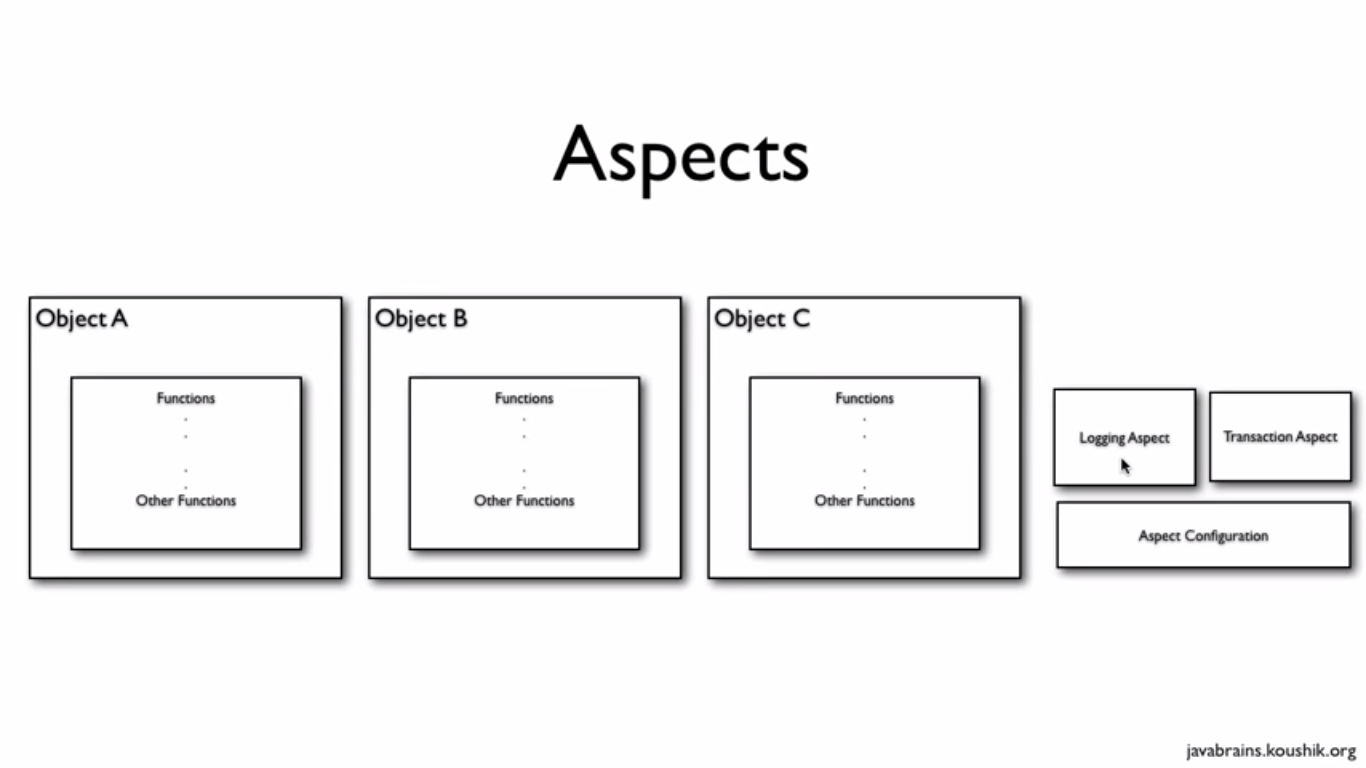
Complex design is involved, graph would become very complex.

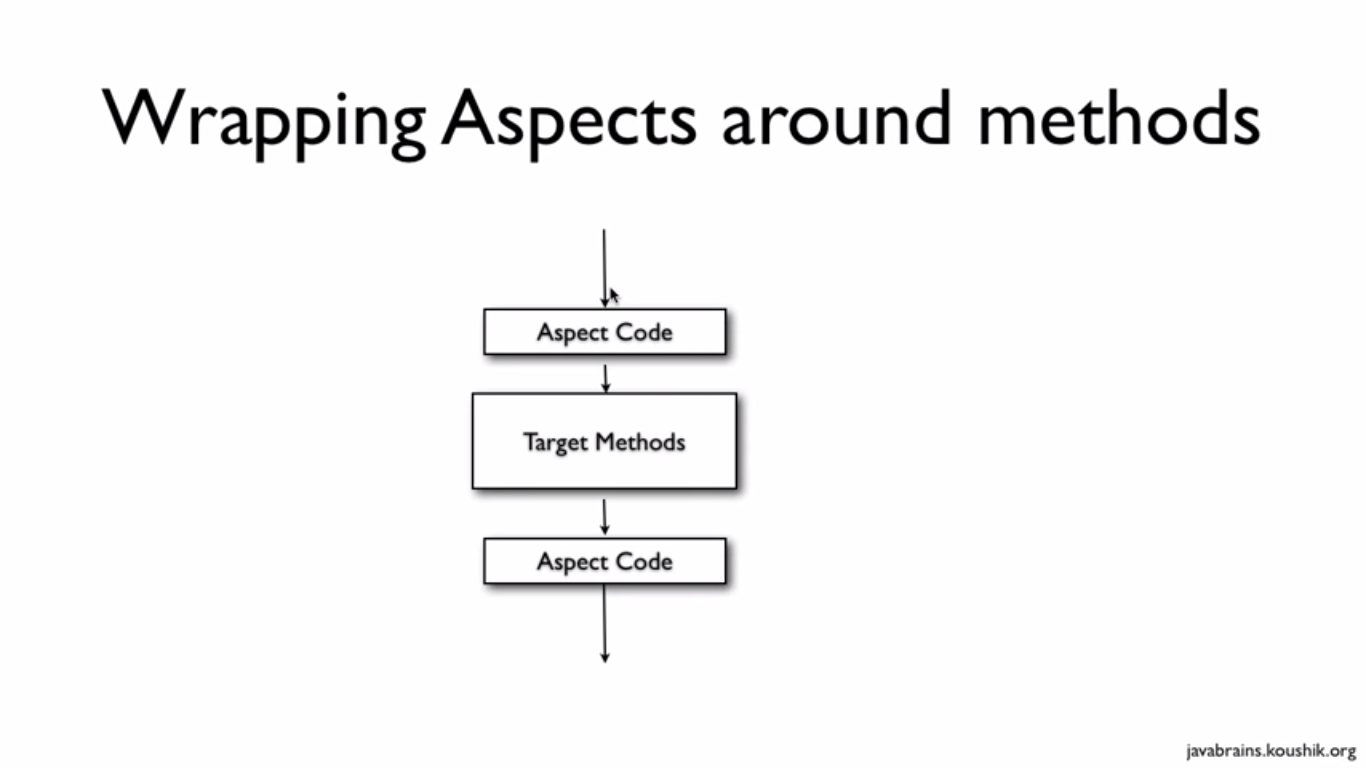
1. **Object oriented programing**
   1. Each object is Individual entity
   2. All the entities can build bigger systems.
   3. Object can be separated for different concerns to achieve the functionality (Each object or few objects together can treated as a concern)
   4. All the concerns related to functionality are **Core concerns**.
   5. But not all the tasks are related to functionality and can be divided into objects (Logging)
   6. Ex –logging () can be called from different object explicitly to log the info.
   7. We can call this logging method in below possible ways.
      1. Create a Logging object and call the logging () method in each object.
      2. Static way of calling
      3. Inherit the Logging Class in each object.
   8. Logging is not a business object and so we call it as a **cross is cutting** object (system wide concerns) which are not a core concerns.
   9. Below are the problems with these cross cutting concerns.
      1. To many relationships to crosscutting concerns from core concerns
      2. Even though we separate the Logging class, still we need to call logging() method explicitly
      3. If we want to change the calling places of logging algorithm them we need to modify the code to effect the changes.

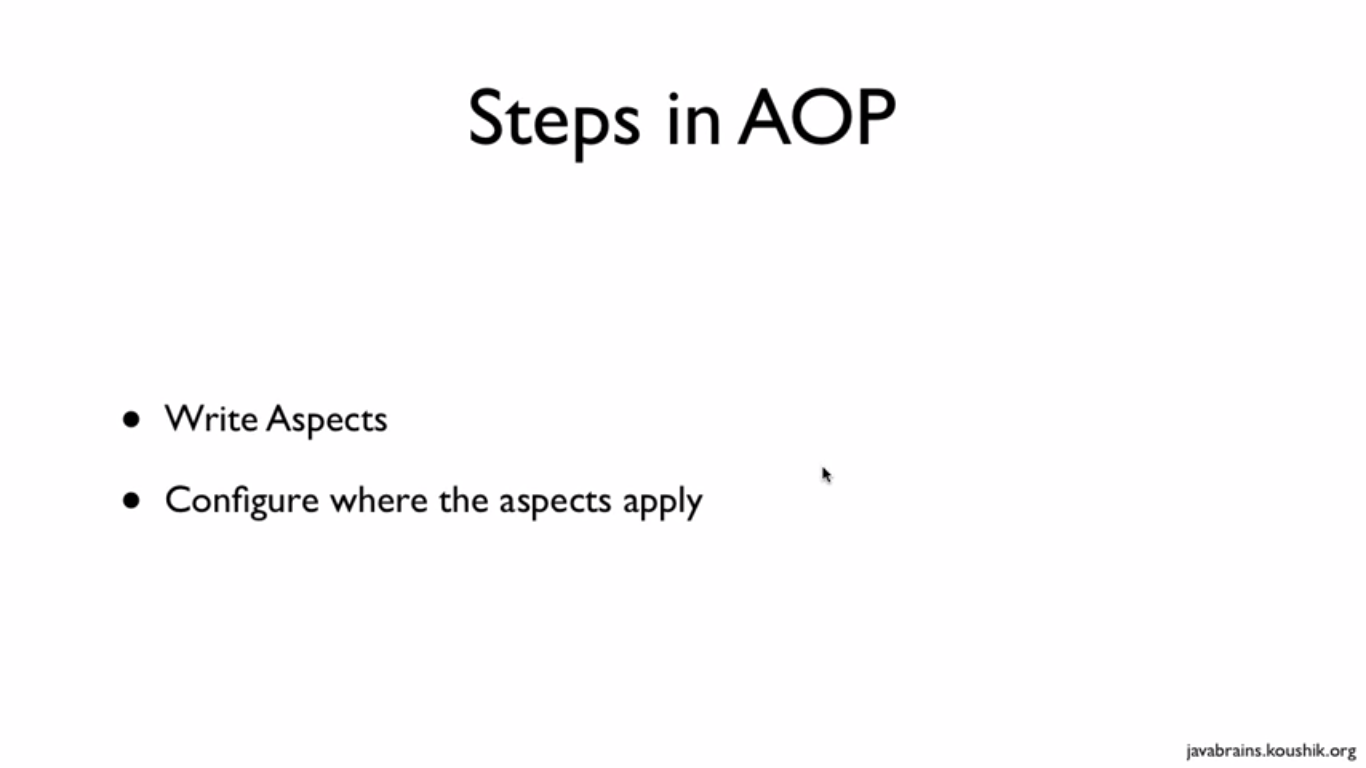












1. **Aspect oriented programing Introduction**
   1. Aspect is a class with special privileges

Ex –

LoggingAspect

TransactionAspect

SecurityAspect.

* 1. We don’t reference and call the Aspects in our regular classes.
  2. To achieve this we define Aspect as configuration in the form of XML or Annotations.
  3. Spring framework will read all the aspects at the time of initialization and create Proxies for us to call the cross cutting concerns at run time.
  4. Configuration can be changed at any time so that calling place of Advices can be changed easily without changing the code.

1. **Steps in Aspect oriented programing**
   1. Writing an Aspect.
   2. Writing the configurations to apply these aspects.
2. **Development environment for AOP**
   1. The links for the dependencies are:   
      AspectJ:

http://www.eclipse.org/aspectj/downloads.php   
AOP Alliance:

http://aopalliance.sourceforge.net/   
CGILIB:

http://cglib.sourceforge.net/   
Spring 3 ASM:

http://asm.ow2.org/ 

* 1. Add core spring library
  2. Add a new library in eclipse by naming it as “AspectJ”
  3. In case of AOP we need below dependency libraries
     1. Core aspect distribution, download and install it.
     2. With in the lib folder we can find below jar files
        1. **Aspectjrt.jar**
        2. aspectjViwere.jar
     3. AOP Aliance .jar
     4. Cgilib.jar
     5. Asm.jar

1. **Developing Aspects**
   1. To develop AOP app we need to write configuration (spring.xml) file.
   2. Search of <aspect-AutoProxy> in the available spring projects.
   3. To get “aop” Namespace information we need to search existing projects.
   4. Need to include Namespace in springs.xml
   5. Aspect is AOP terminology for a class.
   6. Aspect is class with some special privileges
   7. Aspect can have number of Advices and Pointcuts
   8. Writing an Aspect-
   9. To make a class as Aspect use below annotation –

**@aspect**

**Public class LoggingAspect{…..}**

* 1. Advice is AOP terminology for a method.
  2. To make a method as Advice need have an annotation like below –

**@Before("execution(public String getName)")**

**public void secondLogginAdvice(){ …}**

* 1. In the above statement, execution(public String getName)is an **Inline expressions**.
  2. Now we are ready with Aspect which has an Advice.
  3. Only thing left is – we need to tell Spring that we are ready to use AspectJ programming.
  4. To do that we need to use below tag in spring.xml –
     1. **<aop:aspectj-autoproxy>**
  5. Once spring framework reads this tag from spring.xml, and if it finds any Aspects then it will **create proxies** for them to apply advices as configured.
  6. Above Advice will be executed to all the getName() methods in all the classes.
  7. If you want to specify the class/object that we want to apply then need to change the expression like below –
     1. **@Before("execution(public String com.aop.model.Circle.getName()")**
     2. In this case Advice will only execute for Circle. Not for any other classes
  8. Below 2 points are now need to apply our advices as per our convinient –
     1. What if want to apply Advice to different method at different point (multiple methods) as per the configuration.
     2. At one point of execution in code, how can I apply different advices.

1. **How can we apply and Advice to different methods at different points (i.e - to multiple methods) as per the configuration.**
   1. Using wild card expressions
   2. To apply an advice to all getters we need to use below expression

**@Before("execution(public \* get\*()")**

* 1. Above expression will apply to
     1. getCircle()
     2. getName()
  2. Earlier expression is only applies to no argument getter methods, if want to apply an advice to all other getter methods irrespective of arguments then below is the expression -

**@Before("execution(public \* get\*(..)")**

* 1. Wild card meanings –
     1. “..” – 0 or more arguments
     2. “\*” – 1 or more
     3. () – no arguments
     4. Even we can specify a specific argument in the braces like - (String).
  2. We can use package name to have more control, to apply advice to all the classes in model package.

**@Before("execution(public \* com.aop.model.\*.get\*(..)")**

* 1. **Execution** is used to write the wild expressions on methods.
  2. **This expression can be written by using below params**
     1. Access specifiers – ex - Public /private/\* -
     2. Return type – any object/void/\*
     3. Method name – can specify as per our requirement.

1. ***PointCut and wild card expressions* - At one point of execution in the code, how can I apply different advices (Appling multiple advices to a method).**
   1. For this add one more advice to LoggingAspect.

@Before("execution(public \* get\*())")

**public** **void** secondLoggingAdvice(){

System.*out*.println("Second before Logging advice"); }

* 1. Add before advice to the secondLoggingAdvice in the same way as first.
  2. So for all getter () methods these 2 advices will be executed.
  3. **No the problem is –** If we have lot of advices like above then we are repeating the above expressions to all the advices.
  4. To solve this, we have a shortcut called as **PointCut**.
  5. PointCut is an AOP terminology.
  6. **All the points in the code execution of the code to cutten to execute an advice is nothing but Pointcut (Typically methods) .**

**If you have code like below –**

**someMethod() {**

**setName();**

**String x = “abc”**

**x = x+”test”**

**getName(); - This is a PointCut of the Code flow**

**prinln(x)**

**}**

**Point Cuts usually defined by using “Wild card expressions”**

* + 1. Here like getName()
    2. getCricle()
  1. All the points in the code execution of our code (those point will be cutten to execute the advice) where you want this advice method to execute
  2. Here getName(), getCircle() are different Pointcuts from where advices will be executed.
  3. **Below is the code to write a pointcut**

@Pointcut("execution(public String getName())")

public void allGetters(){

}

* 1. Here **allGetterMethod is dummy method** which will not do anything But it holdsing PointCut expression.
  2. Now we can remove lengthy wildcard expression for all the advices and can be replace it with by using Pointcut.
  3. Using **Pointcut** (we use pointcut name) in place of **inline Expressions**

**@Before("allGetters()")**

public void loggingAdvice1(){

System.out.println("Second before Logging advice");

}

**@Before("allGetters()")**

public void secondLogginAdvice1(){

System.out.println("Second before Logging advice");

}

* 1. Two ways –
     1. If **one advice** to be applied for different points/different point cuts – use wild cards.
     2. One point cut expression for **multiple Advices** – then write a pointcut expression and apply it to multiple methods

**Few more Pointcut expression.**

* 1. Logging advice should apply to all other methods in only Circle class.
  2. For this define a pointcut in below way.

**@Pointcut("execution(\* \* com.aop.Circle.\*(..))")**

public void allCircleMethods(){}

* 1. Call the above pointCut at LogginAdvice method.
  2. Applying the Logging advice to all the methods in circle class

**@Pointcut("execution(\* \* com.aop.Circle.\*(..))")**

public void allCircleMethods(){..}

* 1. **within expression applies to classes, whereas**
  2. **execution expression applies to method’s**
  3. other way of applying the Logging advice to all the methods in a circle class

**@Pointcut("within(com.aop.Circle)")**

public void allMethodsInCircle1(){}

* 1. For all the classes in the package

**@Pointcut("within(com.aop.\*)")**

public void allMethodsInCircle2(){}

* 1. For all the classes in the sub packages also

**@Pointcut("within(com.aop..\*)")**

**public void allMethodsInCircle3(){}**

* 1. **args** stands for arguments, this expression checks for arguments ex - Circle or String as argument in the method.

If it match’s the arguments for a method then advice will be applied those methods

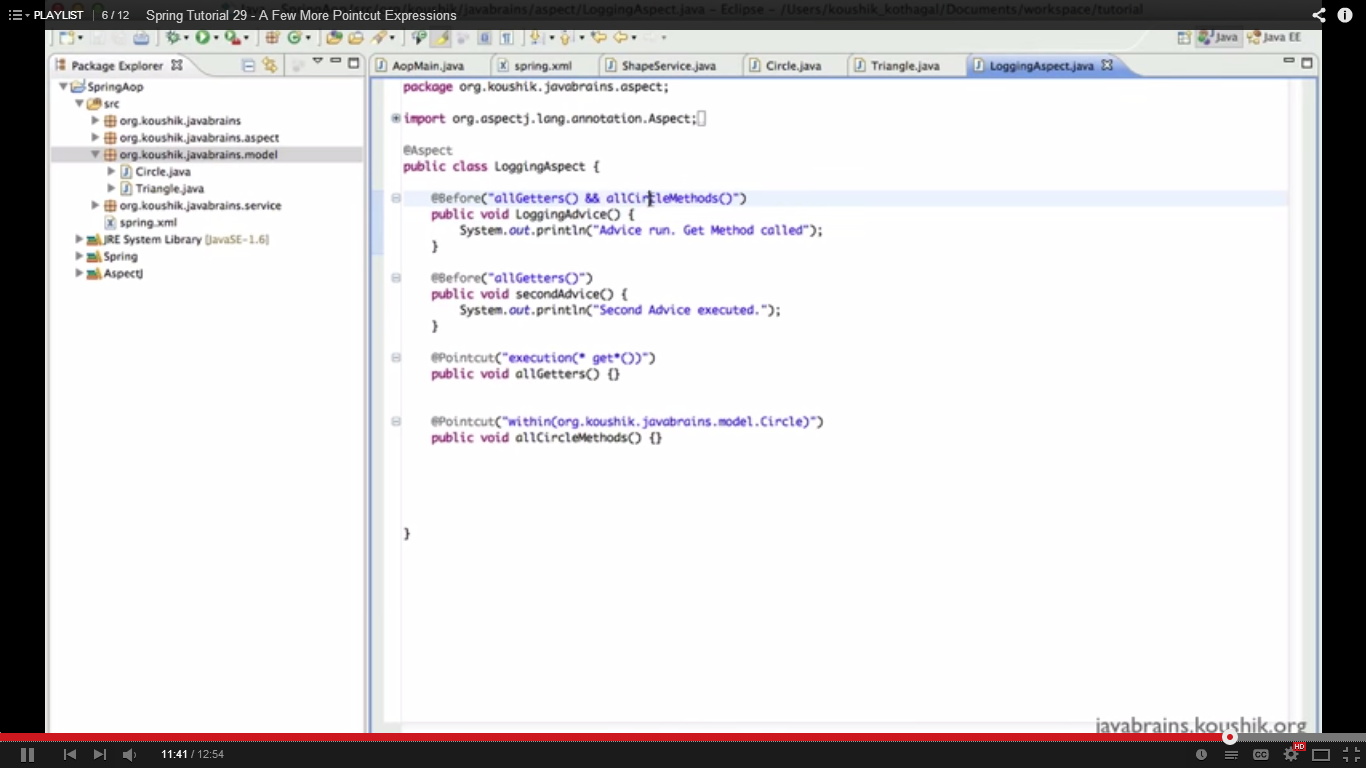
* 1. Other expressions are **Target and this**.
  2. Combining the pointcut expression by using operators, which is a best practice to avoid writing multiple Pointcuts if we have already available combination of Pointcuts.

**@Before("allGetters() && allMethodsInCircle1()")**

**public void combinePointCutAdvice(){**

**System.out.println("combinePointCutAdvice");**

**}**



1. ***JoinPoint’s* and Advice Arguments**
   1. There is some problem with when applying the advice to different methods by using PointCut expressions.
   2. For example above expression applies to all the methods which are in Circle class.
   3. But if the advice is intended to apply only to getter methods still it will applies to other methods like setters.
   4. So Is it possible to know the details of calling method name inside of an ADVICE method to implement the ADVICE logic based on calling method name? so that advice can be applied as per logic written in advice method.
   5. If we know calling method name in Advice then it’s good to have some logic around in advice based on the called method
   6. It’s good to have a provision to inspect target Object (calling object of the calling method) in advice.
   7. **JoinPoint is a AOP terminology**
   8. JoinPoint is all the places in our code that we apply advices.
   9. **In Spring advices are only applied to methods which are nothing but JointPoints.**
   10. **So Method itself is a Joinpoint in Spring, Jointpoint is having the information about Method**
   11. **But Pointcut is an expression which expresses multiple execution points in the code flow to execute Advice, Pointcut tells when to execute the Advice.**
   12. **But in AspectJ advices can be applied to member variables and methods.**
   13. We can get JoinPoint as the argument in advice method signature.
   14. So by using that JoinPoint argument we will get the calling method and calling object information.
   15. We can call the methods on calling object in advice by using JoinPoint argument.
   16. Advice method can accept JoinPoint as argument and it can accept other arguments if it’s using args in its Pointcut expression.
   17. Code for JoinPoint as argument

@Before("allMethodsInCircle1()")

public void joinPointAdvice(**JoinPoint jp**){

System.out.println(**jp**.toString());

System.out.println("combinePointCutAdvice");

}

* 1. **Code with other arguments in advice method**
  2. All the method that take String as argument

@Before("args(**String**)")

public void joinPointAdvice(){

System.out.println("combinePointCutAdvice");

}

* 1. All the methods that take name as argument and type is String, here we are using the name as an argument in the advice.

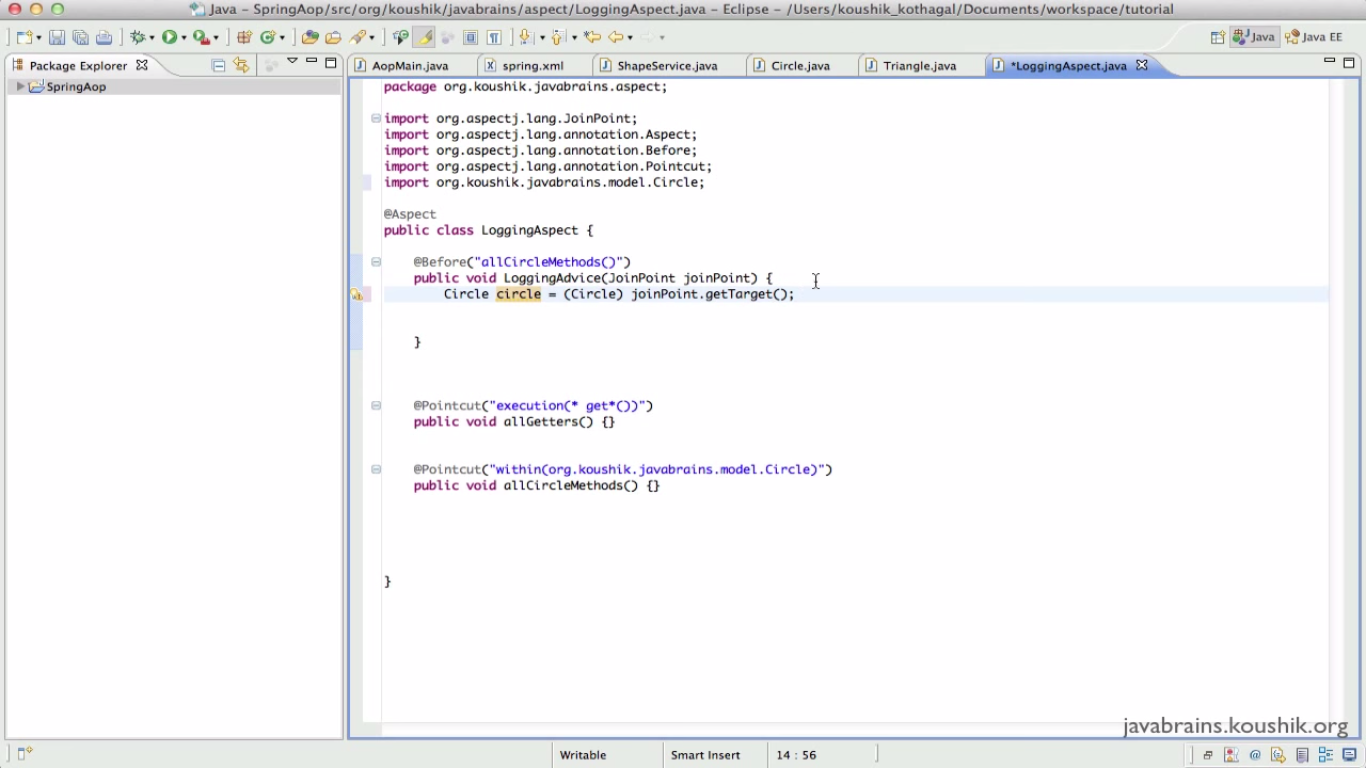
@Before("args(**name**)")

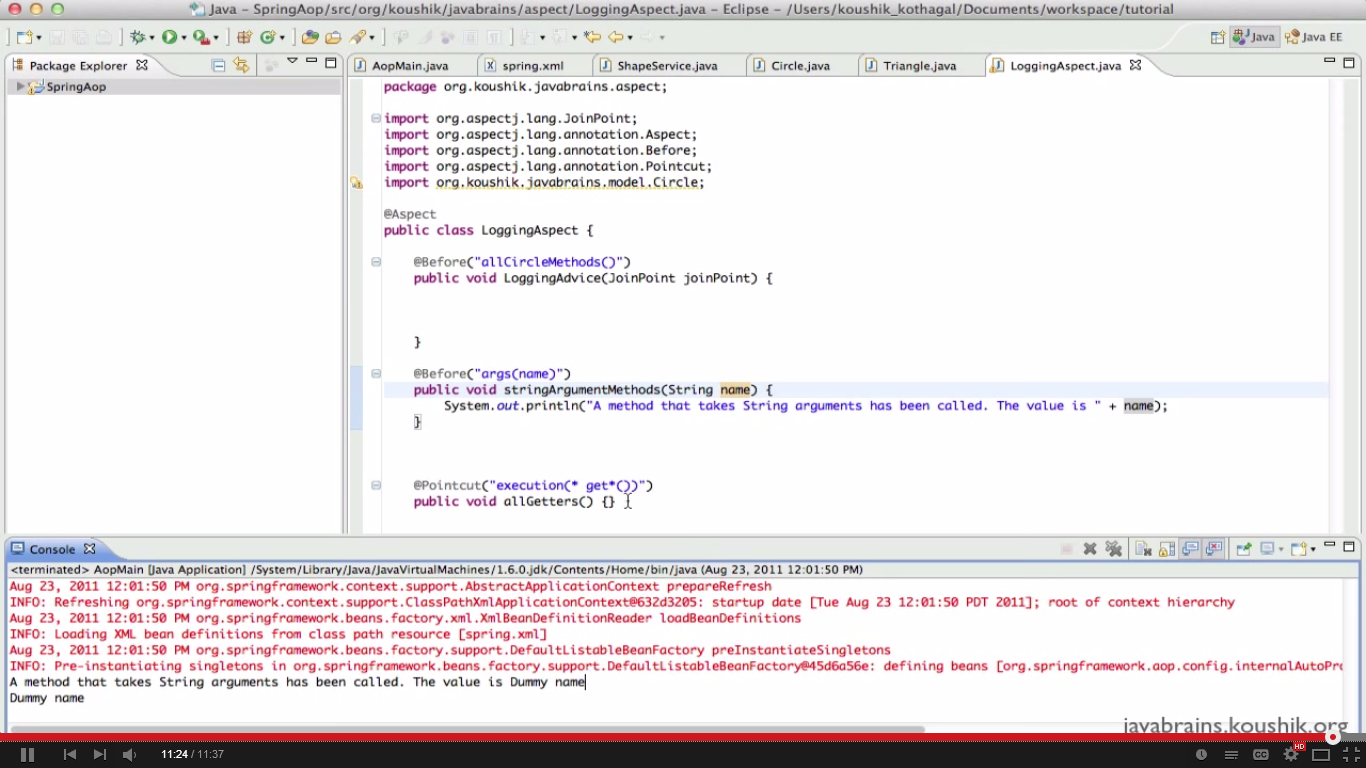
public void joinPointAdvice(**String name**){

System.out.println("combinePointCutAdvice");

}

* 1. We can modify and do some logic around on these arguments inside the advice.





1. ***After advice***
   1. Advice executes only when they called programmatically not at the time of spring container initialization.
   2. Advices will not be executed when spring core container initialization

**For example –** if setter methods are configured in Bean definition and advices also configured on those methods then at the time of bean initialization advices will not be executed.

* 1. To execute an advice after the method execution, we have an annotation called – **After**.
  2. After Advice will execute even we have an exception occurred in method execution. It’s like a finally block/method.

1. ***AfterReturning*** 
   1. This advice will execute only when target method executes successfully.
   2. If we have any exception in the target method this advice will not be executed..
   3. This method can have a target method return value as an argument in adive method.
2. ***AfterThrowing***
   1. This advice will execute only when target method throws exception and not on successfully return.
   2. This method can have an exception that was throw in method execution as an argument in advice method.
3. ***Returning values of in Afterretuning***
   1. We can get the return values of a method in advice, where we can change the object values in the return type.
   2. We need to have retruningType in the AfterReturing
   3. Same way we can capture the Exception details that are occurred in the method by using AfterThrowing exceptions.
4. ***Around Advice***
   1. Advice will run before and after the advice
   2. It will take ProceedingJoinPoint as an argument
   3. Inside the advice call procced() method to execute the actual method.
   4. Before and after the procced() method we can have logic in the advice logic.
   5. Proceed method can have return type form target method
5. ***Method Naming conventions***
   1. Its best practice to have better naming conventions to our methods in the classes.
   2. In that way we can write Pointcut expressions easily.
   3. Otherwise its bit difficult if we don’t find an expression to satisfy multiple method to apply and advice.
6. ***Custom Annotations –***
   1. If we have lot of method which should apply an advice and if we don’t create/have specific pattern (Pointcut expression) to define all these method. Then we can go with custom annotations
   2. Even we don’t aware of method which should be applied in feature.
   3. To identify whether a method is having advice to be applied or not, we should check all the aspects and Pointcut expression whether the expression is matching are not for a particular method.
   4. So, It’s possible to annotate a method with Advice by using custom annotation/advices
   5. Create an Annotation – say Loggable
   6. Use that annotation in Pointcut with expression

**@Before("@Annotation(com.aop.Loggable)")**

public void customAnnotateAdvice(**String name**){

System.out.println("combinePointCutAdvice");

}

* 1. Annotate this pointcut for the methods that they want to apply this custom advice.

@Loggable

Public void getCircle(){  
}

* 1. In the above way Loggable advice will be applied to getCircle method.

1. **XML vs Annotations –**
   1. Legacy code might have used XML’s
      1. Mix of xml
      2. Annotation
   2. **Annotation are simpler and readable to use**
   3. If you treat these aspects as configuration rather than functionality
   4. Aspects can be used based on **configuration or functionality**
   5. **Transaction aspect is functionality – In this case we can use annotation**
2. XML configuration – for debugging logic during the life time of applications and nothing to do with functionality then go with --- xml configuration – To avoid building all the application.