Important definations:

Join points: are the options on the menu and
pointcuts: are the items you select

Aspect =Advices + Point Cut

--> encloses advices (Extra logic) and
--> point cut(Where to apply extra logic

What is the need?

```
public class Magician {
	public void doMagic() {
		System.out.println("do magic abra ka dabra!!!");
	}
}
```

```
class Audiance{
	public void clapping(){
		System.out.println("maza aa gaya");
	}
}
```

AspectJ Annotations

@Aspect, @Before, @After, @AfterReturning, @AfterThrowing, @Around, @PointCut

How it works?

It is new style make use of AspectJ annotations

How it works?
---
1. Create a project, add spring jars,<aop:aspectj-autoproxy />
2. create an target
3. write aspect which encloses advices
4. Enable AutoProxy feature in XML
5. Client invokes target- but the call goes to proxy.

beans.xml

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

```
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                                        http://www.springframework.org/schema/beans/spring-beans-
4.0.xsd
                                        http://www.springframework.org/schema/context
                http://www.springframework.org/schema/context/spring-context-4.0.xsd
                http://www.springframework.org/schema/aop
                http://www.springframework.org/schema/aop/spring-aop-4.0.xsd">

        <aop:aspectj-autoproxy />
        <bean id="m" class="com.Magician"/>
        <!-- Aspect -->
        <bean id="logginaspect" class="com.AudienceAdvice" />

</beans>
```

Target

```
----------
public class Magician {

        public void doMagic() {
                System.out.println("do magic abra ka dabra!!!");
        }
}
```

Aspact

```
--------
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;


//Aspect=point cut+ extra logic to be applied
@Aspect
public class AudienceAdvice {
        @Before("execution(public void doMagic())")
        public void clapping(JoinPoint joinpoint){
                System.out.println("clap clap.....");
                System.out.println("information of target method :"+joinpoint.getSignature().getName());
        }
}
```

Tester

```
--------

        Magician magician = (Magician) context.getBean("m");
        magician.doMagic();
```

More examples:

```
--------------
@Aspect
public class AudienceAdvice2 {
```

```
	@Before("execution(* *.*(..))")
	public void myBeforeMethod(JoinPoint j){
		System.out.println("called Before method
nameOfTheMethod:"+j.getSignature().getName());
	}
	//will execute whether their exception or not
	@After("execution(* *.*(..))")
	public void myAfterMethod(JoinPoint j){
		System.out.println("called After method nameOfTheMethod:"+j.getSignature().getName());
	}

	//only executed if method successfully returns!
	//we can get return value

	@AfterReturning("execution(* *.*(..))")
	public void myAfterReturningMethod(JoinPoint j){
		System.out.println("called after method return
nameOfTheMethod:"+j.getSignature().getName());
	}


	@AfterThrowing("execution(* *.*(..))")
	public void myAfterThrowingMethod(JoinPoint j){
		System.out.println("called after method throws an exception
nameOfTheMethod:"+j.getSignature().getName());
	}
}
```

AroundAdvice

```
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class AudienceAdvice {
	@Around("execution(* *.*(..))")
	public Object myAround(ProceedingJoinPoint pjp) throws Throwable {
		System.out.println("before logic called");

		Object result = pjp.proceed();

		System.out.println("after logic called");
		return result;

	}
}
```

creating an ccc for logging imp business method:

```
@Retention(RetentionPolicy.RUNTIME)
```

```
@Target(ElementType.METHOD)
public @interface Loggable {

}
```

```
@Component
public class Magician {
	@Loggable
	public void doMagic() {
		System.out.println("do magic abra ka dabra!!!");

	}
}
```

```
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.reflect.MethodSignature;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;
@Component
@Aspect
public class MethodLogger {
	private static final Logger logger=LoggerFactory.getLogger(MethodLogger.class);

  @Around("@annotation(Loggable)")
  public Object around(ProceedingJoinPoint point) throws Throwable {
    long start = System.currentTimeMillis();
    Object result = point.proceed();
    logger.info("start "+MethodSignature.class.cast(point.getSignature()).getMethod().getName()+" is
called"+" takes " +(System.currentTimeMillis() - start));
    return result;
  }
}
```

now testing:

```
ApplicationContext ctx=new ClassPathXmlApplicationContext("beans.xml");

Magician magician = (Magician) ctx.getBean("magician");
magician.doMagic();
```

Applying logging to bankapplication:

Pointcut and wildcard expression examples

---

1. @Before("execution(public String getName())")

Before execution of any getName() method in configured bean.

2. @Before("execution(public String com.demo.model.Circle.getName())")

Before execution of any getName() method of com.demo.model.Circle class

3. WildCard: What if i want to apply advice to all getter wheter it is getName() or getAddress()?

execution(public String get*()) :Work for all getter returning string

execution(public * get*()) :Work for all getter returning anything

execution(* get*()) :Work for all getter returning anything no matter wheter it is public or default

execution(* get*(*)) :Work for all getter returning anything, accepting anything (alleat one argument, one-many) no matter wheter it is public or default

execution(* get*(..)) :Work for all getter returning anything, accepting anything (zero-many argument)
no matter wheter it is public or default

execution(* com.demo.model.*.get*()) :Work for all getter of model package returning anything, accepting anything
(zero-many argument) no matter wheter it is public or default

defining alies for injection point? Save typing and less errorprone
---

let say i need to execute getter() advice for two advice

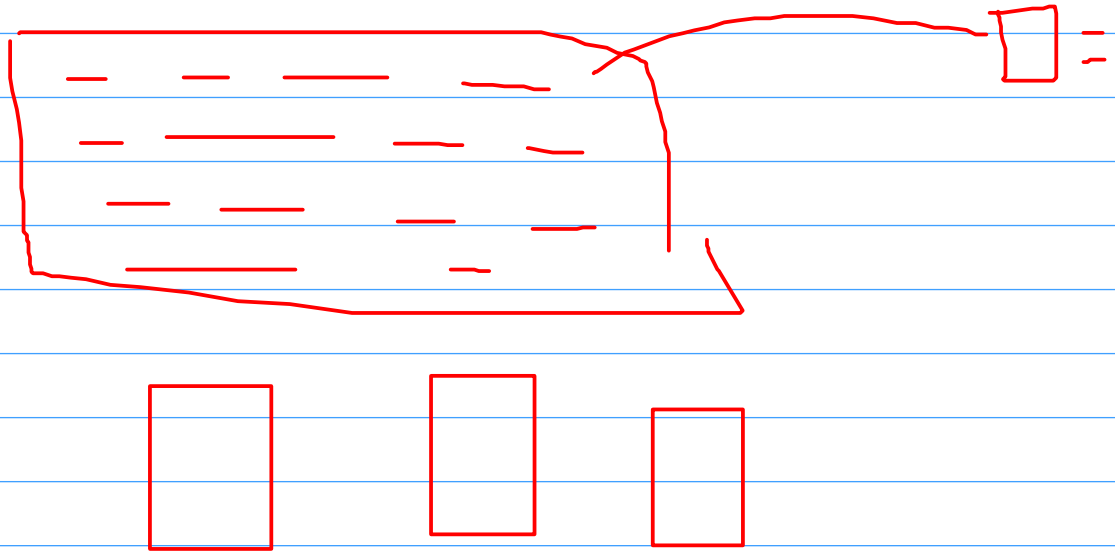
```
@Before("execution( * com.demo.model.*.get*())")
public void loggingAdvice() {
    System.out.println("Advice run. Get method is called");
}

@Before("execution( * com.demo.model.*.get*())")
public void anotherLoggingAdvice() {
    System.out.println("Another Advice run. Get method is called");
}
```

Now we need to repeat @Before("execution(* com.demo.model.*.get*())")

What to do?

40(MCQ) :60(lab)

20



Use pointCut ? Define a pointcut and apply

```
@Pointcut("execution( * get*())")
        public void allGetters(){}
```

allGetter() is an dummy method....Now apply it as:

```
@Before("allGetters()")
public void loggingAdvice()
{
    System.out.println("Advice run. Get method is called");
}
```

More on Pointcut expression & best practices

Let i required to apply logging advice to all the method of Circle class

Now how to do it?

```
execution( * * com.demo.model.Circle.*(..))
```

Looks complicated? Use within

```
@Pointcut("within(com.demo.model.Circle)")
        public void allCircleMethods(){}
```

Now apply it like;

```
@Before("allCircleMethods()")
public void anotherLoggingAdvice()
{
    System.out.println("Another Advice run. Get method is called");
}
```

```
@Pointcut("within(com.demo.model.*)")
public void allCircleMethods(){}
```

for all the classes in model package

```
@Pointcut("within(com.demo.model..*)")
public void allCircleMethods(){}
```

for all the classes in model package and its subpackages

using args

ie applying advice for methods those are accepting
specific method argument.

```
@Pointcut("args(com.demo.model.Circle)")
        public void allCircleMethods2(){}
```

ie for all the methods that accept argument that take Circle as argument.

Combining two pointcut

```
@Pointcut("execution( * get*())")
        public void allGetters(){}

@Pointcut("args(com.demo.model.Circle)")
public void allCircleMethods(){}
```

we can say:

```
@Before("allGetters()|| allCircleMethods()")
    public void loggingAdvice()
    {
        System.out.println("Advice run. Get method is called");
    }

    @Before("allGetters() && allCircleMethods()")
    public void loggingAdvice()
    {
        System.out.println("Advice run. Get method is called");
    }
```

Joint point and method arguments

```
@Pointcut("args(com.demo.model.Circle)")
        public void allMethodsAcceptingCircleAsArgument(){}
```

http://stackoverflow.com/questions/15447397/spring-aop-whats-the-difference-between-joinpoint-and-pointcut
http://www.yegor256.com/2014/06/01/aop-aspectj-java-method-logging.html
http://javabeat.net/annotations-in-java-5-0/