



April 2024, IPT Course  
Introduction to Spring

# SpEL, AOP & Events in Spring

Trayan Iliev  
[tiliev@iproduct.org](mailto:tiliev@iproduct.org)  
<http://iproduct.org>

# Agenda for This Session

- ❖ Spring Expression Language (SpEL)
- ❖ SpEL Operators, Examples, using Application Context
- ❖ Selection and Projection Examples. Variables
- ❖ Type, Constructors, Lists and Maps
- ❖ Aspect Oriented Programming
- ❖ Basic AOP Concepts
- ❖ Types of Advices and Pointcuts
- ❖ Event Publishing and Listening
- ❖ Asynchronous Event Handling
- ❖ Advanced AOP – Introductions and ITD

# Where to Find the Demo Code?

Introduction to Spring 5 demos and examples are available @ GitHub:

<https://github.com/iproduct/course-spring5>

# Spring Expression Language (SpEL)

- ❖ **Spring Expression Language (SpEL)** – expression language that supports querying and manipulating an object graph at runtime
- ❖ SpEL is similar (but more powerful) to Unified EL and offers additional features, most notably **method invocation** and **string templating functionality**
- ❖ Combines two main syntaxes:
  - SpEL expressions syntax: **`#{expression}`**
  - Properties reference syntax: **`${property.name}`**
- ❖ Property placeholders cannot contain SpEL expressions, but expressions can contain property references:  
Ex: **`#{$${my.property.name} + 3}`**

# SpEL Operators

- ❖ **Arithmetic** +, -, \*, /, %, ^, div, mod
- ❖ **Relational** <, >, ==, !=, <=, >=, lt, gt, eq, ne, le, ge
- ❖ **Logical** and, or, not, &&, ||, !
- ❖ **Conditional** ?:
- ❖ **Regex** matches



# SpEL Examples

❖ `@Value("#{ (42 div 5) % 3 + 1} * 10")`  
`private double arithmeticExpressionResult;`

```
double value = parser
    .parseExpression("(42 div 5) % 3 + 1) * 10")
    .getValue(Double.class);
```

❖ `@Value("#{2 == 1 + 1}")`  
`private boolean booleanResult;`

```
boolean value = parser.parseExpression("2 == 1 + 1")
    .getValue(Boolean.class);
```

❖ `@Value("#{@provider}")`  
`private ArticleProvider provider;`

# More Examples

```
boolean value2 = parser.parseExpression("'black' <  
'block'").getValue(Boolean.class);
```

```
boolean falseValue = parser  
    .parseExpression("'xyz' instanceof T(Integer)")  
    .getValue(Boolean.class);
```

```
boolean falseValue2 = parser.parseExpression(  
    "not ('5.00' matches '^-?\\d+(\\.\\d{2})?$')")  
    .getValue(Boolean.class);
```

```
boolean trueValue2 = parser.parseExpression(  
    "! ('5.0023' matches '^-?\\d+(\\.\\d{2})?$')")  
    .getValue(Boolean.class);
```

# Application Context Examples

```
AnnotationConfigApplicationContext ctx = new
```

```
    AnnotationConfigApplicationContext("org.iproduct.spring.spel");  
StandardEvaluationContext context =
```

```
    new StandardEvaluationContext();  
context.setBeanResolver(
```

```
    new BeanFactoryResolver(ctx.getBeanFactory()));
```

```
String beanPropWithDefault = parser.parseExpression(  
    "@beanA.message != null ? @beanA.message : 'default'")  
  
    .getValue(context, String.class);
```

```
String beanPropWithDefaultElvis = parser.parseExpression(  
    "@beanA.message ?: 'default message'")
```



# Selection & Projection Examples. Vars

```
List<String> springTitles = parser.parseExpression(
    "@provider.articles.[title matches '.*Spring.*']![title]")
    .getValue(context, List.class);

System.out.println("Spring titles:" + springTitles);

ExpressionParser parser2 = new SpelExpressionParser();
EvaluationContext context2 =
    SimpleEvaluationContext.forReadOnlyDataBinding().build();
context2.setVariable("primes", primes);

// all prime numbers > 10 from the list (using selection ?{...})
// evaluates to [11, 13, 17]
List<Integer> primesGreaterThanTen =
    (List<Integer>) parser.parseExpression(
        "#primes.[#this>10]").getValue(context2);
System.out.println("SpEL:" + primesGreaterThanTen);
```

# Type, Constructors, Lists and Maps

```
CarPark park =
parser.parseExpression("T(org.iproduct.spel.CarPark).create({" +
    "new org.iproduct.spring.spel.Car(\"Opel\", 3, 2550, 2013),\" +
    "new org.iproduct.spring.spel.Car(\"VW\", 2, 1350, 2010),\" +
    "new org.iproduct.spring.spel.Car(\"Audi\", 5, 2200, 2017)\" +
    "}))").getValue(CarPark.class);
EvaluationContext context3 = new StandardEvaluationContext(park);
Expression expression3 = parser.parseExpression("cars.[make]");
List result3 = (List) expression3.getValue(context3);
System.out.println("SpEL:" + result3);

EvaluationContext context4 =
SimpleEvaluationContext.forReadOnlyDataBinding().build();
Map mapOfMaps = (Map) parser
    .parseExpression("{name:{first:'Nikola',last:'Tesla'},\" +
        \"dob:{day:10,month:'July',year:1856}}")
    .getValue(context4);
System.out.println("SpEL:" + mapOfMaps.get("name"))
```

# Expressions in Bean Definitions

## ❖ XML Configuration

```
<bean id="numberGuess" class="org.springframework.samples.NumberGuess">  
  <property name="randomNumber" value="#{ T(java.lang.Math).random() * 10 }"/>  
  <!-- other properties -->  
</bean>
```

```
<bean id="taxCalculator" class="org.springframework.samples.TaxCalculator">  
  <property name="defaultLocale" value="#{ systemProperties['user.region'] }"/>  
  <!-- other properties -->  
</bean>
```

## ❖ Annotation-based configuration

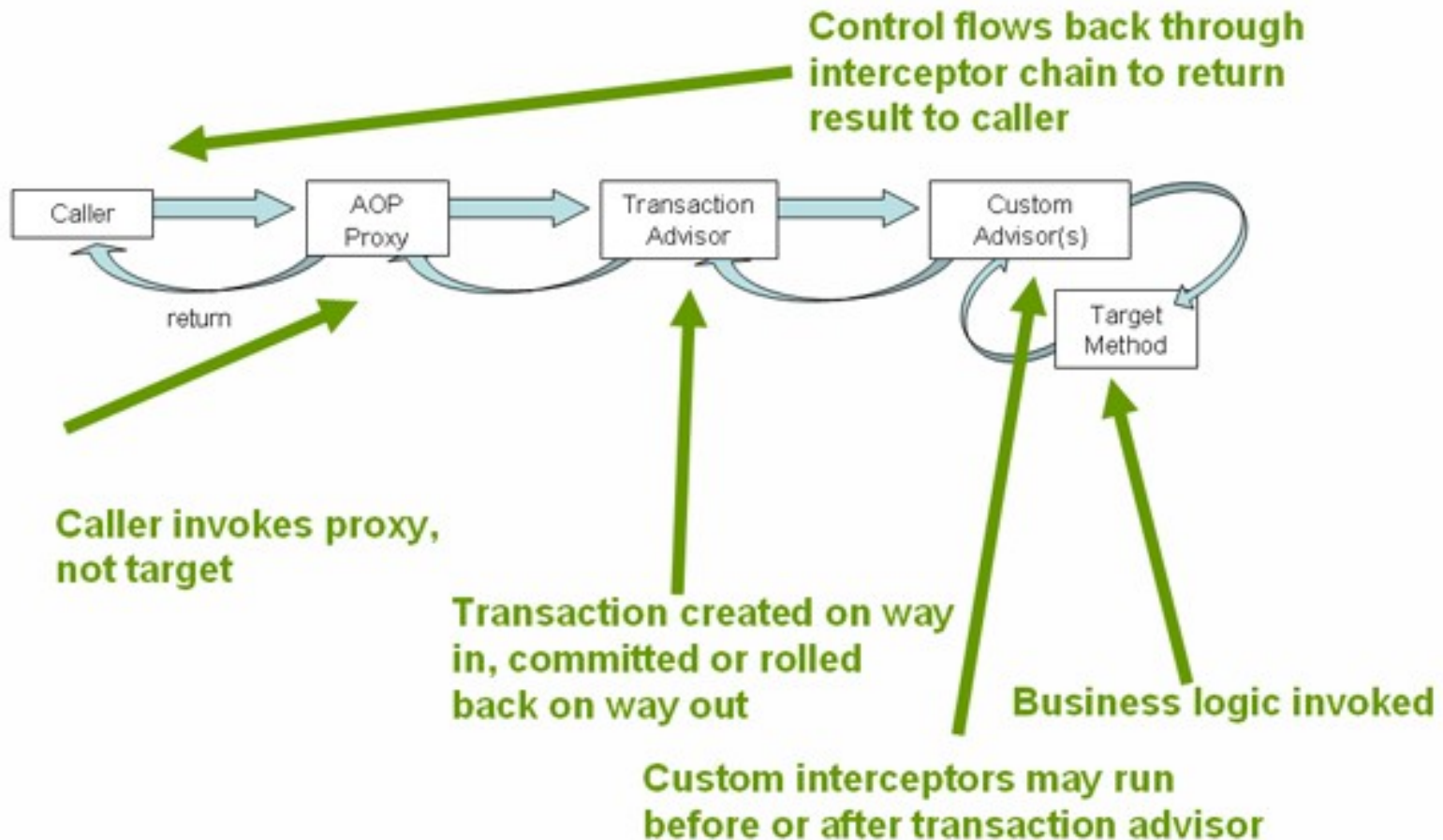
@Autowired

```
public void configure(MovieFinder movieFinder,  
    @Value("#{ systemProperties['user.region'] }") String defaultLocale) { ... }  
}
```

# 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 is the **aspect**.
- ❖ Aspects enable the **modularization of concerns** such as **transaction management** that cut across **multiple types and objects** (crosscutting concerns).
- ❖ AOP is used in the **Spring Framework** to:
  - provide **declarative enterprise services**, especially as a replacement for EJB declarative services – e.g. transaction management.
  - allow users to implement **custom aspects**, complementing their use of OOP with AOP.

# Transactions via AOP Proxies





# Basic AOP Concepts - 1

- ❖ **Aspect** – concern cutting across multiple classes. In Spring AOP, aspects are implemented using regular classes (**schema-based approach**) or regular classes annotated with the **@Aspect** annotation (the **@AspectJ style**).
- ❖ **Join point** – method execution or handling of an exception (always a method execution in Spring AOP).
- ❖ **Advice** – action taken by an aspect at a particular join point. ("**around**", "**before**" and "**after**" advices). (Advice types are discussed below). Modeled as **interceptor** in Spring.
- ❖ **Pointcut** – a predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched. Spring uses the **AspectJ pointcut expression language** by default.



# Basic AOP Concepts - 2

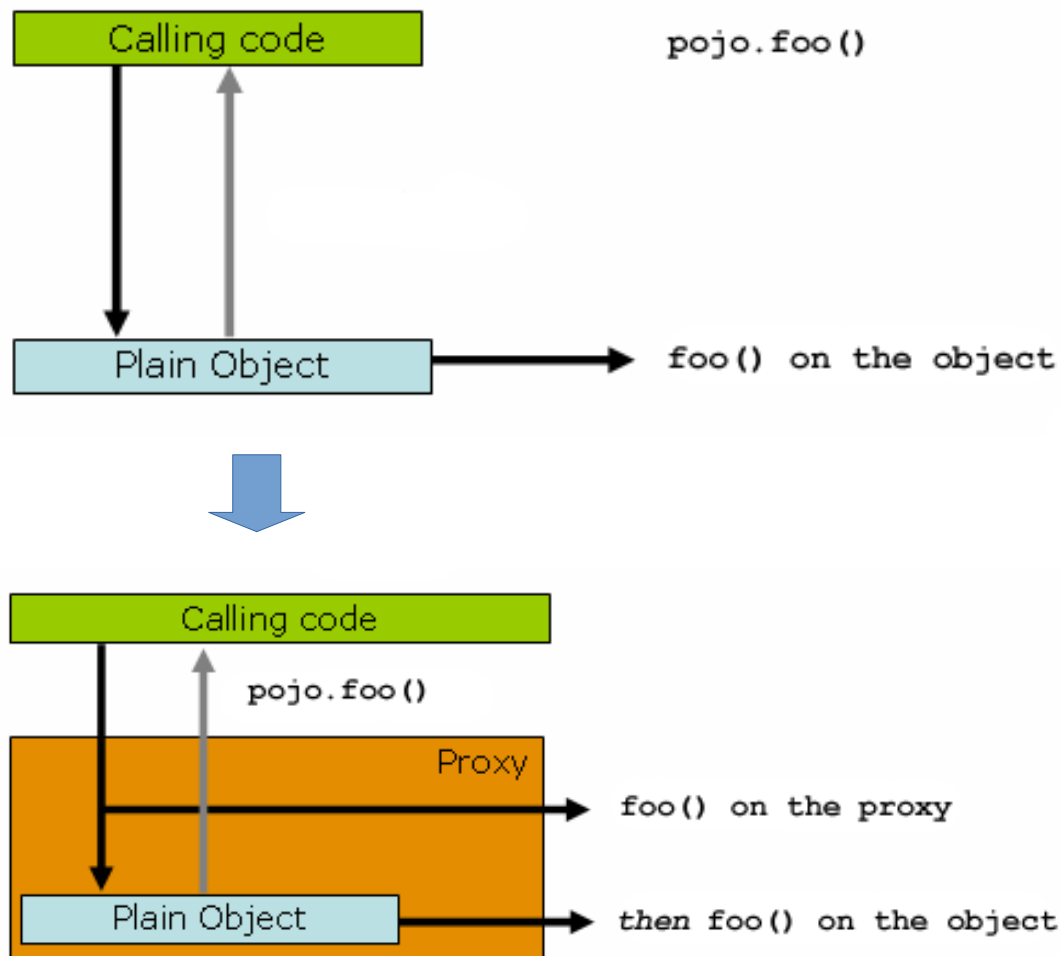
- ❖ **Introduction** (inter-type declaration) – declaring additional methods or fields on behalf of a type. Spring AOP allows to introduce interfaces and corresponding implementations.
- ❖ **Target object** (advised object, proxied object) – object being advised by one or more aspects.
- ❖ **AOP proxy** – an object created by the AOP framework to implement the aspect contracts (advise methods and so on). In Spring: **JDK dynamic proxy** or **CGLIB proxy**.
- ❖ **Weaving** – linking aspects with other application types or objects to create an advised object. This can be done at **compile time** (using the AspectJ compiler), **load time**, or at **runtime**. Spring AOP performs weaving at runtime.

# Types of Advices

- ❖ **Before advice** – executes before a join point, can not prevent proceeding to the join point (unless it throws an exception).
- ❖ **After returning advice** – executed after a join point completes normally: for example, if a method returns without throwing.
- ❖ **After throwing advice** – executed if method throws exception
- ❖ **After (finally) advice** – executed regardless of the means by which a join point exits (normal or exceptional return).
- ❖ **Around advice** – surrounds a join point, can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning own return value or throwing an exception.

# Bean Autoproxying

- ❖ 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.



# Enabling @AspectJ Support

@Configuration

@EnableAspectJAutoProxy

@ComponentScan(basePackages = "org.iproduct.spring.aop")

```
public class AnnotationConfig {
```

```
    @Bean
```

```
    public ArticleProvider provider() {  
        return new MockArticleProvider();  
    }
```

```
    ...
```

```
}
```

❖ In XML config:

```
<aop:aspectj-autoproxy/>
```

# Declaring an Aspect and Pointcut

```
@Component
@Aspect
@Slf4j
public class LoggingAspect {

    @Pointcut("@target(org.springframework.stereotype.Repository) ")
    public void repoMethods() {
    }

    @Before("repoMethods() ")
    public void logMethodCall(JoinPoint jp) {
        String methodName = jp.getSignature().getName();
        log.info("Before invoking: " + methodName);
    }
}
```

# Schema-Based AOP Support

...

```
<bean id="loggingAspect"
```

```
    class="org.iproduct.spring.aop.LoggingAspect" />
```

```
<aop:config>
```

```
    <aop:aspect id="aspects" ref="loggingAspectAspect">
```

```
        <aop:pointcut id="repoMethods" expression=
```

```
            "@target(org.springframework.stereotype.Repository)"/>
```

```
        <aop:before method="logMethodCall"
```

```
            pointcut-ref="repoMethods"/>
```

```
    </aop:aspect>
```

```
</aop:config>
```

...



# Pointcut Expressions

- ❖ **execution** – for matching method execution join points:  
`execution(modifiers-pattern? ret-type-pattern  
declaring-type-pattern?name-pattern (param-pattern)  
throws-pattern?)`  
`execution(public * *(..))` – any public method  
`execution(* set*(..))` – any method with beginning with "set":  
`execution(* com.xyz.service.*Service*(..))` – any method on \*Service class
- ❖ **within** - limits matching to join points within certain types  
`within(com.xyz.service..*)` – any join point (method execution only in Spring AOP) within the service package or a sub-package
- ❖ **this** - limits matching to join points where the bean reference is an instance of the given type
- ❖ **target** - limits matching to join points where the target object (application object being proxied) is instance of given type  
`target(com.xyz.service.AccountService)` – any join point where the target object implements the AccountService interface

# Pointcut Expressions

- ❖ **args** - limits matching to join points where the arguments are instances of the given types  
**args(java.io.Serializable)** – any join point which takes a single parameter, and where the argument passed at runtime is **Serializable**
- ❖ **@target** - limits matching to join points where the class of the executing object has an annotation of the given type
- ❖ **@args** - limits matching to join points where the runtime type of the actual arguments have annotations of the given type(s)
- ❖ **@within** - limits matching to join points within types that have the given annotation
- ❖ **@annotation** - limits matching to join points where the subject of the join point has the given annotation
- ❖ **bean(idOrNameOfBean)** – Spring only – e.g.: **bean(\*Service)**

# @Around Aspect

@Aspect

@Component

@Slf4j

```
public class MethodProfilerAspect {  
    @Pointcut(  
        "within(@org.springframework.stereotype.Repository *)" )  
    public void repositoryClassMethods() {}  
  
    @Around("repositoryClassMethods()")  
    public Object measureMethodExecutionTime(  
        ProceedingJoinPoint pjp) throws Throwable {  
        long start = System.nanoTime();  
        Object retval = pjp.proceed();  
        long end = System.nanoTime();  
        String methodName = pjp.getSignature().getName();  
        log.info("Execution of " + methodName + " took " +  
            (end - start) + " ns");  
        return retval;  
    }  
}
```

# @AfterReturning Aspect

@Component

@Aspect

```
public class EntityCreationPublishingAspect {  
    @Autowired private ApplicationEventPublisher eventPublisher;  
  
    @Pointcut("@target(org.springframework.stereotype.Repository)")  
    public void repositoryMethods() {}  
  
    @Pointcut("execution(* *..add*(..))")  
    public void addMethods() {}  
  
    @Pointcut("repositoryMethods() && addMethods()")  
    public void addRepoMethods() {}  
  
    @AfterReturning(value = "addRepoMethods()", returning =  
        "entity")  
    public void logMethodCall(JoinPoint jp, Object entity) {  
        eventPublisher.publishEvent(new EntityCreationEvent(this,  
            jp.getArgs()[0].getClass().getName(), entity));  
    }  
}
```

# Custom Event Type

```
public class EntityCreationEvent extends ApplicationEvent {  
    private String entityName;  
    private Object entity;  
  
    public EntityCreationEvent(Object source, String entityName,  
        Object entity) {  
        super(source);  
        this.entityName = entityName;  
        this.entity = entity;  
    }  
  
    public String getEntityName() {  
        return entityName;  
    }  
  
    public Object getEntity() {  
        return entity;  
    }  
}
```

# Event Listener

```
@Component
@Slf4j
public class EntityCreationEventListener implements
ApplicationListener<EntityCreationEvent> {
    @Override
    public void onApplicationEvent(EntityCreationEvent event) {
        log.info("Entity created [{}]: {}",
            event.getEntityName(), event.getEntity().toString());
    }
}
```



# Enable Async Event Handling

```
@Configuration
@EnableAspectJAutoProxy
@ComponentScan(basePackages = "org.iproduct.spring.aop")
public class AppConfiguration {
    ...
    @Bean(name = "applicationEventMulticaster")
    public ApplicationEventMulticaster eventMulticaster() {
        SimpleApplicationEventMulticaster eventMulticaster
            = new SimpleApplicationEventMulticaster();

        eventMulticaster.setTaskExecutor(
            new SimpleAsyncTaskExecutor());
        //or Executors.newCachedThreadPool()
        return eventMulticaster;
    }
}
```

# Sharing Common Pointcut Definitions

@Aspect

@Component

```
public class SystemArchitecture {  
  
    @Pointcut("within(org.iproduct.spring.aop..*) && " +  
              "@within(org.springframework.stereotype.Repository)")  
    public void inDaoLayer() {}  
  
    @Pointcut("within(org.iproduct.spring.aop..*) && " +  
              "@within(org.springframework.stereotype.Service)")  
    public void inServiceLayer() {}  
  
    @Pointcut("inDaoLayer() || inServiceLayer()")  
    public void inDomainLayer() {}  
  
}
```

# AOP Introductions (ITDs)

@Component

@Aspect

@Slf4j

```
public class UsageTracking {
```

```
    @DeclareParents(value="org.iproduct.spring.aop.service..*",  
                    defaultImpl=DefaultUsageTracked.class)
```

```
    public static UsageTracked mixin;
```

```
    @Before("org.iproduct.spring.aop.SystemArchitecture" +  
            ".inDomainLayer() && this(usageTracked) && target(target)")
```

```
    public void recordUsage(
```

```
        UsageTracked usageTracked, Object target) {  
        log.info(">>> Incrementing usage count [{}]: {}",  
                target.getClass().getName().toString(),  
                usageTracked.incrementUseCount());  
    }
```

```
}
```

# ITD Interface and Implementation

```
public interface UsageTracked {  
    long incrementUseCount();  
    long getUseCount();  
}
```

**@Component**

```
public class DefaultUsageTracked implements UsageTracked {  
    private AtomicLong counter = new AtomicLong(0L);
```

**@Override**

```
public long incrementUseCount() {  
    return counter.incrementAndGet();  
}
```

**@Override**

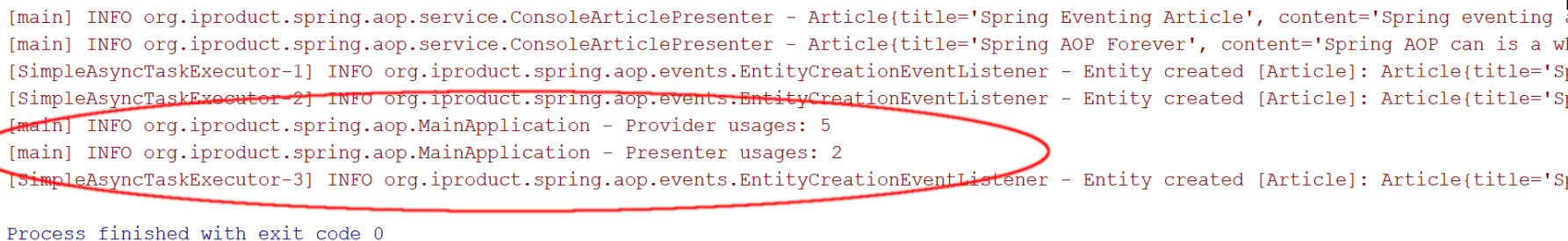
```
public long getUseCount() {  
    return counter.get();  
}
```

```
}
```

# ITD Interface and Implementation

@Slf4j

```
public class MainApplication {  
    public static void main(String... args) throws  
        UsageTracked providerUsage =  
            (UsageTracked) ctx.getBean("provider");  
    log.info("Provider usages: {}",  
        providerUsage.getUseCount());  
    UsageTracked presenterUsage =  
        (UsageTracked) ctx.getBean("presenter");  
    log.info("Presenter usages: {}",  
        presenterUsage.getUseCount());  
}
```



```
[main] INFO org.iproduct.spring.aop.service.ConsoleArticlePresenter - Article{title='Spring Eventing Article', content='Spring eventing  
[main] INFO org.iproduct.spring.aop.service.ConsoleArticlePresenter - Article{title='Spring AOP Forever', content='Spring AOP can is a w  
[SimpleAsyncTaskExecutor-1] INFO org.iproduct.spring.aop.events.EntityCreationEventListener - Entity created [Article]: Article{title='S  
[SimpleAsyncTaskExecutor-2] INFO org.iproduct.spring.aop.events.EntityCreationEventListener - Entity created [Article]: Article{title='S  
[main] INFO org.iproduct.spring.aop.MainApplication - Provider usages: 5  
[main] INFO org.iproduct.spring.aop.MainApplication - Presenter usages: 2  
[SimpleAsyncTaskExecutor-3] INFO org.iproduct.spring.aop.events.EntityCreationEventListener - Entity created [Article]: Article{title='S  
  
Process finished with exit code 0
```

# Additinal Examples

Spring Core Reference Documentation - AOP:

<https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop>

Introduction to Pointcut Expressions in Spring (Baeldung):

<https://www.baeldung.com/spring-aop-pointcut-tutorial>



# Thank's for Your Attention!



**Trayan Iliev**

**CEO of IPT – Intellectual Products  
& Technologies**

<http://iproduct.org/>

<http://robolearn.org/>

<https://github.com/iproduct>

<https://twitter.com/trayaniliev>

<https://www.facebook.com/IPT.EACAD>

<https://plus.google.com/+IproductOrg>