

What is Spring?

- ☐ **Light-weight yet comprehensive framework for building Java SE and Java EE applications.**
- ☐ **Initially built to reduce the complexities of Enterprise Java development.**
- ☐ **Initially developed as an alternative to EJBs.**
- ☐ **Based on POJOs and Interfaces**
- ☐ **JavaBeans-based configuration management, applying Inversion-of-Control principles, specifically using the Dependency Injection technique, which aims to reduce dependencies of components on specific implementations of other components.**
- ☐ **Integration with persistence frameworks Hibernate, JDO and iBATIS.**
- ☐ **Extensive aspect-oriented programming (AOP) framework to provide services such as transaction management**
- ☐ **Spring provides better Maintainability, Testability and Scalability**
- ☐ **MVC web application framework, built on core Spring functionality, supporting many technologies for generating views, including JSP, FreeMarker, Velocity and Tiles.**

Features of Spring

- ☐ **Dependency injection (DI) through IOC**
- ☐ **Aspect-Oriented Programming - Transaction management (AOP)**
- ☐ **Web applications**
- ☐ **Data access (JDBC, Hibernate)**
- ☐ **Messaging**
- ☐ **Testing**
- ☐ **More...**

Spring simplifies Java Development

History of Spring

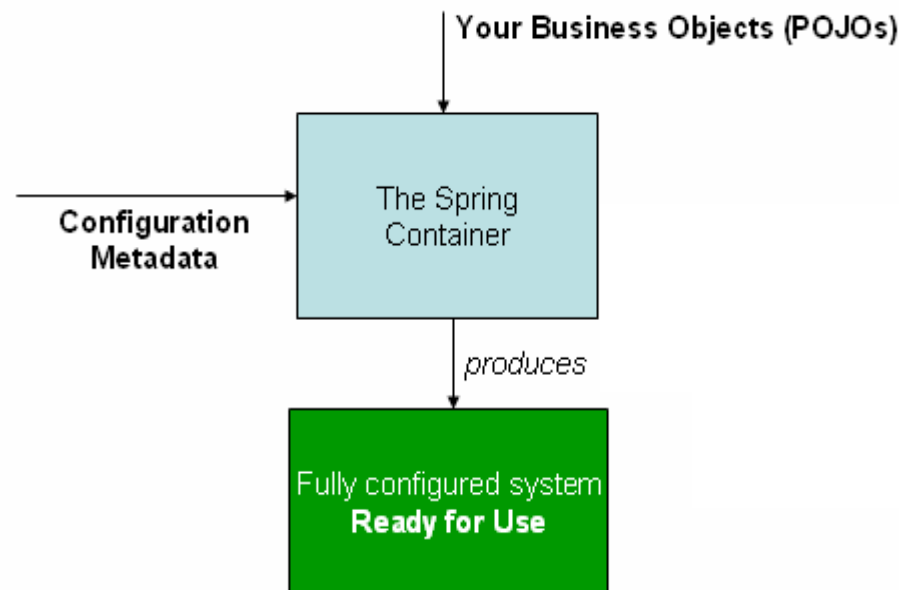
- ❑ First conceived and developed by Rod Johnson
- ❑ Rod Johnson described Spring in his book One-on-One : J2EE Design and Development.
- ❑ First milestone release was in June, 2003
- ❑ Release 1.0 was in 2004
- ❑ Release 2.0 was released in 2006 supported AspectJ
- ❑ Annotations were introduced in 2.5, which was released in 2007
- ❑ 3.0 was released in 2009 supported Java Configuration
- ❑ 3.1 in 2011
- ❑ 3.2 in 2013
- ❑ 4.0 in Dec, 2013 supports Java SE 8 and Java EE 7
- ❑ 4.1 in Oct-2014
- ❑ 4.2.2 on 15-Oct-2015

Key Strategies

- ☐ Lightweight and minimally invasive development with POJOs
- ☐ Loose coupling through DI and interface orientation
- ☐ Declarative Programming through Aspects and common conventions
- ☐ Eliminating boilerplate code with aspects and templates

Spring IOC Container

- ❑ Provides Dependency Injection (DI)
- ❑ DI is the process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments or properties that are set on the object instance after it is constructed
- ❑ The container then *injects* those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name *Inversion of Control* (IoC), of the bean itself controlling the instantiation



Spring Beans

- ☐ The term “bean” is used to refer any component managed by the BeanFactory
- ☐ It is a POJO
- ☐ The “beans” are in the form of JavaBeans (in most cases) - no arg constructor and getter and setter methods for the properties
- ☐ Beans are singletons by default
- ☐ Properties may be simple values or references to other beans
- ☐ Beans can have multiple names
- ☐ These beans are created with the configuration metadata that you supply to the container

Bean Properties

- ☐ Name
- ☐ Scope
- ☐ Constructor arguments
- ☐ Properties
- ☐ Auto-wiring mode
- ☐ Lazy-initialization mode
- ☐ Initialization method
- ☐ Destruction method

Metadata

- ❑ Configuration metadata represents how you as an application developer tell the Spring container to instantiate, configure, and assemble the objects in your application.
- ❑ Configuration metadata traditionally supplied in a simple and intuitive XML
- ❑ Spring 2.5 introduced support for annotation-based configuration metadata.
- ❑ Starting with Spring 3.0 you can define beans external to your application classes by using Java rather than XML files (Java Configuration).

XML Metadata

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-
    beans.xsd">

  <bean id="customer" class="demo.Customer">
    <!-- properties etc. -->
  </bean>
</beans>
```

<import>

<beans>

 <import resource="services.xml"/>

 <import resource="resources/messageSource.xml"/>

 <import resource="/resources/themeSource.xml"/>

 <bean id="bean1" class="..."/>

 <bean id="bean2" class="..."/>

</beans>

ApplicationContext

- ❑ The ApplicationContext is the interface for an advanced factory capable of maintaining a registry of different beans and their dependencies.
- ❑ Using the method **T getBean(String name, Class<T> requiredType)** you can retrieve instances of your beans.
- ❑ The location path or paths supplied to an ApplicationContext constructor are actually resource strings that allow the container to load configuration metadata from a variety of external resources such as the local file system, from the Java CLASSPATH, and so on.
- ❑ ApplicationContext built on top of the BeanFactory (it is a sub-interface) and adds other functionality such as easier integration with Spring's AOP features.

```
ApplicationContext context = new ClassPathXmlApplicationContext  
    (new String[] {"beans.xml", "daos.xml"});  
MessageService service =  
    context.getBean("message", MessageService.class);
```

Application Contexts

- ❑ **AnnotationConfigApplicationContext** —Loads a Spring application context from one or more Java-based configuration classes
- ❑ **AnnotationConfigWebApplicationContext** —Loads a Spring web application context from one or more Java-based configuration classes
- ❑ **ClassPathXmlApplicationContext** —Loads a context definition from one or more XML files located in the classpath, treating context-definition files as classpath resources
- ❑ **FileSystemXmlApplicationContext** —Loads a context definition from one or more XML files in the filesystem
- ❑ **XmlWebApplicationContext** —Loads context definitions from one or more XML files contained in a web application

Creating ApplicationContext

ApplicationContext context =

new FileSystemXmlApplicationContext("d:\\demo\\beans.xml");

ApplicationContext context =

new ClassPathXmlApplicationContext("beans.xml");

ApplicationContext context =

new AnnotationConfigApplicationContext(com.st.demo.BbeansConfig.class);

Dependency Injection

- ❑ It is the job of the container to actually *inject* those dependencies when it creates the bean.
- ❑ Each bean has dependencies expressed in the form of properties, constructor arguments, or arguments to the static-factory method when that is used instead of a normal constructor.
- ❑ These dependencies will be provided to the bean, *when the bean is actually created*.

Constructor dependency Injection

Dependencies are provided through the constructors of the component

Setter dependency injection

Dependencies are provided through the JavaBeanstyle setter methods of the component.

Constructor-based Dependency Injection

```
<beans>
  <bean id="foo" class="x.y.Foo">
    <constructor-arg ref="bar"/>
    <constructor-arg ref="baz"/>
  </bean>
  <bean id="bar" class="x.y.Bar"/>
  <bean id="baz" class="x.y.Baz"/>
</beans>
```

```
<bean id="customer"
  class="examples.Customer ">
  <constructor-arg name="id" value="1"/>
  <constructor-arg name="name" value="Mr. Bill"/>
</bean>
```

Using C Namespace

- ❑ C namespace can be used to specify constructor arguments in xml.
- ❑ First include c-namespace and its schema in xml file.
- ❑ Use c:arg-ref attribute to specify constructor injection

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:c="http://www.springframework.org/schema/c" ...>

    <bean id="employee" class="bean.EmployeeBean"
        c:connection-ref="oracleConnection" />


</beans>
```



argument name
in constructor

Using C Namespace

```
<bean id="employee" class="bean.EmployeeBean"  
      c:_0-ref="oracleConnection" />
```



Argument position in constructor

```
<bean id="employee" class="bean.EmployeeBean"  
      c:job="Programmer" />
```



Assign literal to constructor parameter by name

```
<bean id="employee" class="bean.EmployeeBean"  
      c:_0="Programmer" />
```



Assign literal to constructor parameter by position

Property-based Dependency Injection

```
<bean id="exampleBean" class="examples.ExampleBean">
  <property name="beanOne">
    <ref bean="anotherExampleBean"/>
  </property>
  <property name="beanTwo" ref="yetAnotherBean"/>
  <property name="integerProperty" value="1"/>
</bean>

<bean id="anotherExampleBean"
      class="examples.AnotherBean"/>
<bean id="yetAnotherBean"
      class="examples.YetAnotherBean"/>
```

Using P Namespace

- ❑ P namespace can be used to specify property based injection
- ❑ First include p-namespace and its schema in xml file.
- ❑ Use p:arg-ref attribute to specify property injection

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:p="http://www.springframework.org/schema/p" ...>

    <bean id="employee" class="bean.EmployeeBean"
        p:connection-ref="oracleConnection" />

</beans>
```



Property name

Auto Wiring

- ❑ The Spring container is able to *autowire* relationships between collaborating beans.
- ❑ It is possible to automatically let Spring resolve collaborators (other beans) for your bean by inspecting the contents of the BeanFactory.
- ❑ Autowiring is specified *per* bean and can thus be enabled for some beans, while other beans will not be autowired.
- ❑ Using autowiring, it is possible to reduce or eliminate the need to specify properties or constructor arguments, thus saving a significant amount of typing.
- ❑ When using XML-based configuration metadata, the autowire mode for a bean definition is specified by using the autowire attribute of the <bean/> element.
- ❑ Explicit dependencies in property and constructor-arg settings always override autowiring.
- ❑ You cannot autowire so-called *simple properties such as primitives, Strings, and Classes (and arrays of such simple properties)*. This limitation is by-design.

Auto Wiring Modes

no

No autowiring at all. This is the default.

byName

Autowiring by property name. This option will inspect the container and look for a bean named exactly the same as the property which needs to be autowired.

byType

Allows a property to be autowired if there is exactly one bean of the property type in the container. If there is more than one, a fatal exception is thrown, and this indicates that you may not use *byType* autowiring for that bean. If there are no matching beans, nothing happens; the property is not set. If this is not desirable, setting the `dependency-check="objects"` attribute value specifies that an error should be thrown in this case.

constructor

This is analogous to *byType*, but applies to constructor arguments. If there isn't exactly one bean of the constructor argument type in the container, a fatal error is raised.

Auto Wiring – Advantages and Disadvantages

Advantages

- ☐ Autowiring can significantly reduce the volume of configuration required.
- ☐ Autowiring can cause configuration to keep itself up to date as your objects evolve.
- ☐ Additional dependency to a class can be satisfied automatically without the need to modify configuration.

Disadvantages

- ☐ The relationships between your Spring-managed objects are no longer documented explicitly.
- ☐ Wiring information may not be available to tools that may generate documentation from a Spring container.
- ☐ Autowiring by type will only work when there is a single bean definition of the type specified by the setter method or constructor argument.
- ☐ You need to use explicit wiring if there is any potential ambiguity.

Bean Scope

singleton (Default)

Scopes a single bean definition to a single object instance per Spring IoC container.

prototype

Scopes a single bean definition to any number of object instances.

Request

Scopes a single bean definition to the lifecycle of a single HTTP request; that is, each HTTP request has its own instance of a bean created off the back of a single bean definition.

session

Scopes a single bean definition to the lifecycle of an HTTP Session.

global session

Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a portlet context.

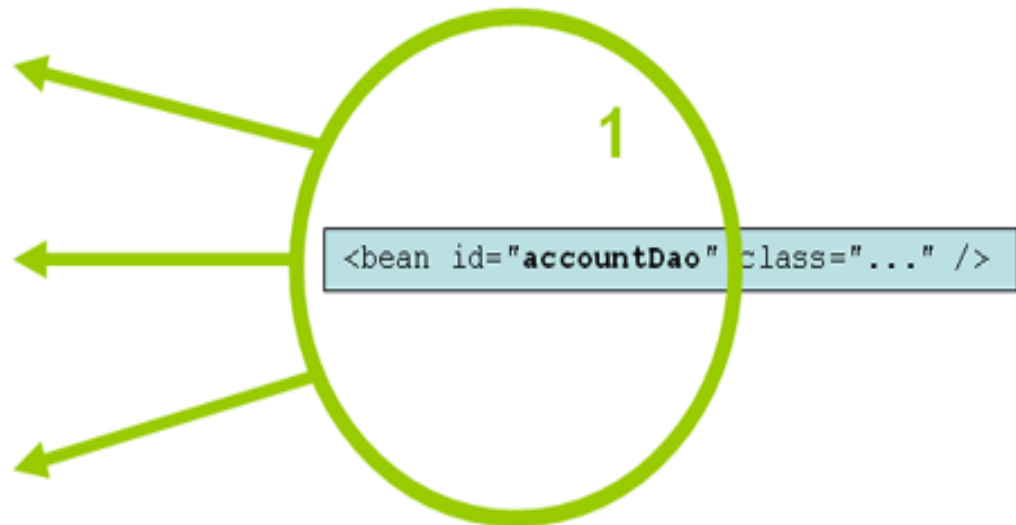
Singleton

```
<bean id="..." class="...">  
  <property name="accountDao"  
    ref="accountDao"/>  
</bean>
```

```
<bean id="..." class="...">  
  <property name="accountDao"  
    ref="accountDao"/>  
</bean>
```

```
<bean id="..." class="...">  
  <property name="accountDao"  
    ref="accountDao"/>  
</bean>
```

Only one instance is ever created...



... and this same shared instance is injected into each collaborating object

Prototype

```
<bean id="..." class="...">  
  <property name="accountDao"  
    ref="accountDao"/>  
</bean>
```

A brand new bean instance is created...

1

```
<bean id="..." class="...">  
  <property name="accountDao"  
    ref="accountDao"/>  
</bean>
```

2

```
<bean id="accountDao" class="..."  
  scope="prototype" />
```

```
<bean id="..." class="...">  
  <property name="accountDao"  
    ref="accountDao"/>  
</bean>
```

3

... each and every time the prototype is referenced by collaborating beans

Initialization Callbacks

In the case of XML-based configuration metadata, you use the **init-method** attribute to specify the name of the method that has a void no-argument signature.

You use the **destroy-method** attribute on the <bean/>

```
<bean id="example"  
      class="examples.ExampleBean"  
      init-method="init" />
```

```
<bean id="example"  
      class="examples.ExampleBean"  
      destroy-method="cleanup" />
```

Annotation based configuration

- ❑ Instead of using XML to describe a bean wiring, the developer moves the configuration into the component class itself by using annotations on the relevant class, method, or field declaration.
- ❑ Annotation injection is performed *before* XML injection, thus the latter configuration will override the former for properties wired through both approaches.
- ❑ `<context:annotation-config/>` looks for annotations on beans in the same application context in which it is defined.
- ❑ The use of `<context:component-scan>` implicitly enables the functionality of `<context:annotation-config>`.
- ❑ Attribute `base-package` specifies packages to search for. It can take a comma-separated list that includes the parent package of each class.
- ❑ Use `@Scope()` annotation to specify scope for beans. Ex:
`@Scope("prototype")`

Annotation based configuration

```
<context:component-scan base-package="catalog" />
```

@Component

- ❑ Is a generic stereotype for any Spring-managed component.
- ❑ Therefore, you can annotate your component classes with **@Component**, but by annotating them with **@Repository**, **@Service**, or **@Controller** instead, your classes are more properly suited for processing by tools or associating with aspects.

@Required Annotation

@Required

Annotation applies to bean property setter methods. The container throws an exception if the affected bean property has not been populated;

@Autowired

As expected, you can apply the @Autowired annotation to "traditional" setter methods. By default, the autowiring fails whenever *zero* candidate beans are available; the default behavior is to treat annotated methods, constructors, and fields as indicating *required* dependencies.

@Autowired

- ❑ As expected, you can apply the @Autowired annotation to "traditional" setter methods
- ❑ By default, the autowiring fails whenever *zero* candidate beans are available; the default behavior is to treat annotated methods, constructors, and fields as indicating *required* dependencies.
- ❑ You can also apply the annotation to methods with arbitrary names and/or multiple arguments.
- ❑ You can apply @Autowired to setter methods, constructors and fields.
- ❑ Attribute **required** indicates that the property is not required for autowiring purposes, the property is ignored if it cannot be autowired.

```
@Autowired(required=false)
public void setMovieFinder(MovieFinder movieFinder) {
    this.movieFinder = movieFinder;
}
```

```
@Autowired
private MovieCatalog movieCatalog;
```

@Autowired

```
@Autowired
```

```
public void prepare(Catalog catalog, Customer customer) {  
}
```

```
// All beans of type MovieCatalog type from the ApplicationContext
```

```
@Autowired
```

```
private MovieCatalog[] movieCatalogs;
```


@Qualifier

- ❑ Autowiring by type may lead to multiple candidates, it is often necessary to have more control over the selection process.
- ❑ `Qualifier` annotation can be used to specify the name of the bean to be used.
- ❑ The `@Qualifier` annotation can also be specified on individual constructor arguments or method parameters

```
@Autowired  
@Qualifier("movies")  
private Catalog catalog;
```

```
@Autowired  
public void prepare(@Qualifier("main") Catalog catalog ) {  
}
```

@Inject and @Named

- ❑ Instead of @Autowired, @javax.inject.Inject may be used.
- ❑ @Inject can be used at the class-level, field-level, method-level and constructor-argument level.
- ❑ If you would like to use a qualified name for the dependency that should be injected, you should use the @Named annotation.
- ❑ Annotation @Named can be used similar to @Component

```
@Inject  
public void setMovieFinder(@Named("main") MovieFinder movieFinder) {  
    this.movieFinder = movieFinder;  
}
```

@PostConstruct and @PreDestroy

- ❑ Introduced in Spring 2.5, the support for these annotations offers yet another alternative to those described in [initialization callbacks](#) and [destruction callbacks](#).
- ❑ A method carrying one of these annotations is invoked at the same point in the lifecycle as the corresponding Spring lifecycle interface method or explicitly declared callback method.
- ❑ In the example below, the cache will be pre-populated upon initialization and cleared upon destruction.

```
public class CachingMovieLister {  
    @PostConstruct  
    public void populateMovieCache()  
    { // populates the movie cache upon initialization... }  
  
    @PreDestroy  
    public void clearMovieCache()  
    { // clears the movie cache upon destruction... }  
}
```

Lazy-Initialized Beans

- ❑ By default, ApplicationContext creates and configure all singleton beans as part of the initialization process.
- ❑ When this behavior is *not* desirable, you can prevent pre-instantiation of a singleton bean by marking the bean definition as lazy-initialized using **lazy-init** attribute.
- ❑ A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.
- ❑ You can also control lazy-initialization at the container level by using the **default-lazy-init** attribute on the <beans/> element.
- ❑ **@Lazy** annotation specifies the same for bean.

```
<bean id="lazy" class="demo.SampleBean" lazy-init="true"/>
```

```
<beans default-lazy-init="true">  
    <!-- define beans here -->  
</beans>
```

XML vs. Annotations vs. Java Configuration

- ☐ Each approach has its pros and cons, and usually it is up to the developer to decide which strategy suits them better.
- ☐ Annotations provide shorter and more concise configuration.
- ☐ However, XML excels at wiring up components without touching their source code or recompiling them.
- ☐ In case of annotations, classes are no longer POJOs and, furthermore, that the configuration becomes decentralized and harder to control.
- ☐ In JavaConfig option, Spring allows annotations to be used in a noninvasive way, without touching the target components source code.

Java based Configuration

- ❑ Spring's new Java-configuration support is provided through **@Configuration** and **@Bean** annotations.
- ❑ The **@Bean** annotation is used to indicate that a method instantiates, configures and initializes a new object to be managed by the Spring IoC container.
- ❑ **@Bean** annotation plays the same role as the **<bean/>** element.
- ❑ You can use **@Bean** annotated methods with any Spring **@Component**, however, they are most often used with **@Configuration** beans.
- ❑ Annotating a class with **@Configuration** indicates that its primary purpose is as a source of bean definitions. Class name and method names are irrelevant.
- ❑ Annotation **@Scope** can be used to specify scope of bean. Ex: **@Scope("prototype")**
- ❑ Annotation **@Lazy** specifies lazy initialization
- ❑ Annotation **@ComponentScan** specifies which packages Spring scans for components. If no package is given with **packages** attribute then Spring uses the package in which this configuration class is present.

Related Annotations

- ☐ **@Configuration**
- ☐ **@Bean**
- ☐ **@ComponentScan**
- ☐ **@Lazy**
- ☐ **@Scope**

Java based Configuration

@Configuration

```
public class AppConfig {  
  
    @Bean  
    public MyService myService() {  
        return new MyServiceImpl();  
    }  
}
```

```
ApplicationContext ctx =  
    new AnnotationConfigApplicationContext  
        (AppConfig.class);
```

```
MyService myService = ctx.getBean(MyService.class);
```


Java based Configuration – Component Scan

```
@Configuration  
@ComponentScan(basePackages = "com.st")  
public class AppConfig {  
    ...  
}
```

Setter Injection

```
@Configuration
@ComponentScan( {"books"})
public class BooksConfiguration {
    @Bean ( name ="javaBooks")
    public Books getBooks() {
        return new JavaBooks();
    }

    @Bean ( name ="catalog")
    public Catalog getCatalog() {
        Catalog cat = new Catalog();
        cat.setBooks(getBooks()); // setter injection
        return cat;
    }
}
```

Constructor Injection

```
public class Catalog {  
    Books books;  
    public Catalog(Books books) {  
        this.books = books;  
    }  
    // others  
}
```

@Configuration

```
@ComponentScan( basePackages ="books" )  
public class BooksConfiguration {  
    @Bean ( name ="oraclebooks")  
    public Books getOracleBooks() {  
        return new OracleBooks();  
    }  
    @Bean ( name ="catalog")  
    public Catalog getCatalog() {  
        Catalog cat = new Catalog( getOracleBooks());  
        return cat;  
    }  
}
```

Auto Wiring

```
public class Catalog {  
    Books books;  
    @Autowired  
    public void setBooks(Books books) {  
        this.books = books;  
    }  
    // others  
}
```

```
@Configuration  
@ComponentScan ( basePackages ="books_java" )  
public class BooksConfiguration {  
    @Bean ( name ="javaBooks")  
    public Books getJavaBooks() {  
        return new JavaBooks();  
    }  
    @Bean ( name ="catalog")  
    public Catalog getCatalog() {  
        Catalog cat = new Catalog();  
        return cat;  
    }  
}
```

AnnotationConfigApplicationContext

```
// Register beans explicitly
AnnotationConfigApplicationContext ctx =
    new AnnotationConfigApplicationContext();
ctx.register(OracleBooks.class, Catalog.class);
ctx.register(AdditionalConfig.class);
ctx.refresh();

Catalog catalog = ctx.getBean(Catalog.class);
```

```
// Scan a particular package
AnnotationConfigApplicationContext ctx =
    new AnnotationConfigApplicationContext();
ctx.scan("books_java");
ctx.refresh();

Catalog catalog = ctx.getBean(Catalog.class);
```

Bean Scopes

singleton (Default)

Scopes a single bean definition to a single object instance per Spring IoC container.

prototype

Scopes a single bean definition to any number of object instances.

request

Scopes a single bean definition to the lifecycle of a single HTTP request. Only valid in the context of a web-aware Spring ApplicationContext.

session

Scopes a single bean definition to the lifecycle of an HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.

application

Scopes a single bean definition to the lifecycle of a ServletContext. Only valid in the context of a web-aware Spring ApplicationContext.

```
<bean id="accountService" class="com.st.AccountService" scope="prototype"/>
```

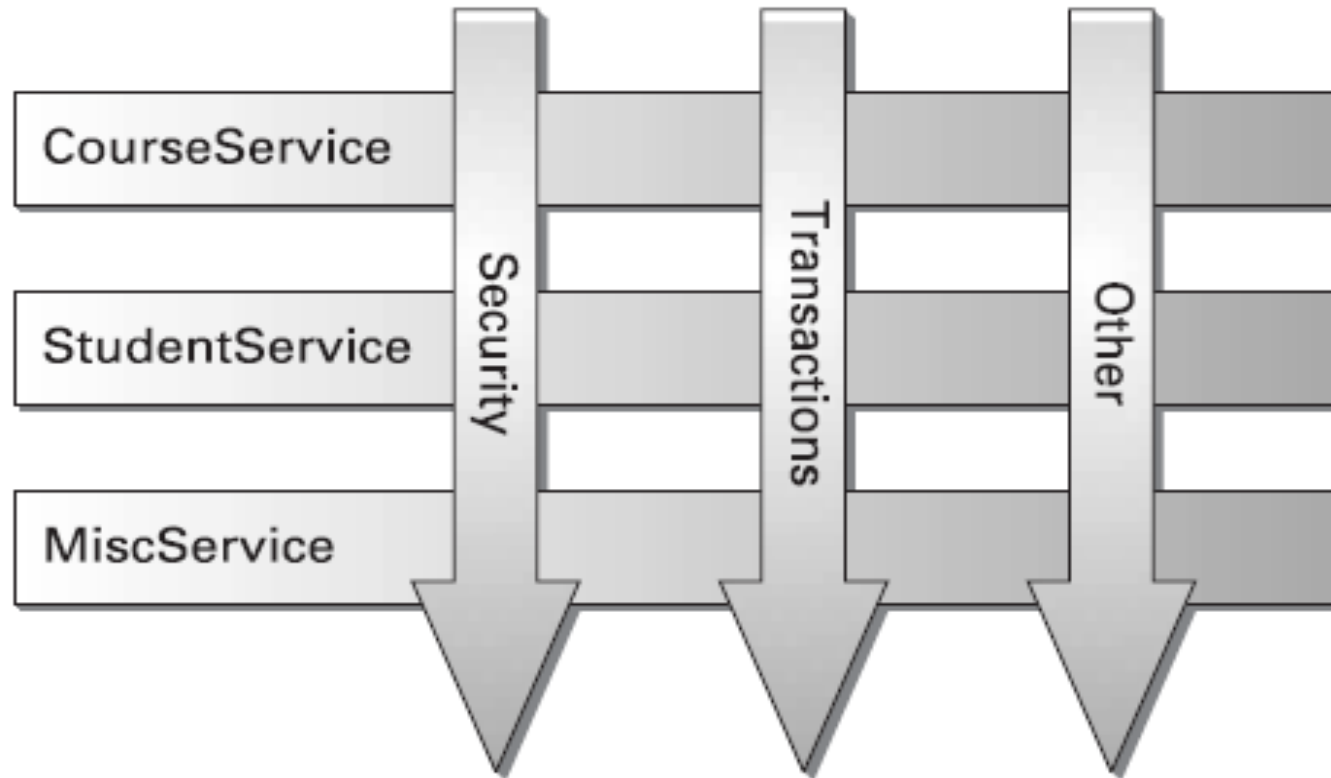
```
@Scope("prototype") // @Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)  
class AccountService {
```

Bean Scopes

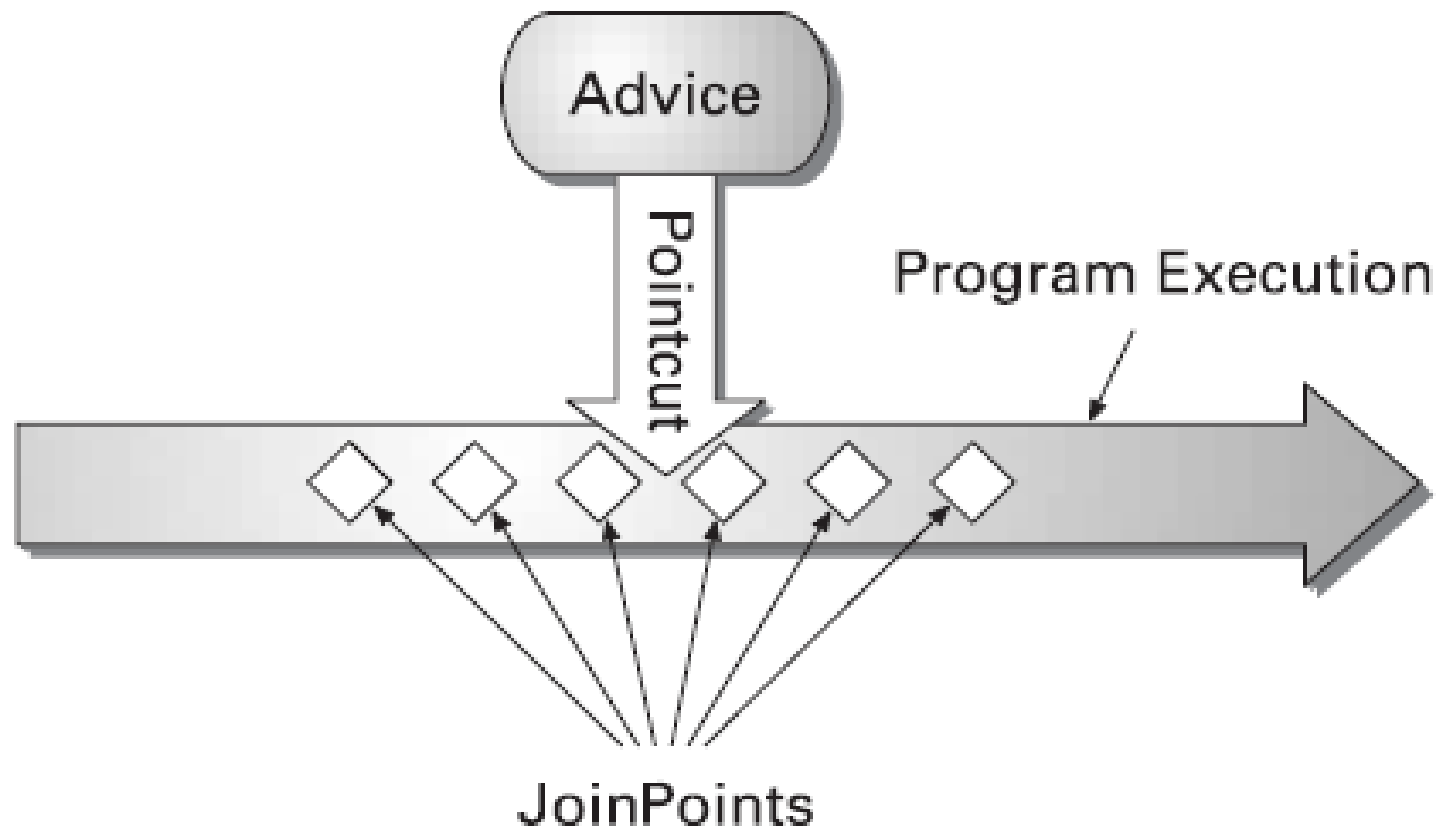
What is AOP?

- ❑ ***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 the unit of modularity is the *aspect*.
- ❑ Aspects enable the modularization of concerns such as transaction management that cut across multiple types and objects.
- ❑ AOP eliminates code tangling, where each method handles multiple concerns
- ❑ AOP also eliminates code scattering, where aspects are implemented in multiple methods.

What is AOP?



What is AOP?



Examples for Aspects

- ☐ **Transaction Management**
- ☐ **Security**
- ☐ **Tracing**
- ☐ **Profiling**
- ☐ **Exception Management**

AOP Terminology

Aspect

A modularization of a concern that cuts across multiple classes. Transaction management is a good example of a crosscutting concern in enterprise Java applications. In Spring AOP, aspects are implemented using regular classes.

Join point

A point during the execution of a program, such as the execution of a method or the handling of an exception. In Spring AOP, a join point *always* represents a method execution.

Advice

Action taken by an aspect at a particular join point. Different types of advice include "around," "before" and "after" advice.

Pointcut

A predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut. Spring uses the AspectJ pointcut expression language by default.

AOP Terminology

Introduction

Declaring additional methods or fields on behalf of a type. Spring AOP allows you to introduce new interfaces (and a corresponding implementation) to any advised object. For example, you could use an introduction to make a bean implement an `IsModified` interface, to simplify caching.

Target object

Object being advised by one or more aspects. Also referred to as the *advised* object. Since Spring AOP is implemented using runtime proxies, 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. This 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 advice

Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).

After returning advice

Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception.

After throwing advice

Advice to be executed if a method exits by throwing an exception.

After (finally) advice

Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).

Around advice

Advice that surrounds a join point such as a method invocation. Around advice 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.

Types of Advices

Before advice

Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).

After returning advice

Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception.

After throwing advice

Advice to be executed if a method exits by throwing an exception.

After (finally) advice

Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).

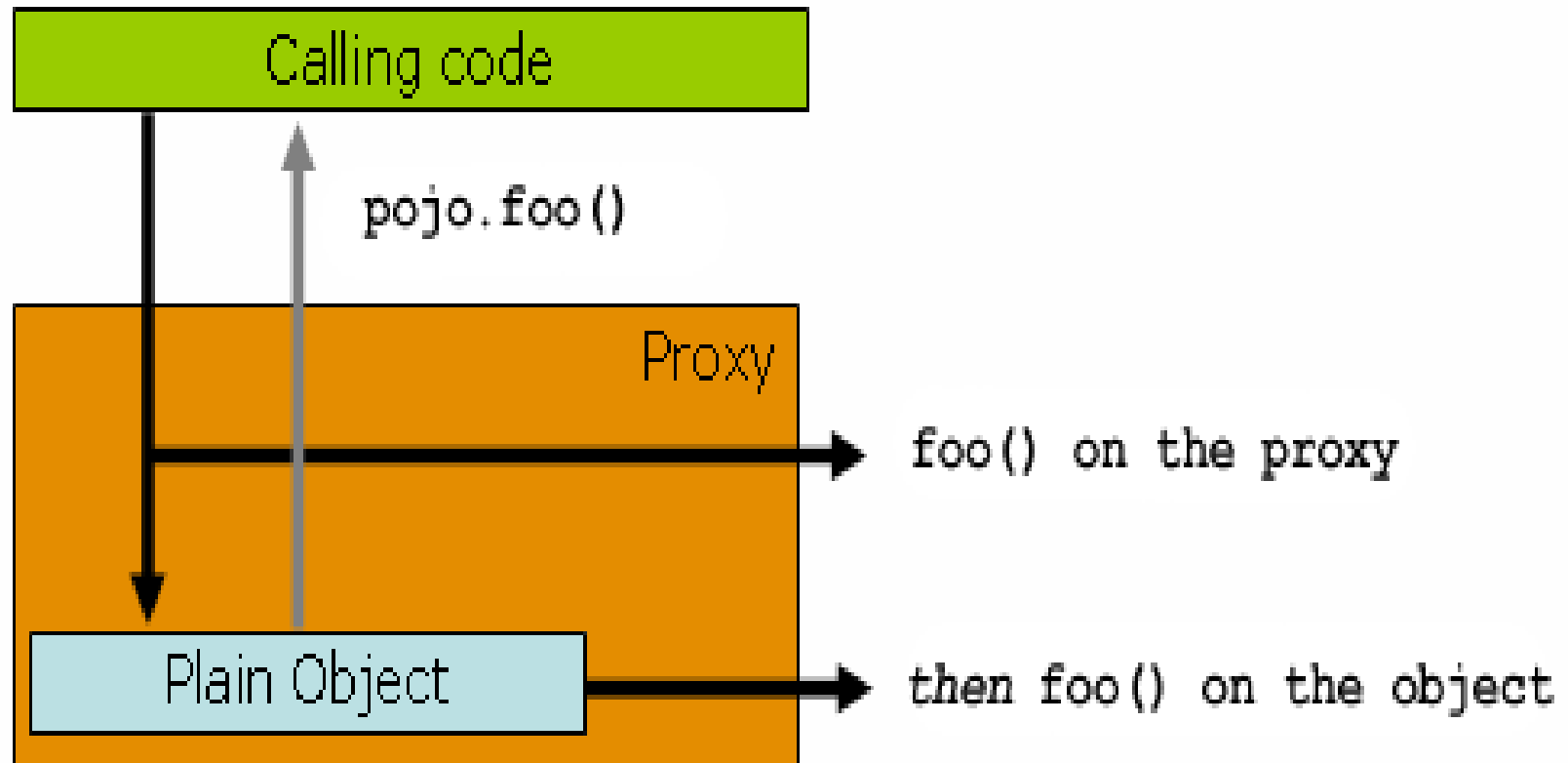
Around advice

Advice that surrounds a join point such as a method invocation. Around advice 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.

AOP Proxies

- ☐ Spring AOP defaults to Java dynamic proxies for AOP proxies.
- ☐ If bean class implements at least one interface then it uses JDK dynamic proxy.
- ☐ All of the interfaces implemented by the bean class will be proxied.
- ☐ If bean class doesn't implement any interface then Spring AOP uses CGLIB proxies.

AOP Proxies



@AspectJ Support

- ❑ It allows aspects to be declared as regular classes annotated with annotations.
- ❑ Spring uses same annotations as @AspectJ
- ❑ But Spring has its own AOP runtime and doesn't depend on @AspectJ compiler or weaver.
- ❑ Use [aspectjweaver.jar](#) to use @AspectJ
- ❑ In case of XML configuration use `<aop:aspectj-autoproxy/>` to enable @AspectJ support
- ❑ Use [@EnableAspectJAutoProxy](#) annotation with configuration class to enable AspectJ support in Java Configuration
- ❑ Any class annotated with [@Aspect](#) becomes an aspect.
- ❑ An aspect should also be annotated with [@Component](#) annotation when it is detected in component scanning.

@Pointcut

- ☐ Use @Pointcut annotation to define pointcuts.
- ☐ Method where Pointcut is declared must return void
- ☐ Pointcut has two parts – pointcut signature and pointcut expression
- ☐ Pointcut signature contains name and parameters
- ☐ Pointcut expression determines exactly which method executions we are interested in

Pointcut Syntax

**Execution (modifiers-pattern? ret-type declaring-type? name-pattern
(param-pattern) throws-pattern?)**

- ❑ All parts except the returning type, name pattern, and parameters pattern are optional.
- ❑ Returning type pattern determines what the return type of the method must be in order for a join point to be matched.
- ❑ The name pattern matches the method name.
- ❑ The parameters pattern is slightly more complex
- ❑ Value **()** matches a method that takes no parameters, whereas **(..)** matches any number of parameters (zero or more).
- ❑ The pattern **(*)** matches a method taking one parameter of any type, **(*,String)** matches a method taking two parameters, the first can be of any type, the second must be a String.
- ❑ Operators **and**, **or** and **not** are also available.

Pointcut Designators(PCD) in Pointcut expression

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

Execution Examples

The execution of any public method:

execution(public * *(..))

The execution of any method with a name beginning with "set" :

execution(* set*(..))

The execution of any method defined by the **AccountService** interface:

execution(* st.AccountService.*(..))

The execution of any method defined in the st package:

execution(* st..(..))

The execution of any method defined in the st package or a sub-package:

execution(* st...(..))

Pointcut Examples

Any join point within the st package:

`within(st.*)`

Any join point within the service package or a sub-package:

`within(com.xyz.service..*)`

Any join point where the proxy implements the AccountService interface:

`this(com.xyz.service.AccountService)`

Any join point where the target object implements the AccountService interface:

`target(com.xyz.service.AccountService)`

Any join point which takes a single parameter, and where the argument passed at runtime is Serializable:

`args(java.io.Serializable)`

Pointcut Examples

```
@Pointcut("execution(public * (..))")  
private void anyPublicOperation() {}
```

```
@Pointcut("within(com.st.trading..)")  
private void inTrading() {}
```

```
@Pointcut("anyPublicOperation() && inTrading()")  
private void tradingOperation() {}
```


AOP Configuration Elements

<aop:advisor>	Defines an AOP advisor. 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:after>	Defines an AOP after advice (regardless of whether the advised method returns successfully).
<aop:config>	The top-level AOP element.
<aop:declare-parents>	Introduces additional interfaces to advised objects that are transparently implemented.
<aop:pointcut>	Defines a pointcut.

AOP with XML Example

```
<aop:config>
  <aop:aspect id="traceaspect" ref="tracebean">
    <aop:pointcut id="processMethod"
      expression="execution(* *.process(int)) and args(value)"/>
    <aop:pointcut id="printMethod"
      expression="execution(* print* (...))" />
    <aop:after pointcut-ref="printMethod" method="afterMethod" />
    <aop:after-throwing pointcut-ref="processMethod"
      method="afterThrowing" throwing="ex" />
    <aop:around pointcut-ref="processMethod"
      method="aroundAdvice" />
  </aop:aspect>
</aop:config>

<bean id="tracebean" class="aop_xml.TraceAspect" />
```

AOP with XML Example

```
public class TraceAspect {  
    public void afterThrowing(RuntimeException ex){  
        System.out.println("Error --> " + ex);  
    }  
    public void aroundAdvice(ProceedingJoinPoint method,int value){  
        System.out.println("About to call method ->" +  
                            method.getSignature());  
        try {  
            method.proceed(method.getArgs());  
            System.out.println("Successfully processed value "+ value);  
        }catch(Throwable ex) {  
            System.out.println("Failure..");  
        }  
        System.out.println("Completed call to call method ->" +  
                            method.getSignature());  
    }  
  
    public void afterMethod(JoinPoint method ) {  
        System.out.printf("Method %s completed\n",  
                            method.getSignature() );  
    }  
}
```

AOP With Annotations

```
<aop:aspectj-autoproxy />  
<context:component-scan  
    base-package="aop_aspectj">  
</context:component-scan>
```

Method Signatures

```
@Before(pointcut="...")
public void doBeforeProcess() {
    // ...
}
```

```
@AfterReturning (pointcut="...", returning="retVal")
public void doAfterProcess(Object retVal) {
    // ...
}
```

```
@AfterThrowing(pointcut="...", throwing="ex")
public void doRecoveryActions(Exception ex) {
    // ...
}
```

```
@After(pointcut="...")
public void doReleaseLock() {
    // ...
}
```

```
@Around(pointcut="...")
public Object doProfiling(ProceedingJoinPoint pjp) throws Throwable {
    // before process
    Object retVal = pjp.proceed();
    // after process
    return retVal;
}
```

AOP With Annotations

```
@Component @Aspect
public class TraceAspect {
    @Pointcut("execution ( * * (..) )")
    public void aopMethods() { }
    @AfterThrowing(pointcut="execution(* process(..))",
        throwing="ex")
    public void afterMethod(JoinPoint method, Throwable ex) {
        System.out.printf("Method %s threw exception : %s \n",
            method.getSignature(), ex.getClass().getName());
    }
    @Around ("execution(* calculate(..))")
    public void aroundMethod(ProceedingJoinPoint method)
        throws Throwable {
        System.out.println("Call:" + method.getSignature());
        method.proceed(method.getArgs());
        System.out.println("Completed " + method.getSignature());
    }
    @Before("execution ( * * (..) ) && args(message)")
    public void beforePrint(String message ) {
        System.out.println("Message : " + message);
    }
}
```

AOP With Annotations

```
public class HelloTest {  
    public static void main(String[] args) {  
        ApplicationContext ctx = new  
            ClassPathXmlApplicationContext("aop_aspectj/context.xml");  
        Hello bean = ctx.getBean ("hello", Hello.class);  
        bean.print();  
        bean.print("Srikanth");  
    }  
}
```

Expression Language

- ❑ **The Spring Expression Language (SpEL for short) is a powerful expression language that supports querying and manipulating an object graph at runtime**
- ❑ **It was created to provide the Spring community with a single well supported expression language that can be used across all the products in the Spring portfolio**

Expression Language Features

- ☐ Literal expressions
- ☐ Boolean and relational operators
- ☐ Regular expressions
- ☐ Class expressions
- ☐ Accessing properties, arrays, lists, maps
- ☐ Method invocation
- ☐ Assignment
- ☐ Calling constructors
- ☐ Bean references
- ☐ Array construction
- ☐ Inline lists
- ☐ Ternary operator
- ☐ Variables
- ☐ User defined functions
- ☐ Collection projection
- ☐ Collection selection
- ☐ Templated expressions

Expression Language - Operators

Relational operators

equal (==, eq)
not equal (!=, ne)
less than (<, lt)
less than or equal (<= , le)
greater than (>, gt)
greater than or equal (>=, ge)

Mathematical operators

Addition (+)
Subtraction (-)
Multiplication (*)
Division (/)
Modulus (%)
Exponential power (^).

Logical operators

and
or
not (!)

Others

? : (Ternary)
? : (Elvis)
Matches

Expression Language Syntax

`# { expression }`

- ☐ Can be a literal - `#{10}`
- ☐ Can refer to a bean property - `#{account.balance}`
- ☐ Can invoke a method - `#{account.deposit(10000)}`
- ☐ Can use T operator to refer to a class so that static members can be accessed - `#{T(java.lang.Math).PI}`
- ☐ Compare values - `#{account.balance == 1000}` or `#{account.balance eq 1000}`
- ☐ Can use ternary operator – `#{account.balance > 1000 ? true : false}`
- ☐ Can use default value for null - `#{account.holder2 ?: 'Single'}`
- ☐ Can evaluate regular expression - `#{account.mobile matches '[0-9]{10}'}`
- ☐ Can evaluate collections - `#{account.trans[0].amount}`

Expression Language API

```
ExpressionParser parser = new SpelExpressionParser();  
Expression exp = parser.parseExpression  
    ("'Hello World'.concat('!')");  
String message = (String) exp.getValue();
```

Expression Language Examples

```
boolean trueValue = parser.parseExpression
    ("2 == 2").getValue(Boolean.class);

String c = parser.parseExpression
    ("abc.substring(2, 3)").getValue(String.class);

int[] numbers2 = (int[]) parser.parseExpression
    ("new int[]{1,2,3}").getValue();

List numbers = (List) parser.parseExpression
    ("{1,2,3,4}").getValue();

@Value("#{addressBean.country}")
private String country;

@Value("#{addressBean.getFullAddress()}")
private String fullAddress;
```

Referring to Beans

```
@Component("customer")
```

```
public class Customer {
```

```
    @Value("Srikanth")
```

```
    private String name;
```

```
    @Value("#{address}") // refers a bean with name address
```

```
    private Address home;
```

```
    @Value("#{address.getFullAddress()}") // invoke method
```

```
    private String homeAddress;
```

```
    @Value("#{contacts.phones['home']}") // get value from Map
```

```
    private String homePhone;
```

```
    @Value("#{contacts.emails[0]}") // get value from List
```

```
    private String primaryEmail;
```

Consuming a RESTful Service

- ☐ **User spring-web and jackson-databind artifacts in pom.xml**
- ☐ **Create a class that represents data sent by RESTful service**
- ☐ **Use @JsonIgnoreProperties to ignore extra properties**
- ☐ **Create an object of RestTemplate class and call getForObject() method with RESTful url.**

POM.XML

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${spring-framework.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>4.2.5.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.7.3</version>
  </dependency>
</dependencies>
```


GitHubUser.java

```
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
```

```
@JsonIgnoreProperties(ignoreUnknown = true)
```

```
public class GitHubUser {
```

```
private String login, email, company, location, created_at;
```

```
    // getter and setter methods
```

```
}
```

GetGitHubUserInfo.java

```
import org.springframework.web.client.RestTemplate;

public class GetGitHubUserInfo {
    public static void main(String[] args) {

        RestTemplate restTemplate = new RestTemplate();
        GitHubUser user = restTemplate.getForObject
            ("https://api.github.com/users/srikanthpragada", GitHubUser.class);

        System.out.println("Login : " + user.getLogin());
        System.out.println("Email : " + user.getEmail());
        System.out.println("Company : " + user.getCompany());
        System.out.println("Location: " + user.getLocation());
        System.out.println("Created : " + user.getCreated_at());
    }
}
```