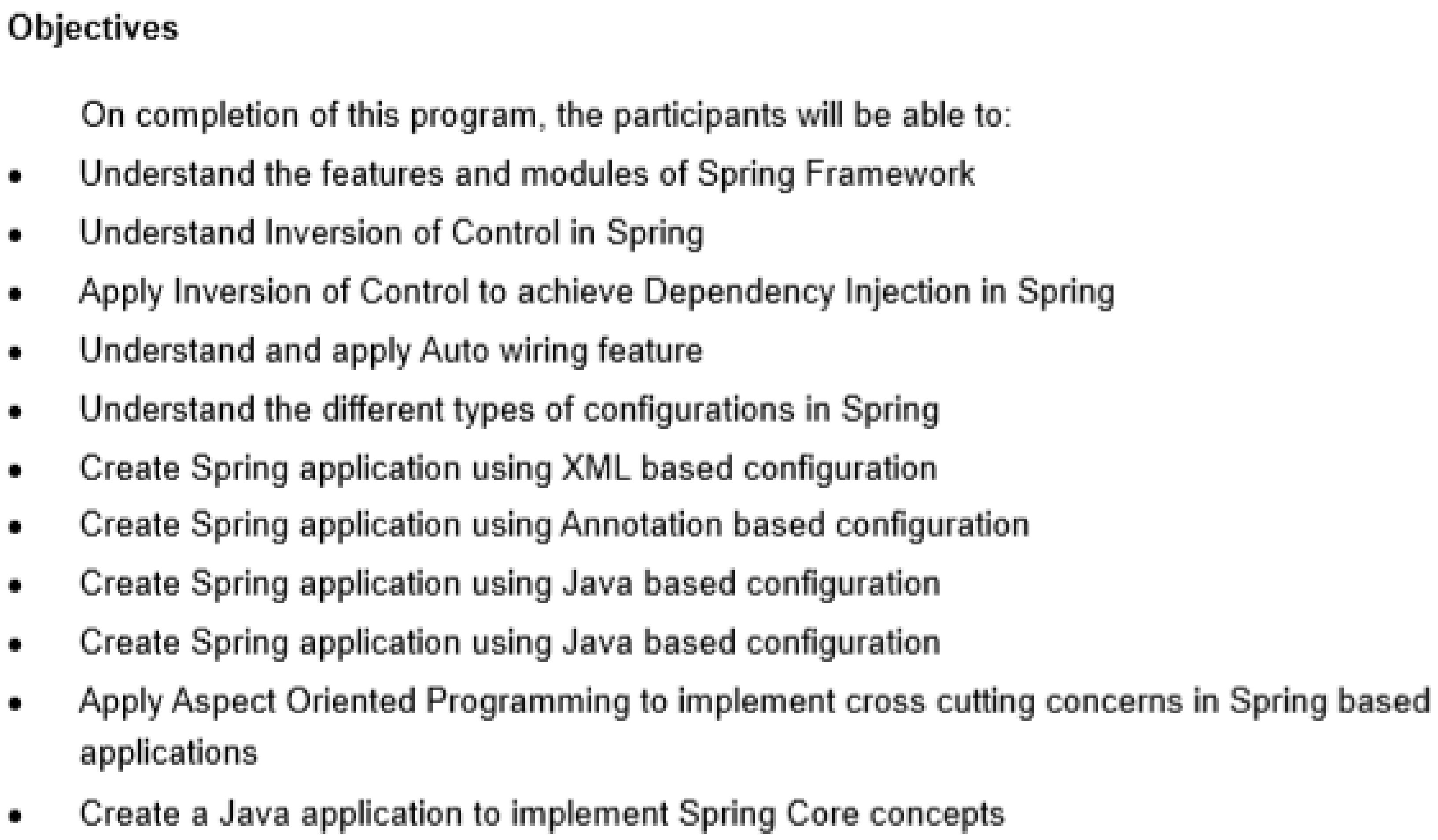
Huawei Technologies Co Ltd. ., All rights reserved

Spring Core
Spring Framework

Huawei Technologies Co Ltd. ., All rights reserved

Course Objectives



Objectives

On completion of this program, the participants will be able to:

- Understand the features and modules of Spring Framework
- Understand Inversion of Control in Spring
- Apply Inversion of Control to achieve Dependency Injection in Spring
- Understand and apply Auto wiring feature
- Understand the different types of configurations in Spring
- Create Spring application using XML based configuration
- Create Spring application using Annotation based configuration
- Create Spring application using Java based configuration
- Create Spring application using Java based configuration
- Apply Aspect Oriented Programming to implement cross cutting concerns in Spring based applications
- Create a Java application to implement Spring Core concepts

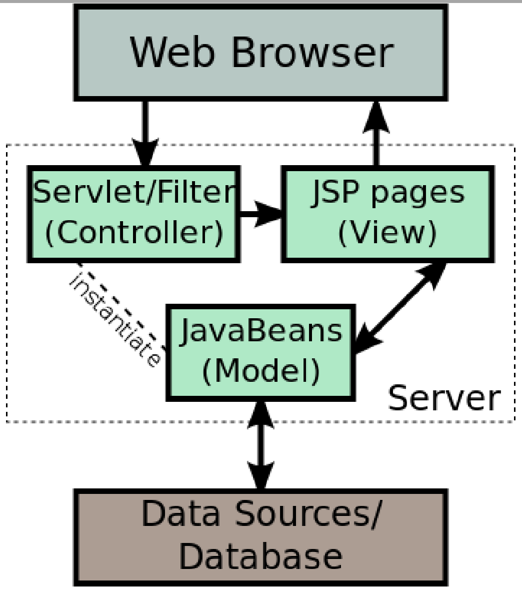
Huawei Technologies Co Ltd. ., All rights reserved

Building Web Applications

$$\begin{tikzcd}\text{Simple Application} \arrow[d] \\ \text{Complex Application}\end{tikzcd}$$

- **Scripting elements calling servlet code directly**
- **Scripting elements calling servlet code indirectly (by means of utility classes)**
- **Beans**
- **Servlet/JSP combo (MVC 2 Architecture)**
- **MVC with JSP expression language (JSTL)**
- **Custom tags**
- **MVC with beans and a framework like Struts or JSF or Spring**

MVC Model 2 Architecture

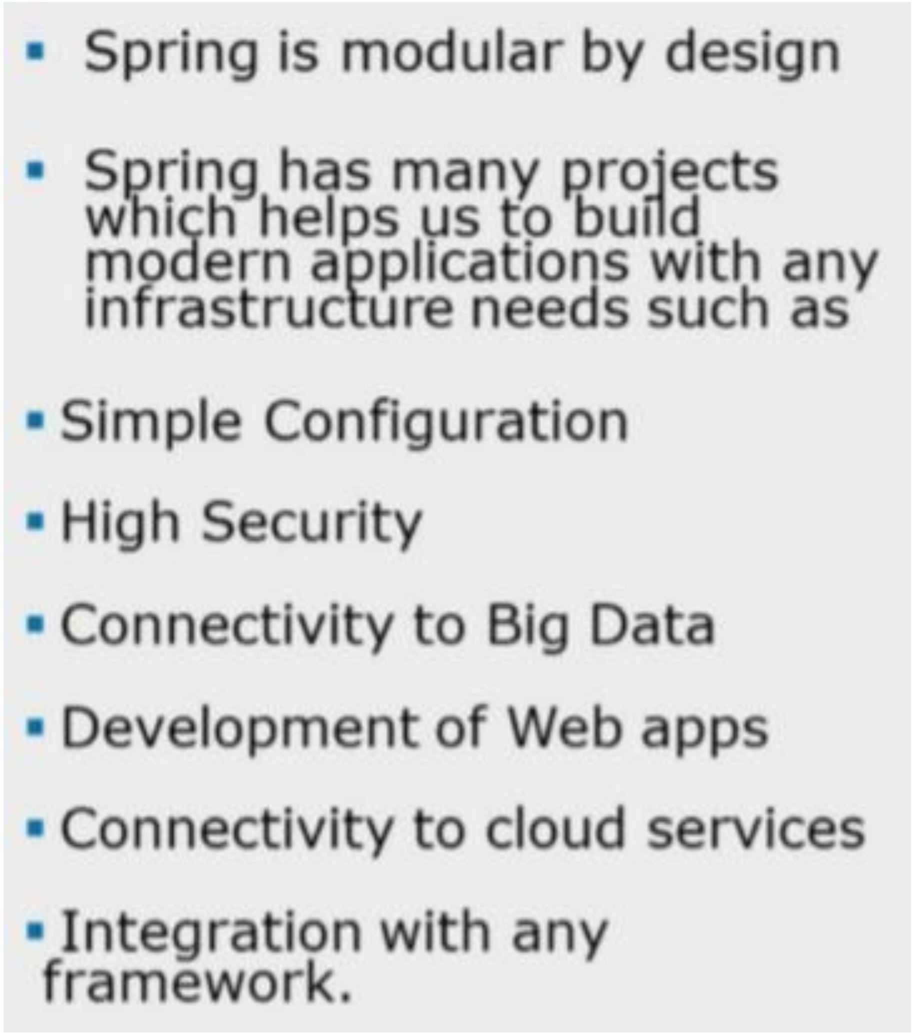


MVC based frameworks are: Struts 2 , JSF , Spring etc.

In JSP Model 2 architecture , JSP is used for creating the view for the application.
A centralized Servlet is used to handle all the request for the application. *The* Servlet *works as the controller for the application*. It then uses the Java beans for processing the business logic and getting the *data (Model)* from the database.
Finally it uses the JSP to render the view which is displayed to the user.

Spring Projects at a glance

http://spring.io/projects



- Spring is modular by design
- Spring has many projects which helps us to build modern applications with any infrastructure needs such as
- Simple Configuration
- High Security
- Connectivity to Big Data
- Development of Web apps
- Connectivity to cloud services
- Integration with any framework.

•Spring Framework
•Spring Data
•Spring Boot
•Spring Integration
•Spring Batch
•Spring Security
•Spring Hateoas
•Spring Social
•Spring AMQP
•Spring REST Docs
•Spring Mobile
•Spring For Android
•Spring Web Flow
•Spring Web Services
•Spring LDAP
•Spring Cloud
•Spring Scala
•Spring ROO
•Spring Flo
•Spring Session

Huawei Technologies Co Ltd. ., All rights reserved

Introduction to the Spring Framework

The Spring Framework provides comprehensive infrastructure support for developing Java applications.

Spring handles the infrastructure so you can focus on your application.

Spring enables you to build applications from "plain old Java objects" (POJOs) and to apply enterprise services non-invasively to POJOs.

The **Spring Framework** is an open source application framework and inversion of control container for the Java platform.

Huawei Technologies Co Ltd. ., All rights reserved

Introduction to the Spring Framework

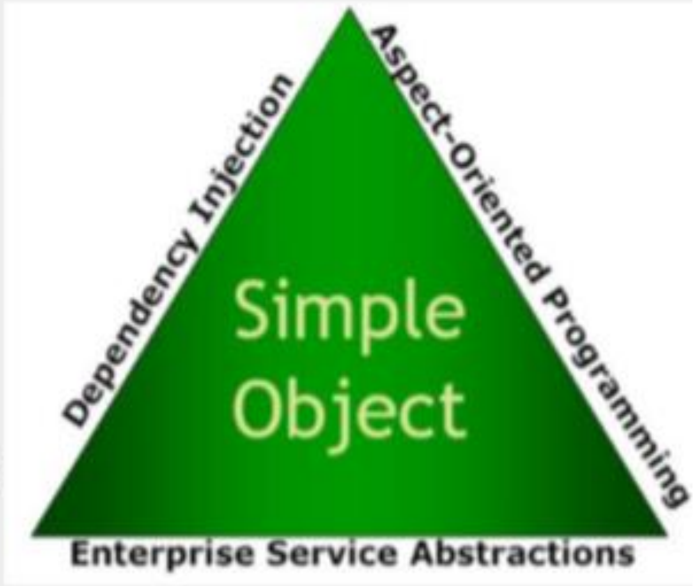
The first version was written by Rod Johnson in October 2002. The framework was first released under the Apache 2.0 license in June 2003

Version	Date
0.9	2002
1.0	2003
2.0	2006
3.0	2009
4.0	2013
5.0	2017

Huawei Technologies Co Ltd. ., All rights reserved

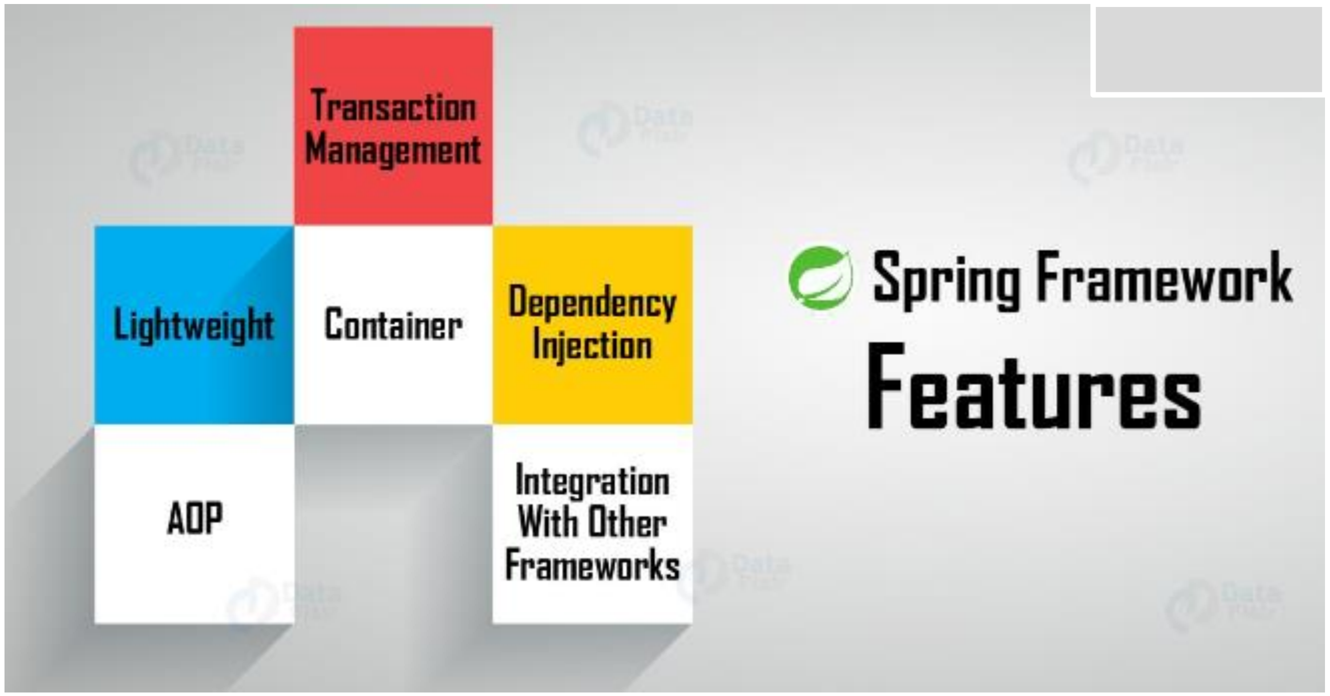
Spring Framework

- Spring is a lightweight inversion of control and aspect-oriented container framework
- Lightweight: in terms of both size and overhead
- Inversion of control: promotes loose coupling
- Aspect-oriented: enables cohesive development by separating application business logic from system services
- Container: contains and manages the life cycle and configuration of application objects
- Framework: possible to configure and compose complex applications from simpler components



Huawei Technologies Co Ltd. ., All rights reserved

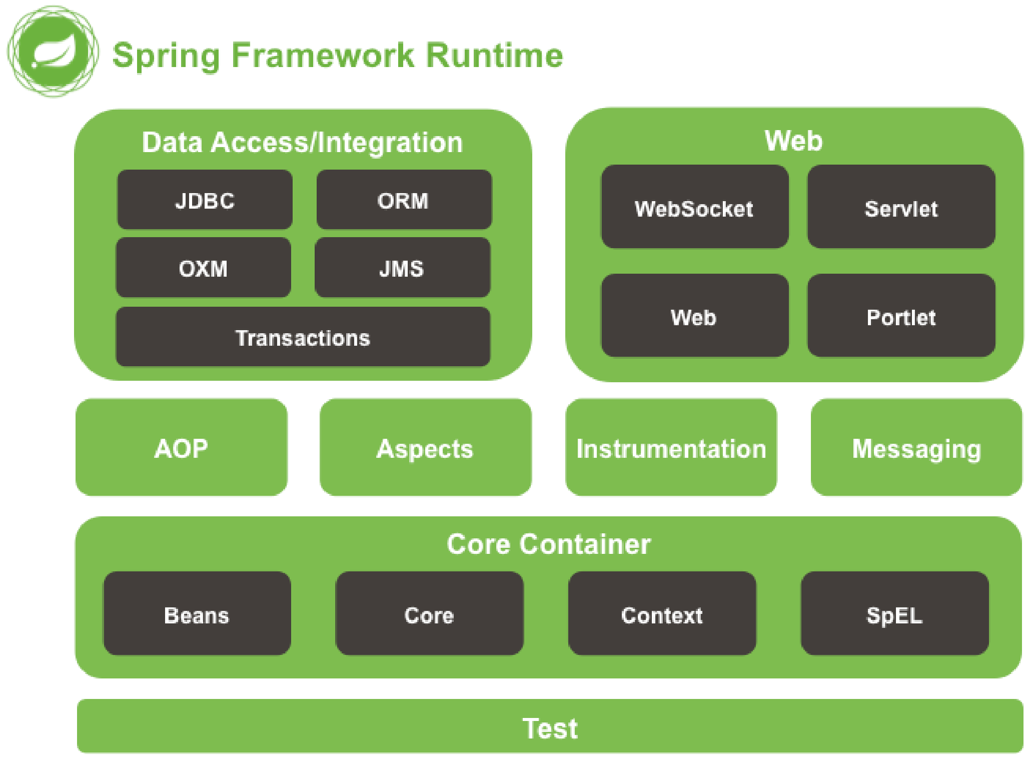
Spring Framework



Huawei Technologies Co Ltd. ., All rights reserved

Spring Framework

The **Spring Framework** consists of features organized into about 20 modules.



The spring-core and spring-beans modules provide the fundamental parts of the framework, including the IoC and Dependency Injection features.

Huawei Technologies Co Ltd. ., All rights reserved

The Spring IoC Container

•The basic concept of the Inversion of Control pattern (also known as dependency injection) is that we do not create our objects but describe how they should be created.

•The **IoC** container enforces the *dependency injection* pattern for our components, leaving them loosely coupled and allowing us to code to abstractions.

The ***org.springframework.beans*** and ***org.springframework.context*** packages are the basis for Spring Framework's IoC container

Huawei Technologies Co Ltd. ., All rights reserved

Dependency Injection (DI)

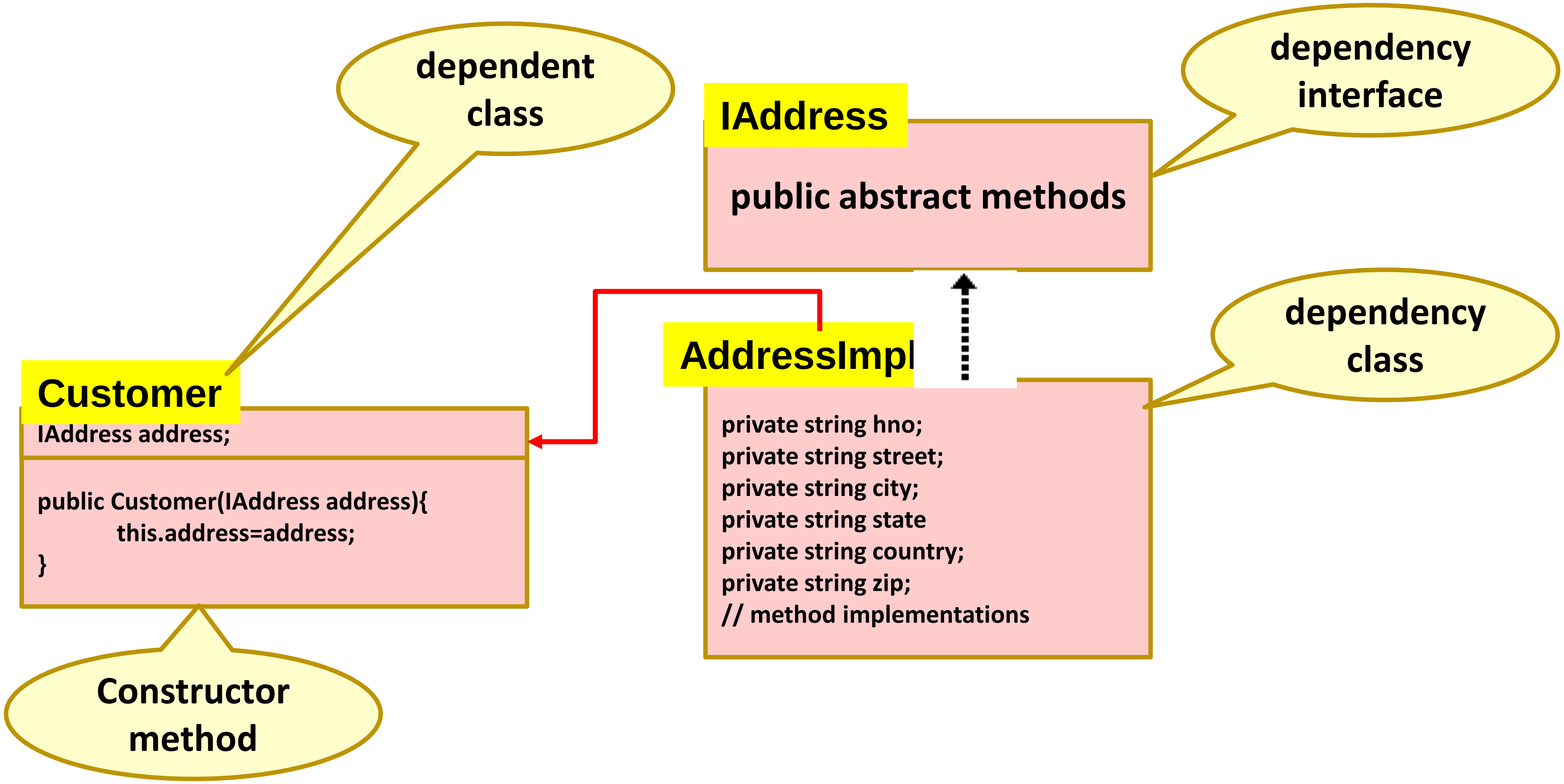
Martin Fowler, a British software engineer popularized the term Dependency Injection as a form of Inversion of Control, identifies three ways in which an object can receive a reference to an external module:

<u>**Three types of dependency injection**</u>

1. *constructor injection*: the dependencies are provided through a class constructor.
2. *setter injection*: the client exposes a setter method that the injector uses to inject the dependency.
3. *interface injection*: the dependency provides an injector method that will inject the dependency into any client passed to it. Clients must implement an interface that exposes a setter method that accepts the dependency**.**

Note: Spring supports constructor and setter injection.

Constructor Injection Example

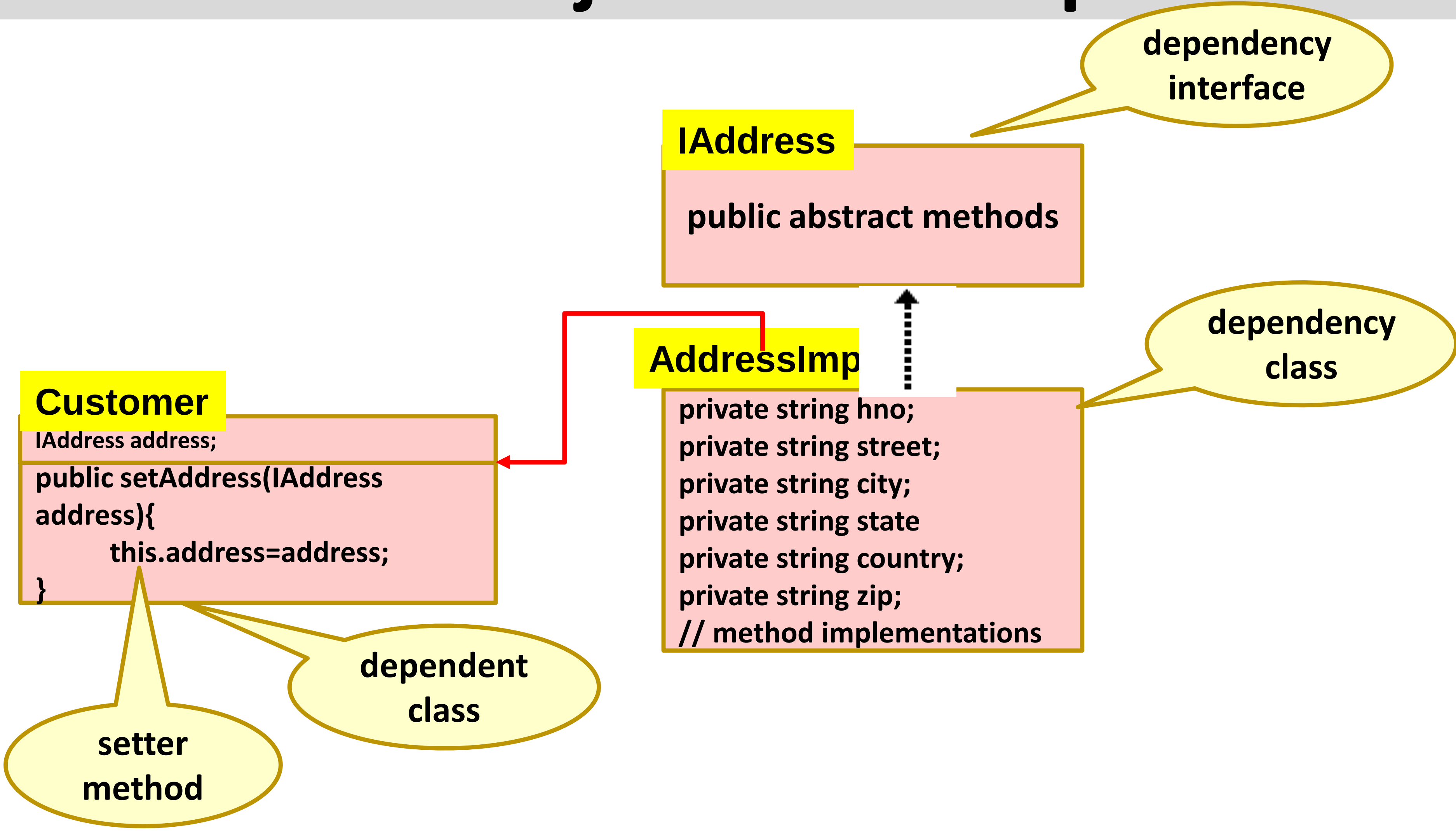


```
IAddress address;

public Customer(IAddress address){
        this.address=address;
}
```

```
private string hno;
private string street;
private string city;
private string state
private string country;
private string zip;
// method implementations
```

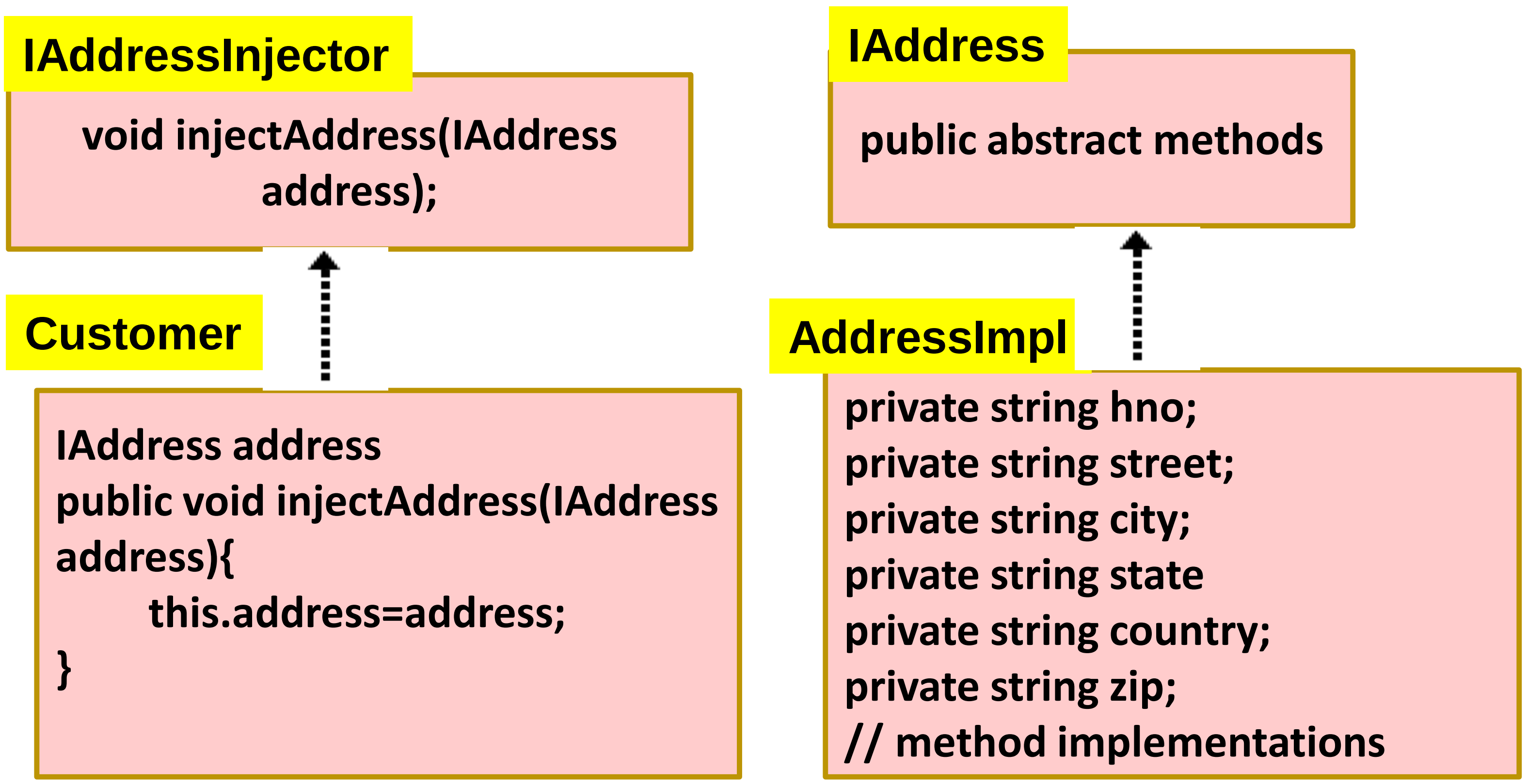
IoC container creates instance of dependency class, Address and passes to the constructor method of dependent class, Customer

Setter Injection Example



IoC container creates instance of dependency class, Address and passes to the setter method of dependent class, Customer

Interface Injection Example



```
void injectAddress(IAddress
address);
```

```
public abstract methods
```

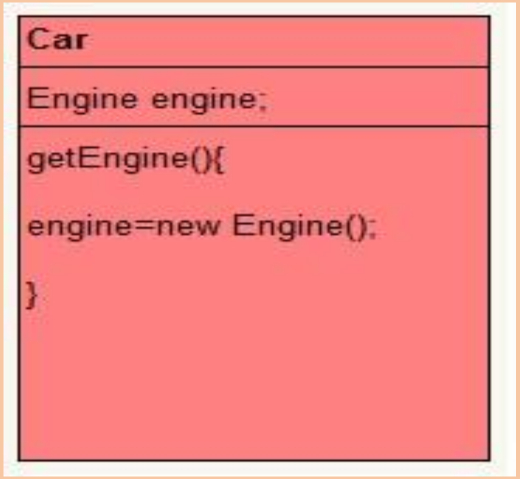
```
IAddress address
public void injectAddress(IAddress
address){
        this.address=address;
}
```

```
private string hno;
private string street;
private string city;
private string state
private string country;
private string zip;
// method implementations
```

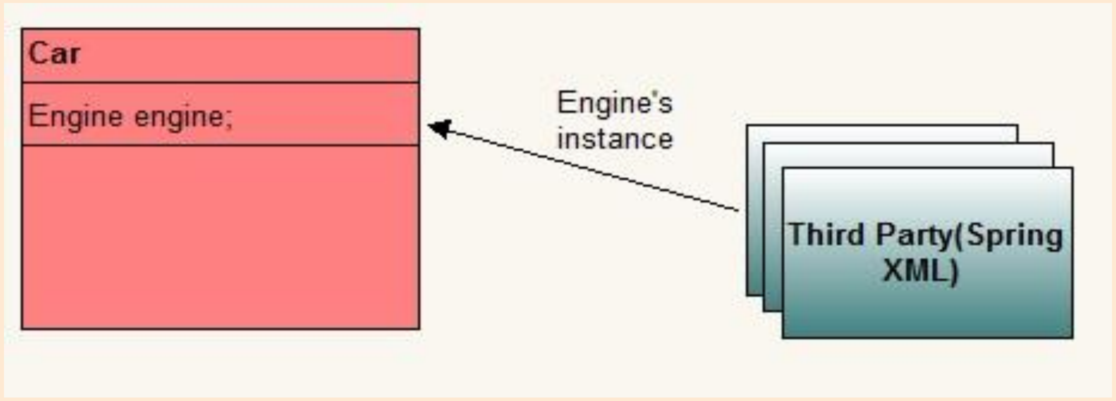
Inversion of Control and Dependency Injection

Inversion of control (IoC) is at the heart of Spring framework.

Normal Method



Dependency Injection



Inversion of Control and Dependency Injection

The container is responsible for managing object lifecycles of specific objects: creating these objects, calling their initialization methods, and configuring these objects by wiring them together.

Objects created by the container are also called managed objects or beans.

The container can be configured by loading XML (Extensible Markup Language) files or detecting specific Java annotations on configuration classes.

These data sources contain the bean definitions that provide the information required to create the beans.

Bean Factory and Application Context

In Spring, the application objects will live within the Spring container.

The Spring IOC container is mainly responsible for creating the objects, wiring them together, configuring them and managing their complete lifecycle i.e. from initializing to destruction.

There are various implementations of Spring Containers.

The two most common implementations are:

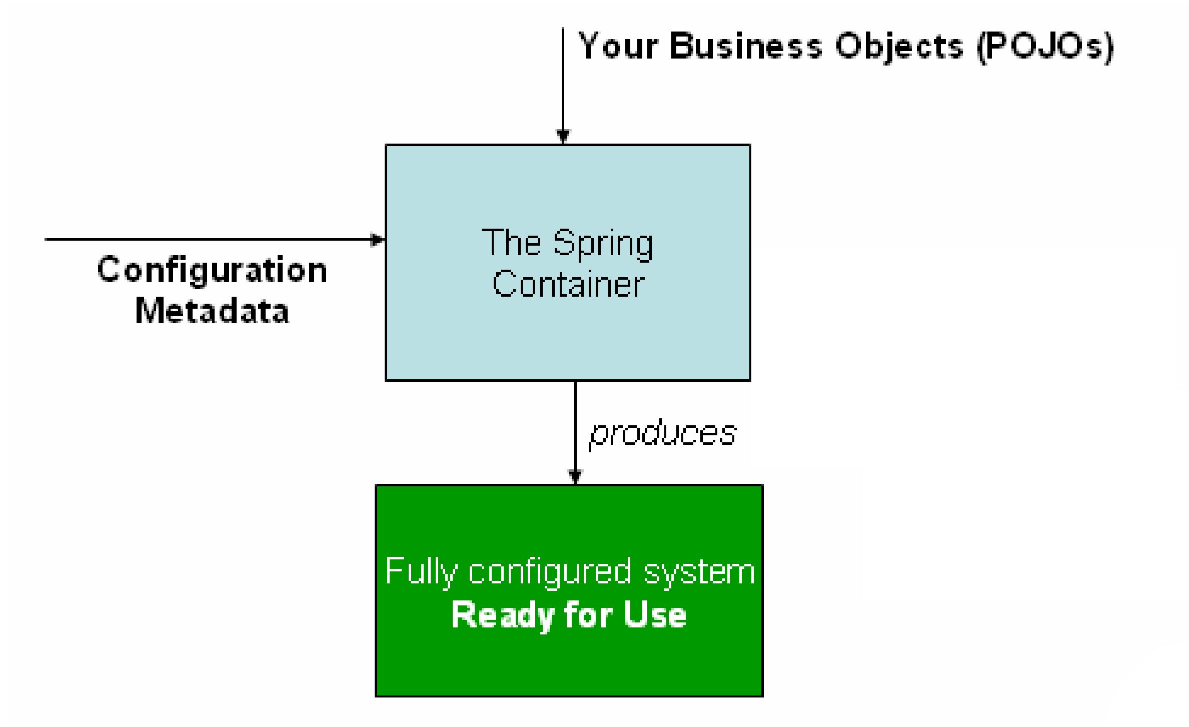
1. **BeanFactory (org.springframework.beans package) and**
2. **ApplicationContext (org.springframework.context package)**

Huawei Technologies Co Ltd. ., All rights reserved

The Spring IoC container

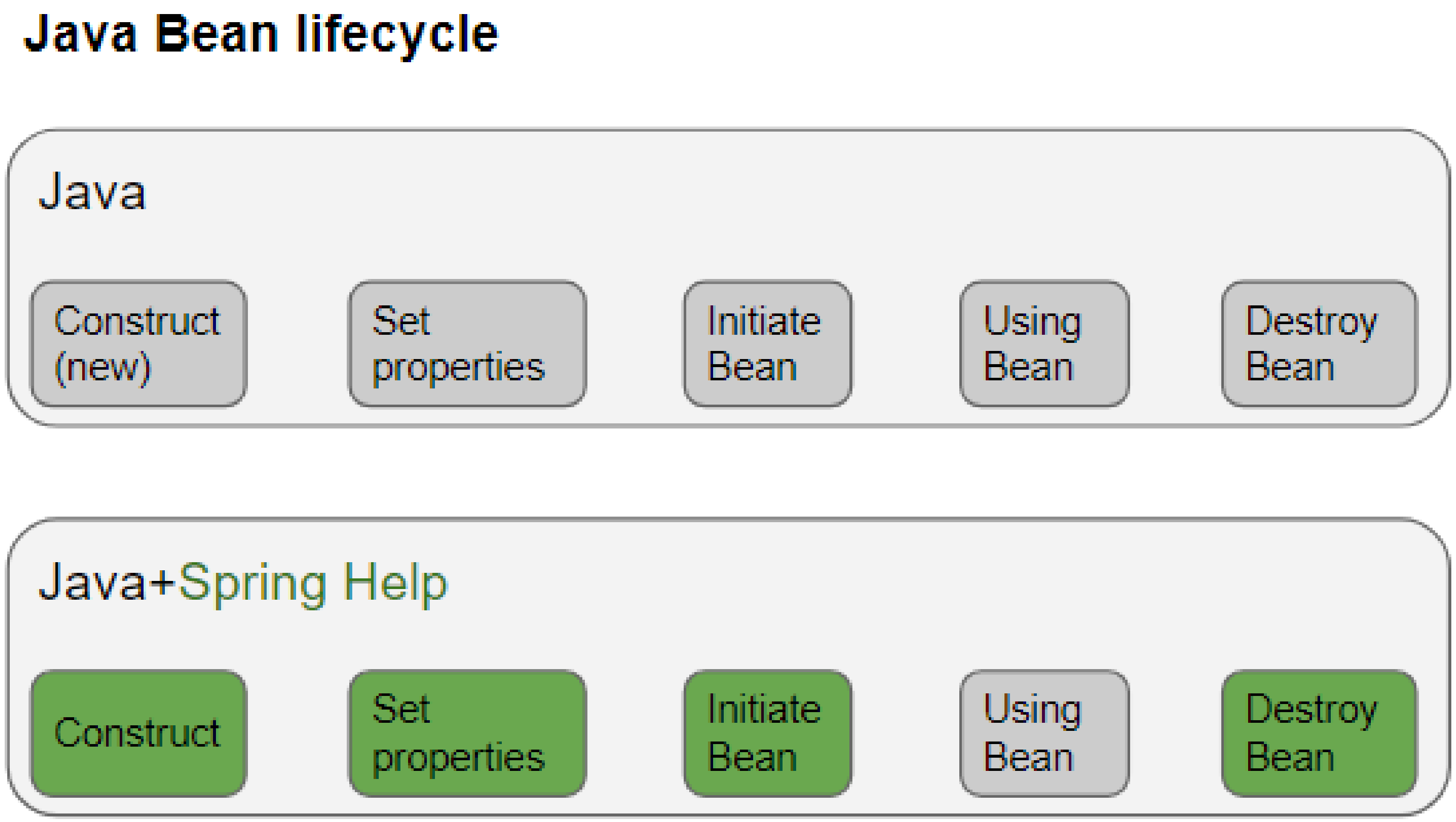
High-level view of how Spring works.

Our application classes are combined with configuration metadata so that after the **ApplicationContext** is created and initialized, we have a fully configured and executable system or application.



Huawei Technologies Co Ltd. ., All rights reserved

Java Bean Lifecycle

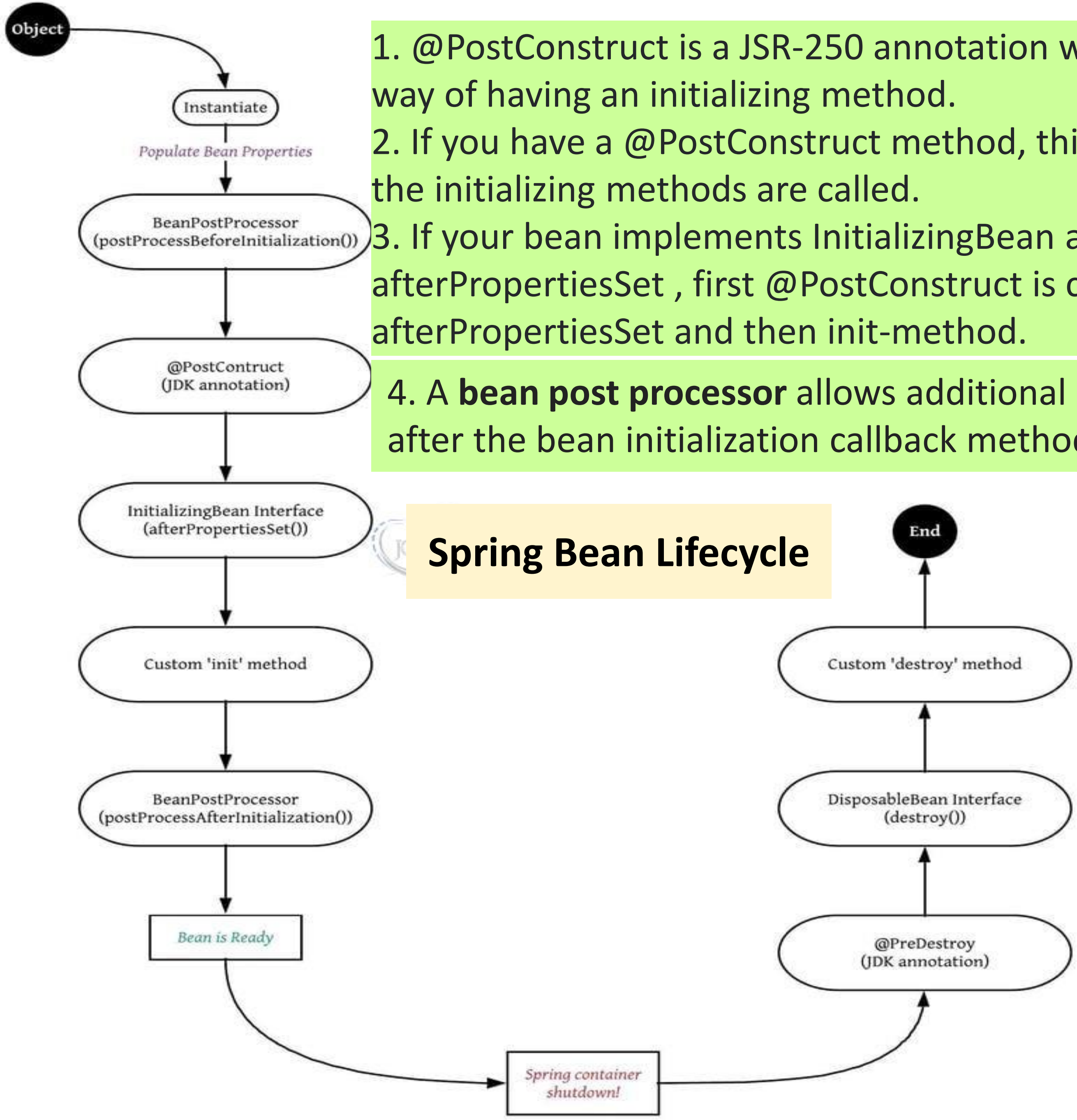


Huawei Technologies Co Ltd. ., All rights reserved

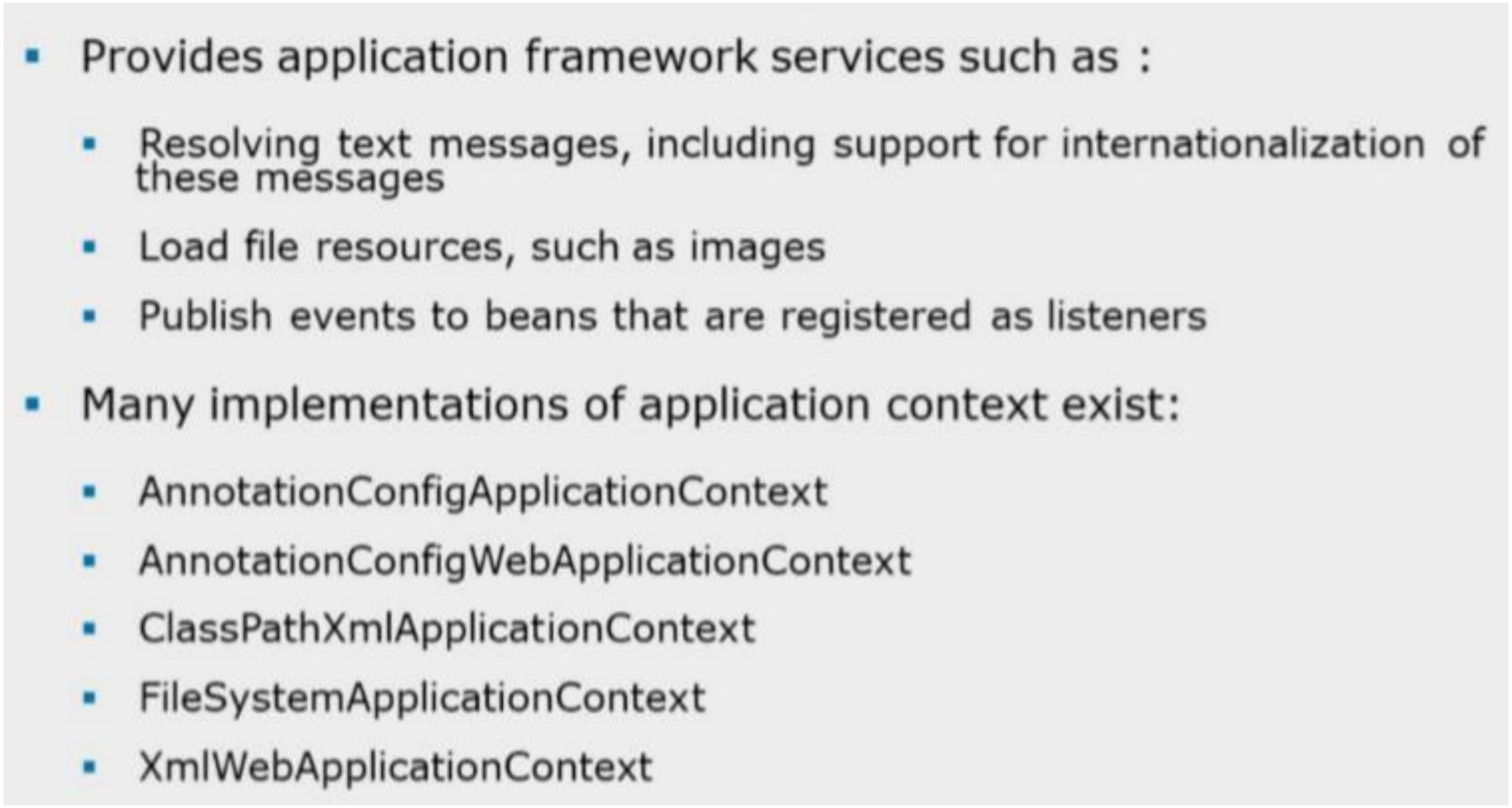
1. @PostConstruct is a JSR-250 annotation while init-method is Spring's way of having an initializing method.
2. If you have a @PostConstruct method, this will be called first before the initializing methods are called.
3. If your bean implements InitializingBean and overrides afterPropertiesSet , first @PostConstruct is called, then the afterPropertiesSet and then init-method.

4. A **bean post processor** allows additional processing before and after the bean initialization callback method.

Spring Bean Lifecycle


Object
Instantiate
Populate Bean Properties
BeanPostProcessor (postProcessBeforeInitialization())
@PostContruct (JDK annotation)
InitializingBean Interface (afterPropertiesSet())
Custom 'init' method
BeanPostProcessor (postProcessAfterInitialization())
Bean is Ready
Spring container shutdown!
@PreDestroy (JDK annotation)
DisposableBean Interface (destroy())
Custom 'destroy' method
End

Bean Container : ApplicationContext



- Provides application framework services such as :
  - Resolving text messages, including support for internationalization of these messages
  - Load file resources, such as images
  - Publish events to beans that are registered as listeners
- Many implementations of application context exist:
  - AnnotationConfigApplicationContext
  - AnnotationConfigWebApplicationContext
  - ClassPathXmlApplicationContext
  - FileSystemApplicationContext
  - XmlWebApplicationContext

Huawei Technologies Co Ltd. ., All rights reserved
Huawei Technologies Co Ltd. ., All rights reserved

Implementations of ApplicationContext

Several implementations of the **ApplicationContext** interface are supplied out-of-the-box with Spring.

The Three most commonly used implementations of **ApplicationContext** are:

FileSystemXmlApplicationContext, ClassPathXmlApplicationContext and XmlWebApplicationContext

Example:

```
ApplicationContext context = new
FileSystemXmlApplicationContext("c://applicationContext.xml");

ApplicationContext context = new
ClasspathXmlApplicationContext("applicationContext.xml");
```

Huawei Technologies Co Ltd. ., All rights reserved
Huawei Technologies Co Ltd. ., All rights reserved

Configuration Metadata

•Configuration metadata is traditionally supplied in a simple XML format.

•Other forms of metadata with the Spring container:

Annotation-based configuration: Spring 2.5 introduced support for annotation-based configuration metadata.

Java-based configuration: Starting with Spring 3.0,

Huawei Technologies Co Ltd. ., All rights reserved

XML-based configuration 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="..." class="...“>
                <!-- collaborators and configuration for this bean go here -->
        </bean>

       <bean id="..." class=“...">
                <!-- collaborators and configuration for this bean go here -->
       </bean>

        <!-- more bean definitions go here -->
</beans>
```

Note : Create source folder, *resource and place configuration file in stand-alone projects*

The id attribute is a string that you use to identify the individual bean definition.
The class attribute defines the type of the bean and uses the fully qualified class name.

Huawei Technologies Co Ltd. ., All rights reserved

Versionless XSD

It is recommended to use the "versionless" XSDs, because they're mapped to the current version of the framework you're using in your application.

So, instead of writing spring-beans-4.0.xsd write as spring-beans.xsd

Applications and tools should never try to fetch those XSDs from the web, since those schemas are included in the JARs.

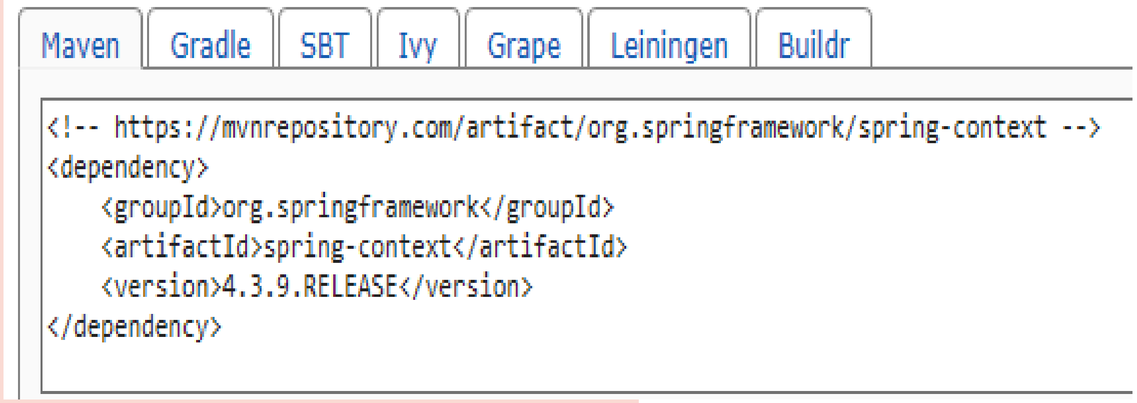
If they do, it usually means your app is trying to use a XSD that is more recent than the framework version you're using, or that your IDE/tool is not properly configured.

Huawei Technologies Co Ltd. ., All rights reserved

Sample Spring application using Maven build tool

pom.xml

```
.......

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <spring.version>4.3.9.RELEASE</spring.version>
  </properties>

  <dependencies>
    <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>4.3.9.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>4.3.9.RELEASE</version>
        </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
.........
```



```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>4.3.9.RELEASE</version>
</dependency>
```

Huawei Technologies Co Ltd. ., All rights reserved

Instantiating a container

Instantiating a Spring IoC container is straightforward.

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 context =
    new ClassPathXmlApplicationContext(new String[] {"services.xml", "daos.xml"});
```

```
ApplicationContext context =
    new ClassPathXmlApplicationContext(“application-context.xml”);
```

Huawei Technologies Co Ltd. ., All rights reserved

Address and Person POJOs

```
public class Address {
private String houseNumber;
private String street;
private String city;
private String state;
private String country;
private Long pinCode;

…

}
```

```
public class Person {
private Long adharCardNumber;
private String personName;
private Address residentialAddress;
private Address permanentAddress;


…
}
```

Huawei Technologies Co Ltd. ., All rights reserved

spring.xml

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

 <bean id="addressBean1" class="com.varaunited.trg.model.Address" >
 <property name="houseNumber" value="3-4-178/3" />
 <property name="street" value="Queens Street"></property>
 <property name="city" value="Hyderabad"></property>
 <property name="state" value="Telanaga"></property>
 <property name="country" value="India"></property>
 <property name="pinCode" value="500028"></property>
 </bean>
 <bean id="addressBean2" class="com.varaunited.trg.model.Address"
 c:houseNumber="5-6-167/8" c:street="Kings Street"
 c:city="Hyderabad" c:state="Telangana" c:country="India"
 c:pinCode="500029" >
 </bean>
<bean id="personBean" class="com.varaunited.trg.model.Person">
 <constructor-arg name="adharCardNumber" value="786745352879"/>
 <constructor-arg name="personName" value="Smith"/>
 <constructor-arg name="residentialAddress" ref="addressBean1"/>
 <constructor-arg name="permanentAddress" ref="addressBean2"/>
 </bean>
 </beans>
```

Tester class

```java
public class App {
    public static void main( String[] args )    {
       ApplicationContext context=
       new ClassPathXmlApplicationContext("spring.xml");

       Address address1= (Address) context.getBean("addressBean1");
       System.out.println(address1);
       Address address2= (Address) context.getBean("addressBean2");
       System.out.println(address2);

       Person person=(Person) context.getBean("personBean");
       System.out.println(person.getAdharCardNumber());
       System.out.println(person.getPersonName());
       /*((AbstractApplicationContext)context).close();*/
       ((AbstractApplicationContext)context).registerShutdownHook();
    }
}
```

Huawei Technologies Co Ltd. ., All rights reserved

Constructor Namespaces (P’s and C’s)

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

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigur
<property name="locations" value="classpath:messages.properties"/>
</bean>

 <bean id="addressBean3" class="com.varaunited.trg.model.Address"
 c:houseNumber="5-6-167/8" c:street="Kings Street"
 c:city="Hyderabad" c:state="Telangana" c:country="India"
 c:pinCode="500029" init-method="initializeBean"
 destroy-method="destroyBean" scope="prototype" autowire-candidate="false">
 </bean>

 <bean id="addressBean4" class="com.varaunited.trg.model.Address"
 p:houseNumber="5-6-197/1" p:street="Queens Street"
 p:city="Hyderabad" p:state="Telangana" p:country="India"
 p:pinCode="500029" autowire-candidate="false">
 </bean>
 </beans>
```

Bean Properties

Properties	Description
class	This attribute is mandatory and specify the bean class to be used to create the bean.
name	This attribute specifies the bean identifier uniquely. In XML-based configuration metadata, you use the id and/or name attributes to specify the bean identifier(s).
scope	This attribute specifies the scope of the objects created from a particular bean definition
constructor-arg	This is used to inject the dependencies
properties	This is used to inject the dependencies.
autowiring mode	This is used to inject the dependencies .
lazy-initialization mode	A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.
initialization method	A callback to be called just after all necessary properties on the bean have been set by the container.
destruction method	A callback to be used when the container containing the bean is destroyed.

Huawei Technologies Co Ltd. ., All rights reserved

Bean Scope

When defining a *<bean>* in Spring, you have the option of declaring a scope for that bean.

If you want Spring to return the same bean instance each time one is needed, you should declare the bean's scope attribute to be **singleton** *(which is default).*

To force Spring to produce a new bean instance each time one is needed, you should declare the bean's scope attribute to be **prototype**.

Huawei Technologies Co Ltd. ., All rights reserved

Bean Scopes

The Spring Framework supports following five scopes, three of which are available only if you use a web-aware **ApplicationContext**.

Scope	Description
singleton	This scopes the bean definition to a single instance per Spring IoC container (default).
prototype	This scopes a single bean definition to have any number of object instances.
request	This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext.
session	This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.
global-session	This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.

Huawei Technologies Co Ltd. ., All rights reserved

The singleton scope

The singleton scope:
If scope is set to singleton, the Spring *IoC* container creates exactly one instance of the object defined by that bean definition.
The default scope is always singleton

```
Message message1 = (Message) applicationContext.getBean("messageBean");
System.out.println(message1);

Message message2=(Message) applicationContext.getBean("messageBean");
System.out.println(message2);
```

If scope is set to prototype, the Spring IoC container creates new bean instance of the object every time a request for that specific bean is made by calling getBean() method.

As a rule, use the *prototype* scope for all *stateful* beans and the *singleton* scope for *stateless* beans.

Huawei Technologies Co Ltd. ., All rights reserved

Stateless and Stateful beans

Stateless beans: beans that are singleton and are initialized **only once**. The only state they have is a shared state.

These beans are created while the *ApplicationContext* is being initialized.

The same bean instance will be returned/injected during the lifetime of this *ApplicationContext.*

Stateful beans: beans that can carry state (instance variables).

These are created every time an object is required (like using the "new" operator in java).

Huawei Technologies Co Ltd. ., All rights reserved

The prototype scope

```
<!-- A bean definition with singleton scope -->
<bean id="..." class="..." scope="prototype">
        <!-- collaborators and configuration for this bean go here -->
</bean>
```

```
Message message1 = (Message) applicationContext.getBean("messageBean");
message1.setMessage("Welcome to Spring4 framework!");
System.out.println(message1.getMessage());
System.out.println(message1);
Message message2=(Message)applicationContext.getBean("messageBean");
System.out.println(message2);
System.out.println(message2.getMessage());
```

The objects references(hexadecimal representation of object's hash code) are displayed.

Huawei Technologies Co Ltd. ., All rights reserved

Bean Life Cycle

When a bean is instantiated, it may be required to perform some initialization to get it into a usable state.

Similarly, when the bean is no longer required and is removed from the container, some cleanup may be required.

Though, there are list of the activities that take place behind the scenes between the time of bean Instantiation and its destruction, two important bean *lifecycle callback methods* are required at the time of bean initialization and its destruction.

To define setup and teardown for a bean, we simply declare the **<bean>** with **init-method** and/or **destroy-method** parameters.

The **init-method** attribute specifies a method that is to be called on the bean immediately upon instantiation.

Similarly, **destroy-method** attribute specifies a method that is called just before a bean is removed from the container.

Huawei Technologies Co Ltd. ., All rights reserved

Initialization Callbacks

Initialization callbacks:
The **org.springframework.beans.factory.InitializingBean** interface specifies a single method:
void afterPropertiesSet() throws Exception; So you can simply implement above interface and initialization work can be done inside *afterPropertiesSet()* method as follows:

```
public class ExampleBean implements InitializingBean {
public void afterPropertiesSet() {
   // do some initialization work
}
}
```

Alternatively, We can provide the information in XML-based configuration metadata

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

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

Following is the class definition:

```
public class ExampleBean { public void init() { // do some initialization work } }
```

Huawei Technologies Co Ltd. ., All rights reserved

Destruction Callbacks

The **org.springframework.beans.factory.DisposableBean** interface specifies a single method:

void destroy() throws Exception; So you can simply implement above interface and finalization work can be done inside destroy() method as follows:

```
public class ExampleBean implements DisposableBean {
public void destroy() {
// do some destruction work
}
}
```

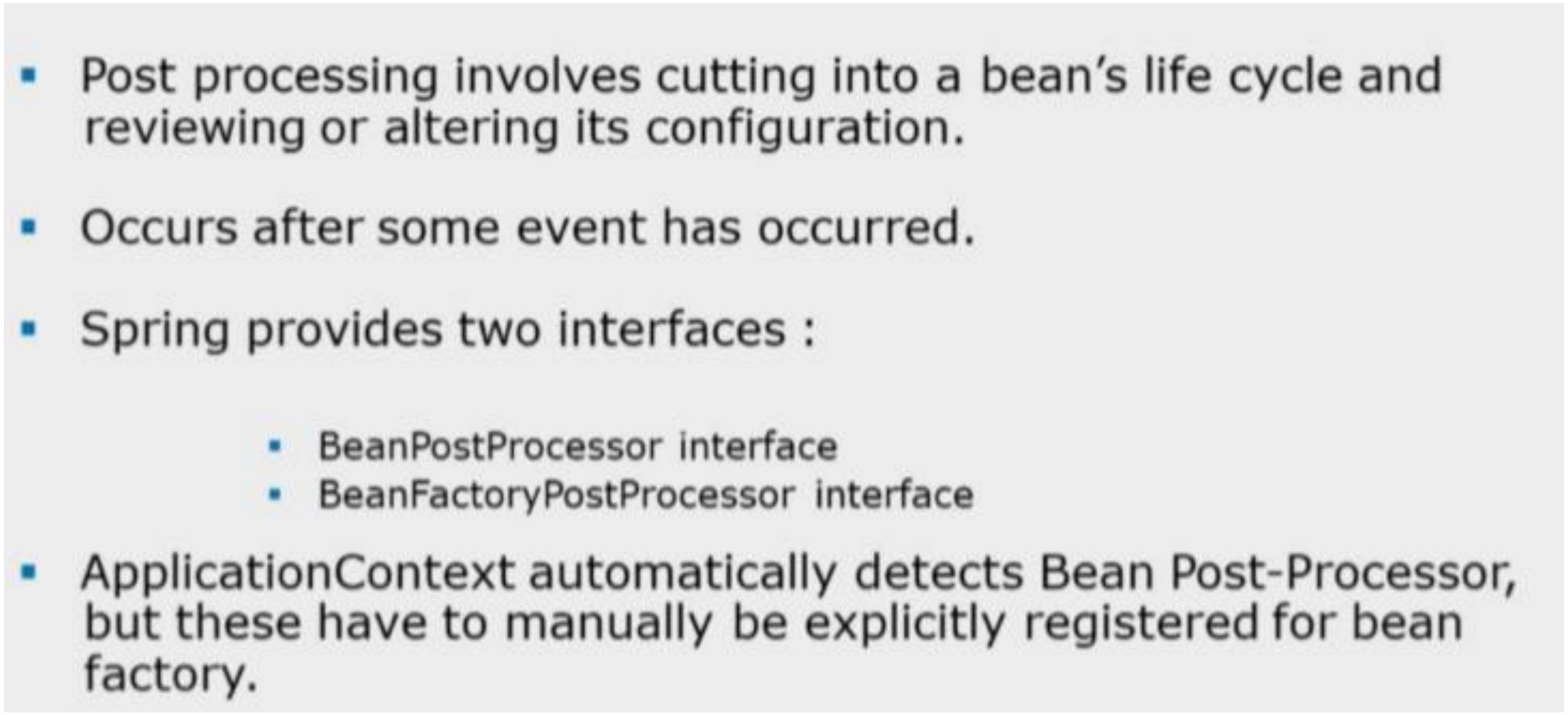
Alternatively, We can provide the information in XML-based configuration metadata

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

Following is the class definition:

```
public class ExampleBean {
            public void destroy() {
            // do some closure work
}
}
```

Customizing beans with BeanPostProcessor



- Post processing involves cutting into a bean's life cycle and reviewing or altering its configuration.
- Occurs after some event has occurred.
- Spring provides two interfaces :
  - BeanPostProcessor interface
  - BeanFactoryPostProcessor interface
- ApplicationContext automatically detects Bean Post-Processor, but these have to manually be explicitly registered for bean factory.

Huawei Technologies Co Ltd. ., All rights reserved
Huawei Technologies Co Ltd. ., All rights reserved

Customizing beans with BeanPostProcessor

A bean post processor allows **additional processing before and after the bean initialization callback method**.

Typically, bean post processors are **used for checking the validity of bean properties or altering bean properties according to certain criteria**.

ApplicationContext automatically detects BeanPostProcessor

Huawei Technologies Co Ltd. ., All rights reserved

Loading ResourceBundle by Spring IoC container

```
package com.deloitte.businesstier;

public class HelloWorld {
private String message1;
private String message2;

public HelloWorld(){

}

public HelloWorld(String message1,
       String message2) {
super();
this.message1 = message1;
this.message2 = message2;
}

public String getMessage1() {
return message1;
}

public void setMessage1(String
message1) {
this.message1 = message1;
}

public String getMessage2() {
return message2;
}

public void setMessage2(String
message2) {
this.message2 = message2;
}

}
```

Loading ResourceBundle by Spring IoC container

messages.properties

Note: Place static files in project's classpath

```
greet=Hello Spring
stream=Java EE
```

spring-resource.xml

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

 <bean
class="org.springframework.beans.factory.config.PropertyPlaceholderCon
figurer">
<property name="locations" value="classpath:messages.properties"/>
</bean>
 </beans>
```

Huawei Technologies Co Ltd. ., All rights reserved

Loading ResourceBundle by Spring IoC container

spring-hello.xml

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

 <import resource="spring-resource.xml"/>

 <bean  id="helloBean" class="com.deloitte.businesstier.HelloWorld">
 <constructor-arg index="0" value="${greet}"/>
 <constructor-arg index="1" value="${stream}"></constructor-arg>
 </bean>

 </beans>
```

Huawei Technologies Co Ltd. ., All rights reserved

Loading ResourceBundle by Spring IoC container

```
package com.deloitte.presentationtier;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.deloitte.businesstier.HelloWorld;

public class Tester {

public static void main(String[] args) {
ApplicationContext context=
        new ClassPathXmlApplicationContext("spring-hello.xml");

HelloWorld helloWorld=(HelloWorld)context.getBean("helloBean");
System.out.println(helloWorld.getMessage1());
System.out.println(helloWorld.getMessage2());
}

}
```

Huawei Technologies Co Ltd. ., All rights reserved

Injecting collections

Spring offers three types of collection configuration elements which are as follows:

Element	Description
<list>	This helps in wiring i.e. injecting list of values, allowing duplicates.
<set>	This helps in wiring a set of values but without any duplicates.
<map>	This can be used to inject a collection of name-value pairs where name and value can be of any type.

Huawei Technologies Co Ltd. ., All rights reserved

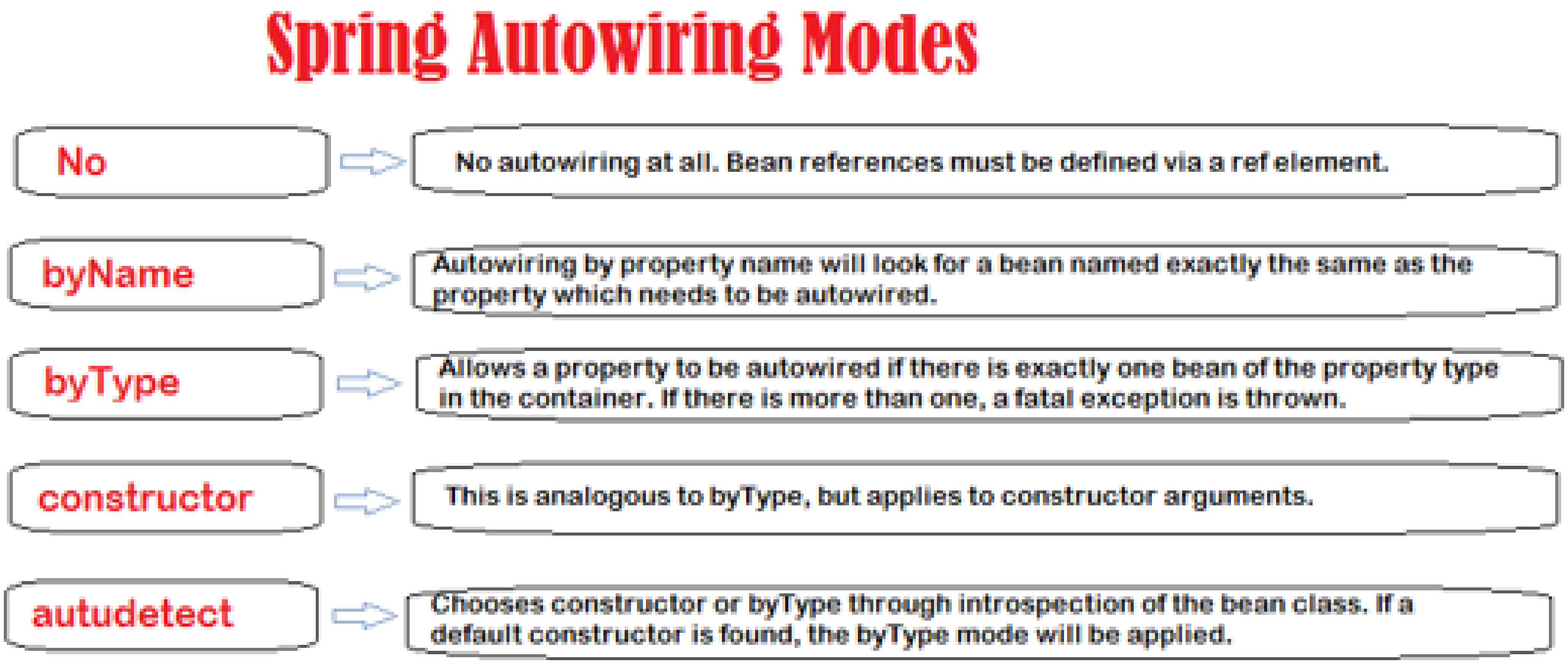
Spring Configuration file

```
…….
<bean id= "countryBean" class=“com.deloitte.businesstier.Country“>
<property name=“countryList">
	<list>
		<value>INDIA</value>
		<value>UK</value>
		<value>USA</value>
		<value>USA</value>
	</list>
 </property>
…….
<property name=“countryMap">
	<map>
		<entry key="1" value="INDIA"/>
		<entry key="2" value=“UK"/>
		<entry key="3" value="USA"/>
		<entry key="4" value="USA"/>
	</map>
</property>
…….
```

```
public class Country {
private List<String> countryList;
private Set<String>  countrySet;
private Map<String,String> countryMap;

// setter and getter methods for other properties

}
```

Auto-Wiring Modes

The autowiring functionality has five modes. The default mode is **no** i.e. by default autowiring is turned off.


Spring Autowiring Modes

No	No autowiring at all. Bean references must be defined via a ref element.
byName	Autowiring by property name will 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.
constructor	This is analogous to byType, but applies to constructor arguments.
autudetect	Chooses constructor or byType through introspection of the bean class. If a default constructor is found, the byType mode will be applied.

Spring bean autowire byType example

```
public class Employee{
    private String fullName;
    private Department department;

    public Department getDepartment() {
        return department;
    }
    public void setDepartment(Department department) {
        this.department = department;
    }
```

```
public class Department {
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

```
.....

   <bean id="employeeBean" class= " com.deloitte.businesstier.Employee" autowire="byType">
       <property name="fullName" value="Ravi Kumar"/>
    </bean>

    <bean id="departmentBean"  class="com.deloitte.businesstier.Department" >
       <property name="name" value="Human Resources" />
    </bean>

....
```

Huawei Technologies Co Ltd. ., All rights reserved

Spring bean autowire byName example

In Spring, **Autowiring by Name** means, if the bean Id is same as the property name of other bean, auto wire it.

```
public class Employee{
    private String fullName;
    private Department department;

    public Department getDepartment() {
        return department;
    }
    public void setDepartment(Department department) {
        this.department = department;
    }
   ........
```

```
public class Department {
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

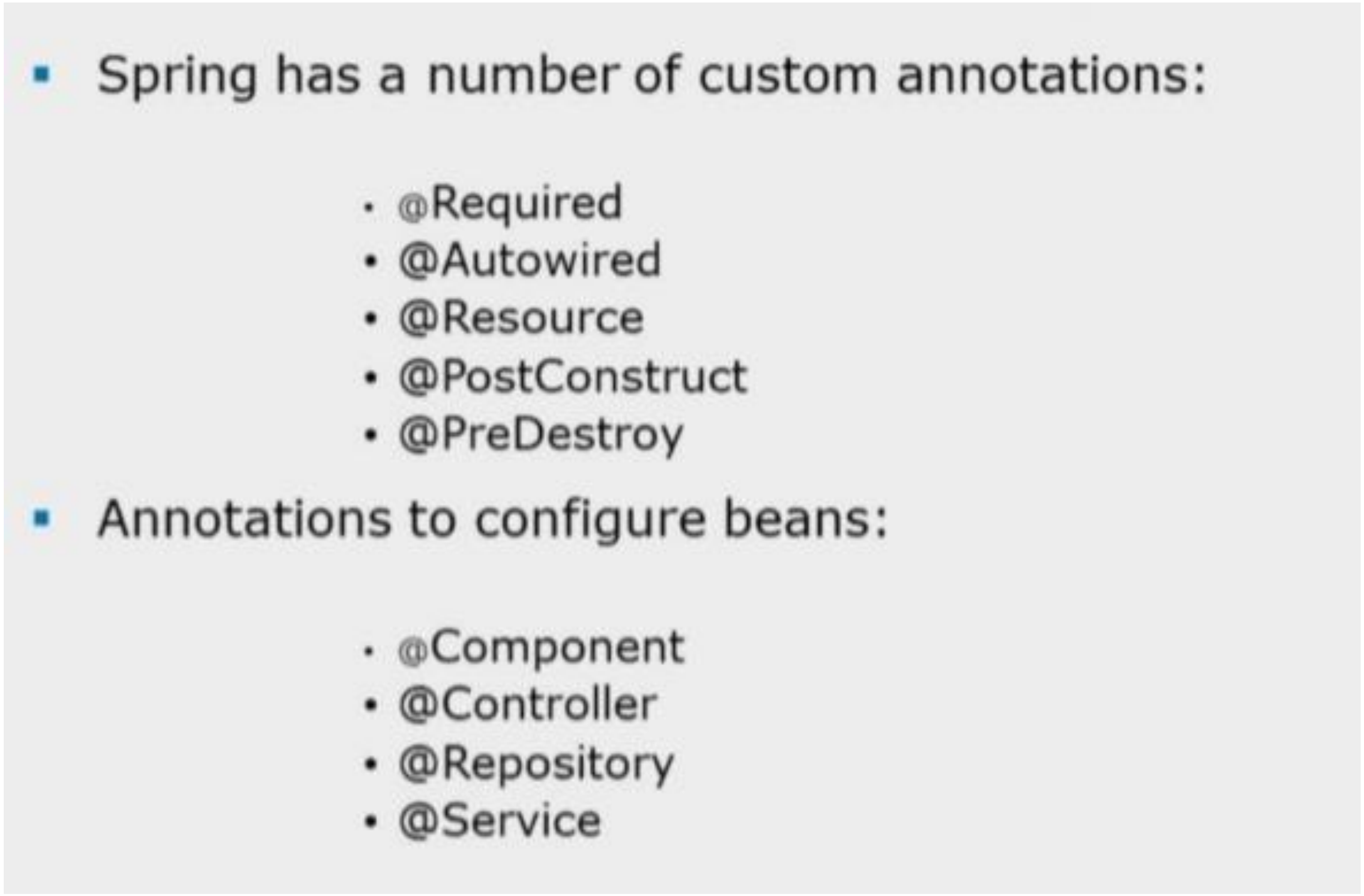
```
.....
  <bean id="employeeBean" class= " com.deloitte.businesstier.Employee" autowire="byName">
      <property name="fullName" value="Ravi Kumar"/>
   </bean>

   <bean id="department"   class="com.deloitte.businesstier.Department" >
      <property name="name" value="Human Resources" />
   </bean>
....
```

Huawei Technologies Co Ltd. ., All rights reserved

Spring Annotations

Huawei Technologies Co Ltd. ., All rights reserved

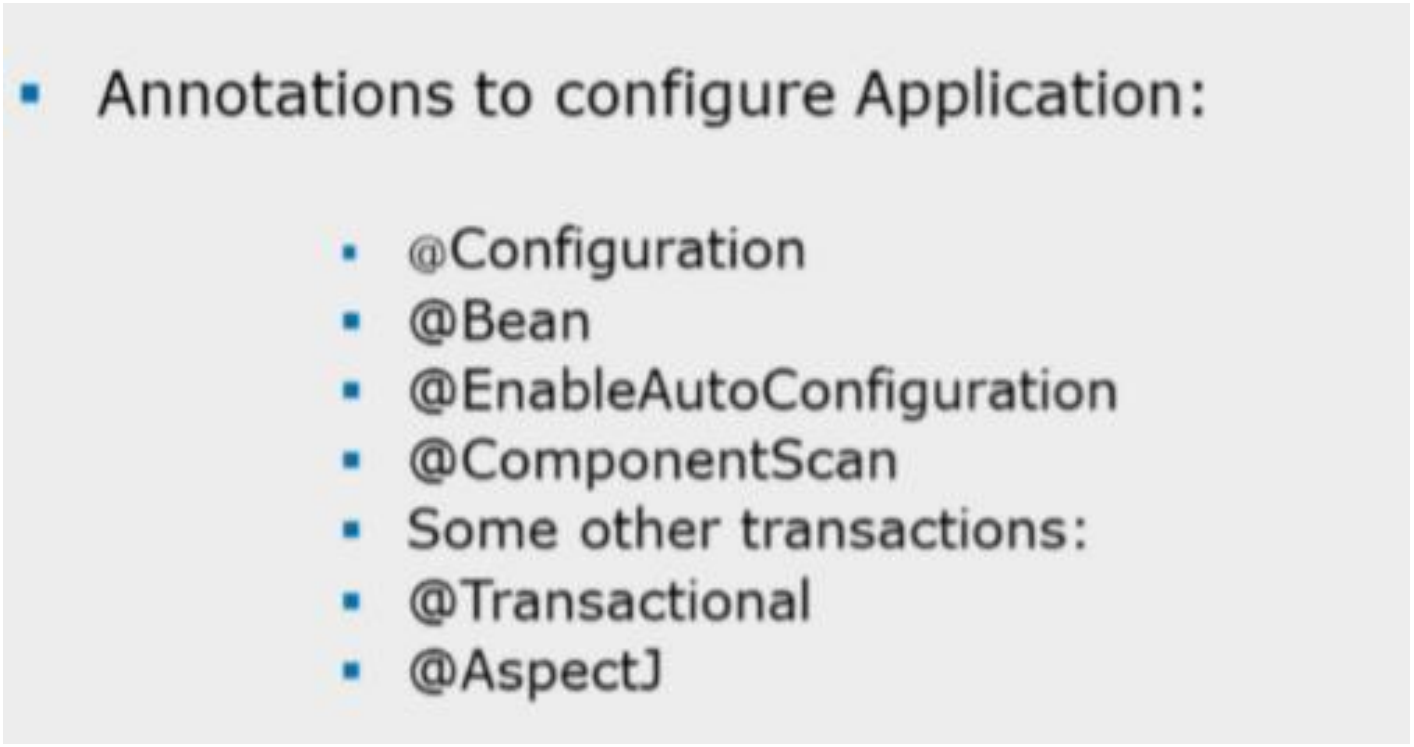
Spring Annotations



- Spring has a number of custom annotations:
  - @Required
  - @Autowired
  - @Resource
  - @PostConstruct
  - @PreDestroy
- Annotations to configure beans:
  - @Component
  - @Controller
  - @Repository
  - @Service

Huawei Technologies Co Ltd. ., All rights reserved

Spring Annotations



- Annotations to configure Application:
  - @Configuration
  - @Bean
  - @EnableAutoConfiguration
  - @ComponentScan
  - Some other transactions:
  - @Transactional
  - @AspectJ

Huawei Technologies Co Ltd. ., All rights reserved

Spring Annotations

Spring MVC Annotations	
@Controller	import org.springframework.stereotype.Controller;
@RequestMapping	import org.springframework.web.bind.annotation.RequestMapping;
@PathVariable	import org.springframework.web.bind.annotation.PathVariable;
@RequestParam	import org.springframework.web.bind.annotation.RequestParam;
@ModelAttribute	import org.springframework.web.bind.annotation.ModelAttribute;
@SessionAttributes	import org.springframework.web.bind.annotation.SessionAttributes;
Spring Security Annotations	
@PreAuthorize	import org.springframework.security.access.prepost.PreAuthorize;

Huawei Technologies Co Ltd. ., All rights reserved

Spring Annotations

Once **<context:annotation-config/>** is configured, we can start annotating our code to indicate that Spring should automatically wire values into properties, methods, and constructors.

Few important annotations to understand how they work:

S.N.	Annotation & Description
1	@Required The @Required annotation applies to bean property setter methods.
2	@Autowired The @Autowired annotation can apply to bean property setter methods, non-setter methods, constructor and properties. ***(specify <context:annotation-config/> in configuration file)***
3	@Qualifier The @Qualifier annotation along with @Autowired can be used to remove the ambiguity by specifying exactly which bean to be wired.
4	JSR-250 Annotations Spring supports JSR-250 based annotations which include **@Resource, @PostConstruct and @PreDestroy annotations *(which are part of javax.annotation package that comes along with Java EE 5 platform).***

Spring Annotations

For spring to process annotations, add the following lines in your application-context.xml file.

```
<context:annotation-config />
```

Note:

Spring supports both Annotation based and XML based configurations.

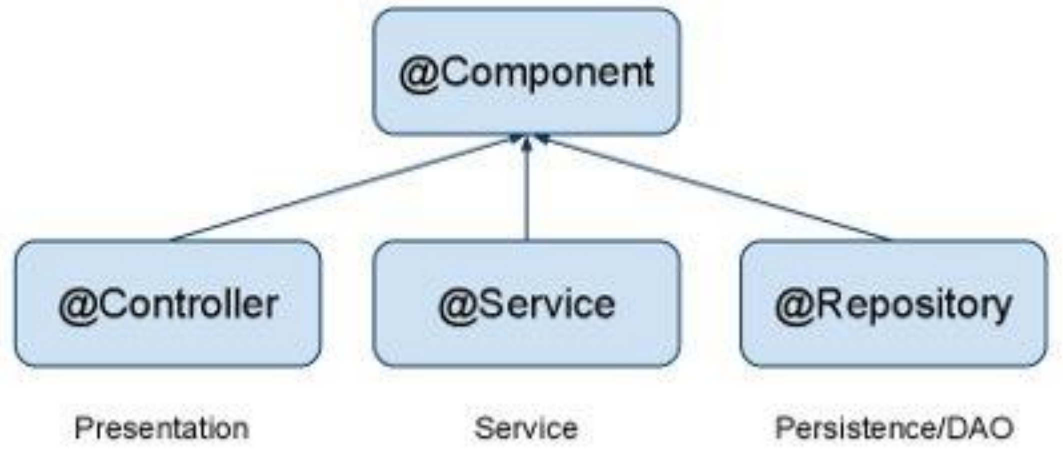
We can even mix them together in which case Annotation injection is performed before XML injection, thus the *later configuration will override the former* for properties wired through both approaches.

Huawei Technologies Co Ltd. ., All rights reserved

@Component Spring Annotation

@Component is a generic stereotype for any Spring-managed component.

@Repository, @Service, and @Controller are specializations of @Component for more specific use cases, for example, in the persistence, service, and presentation layers, respectively.



@Component (and @Service and @Repository) are used to auto-detect and auto-configure beans using class path scanning.

There's an implicit one-to-one mapping between the annotated class and the bean (i.e. one bean per class).

@Service, @Repository and @Component Annotations

@Service

Annotate all your service classes with @Service. All your business logic should be in Service classes.

```
    @Service
    public class CompanyService implements ICompanyService {
        @Autowired
          private ICompanyDAO companyDAO;
}
```

@Repository

Annotate all your DAO classes with @Repository. All your database access logic should be in DAO classes.

```
    @Repository
    public class CompanyDAO implements ICompanyDAO {
    …
    }
```

@Controller

To annotate classes of client/web layer.

```
    @Controller
    public class CompanyController {
    …
    }
```

Huawei Technologies Co Ltd. ., All rights reserved

@Autowired Annotation

*Spring beans can be wired **byName** or **byType**.*

@Autowired by default is a type driven injection.

Autowired byName

Example:

```
@Autowired(required=false)
@Qualifier(value="addressBean1")
private Address residentialAddress;

@Autowired(required=false)
@Qualifier(value="addressBean2")
```

OR

```
@Resource(name="addressBean2")
private Address permanentAddress;
```

Note:
@Qualifier is Spring framework annotation while @Resource is Java EE annotation

Huawei Technologies Co Ltd. ., All rights reserved

@Scope Spring Annotation

@Scope
As with Spring-managed components in general, the default and most common scope for auto-detected components is **singleton**. To change this default behaviour, use @Scope spring annotation.

```
@Component
@Scope("prototype")
public class ClassName {
...
}
```

Note: request, session & global-session are applicable for web applications

Spring Java Based Configuration

Java based configuration option enables you to write most of your Spring configuration without XML but with the help of few Java-based annotations explained below.

@Configuration & @Bean Annotations

Annotating a class with the **@Configuration** indicates that the class can be used by the Spring IoC container as a source of bean definitions. (acts like a configuration file).

The **@Bean** annotation tells Spring that a method annotated with @Bean will return an object that should be registered as a bean in the Spring application context.

Huawei Technologies Co Ltd. ., All rights reserved

Spring Java Based Configuration

The simplest possible @Configuration class would be as follows:

HelloWorldConfig .java

```
@Configuration
public class HelloWorldConfig {
 @Bean
public HelloWorld helloWorld(){
       return new HelloWorld();
}
}
```

Above code will be equivalent to the following XML configuration:

```
<beans> <bean id="helloWorld" class="com.deloitte.businesstier.HelloWorld" /> </beans>
```

Here the method name annotated with @Bean works as bean ID and it creates and returns actual bean.
The above configuration class can have declaration for more than one @Bean

Huawei Technologies Co Ltd. ., All rights reserved

Java Based Configuration Example

```
@Configuration
public class JavaBasedConfiguration {

@Bean(name="addressBean1")
public Address setHouseAddress(){
return new Address("3-4-569","MG Road","Bangalore","Karnataka","India",400156L);
}

@Bean(name="addressBean2")
public Address setOfficeAddress(){
return new Address("3-4-570","Kings Road","Bangalore","Karnataka","India",400156L);
}

@Bean(name="customerBean")
public Customer setCustomer(){
return new
Customer("101","Jones",setHouseAddress(),setOfficeAddress(),"jones@gmail.com");
}
}
```

Huawei Technologies Co Ltd. ., All rights reserved

Java Based Configuration Example

```
public class JavaConfigureTester {

public static void main(String[] args) {
ApplicationContext context=
new AnnotationConfigApplicationContext(JavaBasedConfiguration.class);

Customer customer=(Customer) context.getBean("customerBean");
System.out.println(customer);

}
}
```

Loading multiple configuration classes :

```
AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();
context.register(AppConfig.class, OtherConfig.class);
context.register(AdditionalConfig.class);
```

Huawei Technologies Co Ltd. ., All rights reserved

Configuration data

One can choose the following methods for providing configuration data:

1. Using XML file
2. Only Java Annotations
3. **Java Annotations with minimal XML file**
4. Java Configuration file - a java class with appropriate annotations

Huawei Technologies Co Ltd. ., All rights reserved

Event Handling in Spring

Spring has an eventing mechanism which is built around the *ApplicationContext.*

It can be used to **exchange information between different beans.**

We can make use of application events by listening for events and executing custom code.

Huawei Technologies Co Ltd. ., All rights reserved

Standard Context Events

In fact, there're a variety of built-in events in Spring, that **lets a developer hook into the lifecycle of an application and the context** and do some custom operation.
Even though we rarely use these events manually in an application, the framework uses it extensively within itself.
Event handling in the *ApplicationContext* is provided through the *ApplicationEvent* class and *ApplicationListener* interface.

1. ContextRefreshedEvent:

On either **initializing or refreshing the *ApplicationContext*,** Spring raises the *ContextRefreshedEvent*. Typically a refresh can get triggered multiple times as long as the context has not been closed.
Notice that, we can also have the event triggered manually by calling the *refresh()* method on the *ConfigurableApplicationContext* interface.

2. ContextStartedEvent

3. **ContextStoppedEvent**

4. **ContextClosedEvent**

Huawei Technologies Co Ltd. ., All rights reserved
Huawei Technologies Co Ltd. ., All rights reserved

Event Listener in Spring

If a bean implements the *ApplicationListener*, then every time an *ApplicationEvent* gets published to the ApplicationContext, that bean is notified.

To listen to a context event, a bean should implement the *ApplicationListener* interface which has just one method **onApplicationEvent()**.

```
public class CStartEventHandler implements ApplicationListener<ContextStartedEvent>{
        public void onApplicationEvent(ContextStartedEvent event) {
              System.out.println("ContextStartedEvent Received");
 }
}
```

```
public class CStopEventHandler implements ApplicationListener<ContextStoppedEvent>{
        public void onApplicationEvent(ContextStoppeddEvent event) {
              System.out.println("ContextStoppedEvent Received");
 }
}
```

Huawei Technologies Co Ltd. ., All rights reserved
Huawei Technologies Co Ltd. ., All rights reserved

Event Listener in Spring

Starting from version 4.2, To consume the published events. Spring supports an annotation-driven event listener – *@EventListener.*

This will automatically register an *ApplicationListener* based on the signature of the method :

```
@EventListener
public void handleContextRefreshEvent(ContextStartedEvent ctxStartEvt) {
    System.out.println("Context Start Event received.");
}
```

```
@EventListener
public void handleContextStoppedEvent(ContextStoppedEvent ctxStopEvt) {
    System.out.println("Context Stopped Event received.");
}
```

Huawei Technologies Co Ltd. ., All rights reserved
Huawei Technologies Co Ltd. ., All rights reserved

Running the Application

```
<bean id = "helloWorld" class = "com.trg.HelloWorld">
<property name = "message" value = "Hello World!"/> </bean>
<bean id = "cStartEventHandler" class = "com.trg.CStartEventHandler"/>
<bean id = "cStopEventHandler" class = "com.trg.CStopEventHandler"/>
```

```
public class MainApp {
public static void main(String[] args) {
ConfigurableApplicationContext context =
new ClassPathXmlApplicationContext("beans.xml");
// raise a start event.
context.start();
HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
obj.getMessage();
// raise a stop
event. context.stop(); } }
```

Huawei Technologies Co Ltd. ., All rights reserved
Huawei Technologies Co Ltd. ., All rights reserved
HUAWEI

Custom Event

Let’s create **a simple event class** – just a placeholder to store the event data. In this case, the event class holds a String message:

```
public class CustomSpringEvent extends ApplicationEvent {
    private String message;

    public CustomSpringEvent(Object source, String message) {
        super(source);
        this.message = message;
    }
    public String getMessage() {
        return message;
    }
}
```

Huawei Technologies Co Ltd. ., All rights reserved
Huawei Technologies Co Ltd. ., All rights reserved

Publisher

Now let’s create **a publisher of that event**. The publisher constructs the event object and publishes it to anyone who’s listening.
To publish the event, the publisher can simply inject the *ApplicationEventPublisher* and use the *publishEvent()* API

```
@Component
public class CustomSpringEventPublisher {
    @Autowired
    private ApplicationEventPublisher applicationEventPublisher;

    public void doStuffAndPublishAnEvent(final String message) {
        System.out.println("Publishing custom event. ");
        CustomSpringEvent customSpringEvent = new CustomSpringEvent(this, message);
        applicationEventPublisher.publishEvent(customSpringEvent);
    }
}
```

Huawei Technologies Co Ltd. ., All rights reserved
Huawei Technologies Co Ltd. ., All rights reserved

Listener

Finally, let’s create the listener.
The only requirement for the listener is to be a bean and implement *ApplicationListener* interface:

```
@Component
public class CustomSpringEventListener implements ApplicationListener<CustomSpringEvent> {
    @Override
    public void onApplicationEvent(CustomSpringEvent event) {
        System.out.println("Received spring custom event - " + event.getMessage());
    }
}
```

Huawei Technologies Co Ltd. ., All rights reserved
Huawei Technologies Co Ltd. ., All rights reserved

HUAWEI

Thank You!