# Basic Spring 4.0

Lesson 2:Introduction to Spring Framework, IoC

# Lesson Objectives

- Introduction to Spring Framework
  - Learn about the Spring Framework, its benefits and architecture
  - Learn about the IoC (Inversion of control) and how it allows wiring beans
  - Learns the types of bean factories and life-cycle of beans in these factories
  - Understand how to apply Annotations to Spring applications
- Injecting dependencies through setter and constructor injections
- Wiring Beans
- Bean containers
  - Life cycle of Beans in the factory container
  - BeanPostProcessors and BeanFactoryPostProcessors
- Annotation-based configuration

# Introduction

- December 1996 – JavaBeans makes its appearance.
  - Intended as a general-purpose means of defining reusable application components
  - Used more as a model for building user interface widgets
- Sophisticated applications often require services not directly provided by the JavaBeans specification
- March 1998 – EJB was published.
  - But EJBs are complicated in a different way, that is, they mandate deployment descriptors and plumbing code

# Introduction

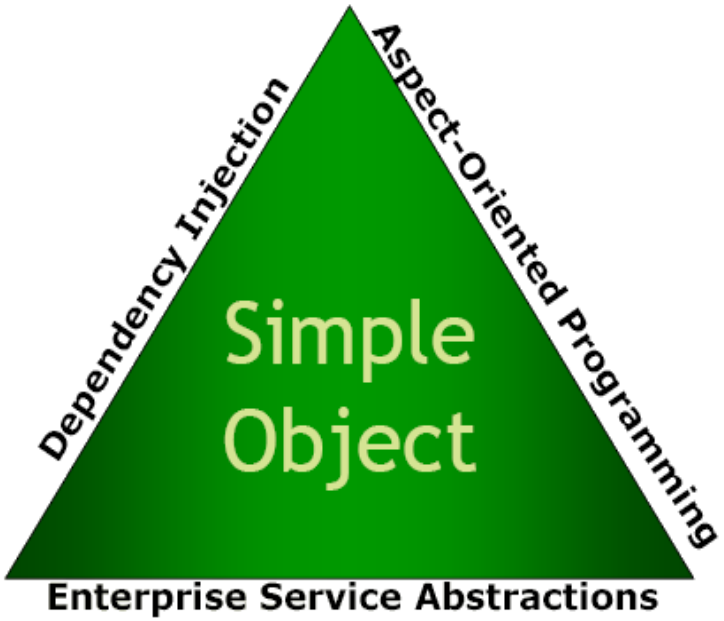
- Many successful applications were built based on EJB
  - But EJB never really achieved its intended purpose, which is to simplify enterprise application development
- Java development comes full circle
  - New programming techniques like including aspect-oriented programming (AOP) and inversion of control (IoC) are giving JavaBeans much of the power of EJB

# What is Spring framework?

- Spring is an open source framework created by Rod Johnson, Juergen Hoeller et all
- Addresses the complexity of enterprise application development
- Any java application can benefit from Spring in terms of simplicity, testability and loose coupling

# What is Spring framework?

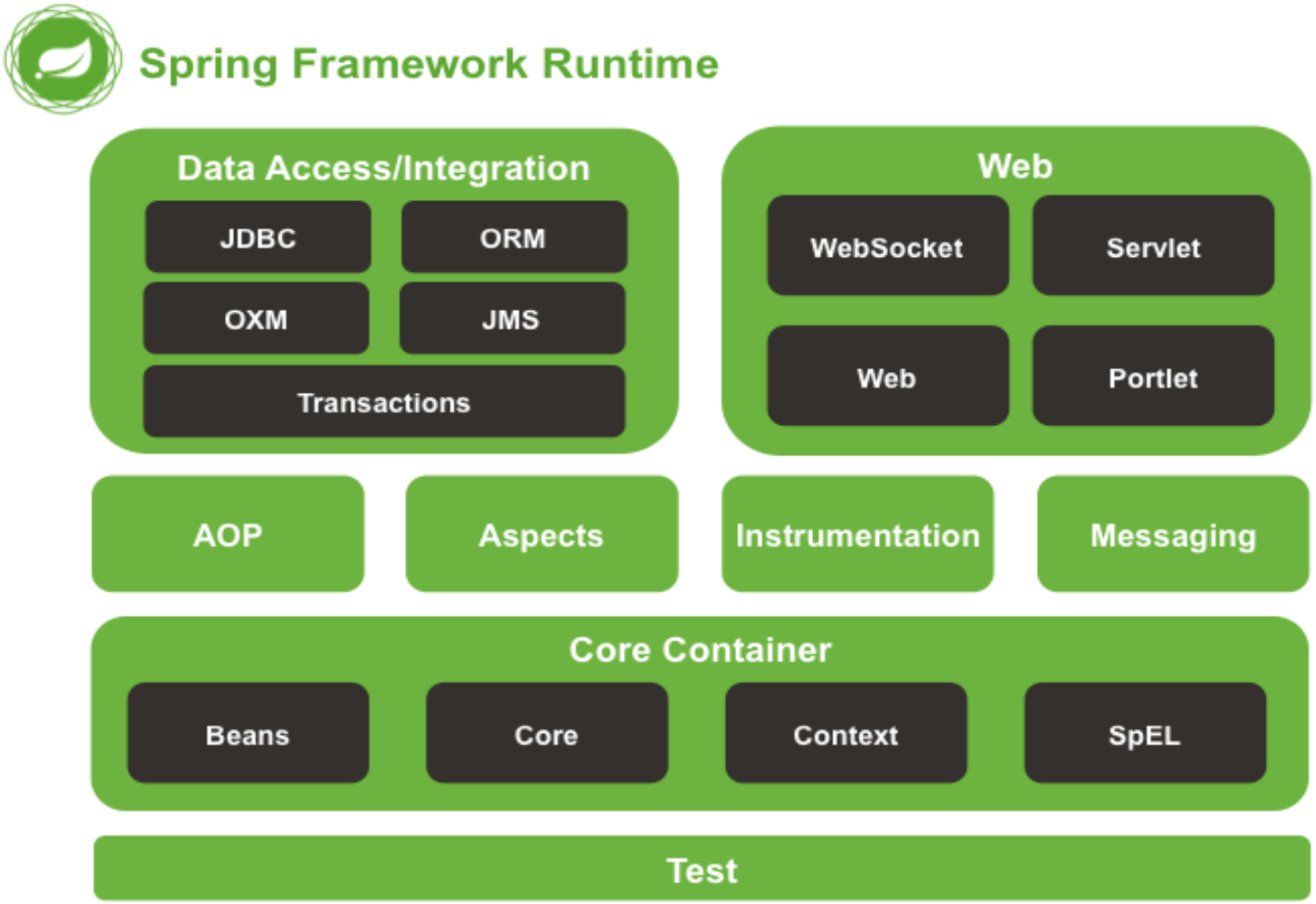
- Spring is a lightweight inversion of control and aspect-oriented container framework
  - Lightweight: in terms of both size and overhead
  - Inversion of control: promotes loose coupling
  - Aspect-oriented: enables cohesive development by separating application business logic from system services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



# Why Spring?

- Spring simplifies Java development
- With Spring, complexity of application is proportional to the complexity of the problem being solved
- Essence of Spring is to provide enterprise services to POJO.
- Spring employs four key strategies:
  - Lightweight and minimally invasive development with plain old Java objects (POJOs)
  - Loose coupling through dependency injection and interface orientation
  - Declarative programming through aspects and common conventions
  - Boilerplate reduction through aspects and templates

# Spring 4.0 architecture



# Dependency Injection

- Any enterprise application has objects  that depend on each other
- Resolving the dependency is termed as ‘Injecting Dependency’ which facilitates loose coupling.
- Choosing the low level implementation to be injected into the reference of the interface in higher level layer, is termed as ‘Inversion Of Control’

# Sample Code

```
package com.igate.di;
public class Person{
        private Address address;
        public Person() {this.address = new Address();}
        public Address getAddress() {return address;}
        public void setAddress(Address address) { this.address=address;}
}
```

- Tight coupling
- Rigid design

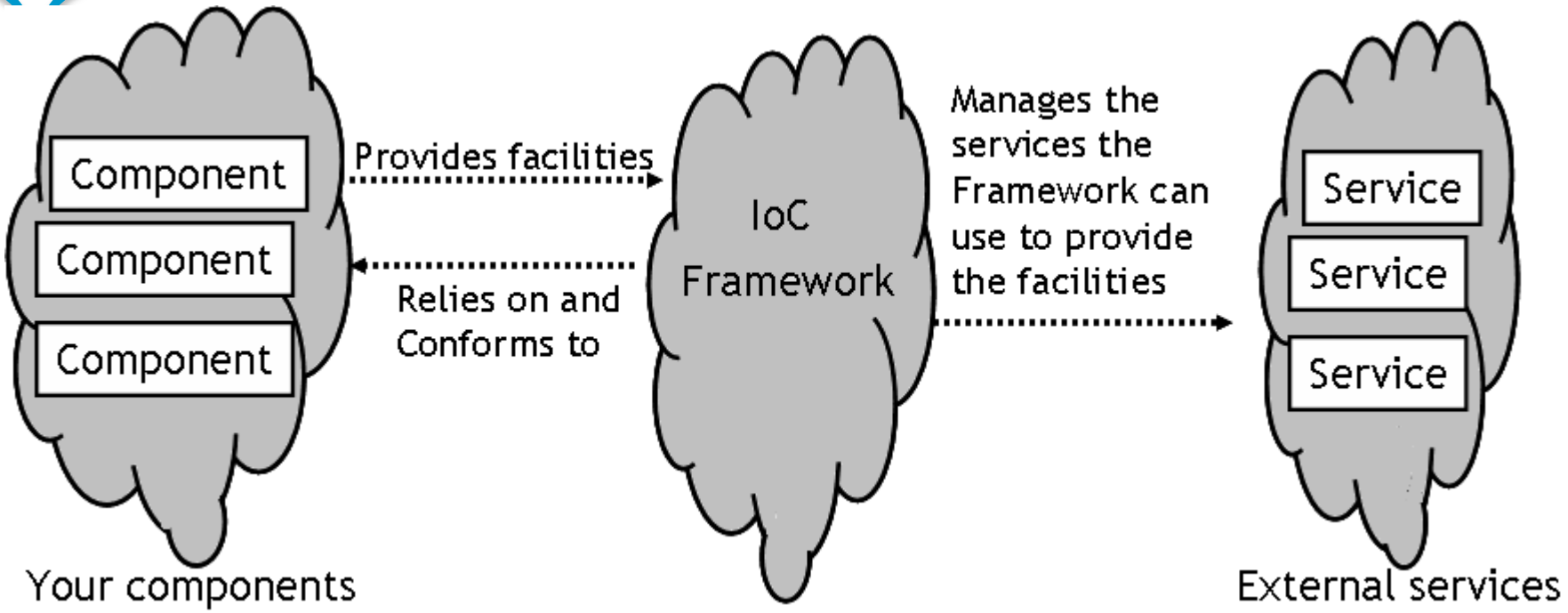
```
package com.igate.di;
public class Person{
        private Address address;
        public Person(Address address) {this.address = address;}
        public Address getAddress() {return address;}
        public void setAddress(Address address) { this.address=address;}
}
```

- Loose coupling
- Flexible Design
- Provisioning DI

```
package com.igate.di;
public class ResidenceAddress implements Address{..... }
```

- Design to Interface
- Enabling Loose Coupling, Flexibility

# IoC Concepts

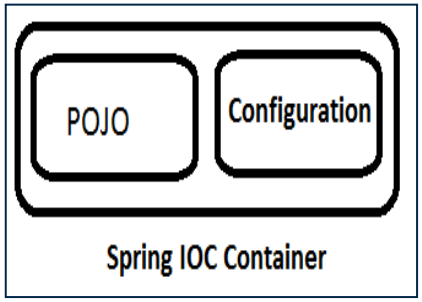



# IoC, Beans and BeanFactories

- Used to achieve loose coupling between several interacting components in an application.
- The IoC framework separates facilities that your components are dependent upon and provides the “glue” for connecting the components.
- DI it is specific type of IoC
- BeanFactory is the core of Spring’s DI container.
- In Spring, the term “bean” is used to refer to any component managed by the container.

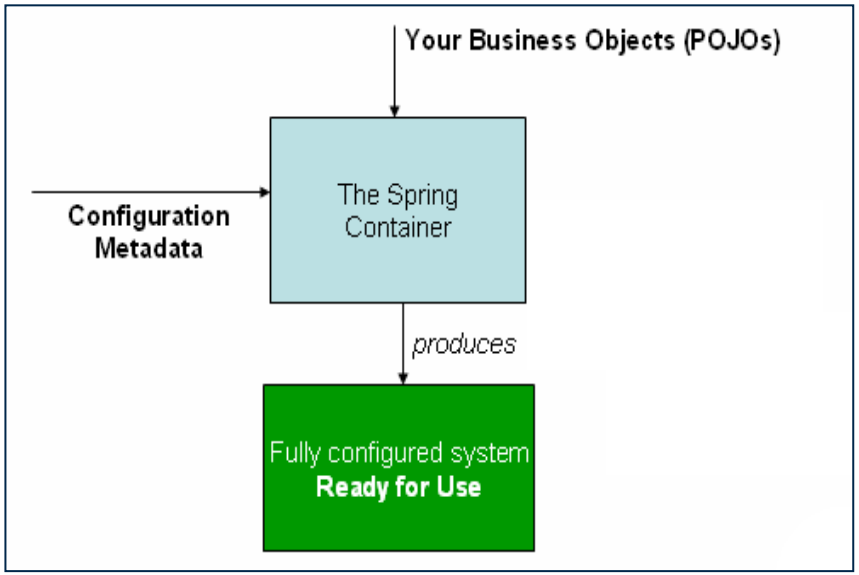
# Spring Beans and Configuration Metadata

- Spring Managed POJOs with business logic are termed as Spring Beans, Spring IOC container manages one or more beans
- Spring meta-data configuration can be …
- XML based
- Java based : Annotations
- Java based configurations are recommended practice since inclusion of Spring JavaConfig project
- Spring Beans and Configuration Metadata is combined and the Spring Framework's IoC container is created & initialized
- Spring IoC container is decoupled from configuration metadata format



# Spring IOC Container

- The Spring IOC container  instantiates, configures, and assembles the beans by reading configuration metadata
- It composes the application and interdependencies between  the objects
- At this stage, the application is fully configured and ready to use
- The Spring IOC Container manages the entire lifecycle of the Spring Beans



# Spring Jumpstart with HelloWorld

```
package training.spring;
   public class HelloWorld {
     public void sayHello(){
        System.out.println("Hello Spring 3.0");
     }
}
```

```
<?xml …..>
<beans ….>
<bean id="HWBean"  class =
      "training.spring.HelloWorld" />
</beans>
```

The Spring configuration file

```
public class HelloWorldClient {
   public static void main(String[] args) {
        XmlBeanFactory beanFactory = new XmlBeanFactory
                              (new  ClassPathResource("HelloWorld.xml"));
        HelloWorld bean = (HelloWorld) beanFactory.getBean("HWBean");
        bean.sayHello();
     }
}
```

Output:
Hello Spring 3.0

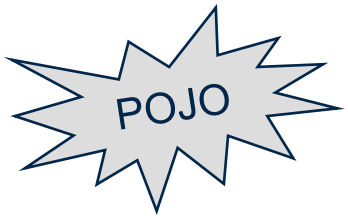
# Inversion of Control Approaches

- IoC pattern uses three different approaches to achieve decoupling of control of services from components:
  - Type 1: Setter Injection
  - Type 2: Constructor injection
  - Type 3: Interface injection

# Injecting dependencies via setter methods

```
public interface CurrencyConverter {
      public double dollarsToRupees(double dollars);
}
```

```
public class CurrencyConverterImpl implements CurrencyConverter {
     private double exchangeRate;
     public double getExchangeRate() {  return exchangeRate;   }
     public void setExchangeRate(double exchangeRate) {
              this.exchangeRate = exchangeRate;              }
     public double dollarsToRupees(double dollars) {
                    return dollars * exchangeRate;
   }
}
```



# Injecting dependencies via setter methods

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd">

    <bean id="currencyConverter"
                    class="training.Spring.CurrencyConverterImpl">
        <property name="exchangeRate" value="44.50" />
    </bean>
</beans>
```

The configuration file (CurrencyConverter.xml)

# Injecting dependencies via setter methods

The client application

```
public class CurrencyConverterClient {
   public static void main(String args[]) throws Exception {
        Resource res = new ClassPathResource("currencyconverter.xml");
        BeanFactory factory = new XmlBeanFactory(res);
        CurrencyConverter curr = (CurrencyConverter)
                                              factory.getBean("currencyConverter");
        double rupees = curr.dollarsToRupees(50.0);
        System.out.println("50 $ is "+rupees+" Rs.");
     }  }
```

Output:
CurrencyConverterImpl()
setExchangeRate()
dollarsToRupees()
50 $ is 2225.0 Rs.

# DemoSpring_1

- This demo illustrates how the container will instantiate the CurrencyConverter service using setter injection.

# Injecting dependencies via constructor

- Bean classes can be programmed with constructors that take enough arguments to fully define the bean at instantiation

```
<bean id="currencyConverter"  class="training.Spring.CurrencyConverterImpl">
    <constructor-arg>
            <value> 44.50 </value>
    </constructor-arg>
</bean>
```

```
public CurrencyConverterImpl(double er) {
    exchangeRate = er;
 }
```

# Injecting dependencies via constructor

- If a constructor has multiple arguments, then ambiguities among constructor arguments can be dealt with in two ways :
  - by index
  - by type

```
<beans>
 <bean id="currencyConverter"
                                class="training.Spring.CurrencyConverterImpl3">
  <constructor-arg><value>44.25</value></constructor-arg>
  <!--<constructor-arg index="0"><value>44.25</value></constructor-arg>-->
  <!--<constructor-arg type="double"><value>44.25</value></constructor-arg>-->
  </bean>
</beans>
```

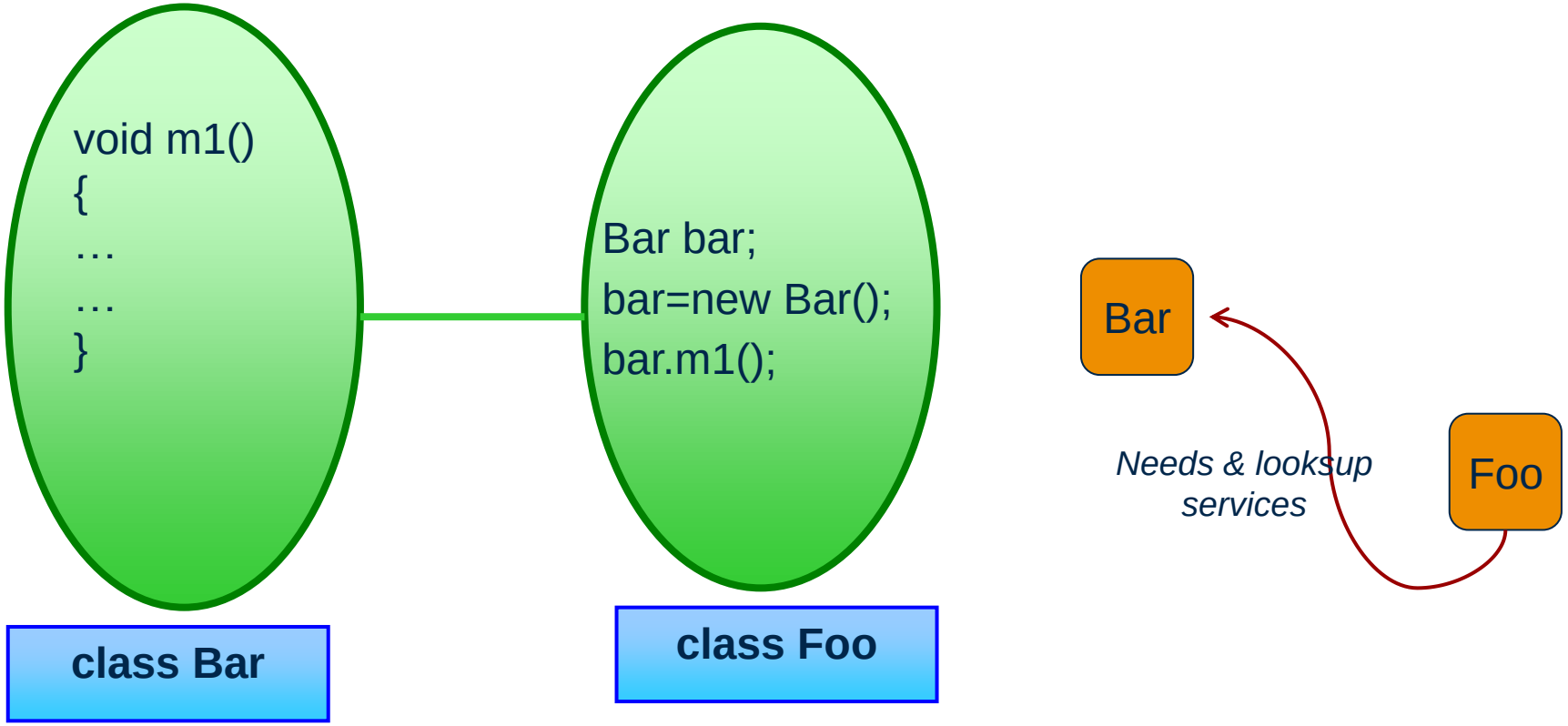
# DemoSpring_2

- This demo illustrates how the container will instantiate the CurrencyConverter service when using the <constructor-arg> element.

# Using collections for injection

- Often, beans need access to collections of objects, rather than just individual beans or values.
- Spring allows you to inject a collection of objects into your beans.
- You can choose either <list>, <map>, <set> or <props> to represent a List, Map, Set or Properties instance.
- You will pass in the individual items just as you would with any other injection.

# Wiring beans



# Wiring Beans - Inner Beans

- Another way of wiring bean references is to embed a <bean> element directly in the <property> element

```
<bean id="currencyConverter" class=“CurrencyConverterImpl">
    <property name="exchangeService">
                        <bean class= "ExchangeServiceImpl" />
        </property>
</bean>
```

  - The drawback here is that the instance of inner class cannot be used anywhere else; it is an instance created specifically for use by the outer bean.

# IoC in action: Wiring Beans

- The act of creating associations between application components is known as wiring.
- In Spring, there are many ways of wiring components together, but most commonly used is XML. An example:

```
<bean id="exchangeService"  class="ExchangeServiceImpl" />
<bean id="currencyConverter" class="CurrencyConverterImpl">
            <property name="exchangeService">
                 <ref bean="exchangeService" />
           </property>
            <!--<property name="exchangeService">
                    <ref local="exchangeService" /> </property> -->
            <!--<property name="exchangeService">
                    <idref local="exchangeService" /> </property> -->
</bean>
```

# DemoSpring_3

- This demo illustrates how the  BeanFactory loads the bean definition and wires the beans together

# Autowiring

- Autowiring allows Spring to wire all bean's properties automatically by setting the autowire property on each <bean> that you want autowired

```
<bean id="foo" class="com.igate.Foo" autowire="autowire type" />
```

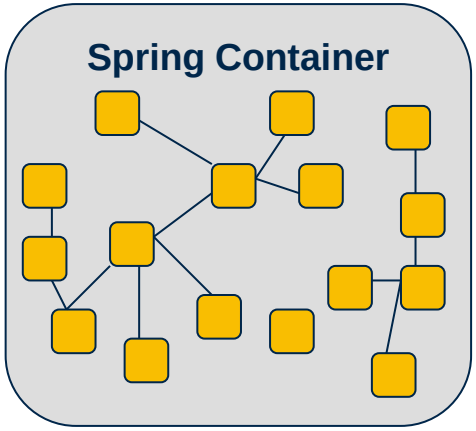
- Four types of autowiring:
  - byName
  - byType
  - constructor
  - Autodetect

# DemoSpring_4

- This demo illustrates automatically wiring your beans

# Bean containers: concept

- The container or bean factory is at the core of the Spring framework and uses IoC to manage components.
- Bean factory is responsible to create and dispense beans.
- It takes part in the life cycle of a bean, making calls to custom initialization and destruction methods, if those methods are defined.
- Spring has two types of containers:
  - Bean factories that are the simplest, providing basic support for dependency injection
  - Application contexts that build on bean factory by providing application framework services



# Bean containers: The BeanFactory

- BeanFactory interface is responsible for managing beans and their dependencies
- Its getBean() method allows you to get a bean from the container by name
- It has a number of implementing classes:
  - DefaultListableBeanFactory
  - SimpleJndiBeanFactory
  - StaticListableBeanFactory
  - XmlBeanFactory

# The XmlBeanFactory

- One of the most useful implementations of the bean factory is instantiated via explicit user code as:

```
Resource res = new FileSystemResource("beans.xml");
XmlBeanFactory factory = new XmlBeanFactory(res);
```

or

```
Resource res = new ClassPathResource("beans.xml");
XmlBeanFactory factory = new XmlBeanFactory(res);
```

# The Resource interface

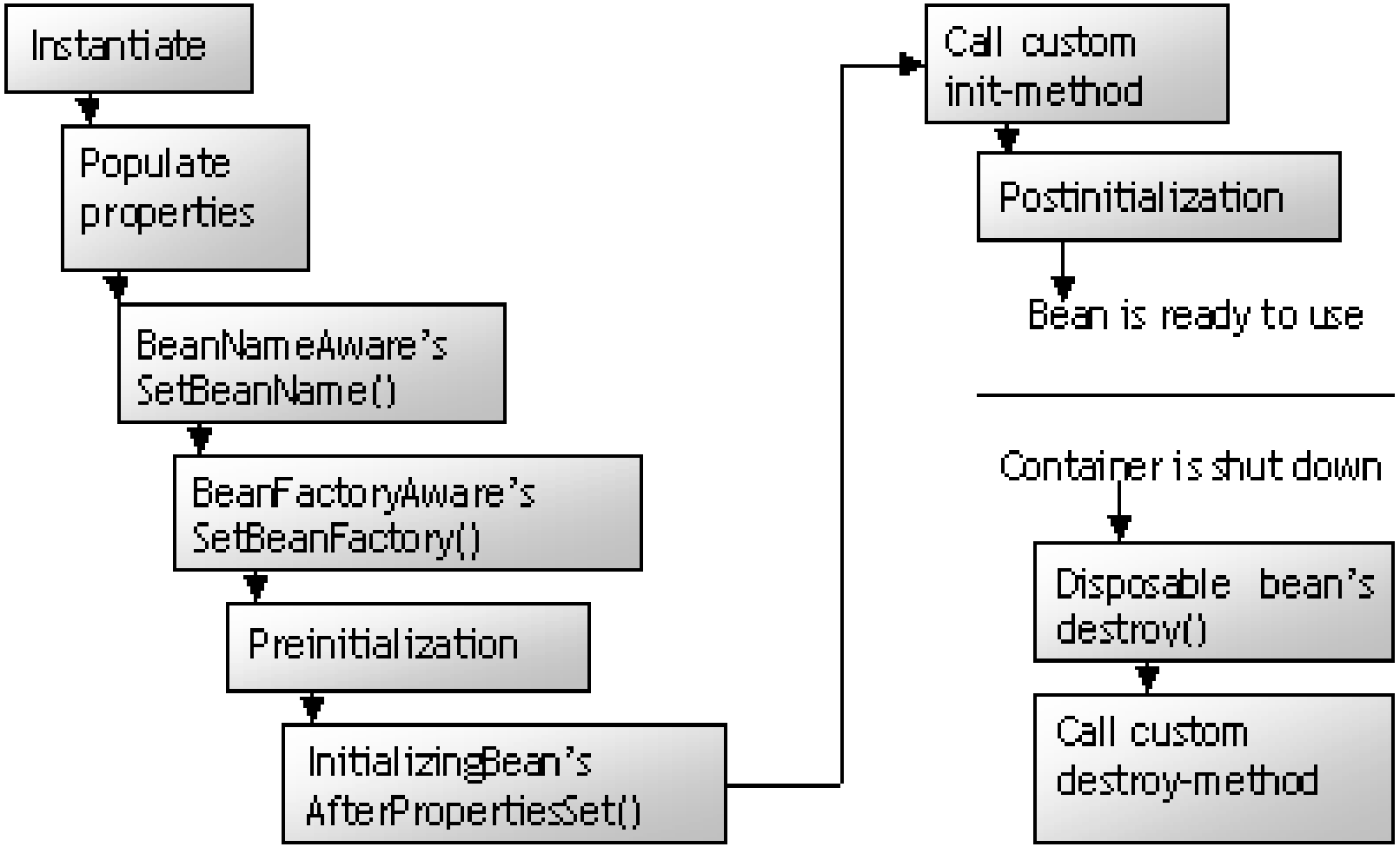
- The Resource interface is a unified mechanism for accessing resources in a protocol-independent manner.
- Some methods:
  - getInputStream(): locates and opens the resource, returning an InputStream for reading from the resource
  - exists(): indicates whether this resource actually exists
  - isOpen(): indicates whether this resource represents a handle with an open stream
  - getDescription(): returns a description for this resource, to be used for error output

# The XmlBeanFactory (Cont…)

- In an XmlBeanFactory, bean definitions are configured as one or more bean elements inside a top-level beans element

```
<?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/bean
s
http://www.springframework.org/schema/beans/spring-beans.xsd”>
      <bean id="..." class="...">
        ...
       </bean>
    ...
</beans>
```

# Life cycle of Beans in Spring factory container



# Initialization and Destruction

- When a bean is instantiated, some initialization can be performed to get it to a usable state
- When the bean is removed from the container, some cleanup may be required
- Spring can use two life-cycle methods of each bean to perform this setup and teardown.
- Example:

```
<bean id="foo"   class="com.spring.Foo"
                       init-method="setup"
                       destroy-method="teardown" />
```

# InitializingBean and DisposableBean

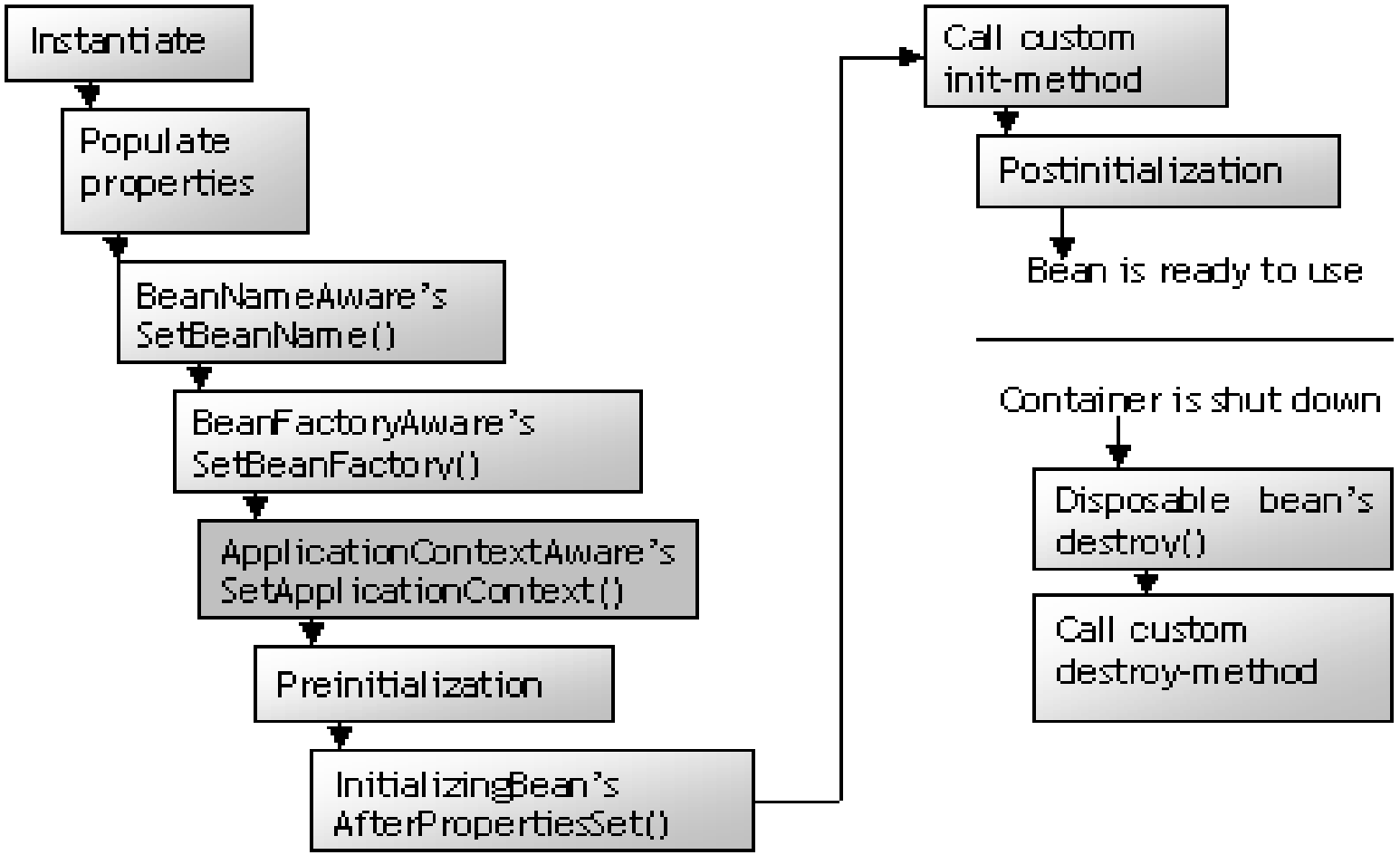
- InitializingBean interface
  - provides afterPropertiesSet() method which is called once all specified properties for the bean have been set.
- DisposableBean interface
  - provides destroy() method which is called when the bean is disposed by the container
- The advantage is that Spring container is able to automatically detect beans without any external configuration.
- The drawback is that the applications' beans are coupled to Spring API.

# Bean containers:Application context

- 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

# ApplicationContext life cycle



# Prototyping Vs Singleton

- By default, all Spring beans are singletons.
- But each time a bean is asked for, prototyping lets the container return a new instance.
- This is achieved through the scope attribute of <bean>
- Example:

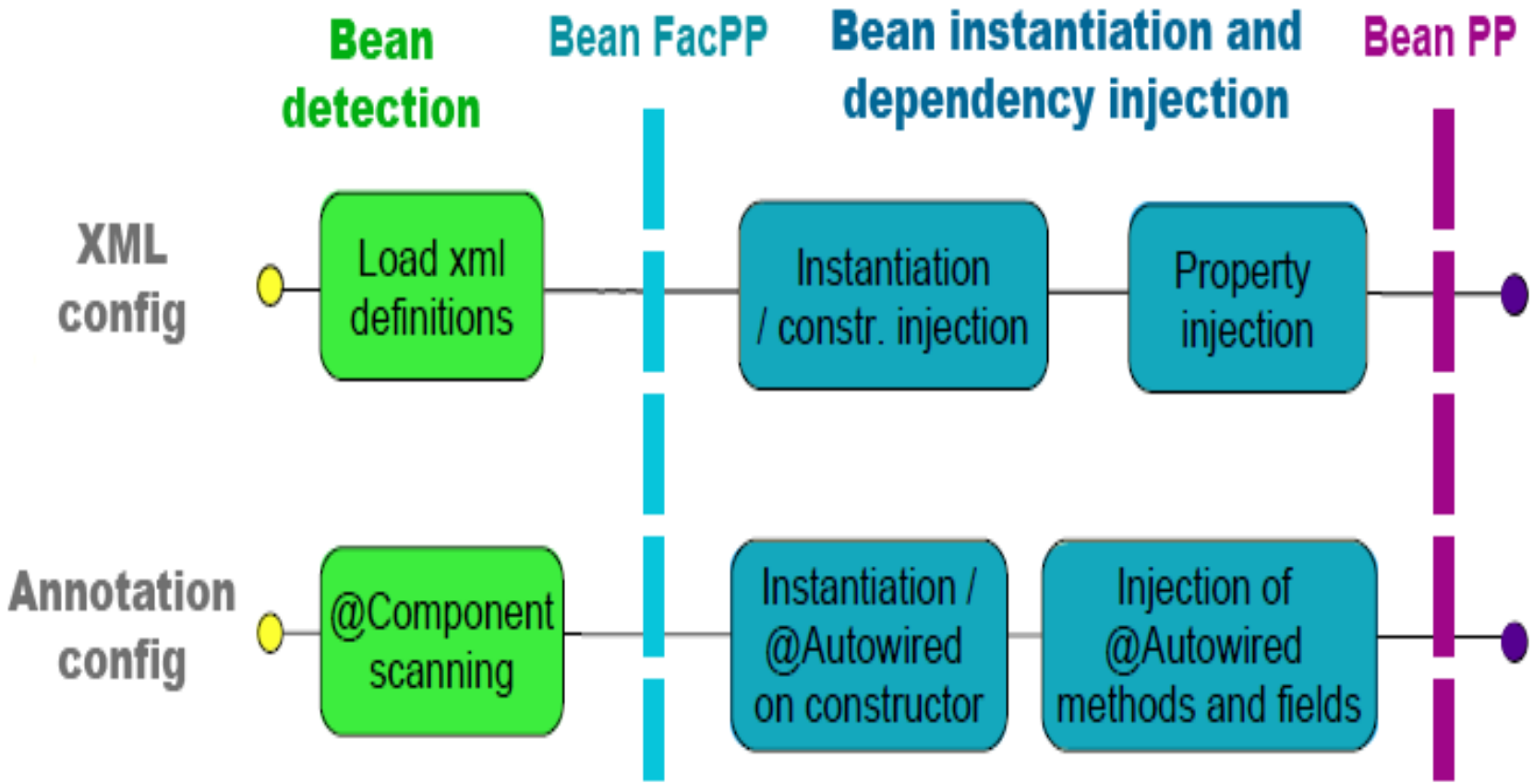
```
<bean id="foo" class="com.igate.Foo" scope="prototype" />
```

- Additional Bean scopes:
  - request
  - session
  - global-session

# 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.

# Lifecycle execution with PostProcessors



# Customizing beans with BeanFactoryPostProcessor

- BeanFactoryPostProcessor performs post processing on the entire Spring container.
- It has a single method, which is postProcessBeanFactory().
- Spring offers a number of pre-existing bean factory post-processors:
  - AspectJWeaving
  - CustomAutowireConfigurer
  - CustomEditorConfigurer
  - CustomScopeConfigurer
  - PropertyPlaceholderConfigurer
  - PreferencesPlaceholderConfigurer
  - PropertyOverrideConfigurer

# PropertyPlaceholderConfigurer

- It is possible to configure entire application in a single bean wiring file.

```
<bean id="datasource" class="com.spring.ConnectionDataSource" >
        <property name="url">
            <value> jdbc:hsqldb:training </value>
        </property>
        <property name="driverclassname">
            <value> org.hsqldb.jdbcDriver </value>
        </property>
    ....
    </bean>
```

- But, sometimes it is beneficial to extract certain pieces of that configuration into a separate property file.

# PropertyPlaceholderConfigurer

- Externalizing properties using PropertyPlaceholderConfigurer indicates Spring to load certain configuration from an external property file.

```
<bean id="placeHolderConfig" class="org.springframework.beans.
                        factory. config.PropertyPlaceholderConfigurer">
    <property name="location"  value="data.properties" />
</bean>
<bean id="dataSource"  class="org.apache.commons.dbcp.BasicDataSource">
      <property name="driverClassName" value="${jdbc.driverClassName}"/>
      <property name="url" value="${jdbc.url}"/>
      <property name="username" value="${jdbc.username}"/>
      <property name="password" value="${jdbc.password}"/>
</bean>
```

```
jdbc.driverClassName=oracle.jdbc.driver.OracleDriver
jdbc.url=jdbc:oracle:thin:@192.168.224.26:1521:trgdb
…..
```

# Demo: DemoSpring_6

- This demo shows how to use the PropertyPlaceholderConfigurer BeanFactoryPostProcessor

# CustomEditorConfigurer

- CustomEditorConfigurer is a bean factory post-processer which allows to convert values in String form to final property values.
- It allows you to register custom implementation of PropertyEditor to translate property wired values to other property types.
- Java.beans.PropertyEditorSupport is a convenience implementation java.beans.PropertyEditor interface that allows setting a non-string property to a string value.
- It has two methods: getAsText() and setAsText(String s)

# CustomEditorConfigurer

```
<bean id="customEditorConfigurer" class="org.springframework.
                        beans.factory.config.CustomEditorConfigurer">
  <property name="customEditors">
     <map>
        <entry key="java.util.Date“ value="MyCustomDateEditor"/>
     </map>
   </property>
</bean>
```

```
CustomEditorConfigurer configurer = (CustomEditorConfigurer)
                              factory.getBean(“customEditorConfigurer “);
Configurer.postProcessBeanFactory(factory);
BeanClass bean = (BeanClass) factory.getBean(“exampleBean”);
```

# Demo: DemoSpring_7

- This demo shows how to use the CustomEditorConfigurer BeanFactoryPostProcessor

# Internationalization: Resolving text messages

- ApplicationContext interface provides messaging functionality by extending MessageSource interface.
- getMessage() is a basic method used to retrieve a message from the MessageSource.
- On loading, ApplicationContext automatically searches for a MessageSource bean defined in the context.
- ResourceBundleMessageSource is a ready-to-use implementation of MessageSource.

# Internationalization: Resolving text messages

```
<bean id="messageSource“ class="org.springframework.context.
                                support.ResourceBundleMessageSource">
    <property name="basename">
        <value>applicationResources</value></property>
</bean>
```

```
MessageSource messageSource = (MessageSource) factory.getBean
                                                ("messageSource");
Locale locale = new Locale("en","US");
String msg = messageSource.getMessage("welcome.message", null, locale);
```

# DemoSpringI18N

- This demo shows how to provide messaging functionality in the application context.

# Annotation-based configuration

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

# Annotation-based configuration

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

# Annotation-based configuration

- Notes Here

# @Autowired annotation

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
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-4.0.xsd">
<context:annotation-config />

    <context:component-scan base-package=“training.spring“ />

     <!-- bean declarations go here -->
</beans>
```

# Execution with Spring Boot

```
@Component("hello")
public class HelloWorld {
        public String sayHello()        {
                return "Hello";
        }
}
```

```
@Configuration
@EnableAutoConfiguration
@ComponentScan("com.igate")
public class Client {
        public static void main(String[] args) {
          ApplicationContext context = SpringApplication.run(Client.class, args);
          HelloWorld bean = (HelloWorld) context.getBean(HelloWorld.class);
            String s=bean.sayHello();
            System.out.println(s);
        }
}
```

# DemoSpring_Anno

- This demo illustrates autowired annotation

# Annotating beans for autodiscovery

- Refer to demos, DemoSpring_Anno

# Lab

- From the lab guide
  - Lab-1 problem-statement-1 2 and 3

# Lesson Summary

- We have so far seen:
  - What is Spring and why spring?
  - The Spring architecture
  - Inversion of control
  - Bean containers
  - Lifecycle of beans in containers.
  - Some popular implementaions of BeanFactoryPostProcessors

# Review Questions

- Question 1: The <constructor-arg> element has an optional _____attribute that specifies the ordering of the constructor arguments.
  - Option 1: By index
  - Option 2: By type
  - Option 3: By order

- Question 2: A _____ bean lets the container return a new instance each time a bean is asked for in a non-web application
  - Option 1: Singleton
  - Option 2: Prototype
  - Option 3: Request
  - Option 4: session

# Review Questions

- Question 3: Specifying the _____ tag will allow Spring to validate at deployment time that the other bean actually exists.
  - Option 1: idref
  - Option 2: ref
  - Option 3: local

- Question 4: The BeanPostProcessor performs post processing on the entire Spring container.
  - Option 1: True
  - Option 2: false