Spring4_1
Spring Framework

Spring Projects

- Spring IO Platform
- Spring Boot
- Spring Framework
- *Spring XD*
- Spring Data
- 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
- *Spring Social*
- *REST Shell*
- Spring STATEMACHINE
- Spring SHELL
- Spring Mobile
- Spring Cloud Data Flow
- *Spring Loaded*

http://spring.io/projects

From configuration to security, web applications to big data – whatever the infrastructure needs of our applications may be, there is a Spring Project to help build it.
Start small and use just what is needed – **Spring is modular by design.**

Advanced Message Queuing Protocol (AMQP)

HATEOAS *(Hypertext As The Engine Of Application State).*

Introduction to the Spring Framework

Spring Framework is open source Java EE framework and inversion of control container.

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

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

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

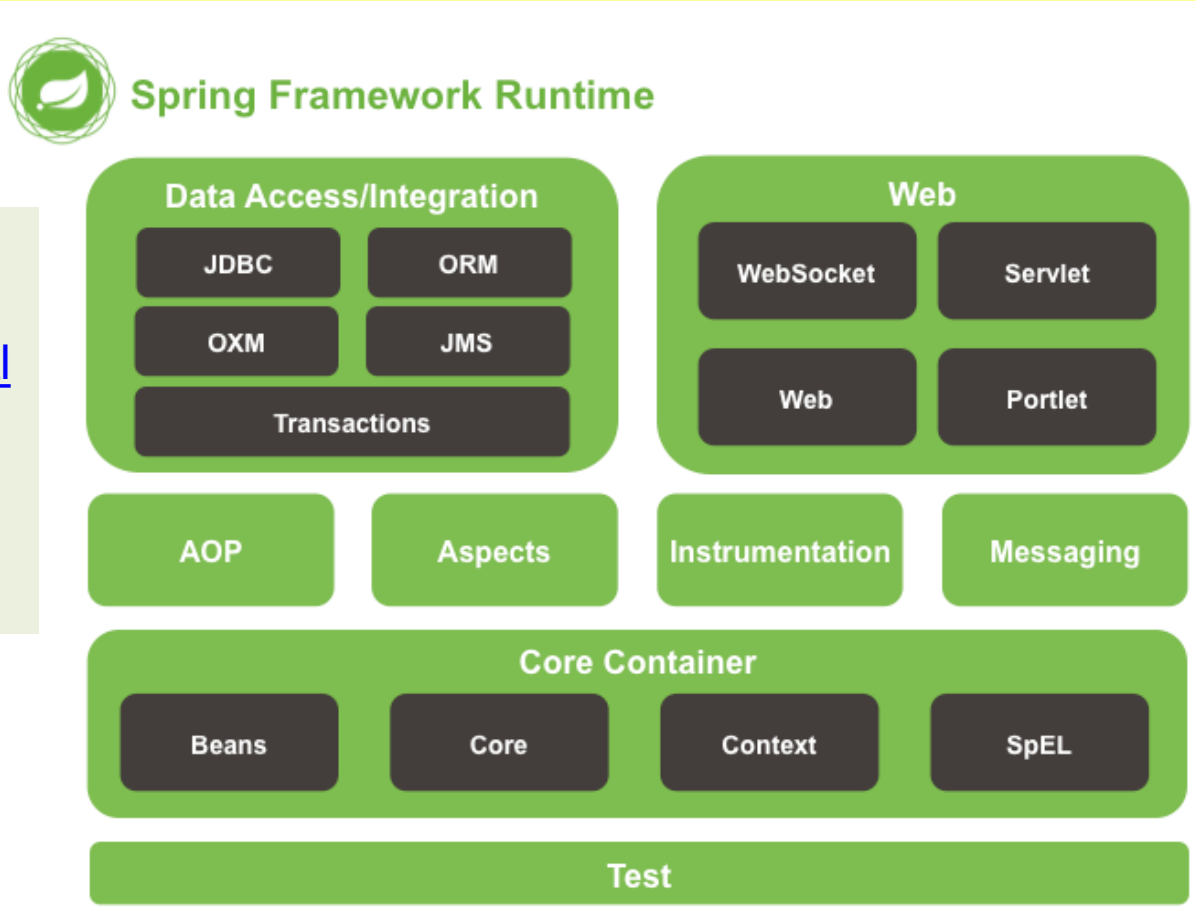
Spring enables you to build applications from "plain old Java objects" (POJOs) and to apply enterprise services non-invasively to POJOs.

Non-invasive: We are not forced to import or extend any Spring APIs, example Struts forces you to extend Action

Spring Framework

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

These modules are grouped into Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming), Instrumentation, and Test, as shown in the following diagram.



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

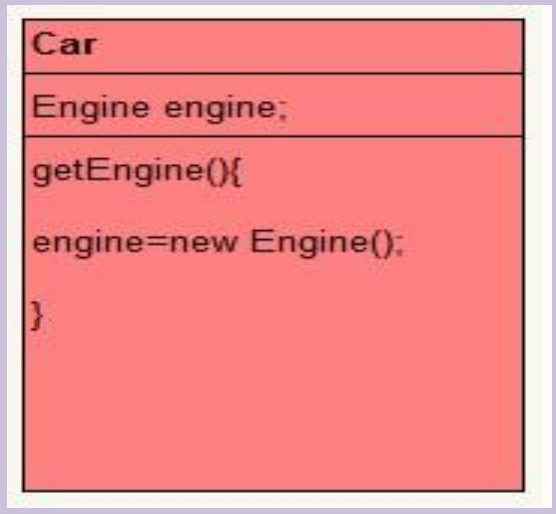
Spring Object XML Mappers (OXM)

Inversion of Control and Dependency Injection

Normal way:

There are many ways to instantiate an object. A simple and common way is with *new* operator.

So here Car class contains object of Engine and we have instantiated using new operator.

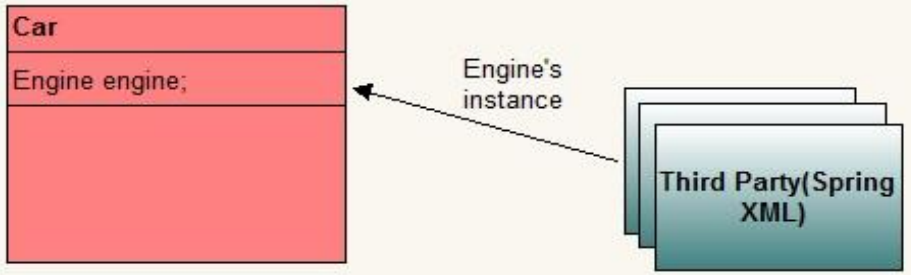


With help of Dependency Injection:

We outsource life-cycle management of Engine object to third party.

Car needs object of Engine to operate but it outsources that job to some third party. The designated third party, decides the moment of instantiation and the type to use to create the instance.

The dependency between class Car and class Engine is injected by a third party. This whole process is called *dependency injection.*



Types of Dependency Injection (DI)

Three types of dependency injection

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:

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.

The Spring IoC Container

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

•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.

•We don't directly connect our components and services together in code but describe which services are needed by which components in a configuration file.

•A container (in the case of the Spring framework, the IOC container) is then responsible for hooking it all up.

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

The Spring IoC Container

The Spring **IoC** container manages java objects(Java beans) – *from instantiation to destruction* – through its **BeanFactory** interface.

Java components that are instantiated by the **IoC** container are called beans, and the **IoC** container manages a bean's scope, lifecycle events, and any AOP features for which it has been configured and coded.

- A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.

- Otherwise, a bean is simply one of many objects in your application.

- Beans, and the *dependencies* among them, are reflected in the *configuration metadata* used by a container.

Spring Framework Core Container

The BeanFactory is a sophisticated implementation of the factory pattern. It removes the need for programmatic singletons and allows you to decouple the configuration and specification of dependencies from your actual program logic.

The **Context** module inherits its features from the Beans module and adds support for internationalization (using, for example, resource bundles), event propagation, resource loading, and the transparent creation of contexts by, for example, a Servlet container.

The Context module also supports Java EE features such as EJB, JMX, and basic remoting. The ApplicationContext interface is the focal point of the Context module.

The **BeanFactory** interface provides the configuration framework and basic functionality, and the **ApplicationContext is sub interface of BeanFactory** adds more enterprise-specific functionality.

The **ApplicationContext** is a complete superset of the **BeanFactory**

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 implementation of Spring Containers.

The two most common implementations are:

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

Bean Factory and Application Context

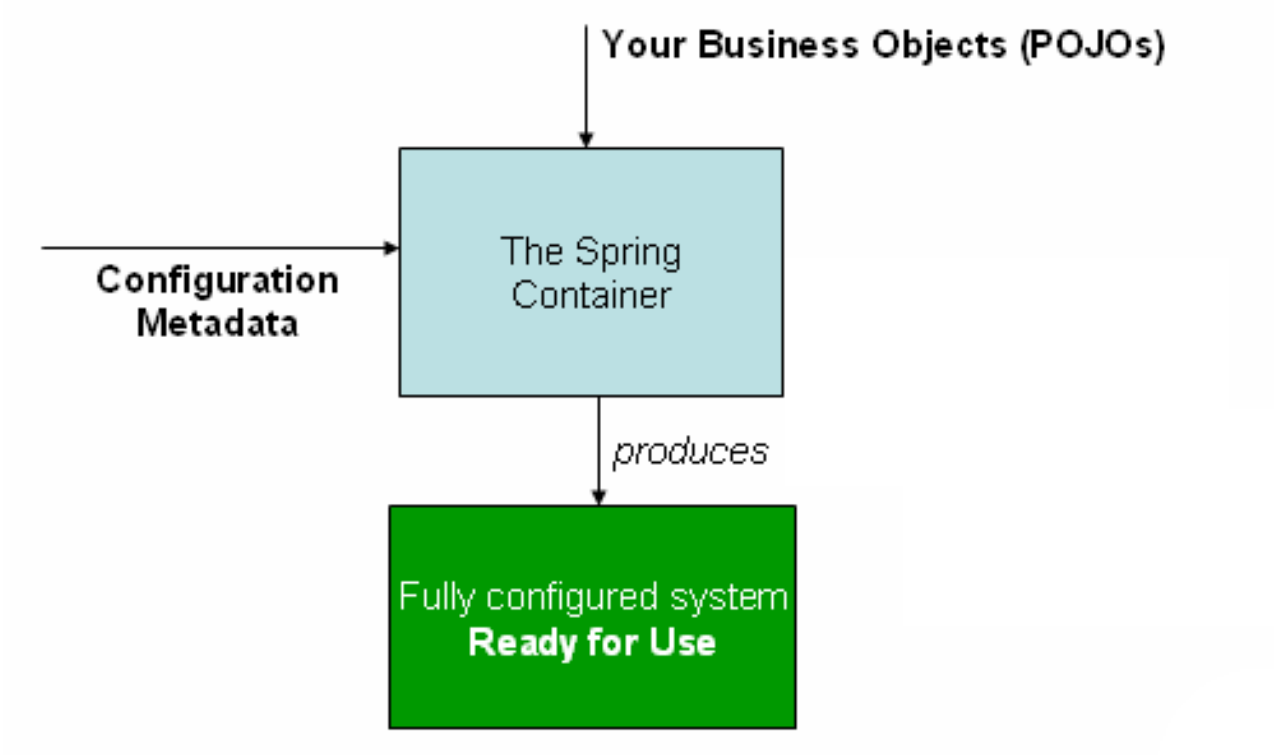
BeanFactory (org.springframework.beans.factory.BeanFactory interface) is the simplest of containers, providing the basic functionality of Dependency Injection.

ApplicationContext (org.springframework.context.ApplicationContext interface) builds on the top of **BeanFactory** and provides application framework services such as the ability to resolve textual messages from a properties file (for i18N), ability to publish application events to interested event listeners etc

The Spring IoC container

•The following diagram is a 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.



org.springframework.context.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");
```

While XML has been the traditional format for defining configuration metadata you can instruct the container to use Java annotations or Java Configuration class as the metadata format

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, many features provided by the *Spring JavaConfig* project became part of the core Spring Framework.

  Thus we can define beans external to our application classes by using Java rather than XML files.

  To use these new features, apply *@Configuration, @Bean, @Import and @DependsOn* annotations.

Spring configuration consists of at least one and typically more than one bean definition that the container must manage.

- XML-based configuration metadata shows these beans configured as **<bean/>** elements inside a top-level **<beans/>** element.
- Java-based configuration typically uses **@Bean** annotated methods within a **@Configuration** class.

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.

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.

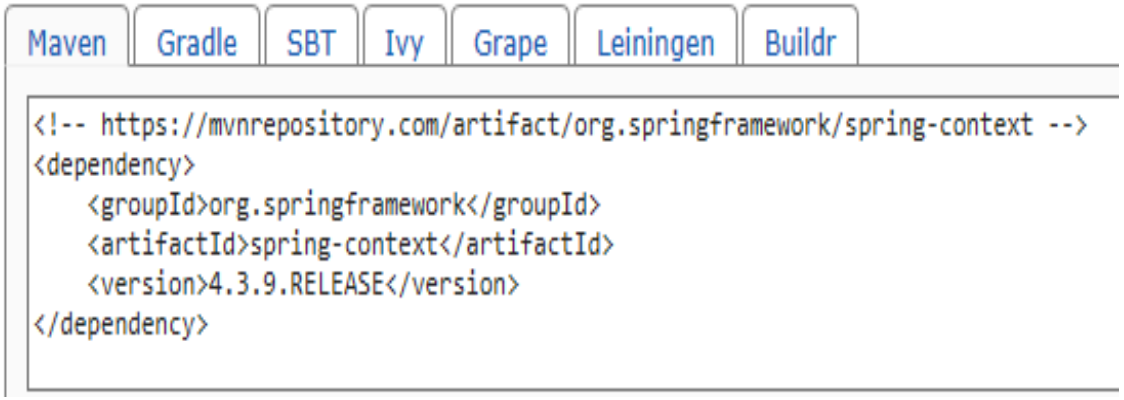
Sample Spring application using Maven build tool

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>
.....
```



Dependency Management

- The process of dependency management involves locating resources (jar files), storing them and adding them to class paths.

- To make this easier Spring is packaged as a set of modules that separate the dependencies as much as possible, so for example to build web application you require the following jar files

- e.g. *spring-core-4.0.6.RELEASE.jar, spring-context-4.0.6.RELEASE.jar, spring-web-4.0.6.RELEASE.jar,spring-webmvc-4.0.6.RELEASE.jar etc.*

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");
```

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;

…
}
```

spring.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

```
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();
    }
}
```

Bean Definition

The bean definition contains the information called **configuration metadata** which is needed for the container to know the followings:

- *How to create a bean*
- *Bean's lifecycle details*
- *Bean's dependencies*

All the above configuration metadata translates into a set of the following properties that make up each bean definition.

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.

Bean Scope

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

For example, 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**.

Similarly 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).*

Bean Definition

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.

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.

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).

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.

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.

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 } }
```

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
}
}
```

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

```
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.PropertyPlaceholderConfigurer">
<property name="locations" value="classpath:messages.properties"/>
</bean>
 </beans>
```

Note: Place static files in project's classpath

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>
```

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());
}

}
```

Injecting collections

We have seen how to configure primitive data type using **value** attribute and object references using **ref** attribute of the **<property>** tag in our Bean configuration file.

Both the cases deal with passing singular value to a bean.

Now what about if we want to pass plural values like Java Collection types ***List, Set and Map.***

To handle the situation, 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.

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

}
```

Injecting null and empty string values

If you need to pass an empty string as a value then you can pass it as follows:

```
<bean id="..." class="ExampleBean">
    <property name="email" value=""/>
</bean>
```

The preceding example is equivalent to the Java code: exampleBean.setEmail("")

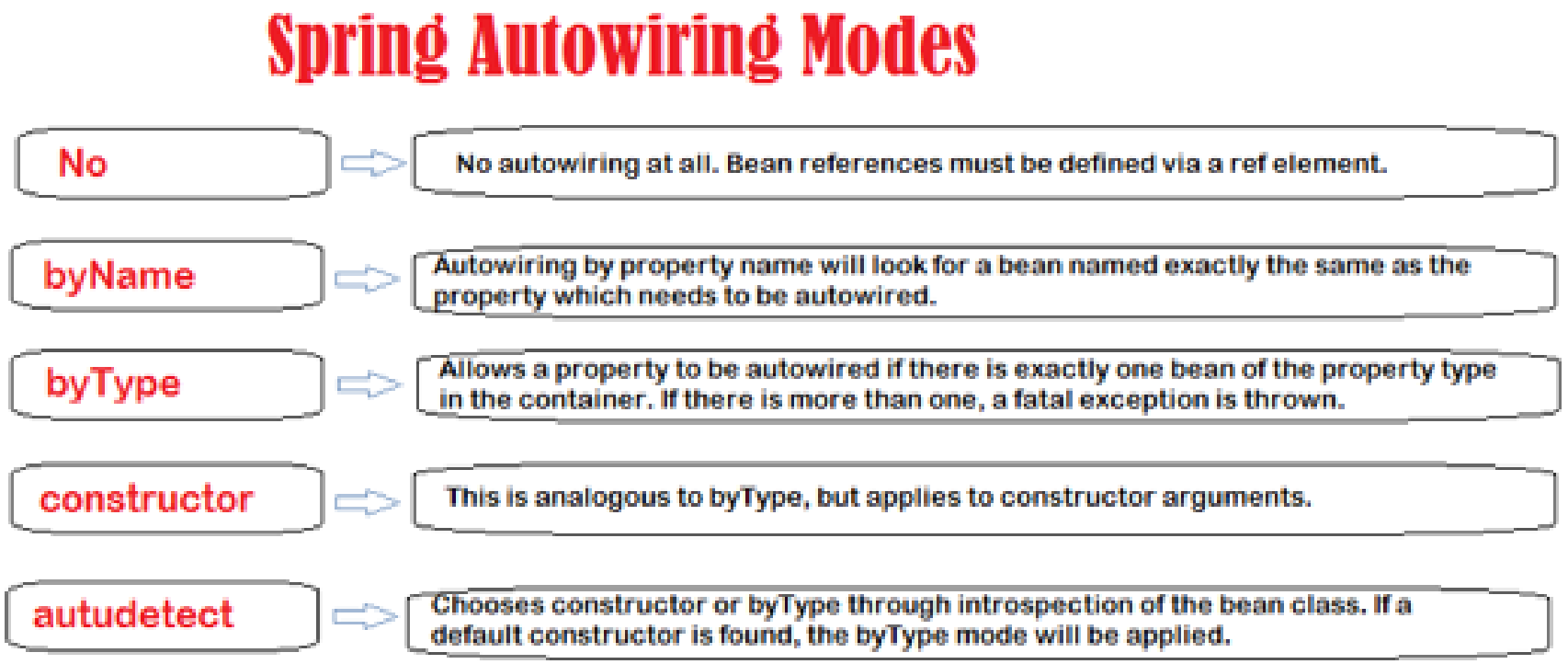
If you need to pass a NULL value then you can pass it as follows:

```
<bean id="..." class="exampleBean">
    <property name="email"><null/></property>
 </bean>
```

The preceding example is equivalent to the Java code: exampleBean.setEmail(null)

Auto-Wiring Modes

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



Spring bean autowire byType

In Spring, **Autowiring byType** means, if data type of a bean is compatible with the data type of other bean property, auto wire it.

For example, a “person” bean exposes a property with data type of “Ability” class, Spring will find the bean with same data type of class “Ability” and wire it automatically. And if no matching found, does nothing.

For example, if a “customer” bean exposes a property with data type Address, Spring will find the bean with same data type of class Address and wire it automatically. If no matching found, does nothing.

If the Spring container finds more than one bean with same data type, exception is thrown.

In these scenarios, we can set attribute of remaining beans whose data type is same to **autowire-candidate="false"**

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>

….
```

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>
….
```

Spring Annotations

Spring Annotations

Annotation	*Package Detail/Import statement*
@Service	import org.springframework.stereotype.Service;
@Repository	import org.springframework.stereotype.Repository;
@Component	import org.springframework.stereotype.Component;
@Autowired	import org.springframework.beans.factory.annotation.Autowired;
@Transactional	import org.springframework.transaction.annotation.Transactional;
@Scope	import org.springframework.context.annotation.Scope;
	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;

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 platfo).***

Spring Annotations

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

<context:annotation-config />

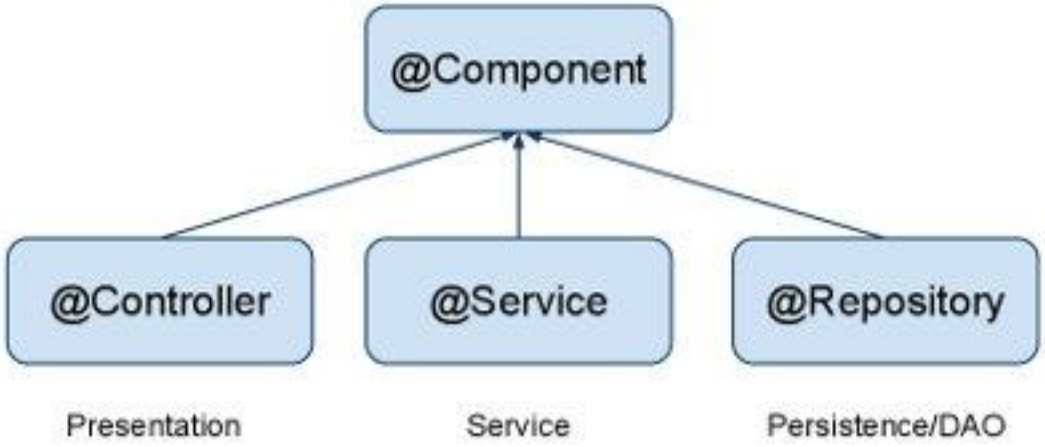
Note:
Spring supports both Annotation based and XML based configurations.

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.

@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 {
...
}
```

@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 {
...
}
```

@Autowired Annotation

Apart from the auto wiring modes provided in bean configuration file, auto wiring can be specified in bean classes also using **@Autowired** annotation.

To use **@Autowired** annotation in bean classes, you must first enable the annotation in spring application using below configuration.

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

Same can be achieved using 'AutowiredAnnotationBeanPostProcessor' bean definition in configuration file.

```
<bean class
="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostP
rocessor"/>
```

When annotation configuration has been enabled, we are free to autowire bean dependencies using **@Autowired,** the way you like.

This is done by three ways

- **@Autowired on properties**
- **@Autowired on property setters**
- **@Autowired on constructors**

@Autowired Annotation

```
@Service
public class CompanyService implements ICompanyService {

  @Autowired
  private ICompanyDAO companyDAO;
  ...
}
```

Note W can use **@Autowired** annotation on setter methods to get rid of the <property> element in XML configuration file.

When Spring finds an **@Autowired** annotation used with setter methods, it tries to perform **byType** autowiring on the method.

@Autowired Annotation

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

Autowired 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 of org.springframework.beans.factory.annotation.Qualifier;

@Resource is of javax.annotation.Resource

@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

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.

Spring Java Based Configuration

So far we have seen how to configure Spring beans using XML configuration file. Alternatively, we can go for **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.

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

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");
}
}
```

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);
```

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

Thank You!