Spring – Introduction

Spring is the most popular application development framework for enterprise Java. Millions of developers around the world use Spring Framework to create high performing, easily testable, reusable code.
The core features of the Spring Framework can be used in developing any Java application, but there are extensions for building web applications on top of the Java EE platform. Spring framework targets to make J2EE development easier to use and promote good programming practice by enabling a POJO-based programming model.

Following is the list of few of the great benefits of using Spring Framework:

- Spring enables developers to develop enterprise-class applications using POJOs
- Spring is organized in a modular fashion. Even though the number of packages and classes are substantial, you have to worry only about ones you need and ignore the rest.
- Spring does not reinvent the wheel instead, it truly makes use of some of the existing technologies like several ORM frameworks, logging frameworks, JEE, Quartz and JDK timers, other view technologies.
- Spring's web framework is a well-designed web MVC framework
- Spring provides a convenient API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO, for example) into consistent, unchecked exceptions.
- Lightweight IoC containers tend to be lightweight, especially when compared to EJB containers.

Dependency Injection (DI):

The technology that Spring is most identified with is the **Dependency Injection (DI)** flavor of Inversion of Control. The Inversion of Control (IoC) is a general concept, and it can be expressed in many different ways and Dependency Injection is merely one concrete example of Inversion of Control.

The dependency part translates into an association between two classes. For example, class A is dependent on class B. Now, let's look at the second part, injection. All this means is that class B will get injected into class A by the IoC.

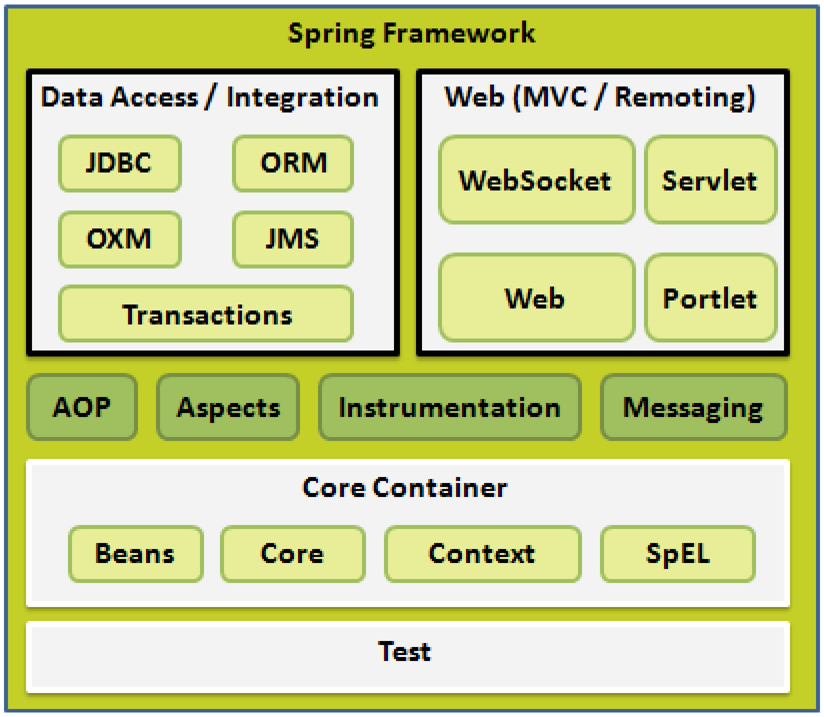
Dependency injection can happen in the way of passing parameters to the constructor or by post-construction using setter methods

Aspect Oriented Programming (AOP):

One of the key components of Spring is the Aspect oriented programming (AOP)framework. The functions that span multiple points of an application are called cross-cutting concerns and these cross-cutting concerns are conceptually separate from the application's business logic. There are various common good examples of aspects including logging, declarative transactions, security, and caching etc.

Spring Framework

The Spring Framework provides about 20 modules which can be used based on an application requirement.



Spring - Hello World Example

Write a Bean Class:

```
package com.h2k.spring.test;

public class HelloWorld {
      private String message;
```

```java
         public void setMessage(String message){
            this.message  = message;
         }

         public void getMessage(){
            System.out.println("Your Message : " + message);
         }
}
```

Main Class Which calls bean method:

```java
package com.h2k.spring.test;

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

public class MainApplication {

      public static void main(String[] args) {
            ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");
            HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
            obj.getMessage();
         }
}
```

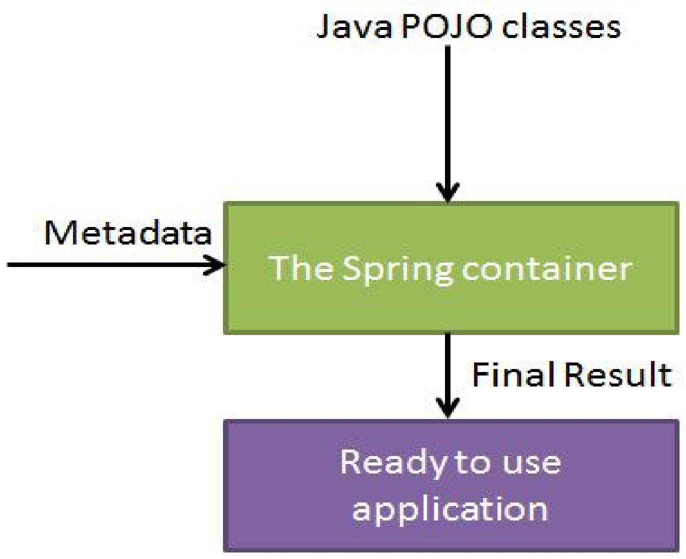
Beans.xml which configures this Bean.

```xml
<?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-3.0.xsd">

   <bean id="helloWorld" class="com.h2k.spring.test.HelloWorld">
       <property name="message" value="Hello H2K Students!!"/>
   </bean>

</beans>
```

Spring - IoC Containers

The Spring container is at the core of the Spring Framework. The container will create the objects, wire them together, configure them, and manage their complete lifecycle from creation till destruction. The Spring container uses dependency injection (DI) to manage the components that make up an application

The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code. The following diagram is a high-level view of how Spring works.



Spring provides following two distinct types of containers.

1. **Spring Bean Factory Container**: This is the simplest container providing basic support for DI and defined by the*org.springframework.beans.factory.BeanFactory* interface.
2. S**pring ApplicationContext Container**: This container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners. This container is defined by the*org.springframework.context.ApplicationContext* interface.

The *ApplicationContext* container includes all functionality of the *BeanFactory* container, so it is generally recommended over the *BeanFactory*.

```
package com.h2k.spring.test;

import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

public class BeanFactoryExample {

	public static void main(String[] args) {
```

```
                    XmlBeanFactory factory = new XmlBeanFactory(new
ClassPathResource("Beans.xml"));

                   HelloWorld obj = (HelloWorld) factory.getBean("helloWorld");
                   obj.getMessage();
                }

}
```

Spring ApplicationContext Container

The most commonly used ApplicationContext implementations are:

- **FileSystemXmlApplicationContext**: This container loads the definitions of the beans from an XML file. Here you need to provide the full path of the XML bean configuration file to the constructor.
- **ClassPathXmlApplicationContext** This container loads the definitions of the beans from an XML file. Here you do not need to provide the full path of the XML file but you need to set CLASSPATH properly because this container will look bean configuration XML file in CLASSPATH.
- **WebXmlApplicationContext:** This container loads the XML file with definitions of all beans from within a web application.

Spring - 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

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 and it will be discussed in bean scopes chapter.
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.

```
<?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-3.0.xsd">
   <!-- A simple bean definition -->
   <bean id="..." class="...">
       <!-- collaborators and configuration for this bean go here -->
   </bean>
   <!-- A bean definition with lazy init set on -->
   <bean id="..." class="..." lazy-init="true">
       <!-- collaborators and configuration for this bean go here -->
   </bean>
   <!-- A bean definition with initialization method -->
   <bean id="..." class="..." init-method="...">
       <!-- collaborators and configuration for this bean go here -->
   </bean>
   <!-- A bean definition with destruction method -->
   <bean id="..." class="..." destroy-method="...">
       <!-- collaborators and configuration for this bean go here -->
   </bean>
   <!-- more bean definitions go here -->
</beans>
```

Spring - Bean Scopes

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**

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.

```
<!-- A bean definition with singleton scope -->
```

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

Example:

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

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");
      HelloWorld objA = (HelloWorld) context.getBean("helloWorld");
      objA.setMessage("I'm object A");
      objA.getMessage();
      HelloWorld objB = (HelloWorld) context.getBean("helloWorld");
      objB.getMessage();
   }
}
```

Try same example with prototype scope. You will receive null output for objB.

Spring - Bean Life Cycle

The life cycle of a Spring bean is easy to understand. 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.

Initialization callbacks:

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

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

Destruction callbacks

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

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

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

Example:

```
public class HelloWorld {
   private String message;
   public void setMessage(String message){
      this.message  = message;
   }
   public void getMessage(){
      System.out.println("Your Message : " + message);
   }
   public void init(){
      System.out.println("Bean is going through init.");
   }
   public void destroy(){
      System.out.println("Bean will destroy now.");
   }
}
```

Default initialization and destroy methods:

If you have too many beans having initialization and or destroy methods with the same name, you don't need to declare init-method and destroy-method on each individual bean. Instead framework provides the flexibility to configure such situation using default-init-method anddefault-destroy-method attributes on the <beans> element as follows:

```
<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-3.0.xsd"
    default-init-method="init"
    default-destroy-method="destroy">

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

Spring - Bean Definition Inheritance

A bean definition can contain a lot of configuration information, including constructor arguments, property values, and container-specific information such as initialization method, static factory method name, and so on.

A child bean definition inherits configuration data from a parent definition. The child definition can override some values, or add others, as needed.

Spring Bean definition inheritance has nothing to do with Java class inheritance but inheritance concept is same

```
<?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-3.0.xsd">

   <bean id="helloWorld" class="com.h2kinfosys.HelloWorld">
      <property name="message1" value="Hello World!"/>
      <property name="message2" value="Hello Second World!"/>
   </bean>

   <bean id="helloIndia" class="com.h2kinfosys.HelloIndia" parent="helloWorld">
      <property name="message1" value="Hello India!"/>
      <property name="message3" value="Namaste India!"/>
   </bean>
</beans>
```

Class Should look like:

```
package com.h2kinfosys;

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

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

      HelloWorld objA = (HelloWorld) context.getBean("helloWorld");

      objA.getMessage1();
      objA.getMessage2();


      HelloIndia objB = (HelloIndia) context.getBean("helloIndia");
      objB.getMessage1();
      objB.getMessage2();
      objB.getMessage3();
   }
}
```

Bean Definition Template:

You can create a Bean definition template which can be used by other child bean definitions without putting much effort. While defining a Bean Definition Template, you should not specifyclass attribute and should specify abstract attribute with a value of true as shown below:

```
<?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-3.0.xsd">

   <bean id="beanTeamplate" abstract="true">
      <property name="message1" value="Hello World!"/>
      <property name="message2" value="Hello Second World!"/>
      <property name="message3" value="Namaste India!"/>
   </bean>

   <bean id="helloIndia" class="com.h2kinfosys.HelloIndia" parent="beanTeamplate">
      <property name="message1" value="Hello India!"/>
      <property name="message3" value="Namaste India!"/>
   </bean>

</beans>
```