Spring basics

Agenda

Objectives

At the end of this session, you will be able to:

- Understand Core Spring framework
- Understand the steps involved in creating a simple Spring Application

Spring Core

Core Spring

- The Core Spring can be thought of a **Framework** and a **Container** for managing Business Objects and their relationship
- With Spring Framework, most of the times we don't need to depend on Spring specific Classes and Interfaces
- This is unlike other Frameworks, where the framework will force the Client Applications to depend on their propriety Implementations
- **Business Components** in Spring are **POJO** (Plain Old Java Object) or **POJI** (Plain Old Java Interface)
- These Business components are **configured** to the **Spring Container** for rendering

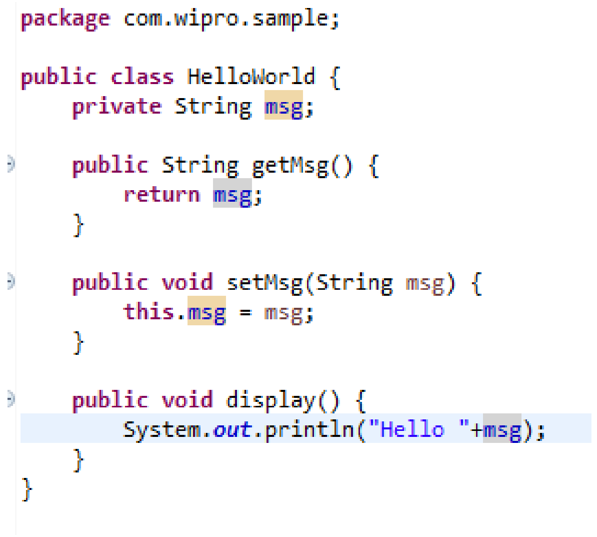
First Spring Application

First Spring Application

- Let us understand the various components
- What is needed?
  - Simple Java classes to represent our business needs (POJO)
  - Configuration details to instruct how to manage the business objects
  - And Spring jars for bringing both together
- We would create Maven Projects to take care of the Spring dependencies
  - i.e. Required Spring jars and its compatibilities between them
- Let us take a Simple HelloWorld Application; here our base requirement is to display the content contained by the data field *msg*

First Spring Application

- HelloWorld Class

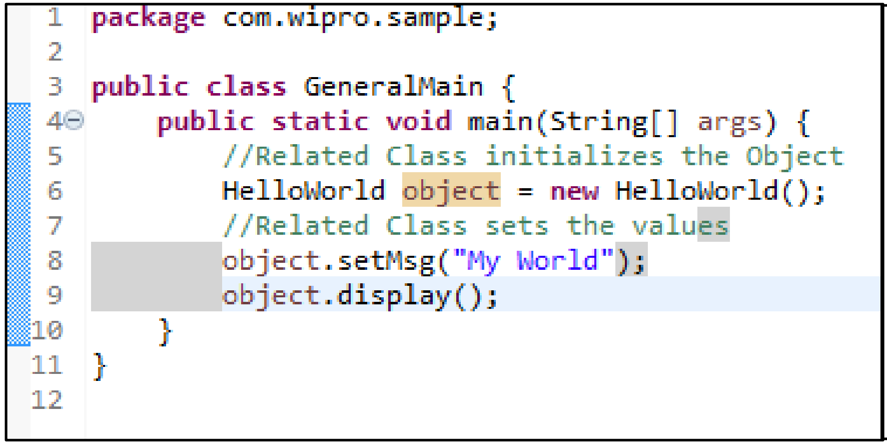


```java
package com.wipro.sample;

public class HelloWorld {
    private String msg;

    public String getMsg() {
        return msg;
    }

    public void setMsg(String msg) {
        this.msg = msg;
    }

    public void display() {
        System.out.println("Hello "+msg);
    }
}
```

NOTE: We are just understanding the components :
Step by step guide to create the application is given at the end

First Spring Application

- A Normal Application for wanting to use this HelloWorld class would look like this



```java
package com.wipro.sample;

public class GeneralMain {
    public static void main(String[] args) {
        //Related Class initializes the Object
        HelloWorld object = new HelloWorld();
        //Related Class sets the values
        object.setMsg("My World");
        object.display();
    }
}
```

- Here the Related class(*GeneralMain*) creates and maintains the required objects of *HelloWorld* class
- Object Graph created for this scenario would also look simple with 2 class
- When the application grows; the no of object references increases with Object Graph complexity

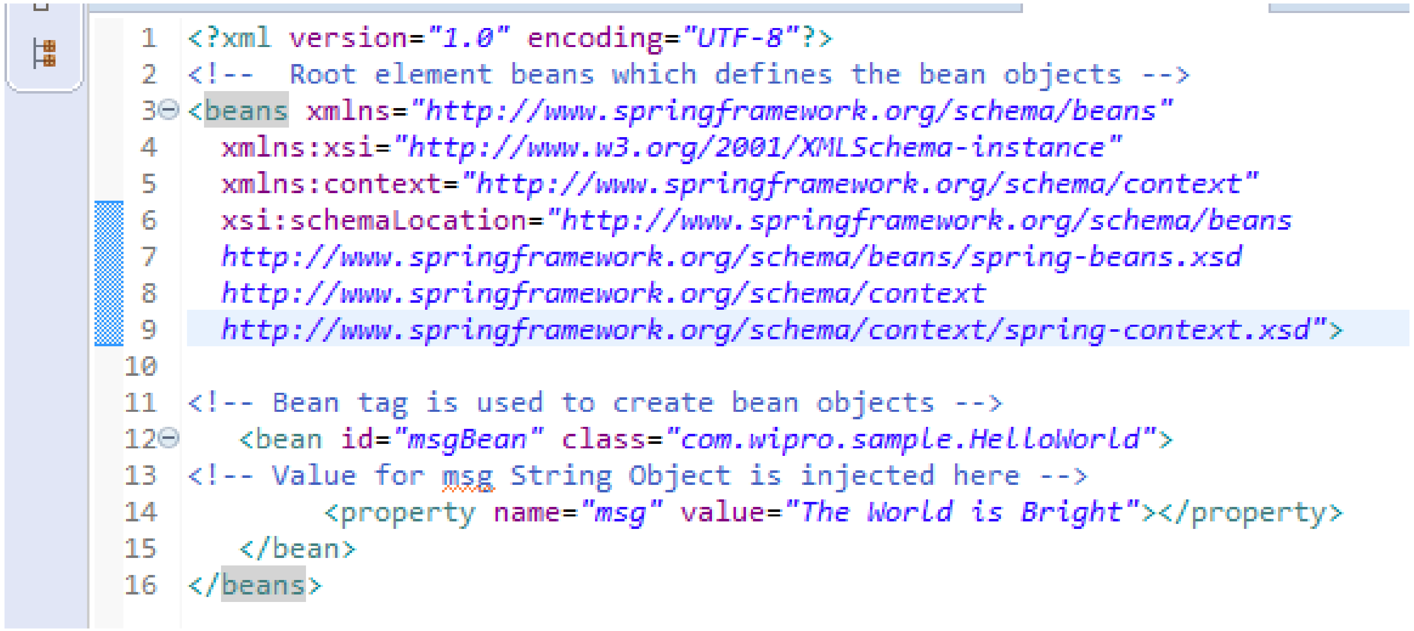
- Spring removes this overhead; It creates and delivers the Objects to the related class
- Delivering of Objects to the related class is called Dependency Injection(DI) or
- IOC Inversion Of Control as Spring takes the control in Object life cycle

First Spring Application

- We need to configure the Spring Container to express
  - The spring beans
  - Spring bean’s dependencies
  - Services needed by these beans
- Ways to implement the configuration in Spring are
  - XML-Based Configuration
    - Widely used approach. Where <bean> tag is used to define Spring beans
  - Java-Based Configuration
    - Java class is used to define the configuration using Annotations like
      - @Configuration – to annotate the class as Configuration class
      - @Bean – to annotate the method defining the bean class
  - Annotation-Based Configuration
    - It’s a combination of
    - XML Configuration to express auto scanning feature
    - Annotations to express the components like
      - @Component, @Service
      - @Autowired

First Spring Application

- For Initial Demos we use XML configuration file
- Let us look at the configuration File for our HelloWorld class
- Named as beans.xml



```xml
<?xml version="1.0" encoding="UTF-8"?>
<!--  Root element beans which defines the bean objects -->
<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.xsd">

<!-- Bean tag is used to create bean objects -->
   <bean id="msgBean" class="com.wipro.sample.HelloWorld">
<!-- Value for msg String Object is injected here -->
        <property name="msg" value="The World is Bright"></property>
   </bean>
</beans>
```

First Spring Application

- Last step is to create the client class to use the Spring framework functionality
- There are 2 ways
  - BeanFactory
    - Provides Advanced Configuration for managing any type of Bean, with any type of storage facility

```
Resource resource = new FileSystemResource("src/main/resources/beans.xml");
BeanFactory beanFactory = new XmlBeanFactory(resource);
HelloWorld helloWorld = beanFactory.getBean(HelloWorld.class);
helloWorld.display();
```

  - ApplicationContext
    - Is built over BeanFactory with added functionalities like
      - Easy integration with Spring AOP
      - Event Propogation
      - Enterprise centric functionalities and more

```
ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
HelloWorld helloWorld = context.getBean(HelloWorld.class);
helloWorld.display();
```

First Spring Application

For the Step by Step Guide of creating First Spring Application
Refer: **First Spring Projects.pdf**

BeanFactory – A better understanding

- BeanFactory is a container that manages all the beans
  - Configuration
  - And life cycle [Initialization , rendering , destruction]
- BeanFactory Interface has multiple implementation classes
  - Commonly used class for instantiation of BeanFactory interface is XMLBeanFactory
  - XMLBeanFactory is deprecated with Spring 3.1
    - Alternatively we can use

```
BeanDefinitionRegistry beanDefinitionRegistry = new DefaultListableBeanFactory();
XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(beanDefinitionRegistry);
reader.loadBeanDefinitions(new ClassPathResource("SPRING_CONFIGURATION_FILE"));
```

Configuration File – A better understanding

- Bean Life Cycle
- The Bean objects defined in the Xml Configuration File undergoes a Standard Lifecycle Mechanism
- We can enhance or modify the lifecycle of bean objects by using interfaces like *InitializingBean* and *DisposableBean*
- The InitializingBean interface has a single method called *afterPropertiesSet*() which will be called immediately after all the property values that have been defined in the Xml Configuration file is set
- The DisposableBean has a single method called *destroy*() which will be called during the shut down of the Bean Container

Configuration File – A better understanding

- Example code illustrating the usage of ‘Life Cycle Interfaces’

```
import org.springframework.beans.factory.*;
public class Employee implements InitializingBean, DisposableBean {
private String name;
private String id;
public void afterPropertiesSet() throws Exception {
        System.out.println("Employee->afterPropertiesSet() method called");
        }
public void destroy() throws Exception {
        System.out.println("Employee->destroy() method called");
        }
}
```

Configuration File – A better understanding

- Order of Creation of Beans
- We can control the bean creation order by using the *depends-on* attribute of the bean *tag*
- *depends-on* attribute take the bean identifier names which needs to be defined prior to the current bean
- Example code – controlling the order of creation of Beans

```
<bean id = "joseph" class = "spring.complex.Employee" depends-on = "admin">
</bean>
<bean id = "admin" class = "spring.complex.Department" depends-on = "oracle">
</bean>
<bean id = "oracle" class = "spring.complex.Organisation">
</bean>
```

Thank you