Enterprise Application Development

An Introduction to Spring Framework

Sadegh Aliakbary

Outline

- Dependency Injection and IoC
- Aspect Oriented Programming
- Spring Framework

Introduction to Spring Framework

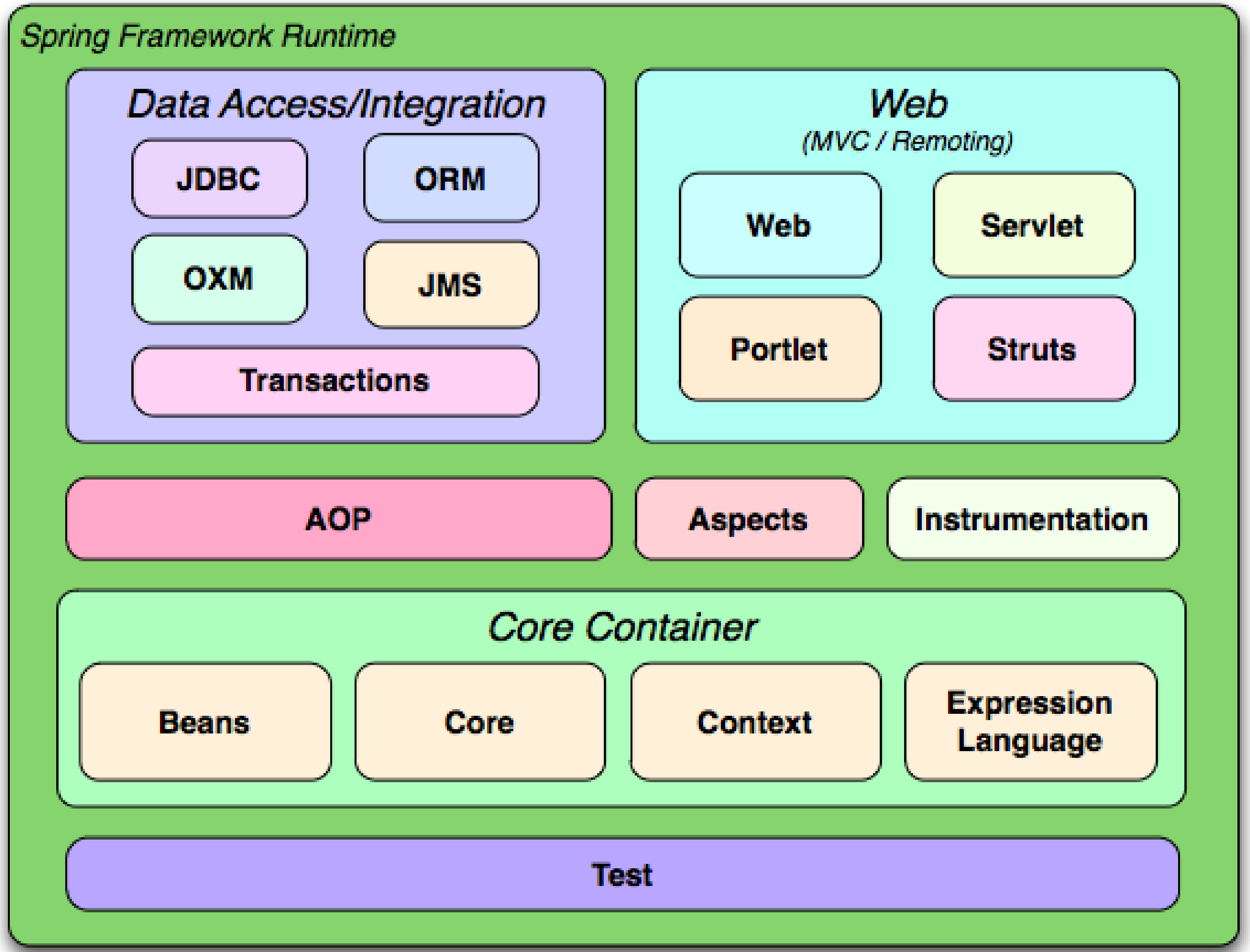
- An **open source** Java platform
- Initially released under the Apache 2.0 license in **2003**
- Spring is **lightweight**: the basic version = 2MB
- The core features can be used in any Java application
- But there are extensions for web applications on top of Java EE platform
- Spring targets to make J2EE development easier to use
- Promote good programming practice
- By enabling a POJO-based programming model

About Spring

- Provides to create high performing, easily testable and
- reusable code.
- is organized in a modular fashion
- simplifies java development

Spring Modules

- Spring is modular
- Allowing you to choose which modules are applicable to you
- Provides about 20 modules


Spring Framework Runtime
Data Access/Integration
JDBC
ORM
OXM
JMS
Transactions
Web
(MVC / Remoting)
Web
Servlet
Portlet
Struts
AOP
Aspects
Instrumentation
Core Container
Beans
Core
Context
Expression Language
Test

Two Key Components of Spring

- Dependency Injection (DI)
- Aspect Oriented Programming (AOP)

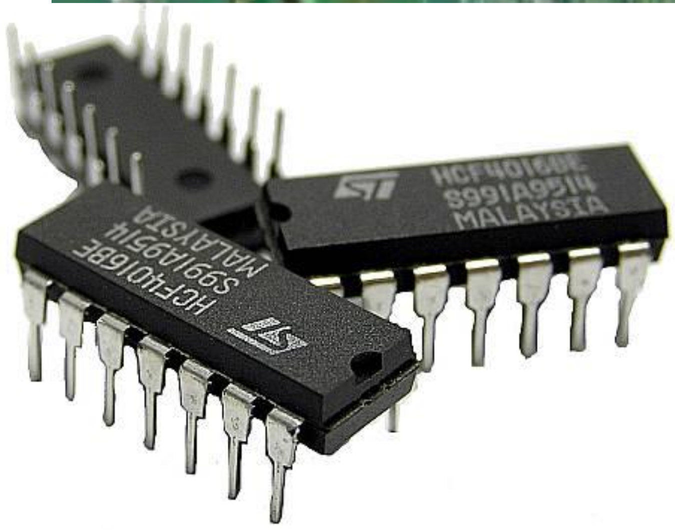
Dependency Injection (cont'd)

- application classes should be as independent as possible
  - To increase the possibility to reuse these classes
  - and to test them independently
- **Dependency**: an association between two classes
  - E.g., class A is dependent on class B
- **Injection**: class B will get injected into class A by the IoC
- Dependency injection
  - in the way of passing parameters to the constructor
  - or by post-construction using setter methods

Library vs Framework

- Framework:

- Library:



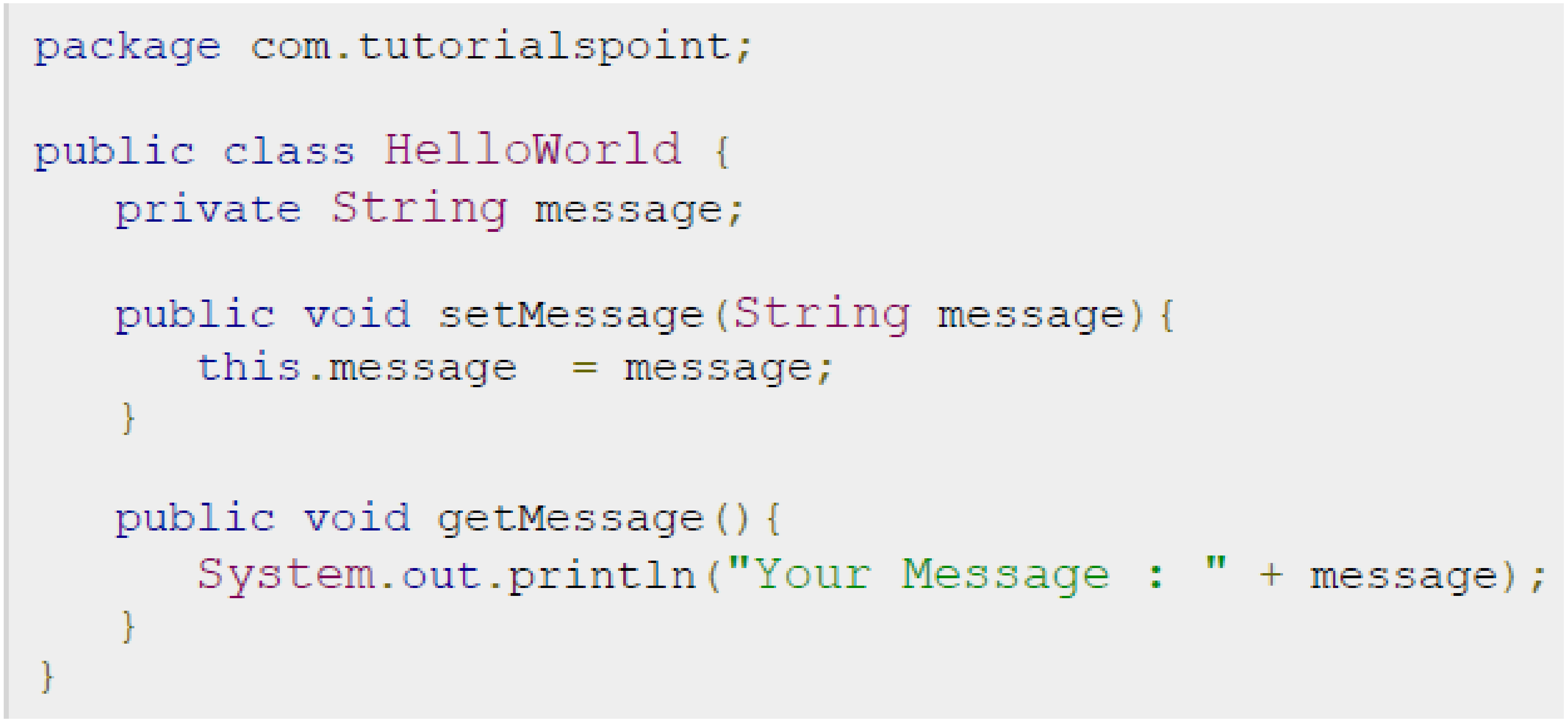
Aspect Oriented Programming (AOP)

- **cross-cutting concerns**
  - The functions that span multiple points of an application
- cross-cutting concerns are conceptually separate from the application's business logic
  - E.g., logging, declarative transactions, security, and caching
- The key unit of modularity
  - in OOP: the class
  - in AOP: the aspect.
- DI helps you decouple application objects from each other
- AOP helps you decouple cross-cutting concerns from the objects that they affect

Spring – Hello World

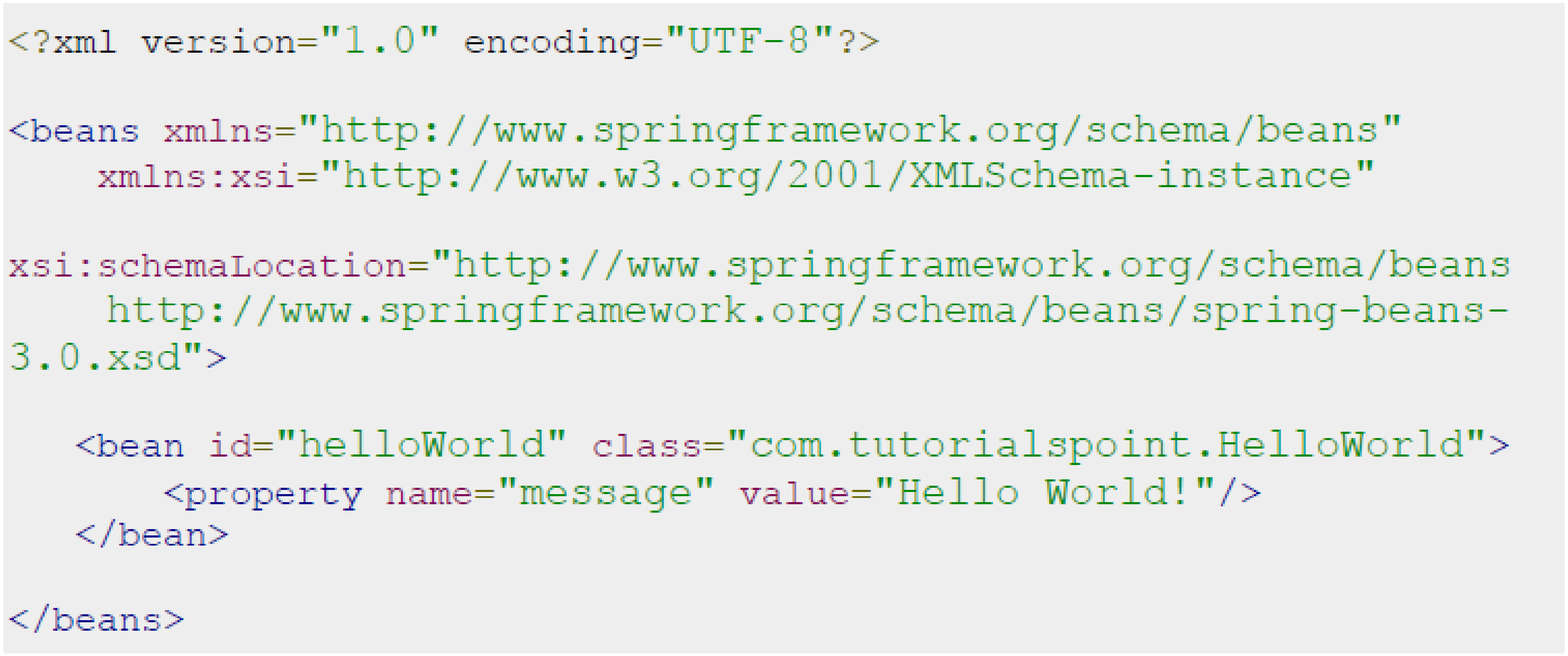
- Create your java project
  - Simple application
  - Web application
- Create source files
  - Class of beans
- Create bean configuration file (XML)
- Retrieve beans

Source: Bean Classes



```
package com.tutorialspoint;

public class HelloWorld {
   private String message;

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

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

Bean Configuration File

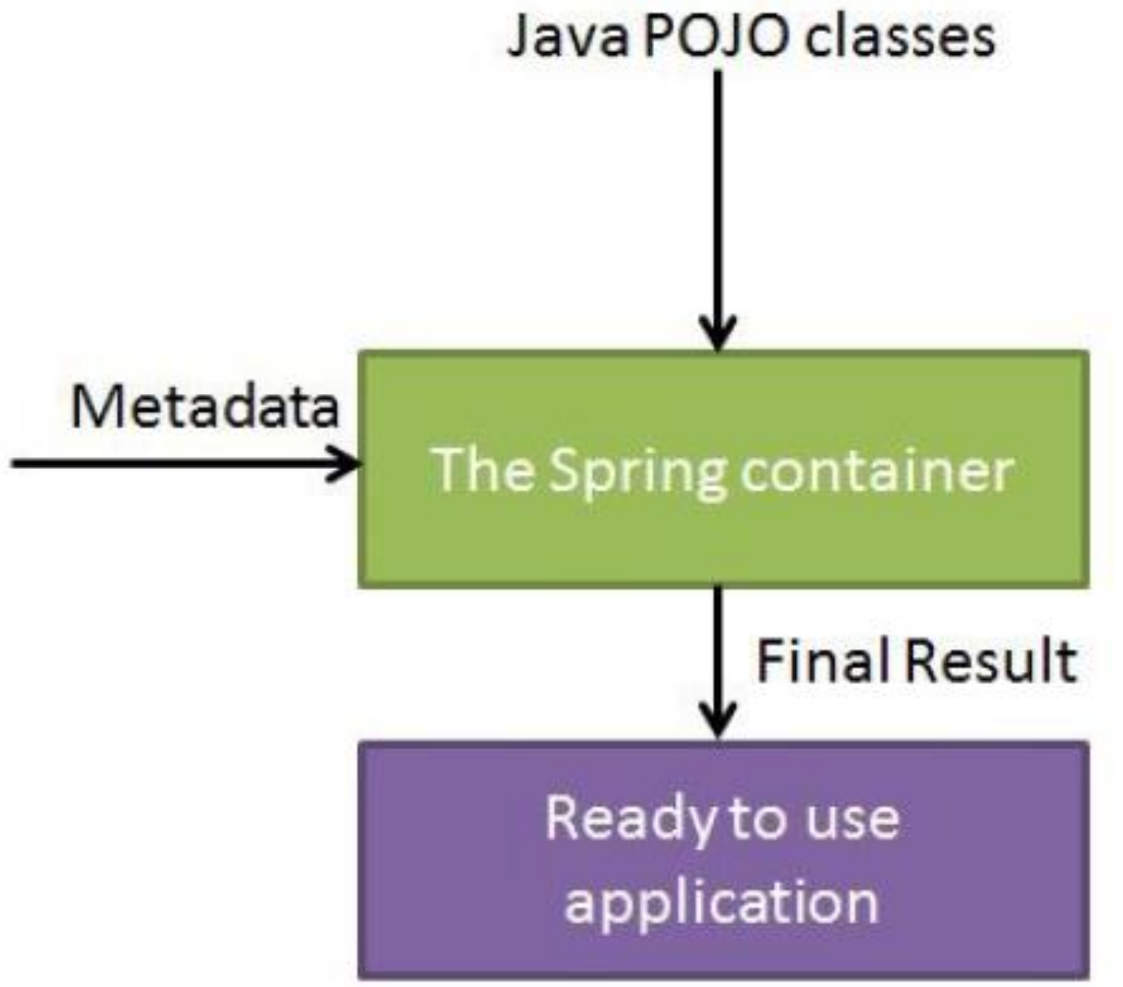


```
<?xml version="1.0" encoding="UTF-8"?>

<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.tutorialspoint.HelloWorld">
       <property name="message" value="Hello World!"/>
   </bean>

</beans>
```

Retrieve Beans

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

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

      obj.getMessage();
   }
}
```

Spring Container



Spring Configuration Metadata

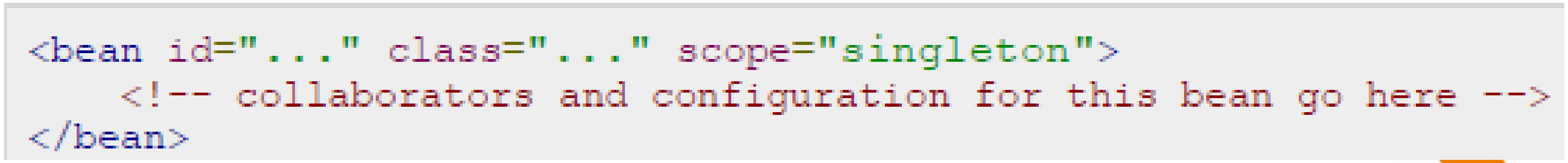
- XML based configuration file.
- Annotation-based configuration
- Java-based configuration

Spring Bean Definition

- class
- name (id)
- scope
- constructor-arg
- properties
- autowiring mode
- lazy-initialization mode
- initialization method
  - A callback, invoked just after all properties on the bean have been set
  - For the sake of post-processing the bean creation
- destruction method
  - A callback, invoked when the container is destroyed.

Spring Bean Scopes

- Common Scopes
  - singleton
  - prototype
    - container creates new bean instance of the object every time a request for that specific bean is made.
- Web-aware applications
  - request
  - session
  - global-session



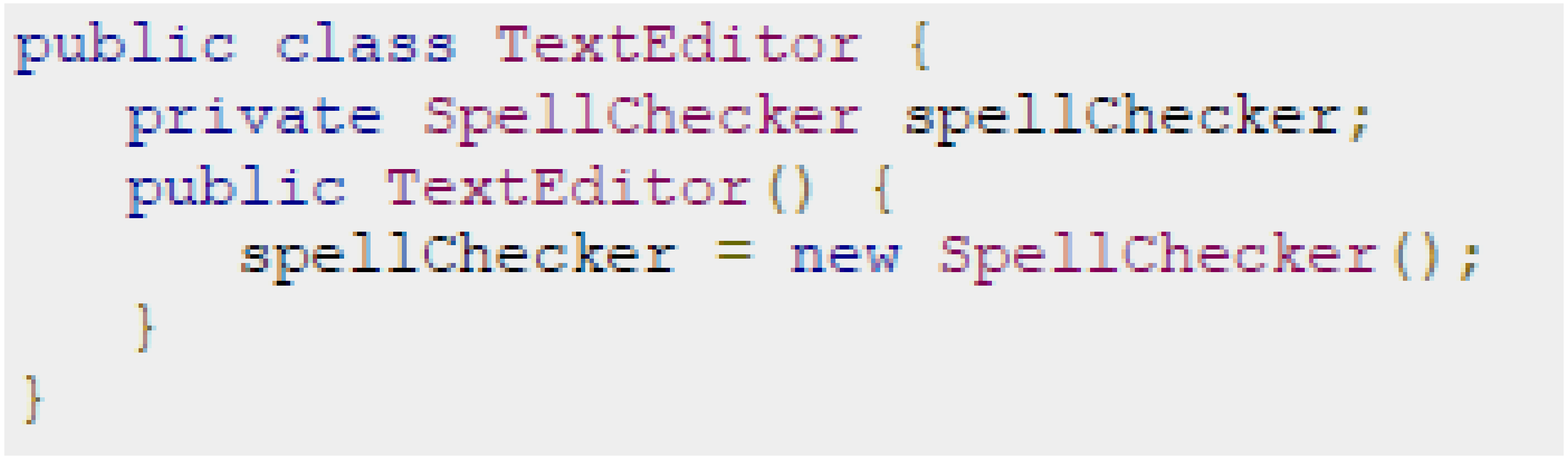
```xml
<bean id="..." class="..." scope="singleton">
    <!-- collaborators and configuration for this bean go here -->
</bean>
```

Dependency Injection

- Every java based application has a few objects that work together
- In a complex Java application, application classes should be as independent as possible
- To increase the possibility to reuse these classes
- and to test them independently
- Dependency Injection (or sometime called **wiring**) helps in gluing these classes together
- and same time keeping them independent.

Example of a Dependency:

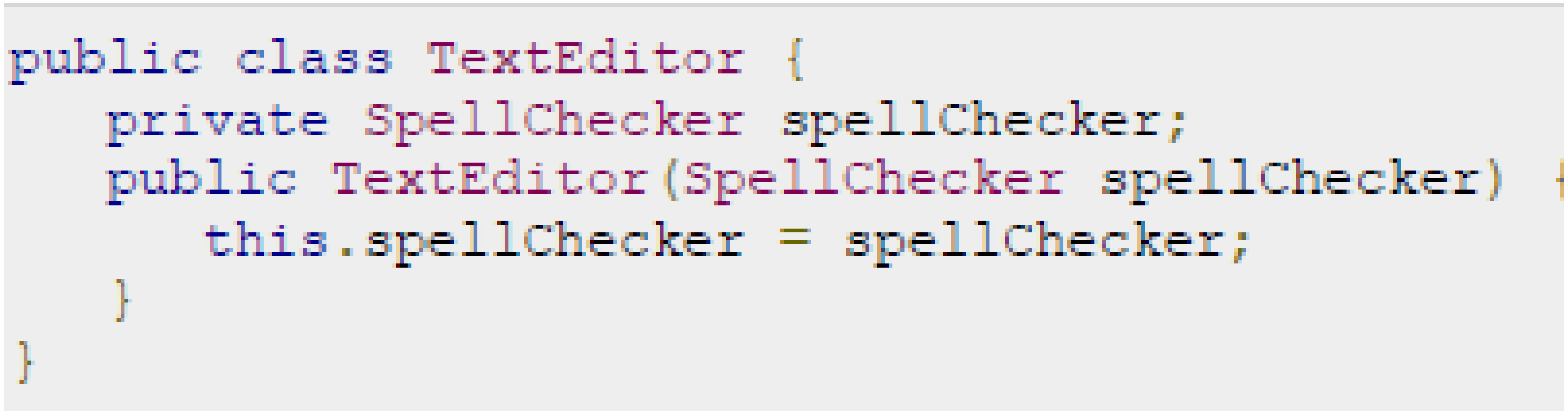
- What is wrong with this code?
- we have created a dependency between the TextEditor and the SpellChecker concrete class



```
public class TextEditor {
    private SpellChecker spellChecker;
    public TextEditor() {
        spellChecker = new SpellChecker();
    }
}
```

Solution

- inversion of control
- like this:



```
public class TextEditor {
   private SpellChecker spellChecker;
   public TextEditor(SpellChecker spellChecker) {
      this.spellChecker = spellChecker;
   }
}
```

Dependency Injection Types

- Constructor-based dependency injection
- Setter-based dependency injection

Constructor-based dependency injection

```xml
<!-- Definition for textEditor bean -->
<bean id="textEditor" class="com.tutorialspoint.TextEditor">
   <constructor-arg ref="spellChecker"/>
</bean>

<!-- Definition for spellChecker bean -->
<bean id="spellChecker" class="com.tutorialspoint.SpellChecker">
</bean>
```

Setter-based

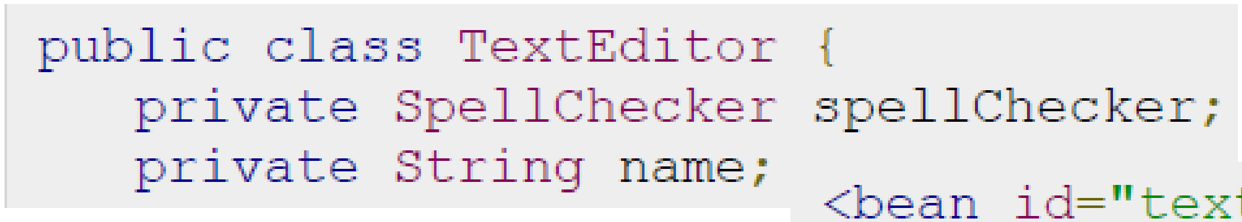
```xml
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"

   <!-- Definition for textEditor bean -->
   <bean id="textEditor" class="com.tutorialspoint.TextEditor">
      <property name="spellChecker" ref="spellChecker"/>
   </bean>

   <!-- Definition for spellChecker bean -->
   <bean id="spellChecker" class="com.tutorialspoint.SpellChecker">
   </bean>

</beans>
```

Auto-wiring

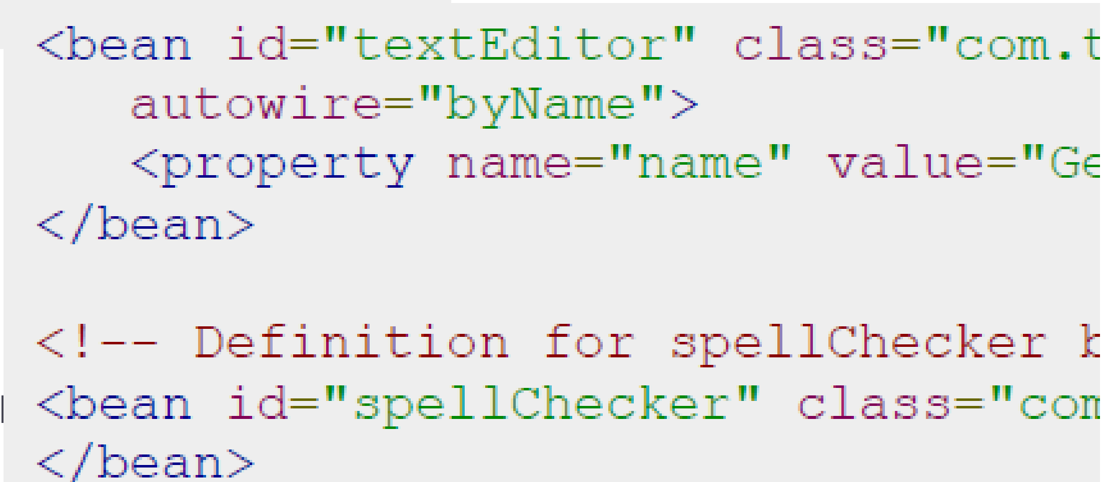
- We can **autowire** relationships between collaborating beans
- Without using <constructor-arg> or <property> elements
- Decreases the amount of XML configuration you write
- Use the **autowire** attribute of the <bean/> element
  - byName
  - byType
  - constructor
- If a bean is autowired
  - Its properties are automatically set by other defined beans

Auto-wiring : byName

- Autowiring by property name.
- Spring looks at the properties of the beans on which *autowire* attribute is set to *byName*
- Tries to match and wire its properties with the beans defined by the same names
  - If matches are not found, does nothing!



```
public class TextEditor {
   private SpellChecker spellChecker;
   private String name;
```



```
<bean id="textEditor" class="com.t
   autowire="byName">
   <property name="name" value="Ge
</bean>

<!-- Definition for spellChecker b
<bean id="spellChecker" class="com
</bean>
```

Annotation-based Configuration

- Since Spring 2.5
- to configure the dependency injection using **annotations**
- instead of using XML to describe a bean wiring
- The bean configuration is specified in the class itself by using annotations
- Annotation injection is performed before XML injection
  - thus the latter configuration will override the former
- Typical spring annotations
  - @Component
  - @Autowired

```
@Component
public class CustomerManager{
  @Autowired
  CustomerDAO customerDAO;
}
```

Spring and JSR-330 Standard

- Spring is not a javaee standard implementation
  - Servlet, JSP, JPA, EJB, JAX-RS, … are java standards
- But javaee has a new standard for dependency injection:
  - JSR 330: Dependency Injection for Java.
- Since Spring 3.0, Spring supports the JSR 330
- **@Inject** instead of Spring's **@Autowired**
  - to inject a bean
- **@Named** instead of Spring's **@Component**
  - to declare a bean

```java
@Named("contactService")
public class ContactServiceImpl {
  @Inject
  ContactManager manager;
}
```

XML Approach

```java
package ir.asta.training.contacts.dao;
public class ContactDao {
    ...
}
```

```java
package ir.asta.training.contacts.manager;
import ir.asta.training.contacts.dao.ContactDao;
public class ContactManager {
   ContactDao dao;
   ...
}
```

```xml
<bean id="contactManager"
class="ir.asta.training.contacts.manager.ContactManager">
    <property name="dao" ref="contactDao" />
</bean>
<bean id="contactDao" class="ir.asta.training.contacts.dao.ContactDao" />
```

```java
public static void main( String[] args ){
  ApplicationContext context = new ClassPathXmlApplicationContext("conf.xml");
  ContactManager cust = (ContactManager)context.getBean("contactManager");
}
```

Auto scanning with XML Approach

```java
package ir.asta.training.contacts.dao;
import javax.inject.Named;
@Named("contactDao")
public class ContactDao {
   ...
}
```

```java
package ir.asta.training.contacts.manager;
import javax.inject.Inject;
import javax.inject.Named;
import ir.asta.training.contacts.dao.ContactDao;
@Named("contactManager")
public class ContactManager {
   @Inject
   ContactDao dao;
   ...
}
```

```xml
<context:component-scan base-package="ir.asta.training.contacts" />
```

```java
 public static void main( String[] args ){
    ApplicationContext context = new ClassPathXmlApplicationContext("conf.xml");
    ContactManager cust = (ContactManager)context.getBean("contactManager");
}
```

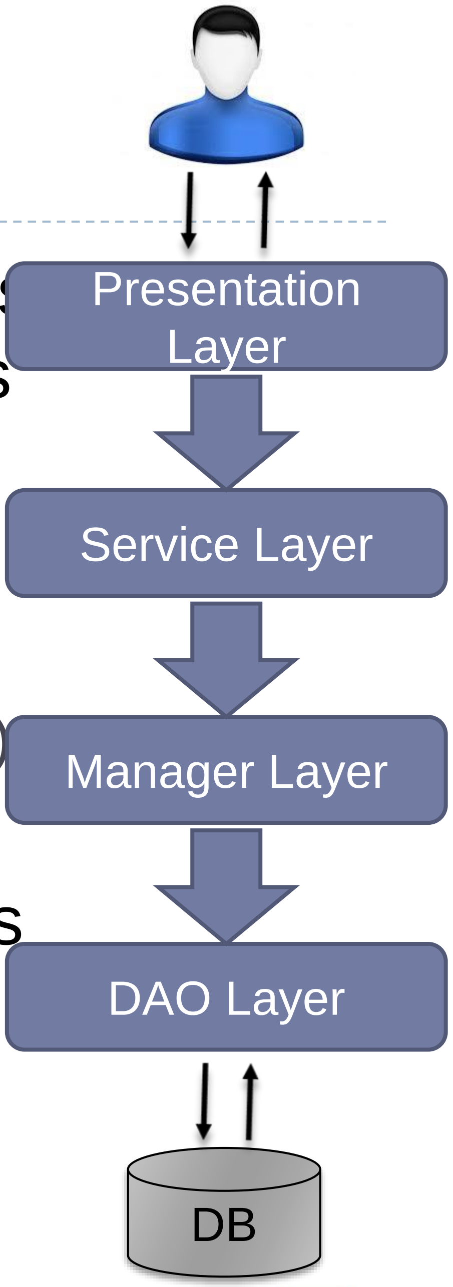
Unit Testing of Spring Beans

```
public class ContactManagerTest {
    @Test
    public void testContactManager() {
    ApplicationContext context =
    new ClassPathXmlApplicationContext("config.xml");
    ContactManager contactManager =
    (ContactManager)context.getBean("contactManager");
        //asserts
    }
}
```

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {"config.xml"})
public class ContactManagerTest {
    @Inject
    ContactManager contactManager;

    @Test
    public void testContactManager() {
        //asserts
    }
}
```

Layered Architecture

- A layer is a group of reusable components that are reusable in similar circumstances
- Common Layers:
  - Presentation layer (UI, view)
  - Service layer (web services)
  - Manager layer (business logic, domain layer)
  - Data access layer (DAO, persistence layer)
- Usually for each layer, the class instances are declared as spring beans
  - E.g., ContactDAO, ContactManager, ContactService, etc.



Exercise

- Write a web application
- “Add” servlet for telephone contacts
- With all layers
  - Dao → dummy implementation
  - Manager → just delegate
  - Servlet → Spring-enable your servlets
    - How to spring-enable a servlet?!
- Define the beans and spring-enable your project

References and Material

- Spring Framework Reference Documentation
  - http://www.springsource.org/documentation
- Spring Framework Tutorial
  - www.tutorialspoint.com/spring/spring_tutorial.pdf

Any questions?