# SPRING FRAMEWORK

Instructor: DieuNT1

# Table Content

# Learning Goals

- **After the course, attendees will be able to:**

Understand Spring Framework and its core technologies.

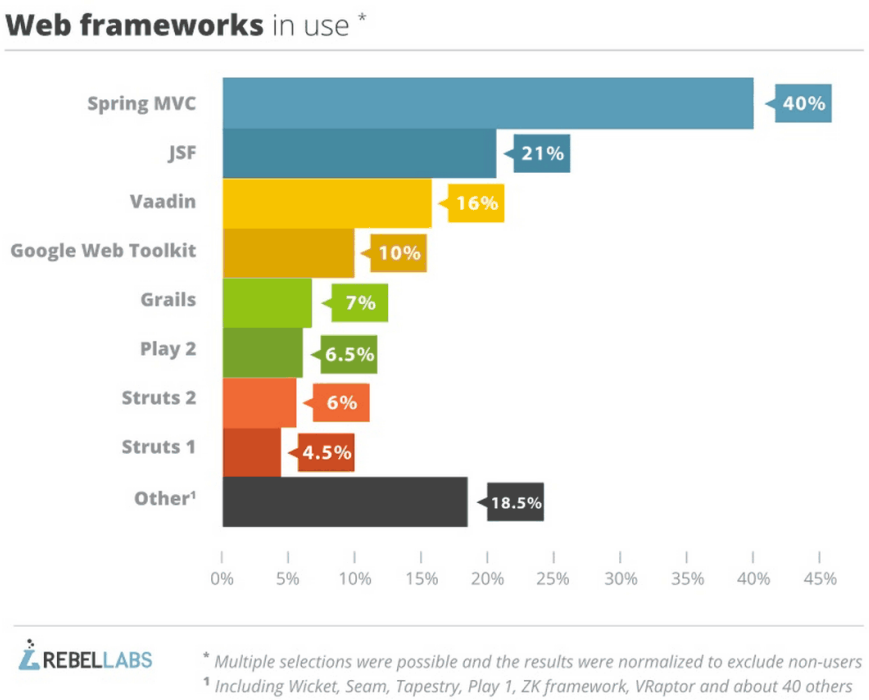
Know how to write a Web application with Spring  Framework.

Section 1

# OVERVIEW OF THE SPRING FRAMEWORK

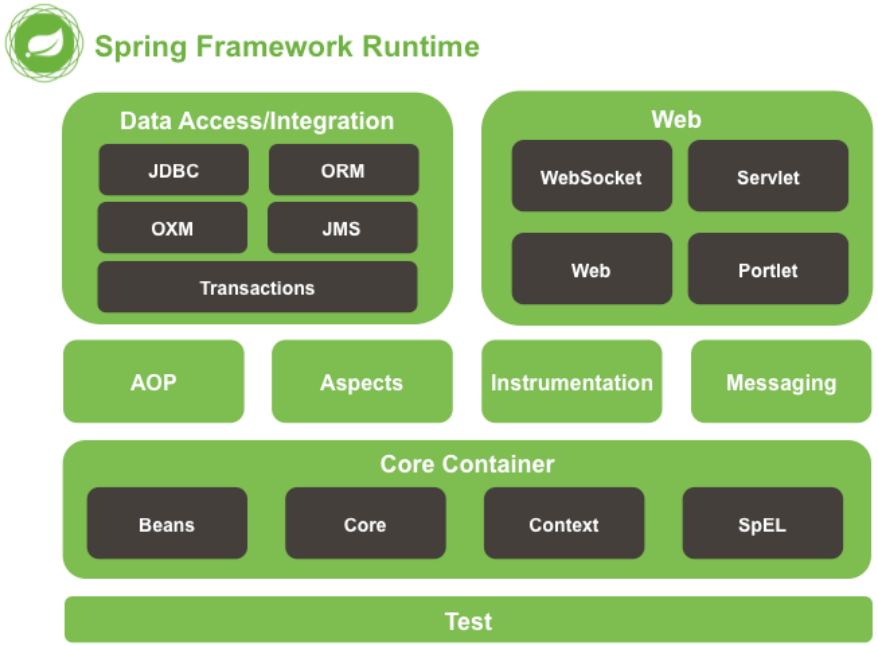
# Introduction (1/2)

- The **Spring Framework** is a Java platform that provides comprehensive infrastructure support for developing Java applications.
- **Spring framework** is one of the most popular application development frameworks used by java developers.



# Introduction (2/2)

- It consists of a **large number of modules** providing a range of services:
  - ✓ Core Container;
  - ✓ Data Access/Integration;
  - ✓ Web;
  - ✓ Test.
  - ✓ AOP (Aspect Oriented Programming);
  - ✓ Instrumentation;
  - ✓ Messaging;



# Modules (1/3)

## ❖ Data Access/Integration

- ✓ **JDBC** module provides a JDBC-abstraction layer
- ✓ ***ORM*** *(*object-relational mapping APIs): *integrate with JPA,* JDO, Hibernate, and iBatis.
- ✓ **OXM** (Object/XML mapping) implemente for JAXB, Castor, XMLBeans, JiBX and XStream.
- ✓ **JMS** (Java messaging service): producing and consuming messages.
- ✓ **Transaction**: supports programmatic and declarative transaction management.

# Modules (2/3)

## ❖ Web

- ✓ **Web**: Support some features in web application such as : file upload, file download
- ✓ **Web-Servlet**: contains Spring's model-view-controller (*MVC) implementation for web* applications
- ✓ **Web-Struts**: contains the support classes for integrating a classic Struts web tier (struts 1 or struts 2) within a Spring application
- ✓ *Web-Portlet module provides the MVC implementation to be used in a portlet environment and* mirrors the functionality of Web-Servlet module.

# Modules (3/3)

## ❖ AOP and Instrument

- ✓ Spring's *AOP module provides an AOP Alliance-compliant aspect-oriented programming* implementation allowing you to define
- ✓ *Aspects module provides integration with AspectJ.*
- ✓ *Instrumentation module provides class instrumentation support and classloader implementations* to be used in certain application servers.

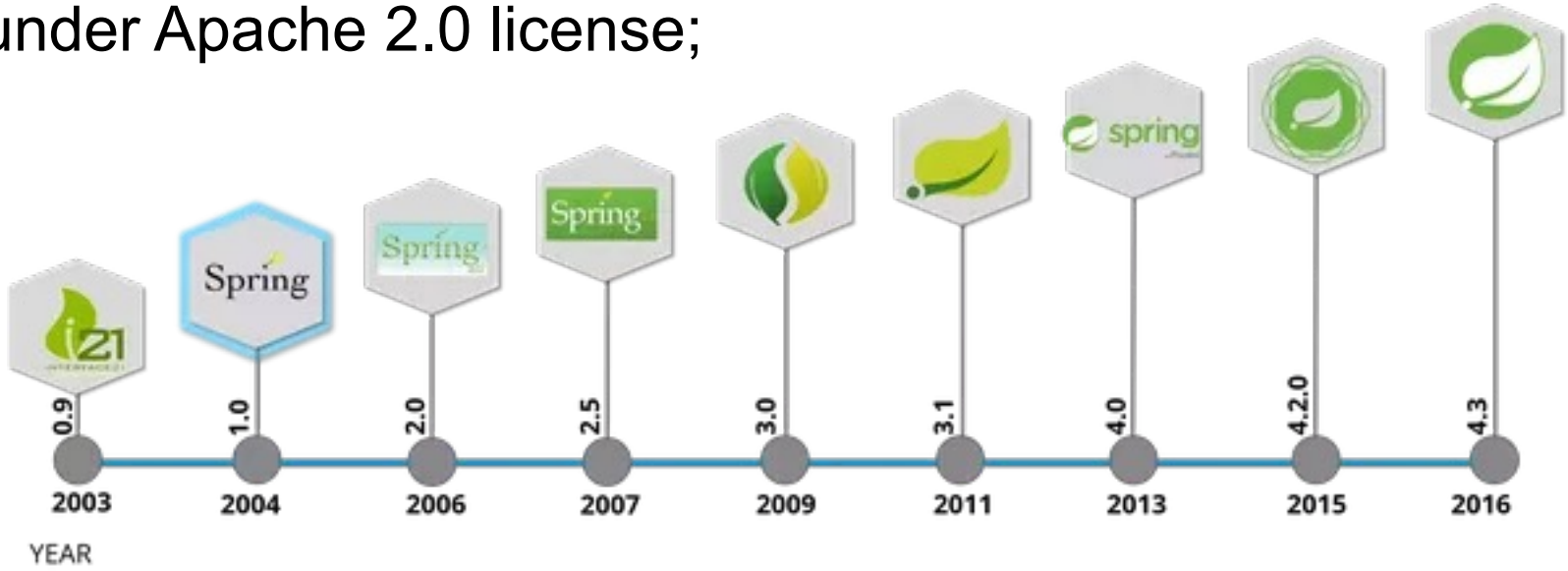
## ❖ Test

- ✓ The *Test module supports the testing of Spring components with JUnit or TestNG*

# History of Spring Framework

- In **October 2002** by Rod Johnson;
  - ✓ He proposed a simpler solution based on ordinary java classes (**POJO** – plain old java objects) and dependency injection (DI or IoC).

- In **June 2003**, spring 0.9 was released under Apache 2.0 license;



Section 2

# SPRING IOC

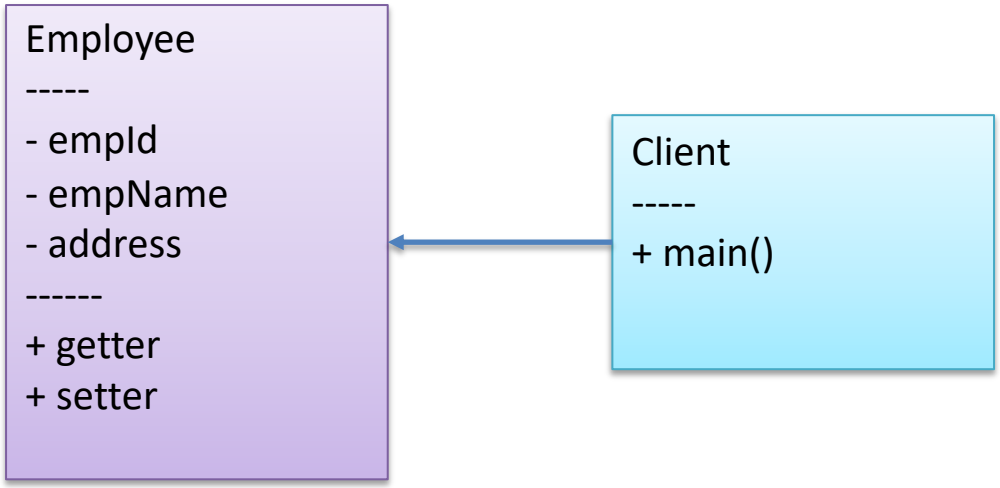
# What is Spring Inversion of Control(IoC )?

- **Let's first understand the issue, consider the following class:**

```
package com.fsoft.bean;

public class Employee {
     private int empId;
     private String empName;
     private String address;

     public Employee() {
     }

     public Employee(int empId, String empName, String address) {
          this.empId = empId;
          this.empName = empName;
          this.address = address;
     }

     //getter-setter methods
}
```

## Standard code that without IoC

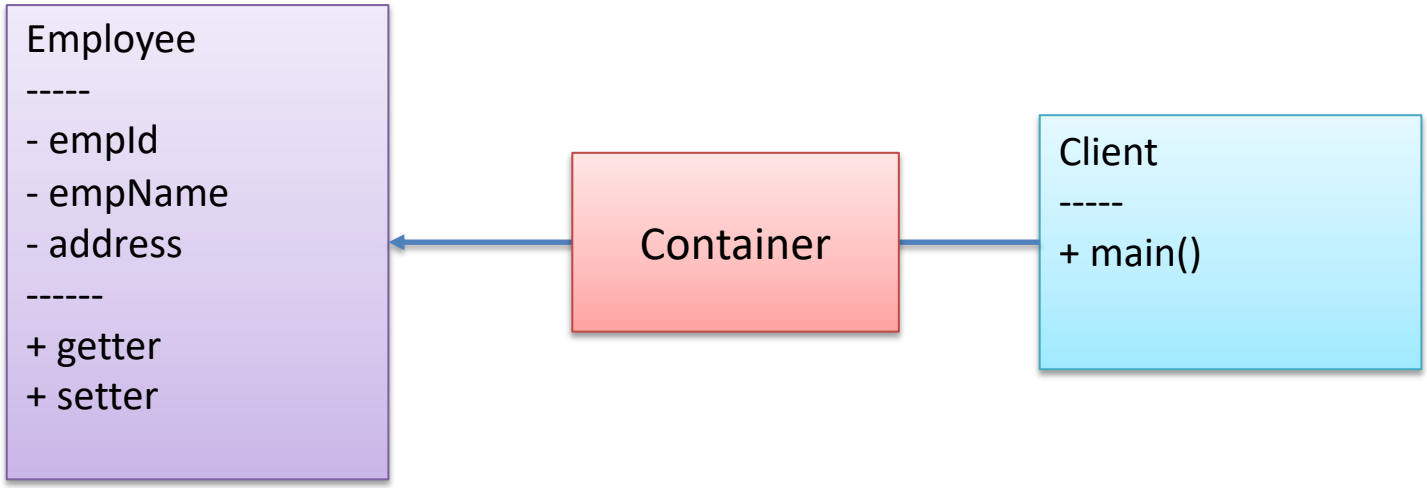
```
package com.fsoft.bean;
public class Client {
    public static void main(String[] args) {
        Employee employee = new Employee();
        employee.setEmpId(1);
        employee.setEmpName("John Watson");
        employee.setAddress("New York");
        System.out.println("Employee details: " + employee);
    }
}
```



# Spring IoC

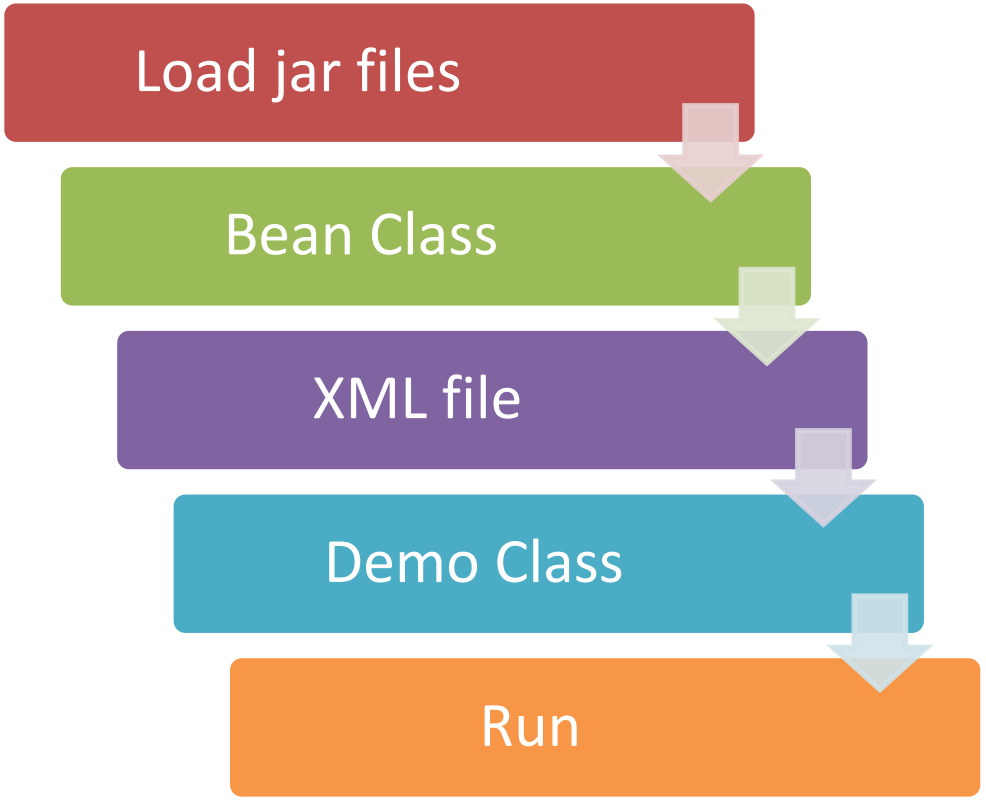
## ❖ With IoC

- ✓ You don't create objects. Using Bean Configuration File;
- ✓ Create an application context where we used framework API **ClassPathXmlApplicationContext()**.
- ✓ This API loads beans configuration file and based on the provided API, it will create and initialize all the objects.



# Spring IoC Demo

## ❖ Start Coding in 5 Simple Steps



# Required Maven Dependency

## ❖ Add maven dependency in pom.xml file.

```
<project ...>
<properties>
      <project.build.sourceEncoding>UTF-8
      </project.build.sourceEncoding>
      <spring.version>4.3.10.RELEASE</spring.version>
      <junit.version>4.12</junit.version>
      <servlet.version>3.1.0</servlet.version>
</properties>
<dependencies>
<!-- Junit -->
<dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>${junit.version}</version>
      <scope>test</scope>
</dependency>
<!--Servlet-Api -->
<dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>javax.servlet-api</artifactId>
      <version>${servlet.version}</version>
</dependency>
```

```
<!-- Spring Framework -->
<dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>${spring.version}</version>
</dependency>
<dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-beans</artifactId>
      <version>${spring.version}</version>
</dependency>
<dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>${spring.version}</version>
</dependency>
```

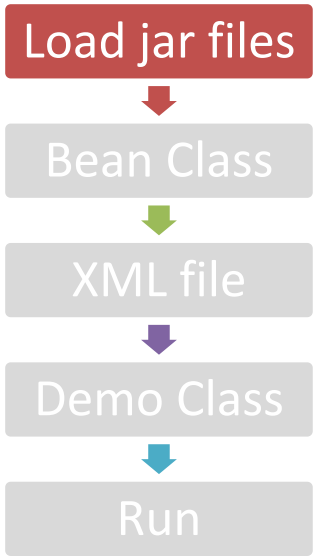
# Required Maven Dependency

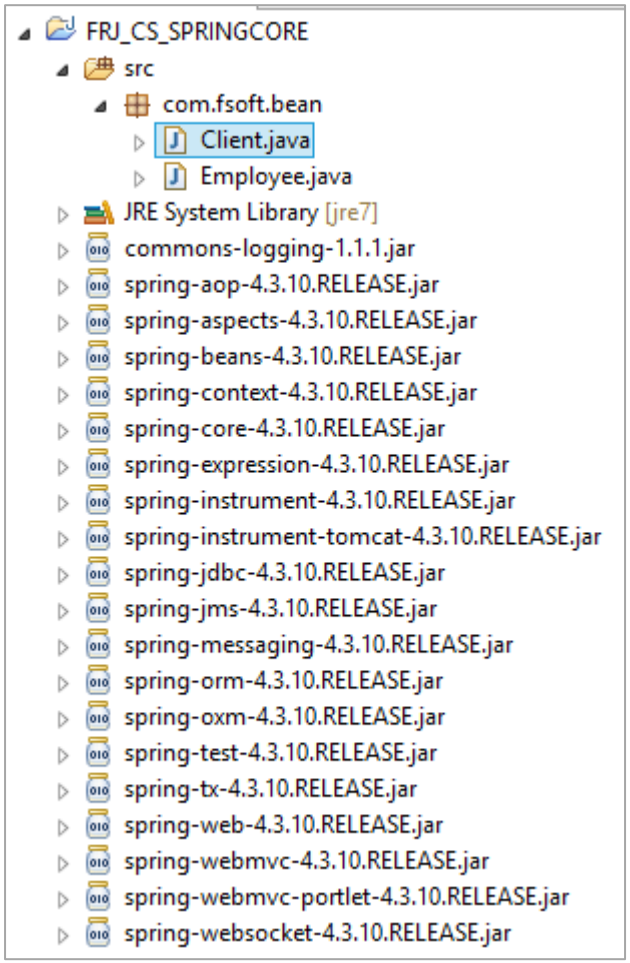
- **Add maven dependency in pom.xml file.**

```
<!-- Spring JDBC (if need) -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>${spring.version}</version>
</dependency>
<!-- MS SQL Server (if need) -->
<dependency>
    <groupId>com.microsoft.sqlserver</groupId>
    <artifactId>mssql-jdbc</artifactId>
    <version>6.1.0.jre7</version>
</dependency>
</dependencies>
<build>
    <finalName>FRESHERACADEMY</finalName>
</build>
</project>
```

# Spring IoC Demo

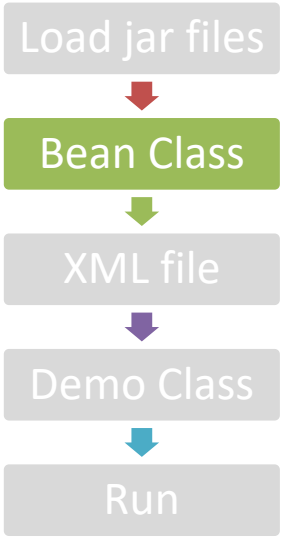
❖ Create **Java Project** and **Load jar files**



# Spring IoC Demo

## ❖ Create Bean Configuration File

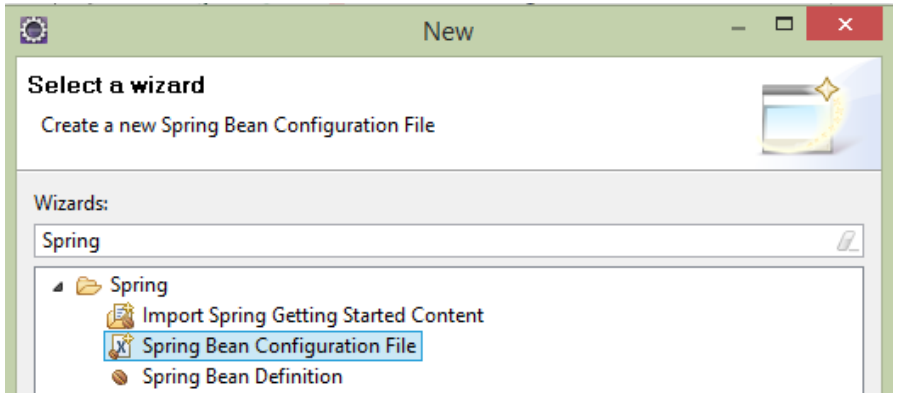



```
package com.fsoft.bean;

public class Employee {
	private int empId;
	private String empName;
	private String address;

	public Employee() {
	}

	public Employee(int empId, String empName, String address) {
		this.empId = empId;
		this.empName = empName;
		this.address = address;
	}

	//getter-setter
}
```

# Spring IoC Demo

## ❖ Create Bean Configuration File

Load jar files
↓
Bean Class
↓
**XML file**
↓
Demo Class
↓
Run

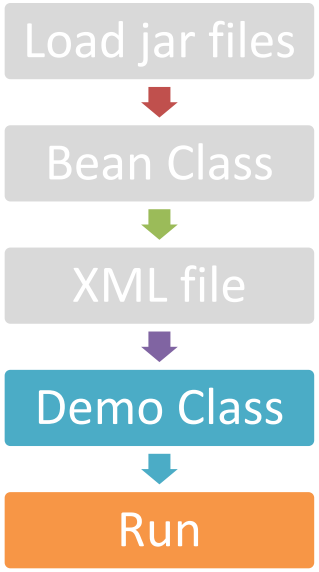



*employeeBean.xml

```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.xsd">

    <bean id="emp1" class="com.fsoft.bean.Employee">
        <property name="empId" value="1" />
        <property name="empName" value="Jack" />
        <property name="address" value="Eastern Shores" />
    </bean>

    <bean id="emp2" class="com.fsoft.bean.Employee">
        <property name="empId" value="2" />
        <property name="empName" value="Jennie" />
        <property name="address" value="Shouthern Shores" />
    </bean>
</beans>
```

# Spring IoC Demo

## ❖ Create Source Files



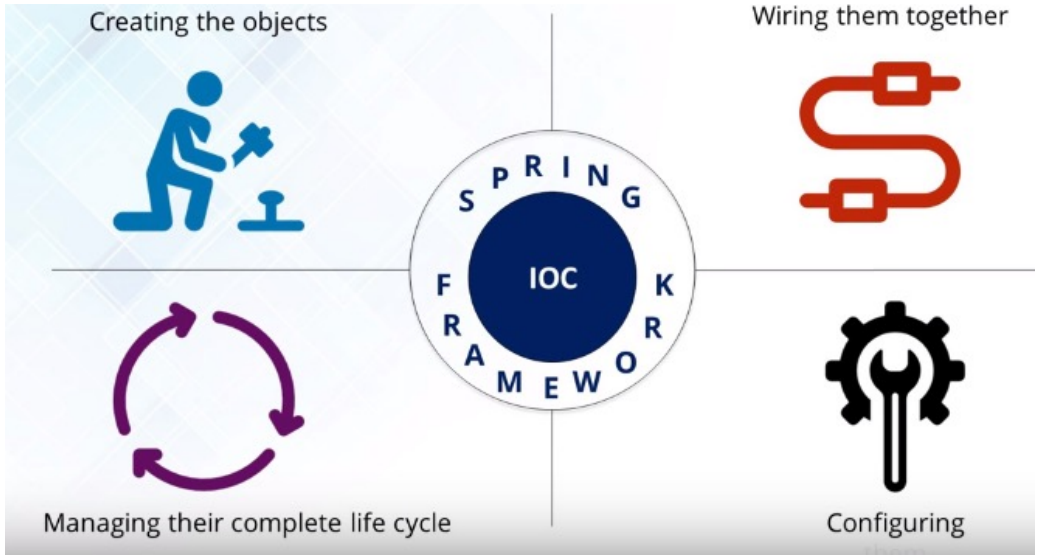

```java
package com.fsoft.bean;

public class Client {
public static void main(String[] args) {

     ApplicationContext context = new
                ClassPathXmlApplicationContext("employeeBean.xml");

     Employee emp1 = (Employee) context.getBean("emp1");
     Employee emp2= (Employee) context.getBean("emp2");

     System.out.println("Employee details: " + emp1);
     System.out.println("Employee details: " + emp2);
}
}
```

**Result**

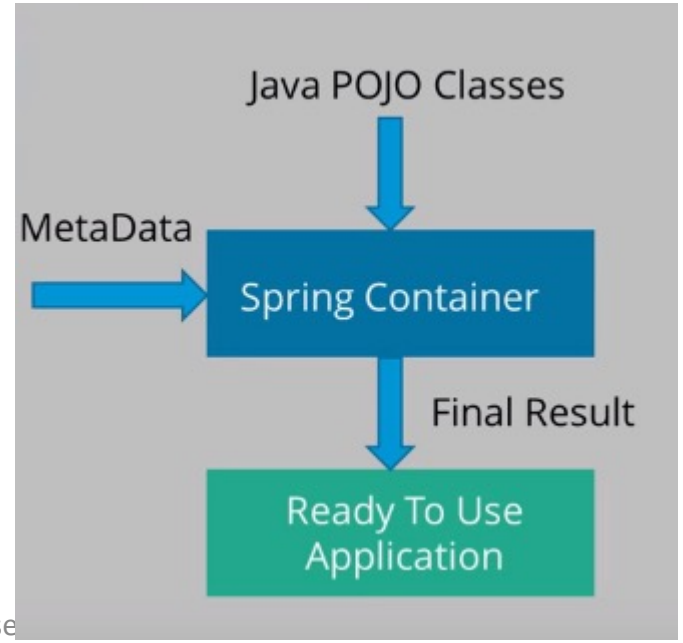
```
Initial!
Employee details: Employee [empId=1, empName=Jack, address=Eastern Shores]
Employee details: Employee [empId=2, empName=Jennie, address=Shouthern Shores]
Destroy!
```

# IOC Container Features

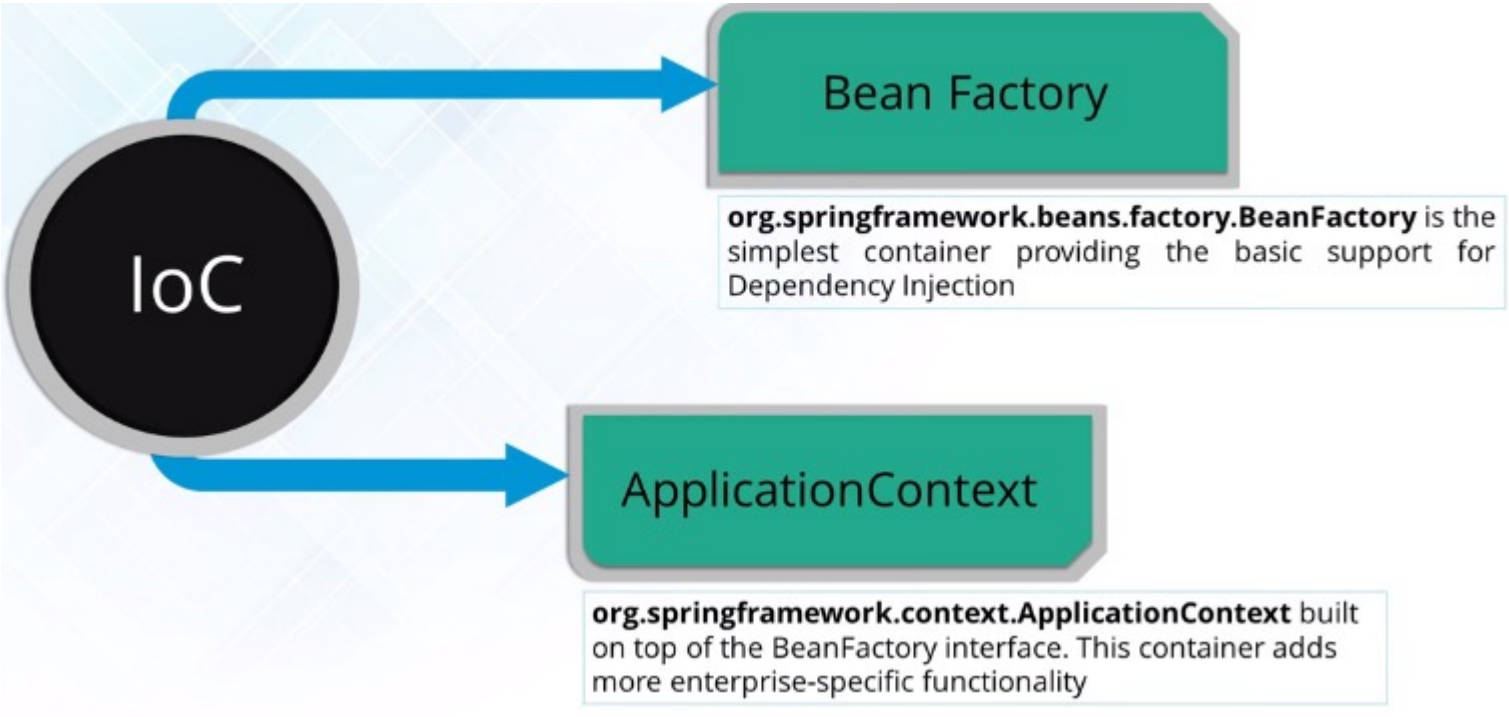
- IoC




The **Spring IoC** container by using Java POJO classes and configuration metadata procedures a fully configured and executable system or application.



# Types Of IoC Container



Section 3

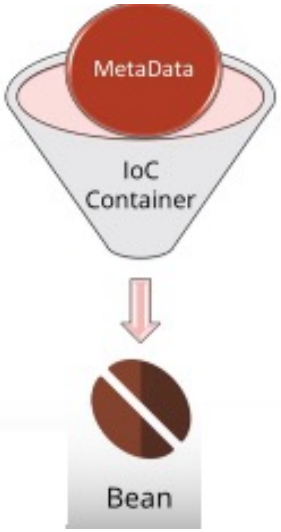
# SPRING BEAN

# Bean Object

**Beans** are the objects that form the backbone of our application and are managed by the **Spring IoC container.**

**Spring IoC container** *instantiates*, *assembles*, and *manages* the bean object.

The configuration metadata that are supplied to the container are used **create Beans object.**



# Some Bean Properties

| Property | Explain |
|---|---|
| **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 and will be discussed in subsequent chapters. |
| **properties** | Define properties of class. |
| **autowire mode** | Set autowire for bean. |
| **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. |

# class property

```java
package com.fsoft.bean;
public class Address {
	private String city;
	private String street;

	public Address() {

	}

	public Address(String city, String street) {
		this.city = city;
		this.street = street;
	}

	// getter-setter methods
}
```

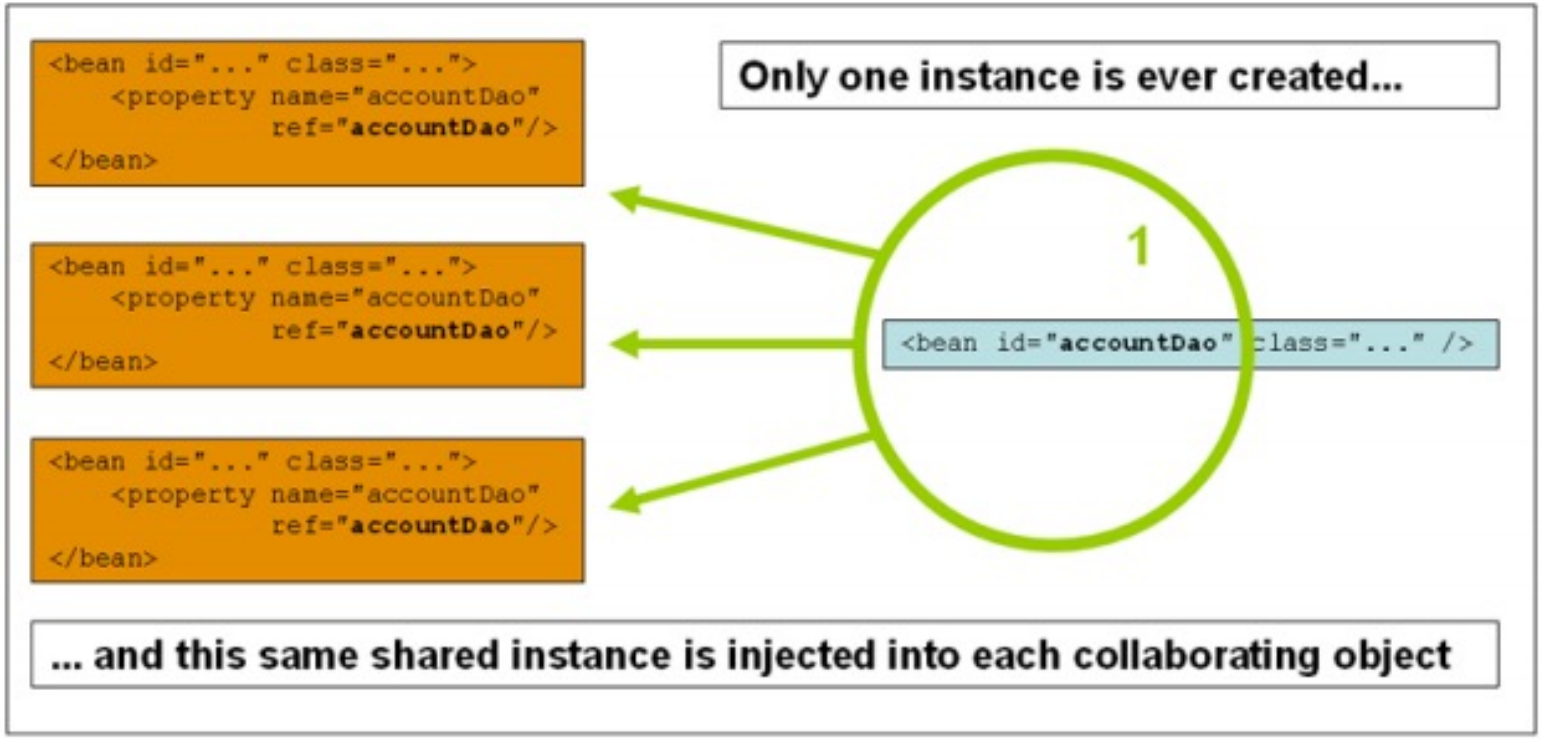
```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="addr" class="com.fsoft.bean.Address">
        <property name="city" value="Hanoi" />
        <property name="street" value="Duytan" />
    </bean>
</beans>
```

# scope property

| Scope | Explain |
|---|---|
| **singleton** | (Default) Scopes a single bean definition to a single object instance per Spring IoC container. |
| **prototype** | Scopes a single bean definition to any number of object instances. |
| **request** | Scopes a single bean definition to the lifecycle of a single HTTP request; that is, each HTTP request has its own instance of a bean created off the back of a single bean definition. |
| **session** | Scopes a single bean definition to the lifecycle ofcan HTTP Session. Only valid in the context of acweb-aware Spring ApplicationContext. |
| **global session** | Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a portlet context. Only valid in the context of a web-aware Spring ApplicationContext. |

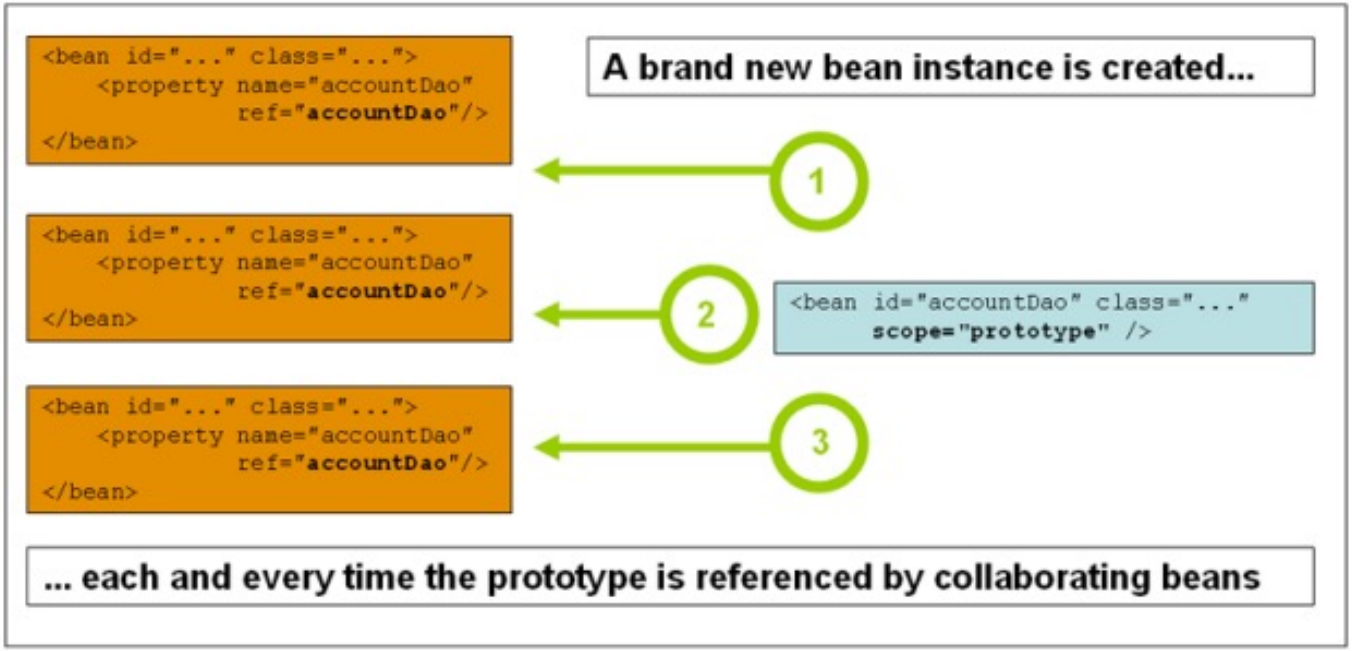
# scope property

❖ Scope "singleton"



# scope property

❖ Scope "prototype"



Section 4

# SPRING DI

# Dependency Injection (DI)

It is a **design pattern** which removes the dependency from the programming code, that makes the Application easy to manage and test.

**Dependency Injection** makes our programming code *loosely coupled*, which means change in implementation doesn't affects the use.

# Spring DI

- Consider you have an application which has a employee component and you want to identify a their address.
- Your **standard code** would look something like this:

```
public class Employee {

    private int empId;
    private String empName;
    private Address address; // HAS-A relationship
    /*private String address;*/

    public Employee() {
        this.empId = 0;
        this.empName = "N/A";
        this.address = new Address();
    }
}
```

# Spring DI

- Let's create a dependency between the Employee and the Address.
- In an IoC scenario, we would instead do something like this:

```
public class Employee {

    private int empId;
    private String empName;
    private Address address; // HAS-A relationship

    public Employee(Address address) {
        super();
        this.address = address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }
}
```

- We can inject the dependancies using the **setter** or **constructor** injection.

# Type of Dependency Injection

Spring framework avails two ways to inject dependency :

| | | |
|---|---|---|
| By Constructor | 1 | The **<constructor-arg>** subelement of **<bean>** is used for constructor injection |
| By Setter method | 2 | The **<property>** subelement of **<bean>** is used for setter injection |

# Spring DI Demo

## ❖ By Constructor

```
<?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="addr" class="com.fsoft.bean.di.Address">
        <property name="city" value="Hanoi" />
        <property name="street" value="Duytan" />
    </bean>

    <bean id="emp3" class="com.fsoft.bean.di.Employee">
        <property name="empId" value="3"/>
        <property name="empName" value="My"/>
        <property name=" address " ref="addr">/> <--setter-->

        <constructor-arg name="address" ref="addr" />
    </bean>
</beans>
```

Using **<constructor-arg>** subelement to initialize instance variables

# Spring DI Demo

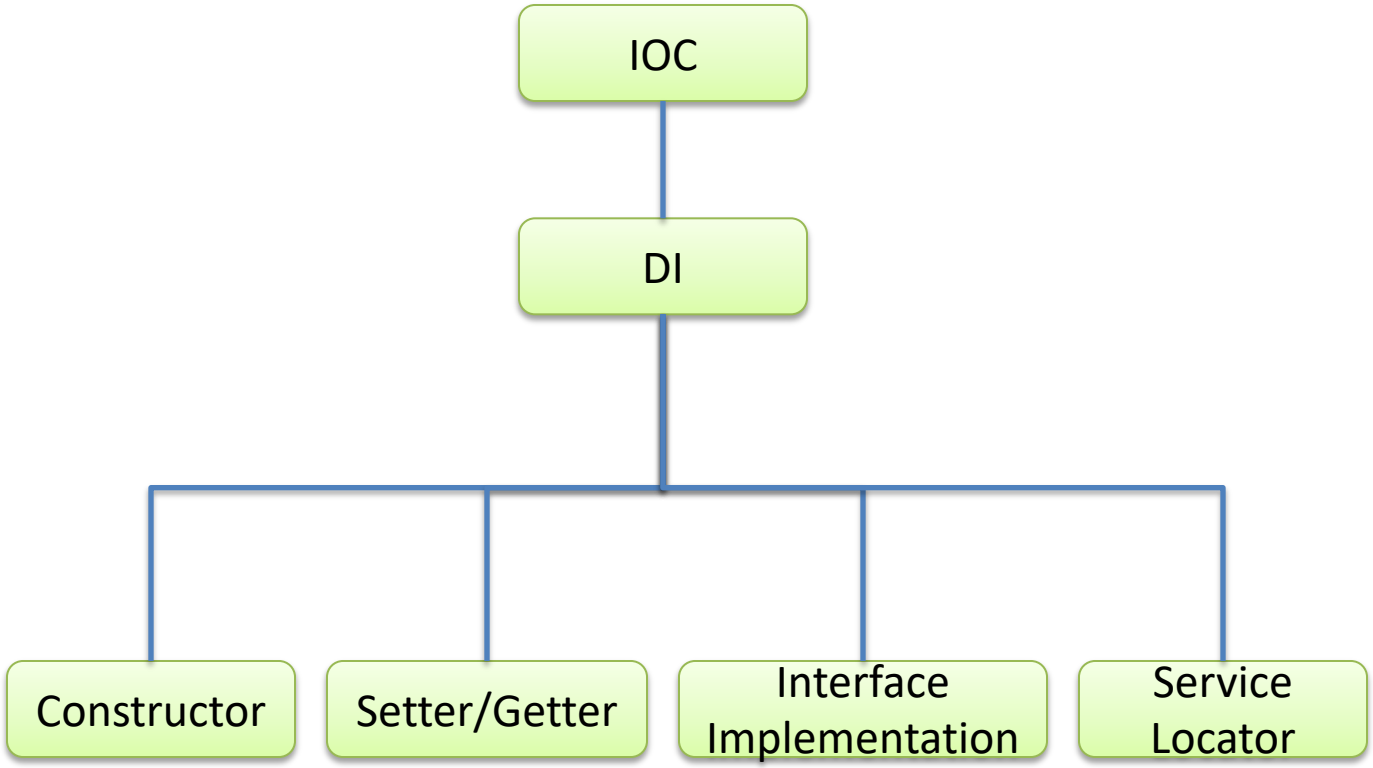
## ❖ By Setter

```
<?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="addr" class="com.fsoft.bean.di.Address">
        <property name="city" value="Hanoi" />
        <property name="street" value="Duytan" />
    </bean>

    <bean id="emp4" class="com.fsoft.bean.di.Employee">
        <property name="empId" value="4"/>
        <property name="empName" value="My"/>
        <property name="address" ref="addr" />
    </bean>
</beans>
```

Using **<property>** subelement to initialize instance variables

# Spring IOC and DI

❖ Ways of implement IOC

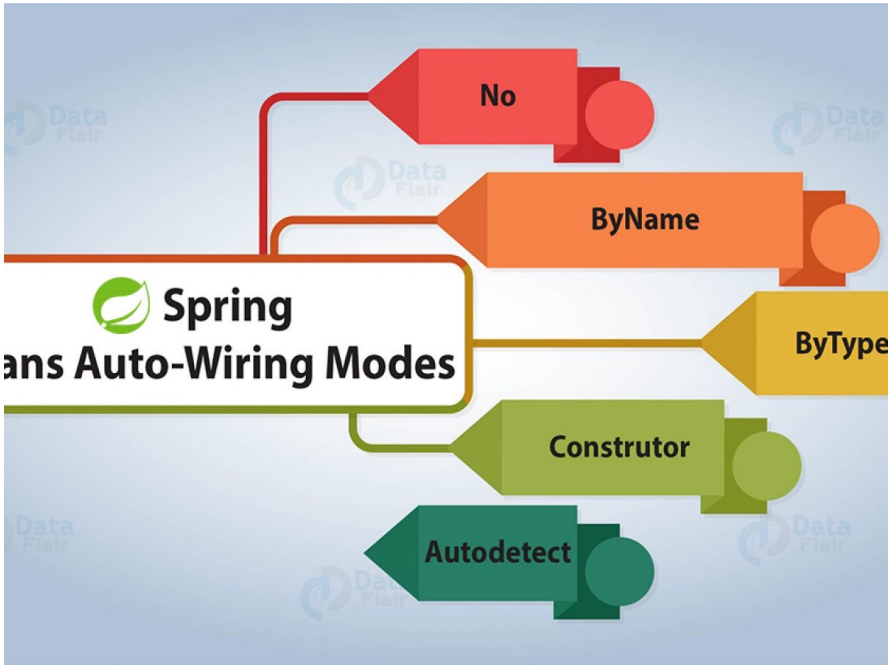


Section 5

# AUTOWIRING IN SPRING

# Introduction

- Spring provides a way to automatically detect the relationships between various beans
- The XML-configuration-based autowiring functionality has five modes – no, byName, byType, constructor, and autodetect. The default mode is no.



# Example classes

```java
public class Department {
	private String deptName;

	public String getDeptName() {
		return deptName;
	}

	public void setDeptName(String deptName) {
		this.deptName = deptName;
	}
}
```

```java
public class Employee {
	private int eid;
	private String ename;
	private Department department;

	public int getEid() {
		return eid;
	}

	public void setEid(int eid) {
		this.eid = eid;
	}

	public String getEname() {
		return ename;
	}

	public void setEname(String ename) {
		this.ename = ename;
	}

	public Department getDepartment() {
		return department;
	}

	public void setDepartment(Department department) {
		this.department = department;
	}

	public void showEployeeDetails() {
		System.out.println("Employee Id : " + eid);
		System.out.println("Employee Name : " + ename);
		System.out.println("Department : " +
						department.getDeptName());
	}
}
```

# Autowiring Modes

- no: It's the default autowiring mode. It means no autowiring.
- byName: The byName mode injects the object dependency according to name of the bean.
  - ✓ In such a case, the **property** and **bean name** should be the same.
  - ✓ It internally calls the **setter method**.

```
..
<bean id="department" class="fa.training.entities.Department">
     <property name="deptName" value="Information Technology" />
</bean>

<bean id="employee" class="fa.training.entities.Employee" autowire="byName">
     <property name="eid" value="100"/>
     <property name="ename" value="100"/>
</bean>
```

```
Employee employee = (Employee) applicationContext.getBean("employee");
employee.showEployeeDetails();
```

**Output:**

```
Employee Id : 100
Employee Name : 100
Department : Information Technology
```

# Autowiring Modes

- byType: The byType mode injects the object dependency according to type.
  - ✓ So it can have a **different property and bean name**.
  - ✓ It internally calls the **setter method**.

```
..
<bean id="dept" class="fa.training.entities.Department">
    <property name="deptName" value="Information Technology" />
</bean>

<bean id="employee" class="fa.training.entities.Employee” autowire="byType">
    <property name="eid" value="100"/>
    <property name="ename" value="100"/>
</bean>
```

# Autowiring Modes

- constructor: The constructor mode injects the dependency by calling the constructor of the class.
  - ✓ It calls the constructor having a large number of parameters.

```
..
<bean id="dept" class="fa.training.entities.Department">
    <property name="deptName" value="Information Technology" />
</bean>

<bean id="employee" class="fa.training.entities.Employee”
                                        autowire="constructor">
    <property name="eid" value="100"/>
    <property name="ename" value="100"/>
</bean>
```

```
class Employee {
    public Employee(Department department) {
        super();
        this.department = department;
    }
}
```

Section 6

# SpEL

# Spring EL

- The Spring Expression Language (**SpEL** for short) is a powerful expression language that supports querying and manipulating an object graph at runtime.
- The Spring EL is similar with OGNL and JSF EL, and evaluated or executed during the bean creation time.
- All Spring expressions are available via XML or annotation.

# SpEL in XML based configuration

- The SpEL are enclosed with #{ SpEL expression }:

```
<?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="addr" class="com.fsoft.bean.di.Address">
    <property name="city" value="Hanoi" />
    <property name="street" value="Duytan" />
</bean>
<bean id="employee" class="com.fsoft.bean.di.Employee">
   <property name="empId" value="12"/>
   <property name="empName" value="My"/>
    <property name="address" value="#{addr}"></property>
</bean>
</beans>
```

# SpEL in Annotation-based configuration

```java
package com.fsoft.bean;

import org.springframework.beans.factory.annotation.Value;

@Component("addr")
public class Address {
    @Value("Hanoi")
    private String city;

    @Value("Duytan")
    private String street;

    public Address() {

    }
    // ...
}
```

# Spring EL in Annotation

- To use SpEL in annotation, you must register your component via annotation. If you register your bean in XML and define @Value in Java class, the @Value will failed to execute.

```
<?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"
    xmlns:context="http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-
        3.0.xsd"  >

    <context:component-scan base-package="com.fsoft.bean" />
    // ...
</beans?
```

# SUMMAY

1. Overview of the Spring Framework
2. Spring IoC
3. Spring Bean
4. Dependency Injection
5. SpEL

# Thank you