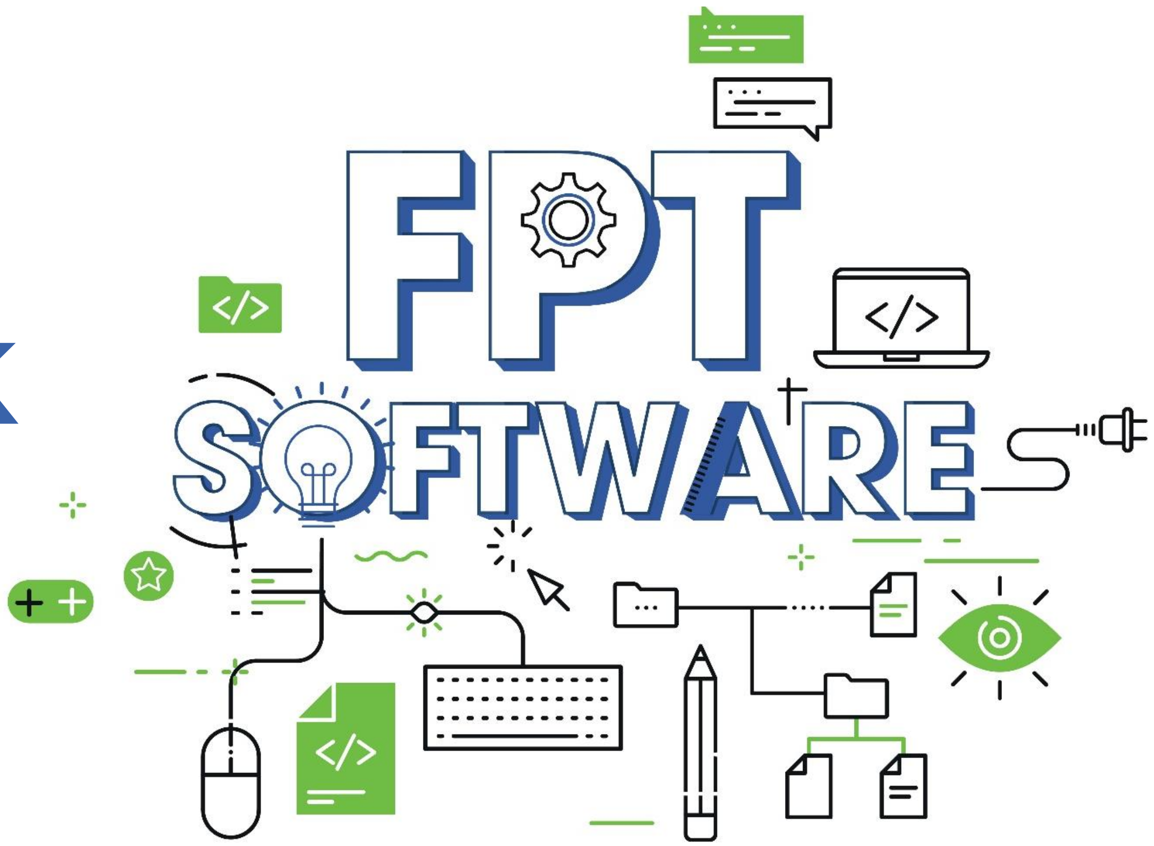
SPRING FRAMEWORK

Design by: DieuNT1



Agenda

Lesson Objectives

1
- Understand Spring Framework and its core technologies.

2
- Understand the architectural components of the Spring Framework: IoC, DI

3
- Understand the Spring modules

4
- Able to Setting up a Spring Development Environment

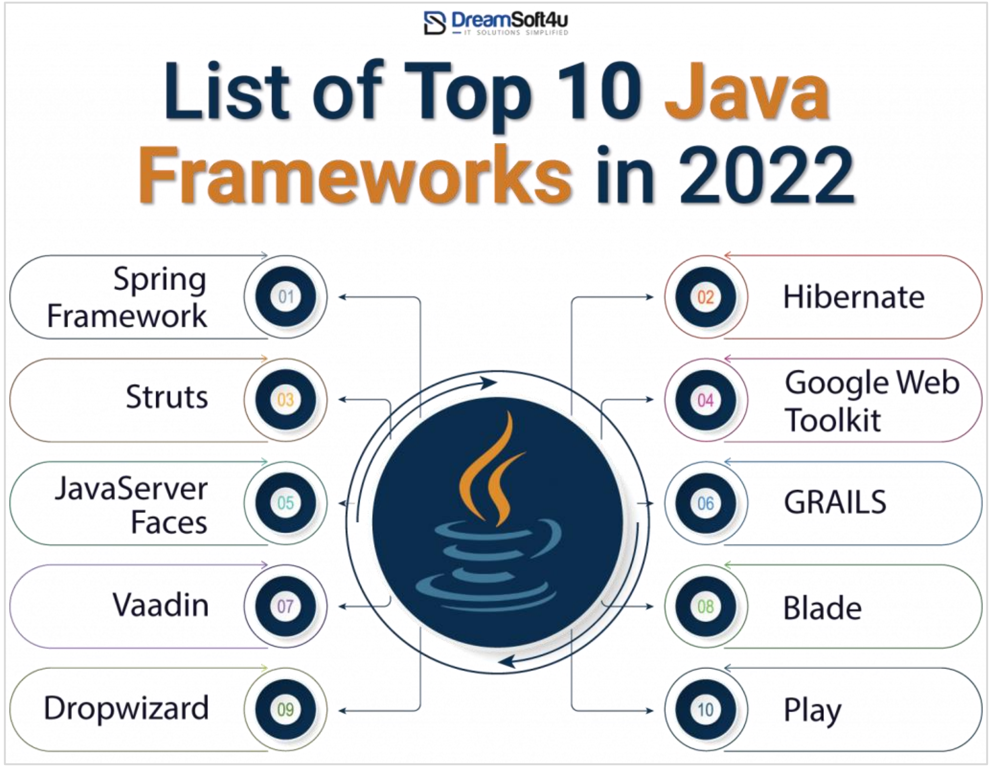
5
- Creating and Configuring Spring Applications

Section 1

Overview of the Spring Framework

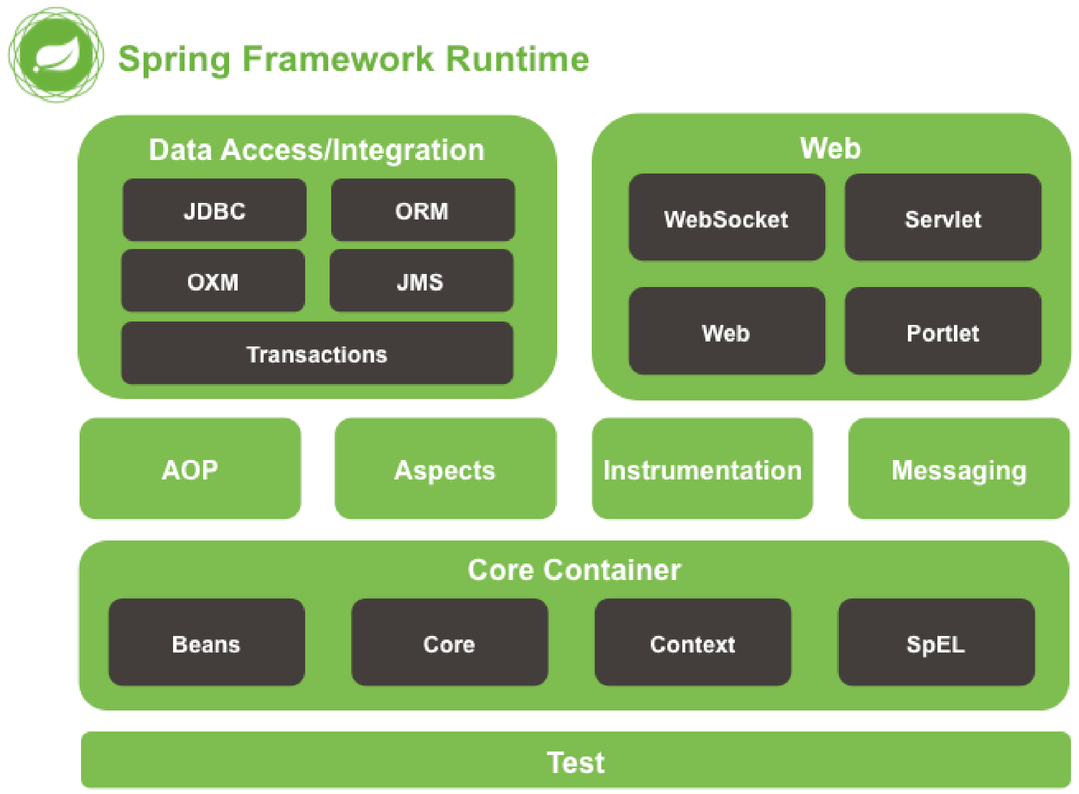
Introduction

- 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

- **Spring framework** 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;



Spring Modules

- **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.

Spring Modules

- **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.

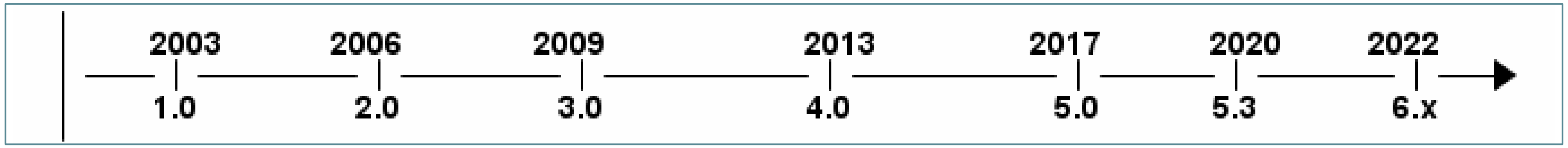
Spring Modules

- **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;



- ✓ 6.1.x is the upcoming feature branch (November 2023).
- ✓ 6.0.x is the main production line as of November 2022. This new generation of the framework comes with a JDK 17 and Jakarta EE 9 baseline.
- ✓ 5.3.x is the final feature branch of the 5th generation, with long-term support provided on JDK 8, JDK 11, JDK 17 and the Java EE 8 level.
- ✓ *4.3.x reached its official EOL (end-of-life) on December 31st, 2020. No further maintenance and security patches are planned in that line.*
- ✓ *3.2.x reached its official EOL (end-of-life) on December 31st, 2016. No further maintenance and security patches are planned in that line.*

Section 2

Spring IOC Container Overview

What We'll Learn?

1. What Is the Spring Container?
2. What is Configuration Metadata?
3. How to Create a Spring Container?
4. How to Retrieve Bean from Spring Container?
5. Spring IOC Container XML Config Example
6. Spring IOC Container Java Config Example

What is the Spring Container?

The **Spring container** is responsible for ***instantiating***, ***configuring***, and ***assembling*** the Spring beans.

The container **gets its instructions on what objects to instantiate**, **configure**, and **assemble** by reading configuration metadata.

- The configuration metadata is represented in **XML**, **Java annotations**, or **Java code**.
- **The responsibilities of IOC container are:**
  - ✓ Instantiating the bean
  - ✓ Wiring the beans together
  - ✓ Configuring the beans
  - ✓ Managing the bean's entire life-cycle

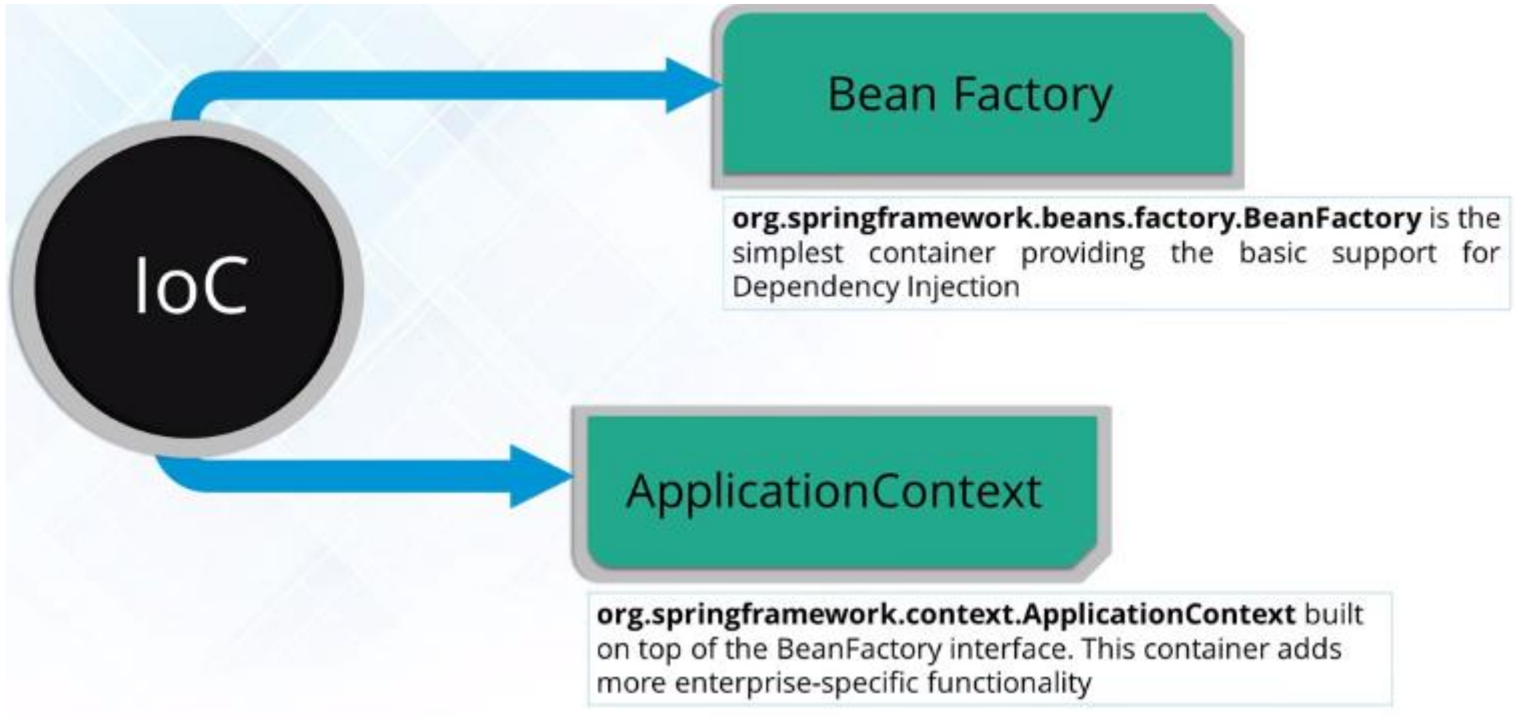
What is the Spring Container?

- The `org.springframework.beans` and `org.springframework.context` pa -ckages are the basis for Spring Framework's IoC container.
- Spring framework provides two distinct types of containers:
  - ✓ **BeanFactory** container
  - ✓ **ApplicationContext** container

One main difference between *BeanFactory* and *ApplicationContext* is that *BeanFactory* only instantiates bean when we call *getBean()* method while *ApplicationContext* instantiates singleton bean when the container is started, It doesn't wait for *getBean()* method to be called.

BeanFactory is the root interface of Spring IOC container. *ApplicationContext* is the child interface of *BeanFactory* interface that provides Spring AOP features, i18n etc.

Types Of IoC Container



org.springframework.beans.factory.BeanFactory is the simplest container providing the basic support for Dependency Injection

org.springframework.context.ApplicationContext built on top of the BeanFactory interface. This container adds more enterprise-specific functionality

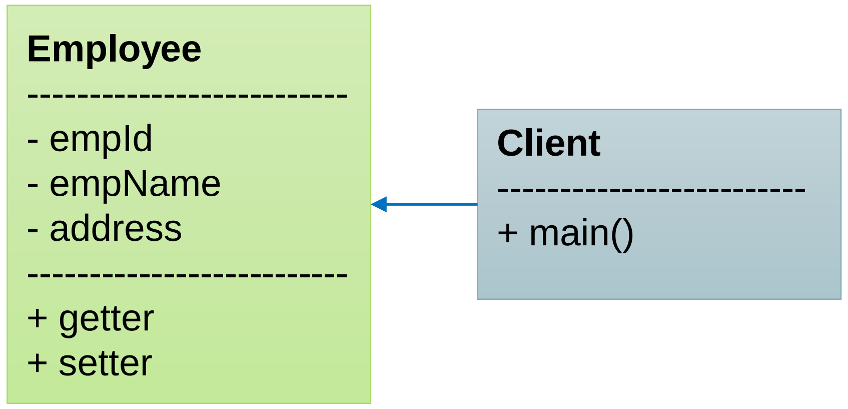
What is Spring Inversion of Control (IoC)?

- 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
}
```

Spring IoC

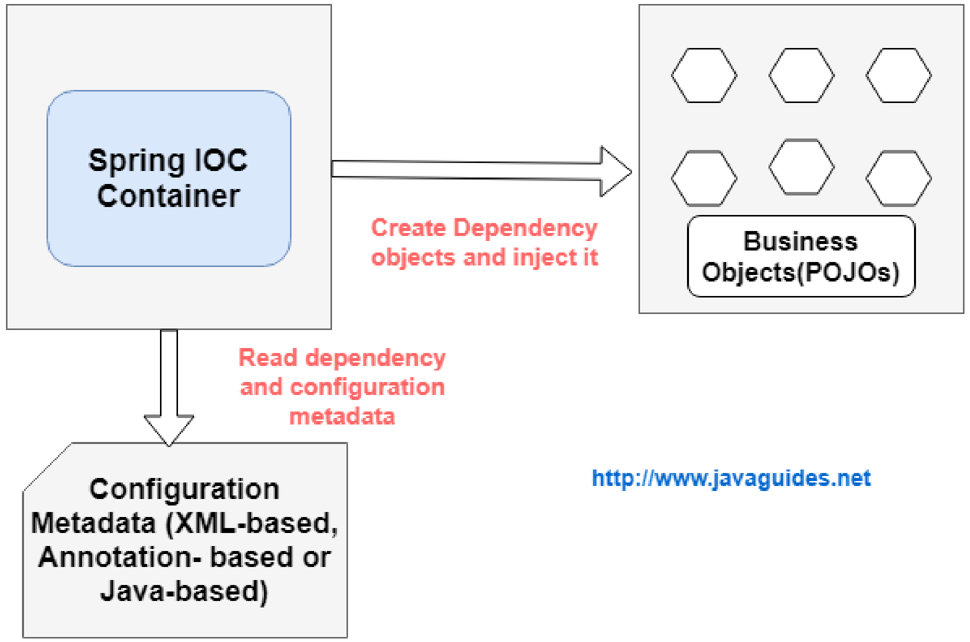
- **Standard code that without IoC**

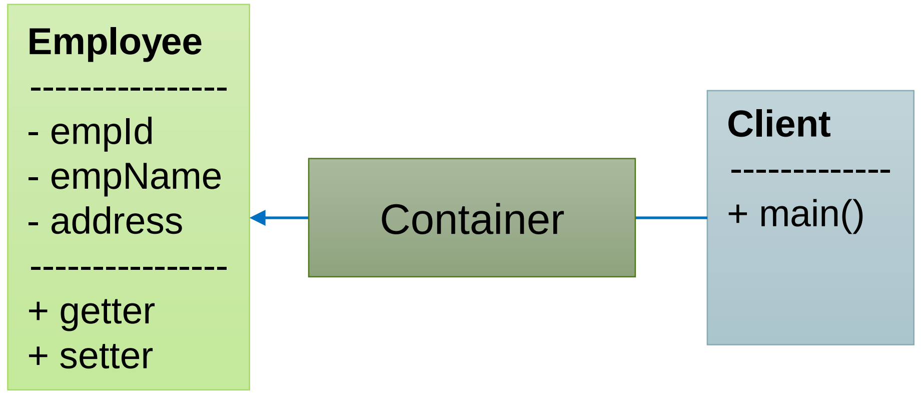


```
package com.fsoft.bean;
public class Client {
    public static void main(String[] args) {
        Employee employee = new Employee();
        employee.setEmpId(1);
        employee.setEmpName("John Watson");
        employee.setAddress("New York");
        System.out.println("Employee details: " + 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 Container
Create Dependency objects and inject it
Business Objects(POJOs)
Read dependency and configuration metadata
Configuration Metadata (XML-based, Annotation- based or Java-based)
http://www.javaguides.net



What is Configuration Metadata?

- Spring IoC container consumes a form of configuration metadata.
- **Three ways** we can supply Configuration Metadata to Spring IoC container
  - ✓ XML-based configuration
  - ✓ Annotation-based configuration
  - ✓ Java-based configuration

How to Create a Spring Container?

Spring provides many **ApplicationContext** interface implementations that we use are:

- *AnnotationConfigApplicationContext*: using Spring in standalone Java applications and using annotations for **Configuration**.

```
AnnotationConfigApplicationContext context = new
        AnnotationConfigApplicationContext(AppConfig.class);
```

Note that we are supplying configuration metadata via applicationContext.xml file(XML-based configuration).

How to Create a Spring Container?

- *ClassPathXmlApplicationContext*: spring bean configuration XML file in a standalone application.

```
ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
```

Note that we are supplying configuration metadata via AppConfig.class file.

- *FileSystemXmlApplicationContext*: This is similar to *ClassPathXmlApplicationContext* except that the XML configuration file can be loaded from anywhere in the file system.

AnnotationConfigWebApplicationContext and *XmlWebApplicationContext* for web applications.

How to Retrieve Bean from Spring Container?

ApplicationContext getBean() Example:

```
ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
```

BeanFactory getBean() Example:

```
XmlBeanFactory factory = new XmlBeanFactory (new ClassPathResource("beans.xml"));
HelloWorld obj = (HelloWorld) factory.getBean("helloWorld");
```

Spring IOC Container XML Config Example

1. Create a simple Maven Project
2. Add Maven Dependencies
3. Configure HelloWorld Spring Beans
4. Create a Spring Container
5. Retrieve Beans from Spring Container

Spring IOC Container XML Config Example

Tools and technologies used

- Spring Framework - 6.x
- JDK - 17 or later
- Maven - 3.2+
- IDE - Eclipse/STS

Spring IOC Container XML Config Example

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

```xml
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
    <spring.version>6.0.10</spring.version>
</properties>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-core -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>${spring.version}</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${spring.version}</version>
</dependency>
```

Spring IOC Container XML Config Example

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

```xml
<!-- https://mvnrepository.com/artifact/org.springframework/spring-beans -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-beans</artifactId>
    <version>${spring.version}</version>
</dependency>
```

Spring IOC Container XML Config Example

- Create Bean Configuration File

```java
package fa.training.entities;

public class Book {

    private int bookId;

    private String title;

    private int year;

    private String version;

    // getter, setter and constructor methods
}
```

Spring IOC Container XML Config Example

- **Create Bean Configuration File**

context.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"
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 id="book" class="fa.training.entities.Book">
        <constructor-arg name="bookId" type="int" value="1" />
        <constructor-arg name="title" value="Java SE Programming Language" />
        <constructor-arg name="year" type="int" value="2023" />
        <constructor-arg name="version" value="2" />
    </bean>

</beans>
```

Spring IoC Demo

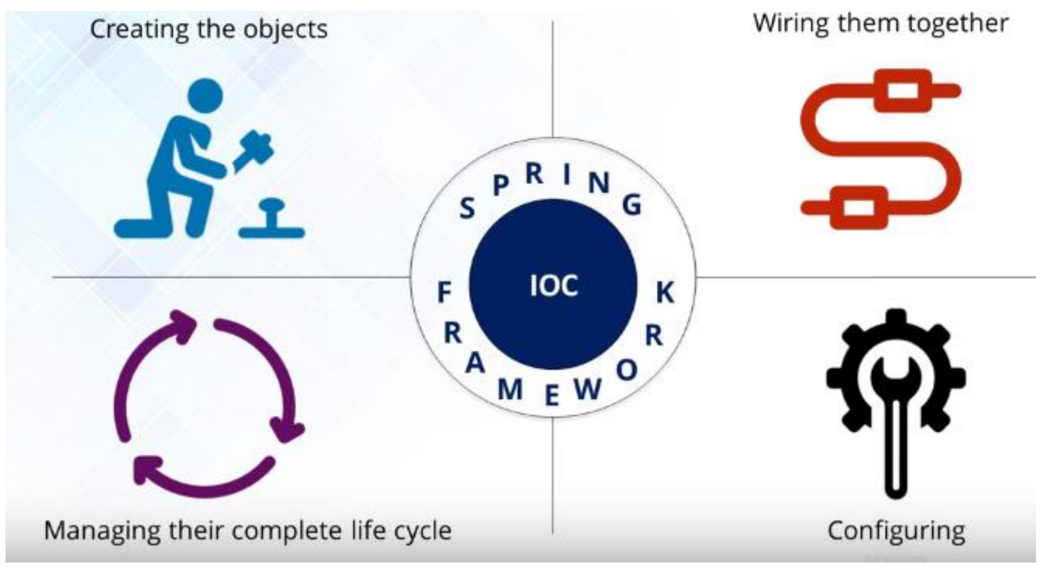
▪ **Create a Spring Container**

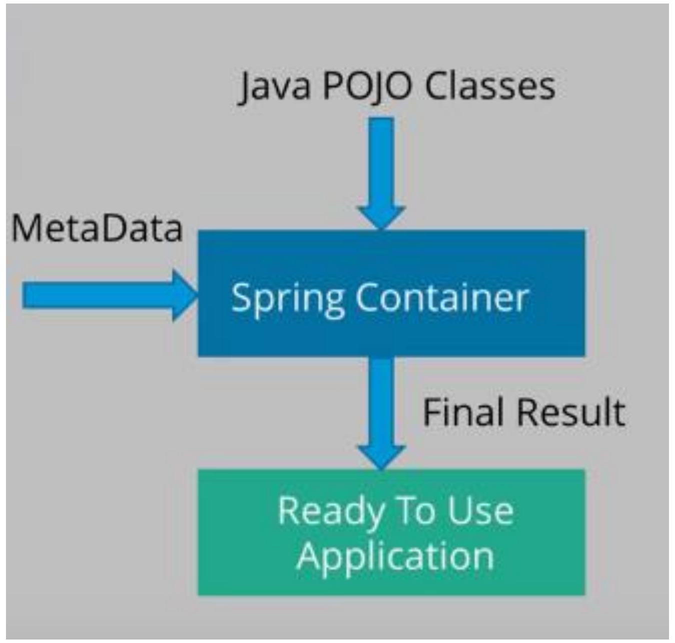
```
public class Client {
public static void main(String[] args) {

    ApplicationContext context = new
                ClassPathXmlApplicationContext("context.xml");
        Book book = (Book) context.getBean("book");

        AppLogUtils.getLog().info(book);

        }
}
```

Result

```
[INFO ] 2023-08-23 11:18:35 [main] Main 20 - Book(bookId=1, title=Java SE Programming Language, year=2023, version=2)
```

IOC Container Features





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

Section 3

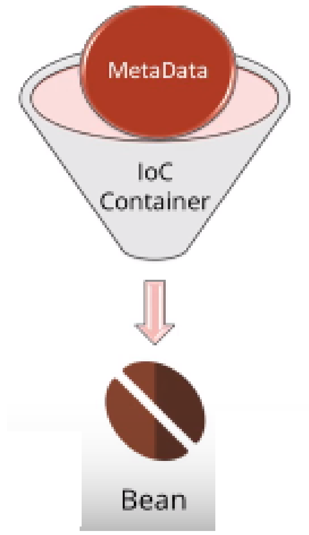
Spring Beans

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.
property	Define properties of class.
autowire	Set autowire for bean.
lazy-init	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
}
```

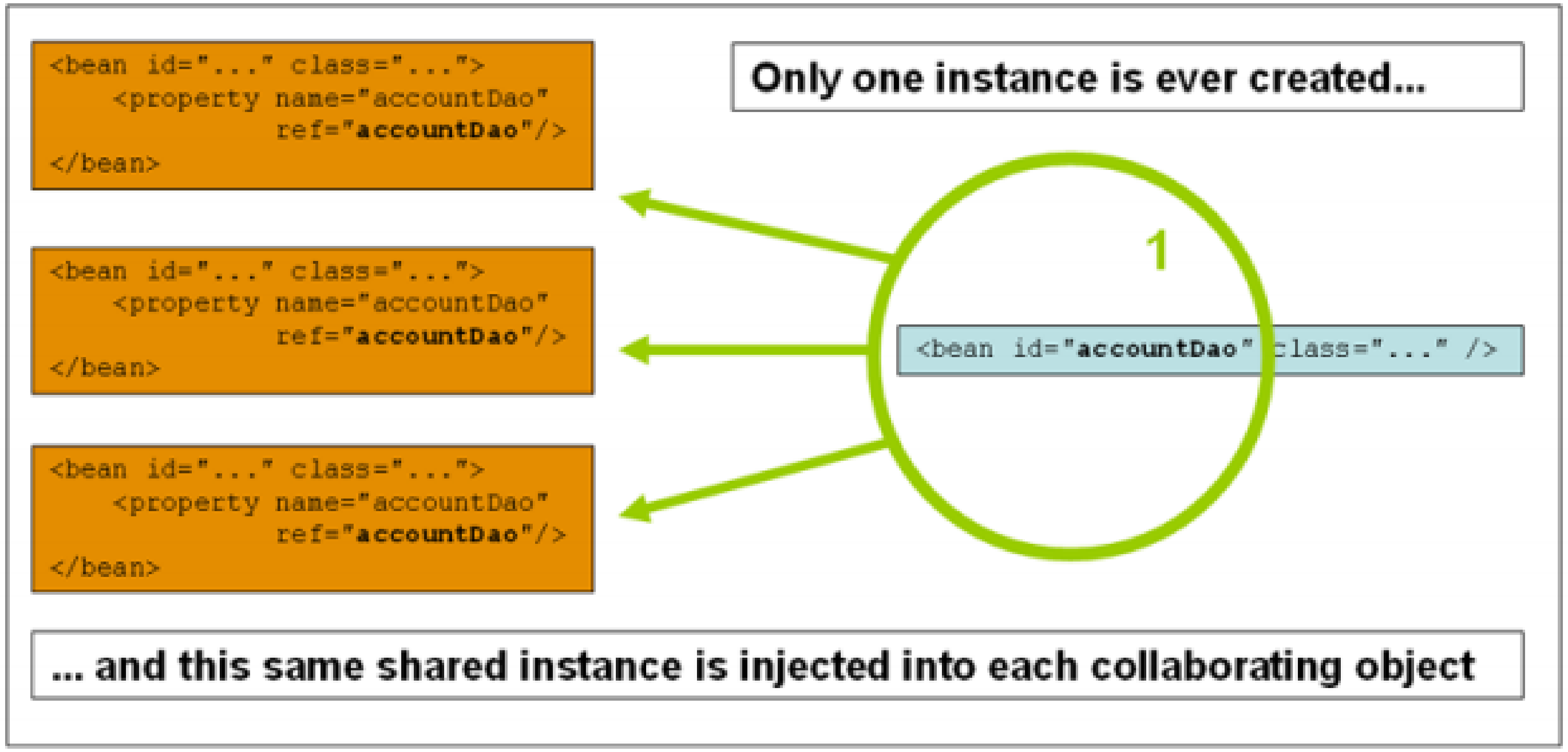
```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
	http://www.springframework.org/schema/beans/spring-beans.xsd">

    <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.
application	The *application* scope creates the bean instance for the lifecycle of a *ServletContext.*
websocket	The same instance of the bean is then returned whenever that bean is accessed during the entire *WebSocket* session.

scope property

- Scope "singleton"



Only one instance is ever created...

```
<bean id="..." class="...">
    <property name="accountDao"
              ref="accountDao"/>
</bean>
```

```
<bean id="..." class="...">
    <property name="accountDao"
              ref="accountDao"/>
</bean>
```

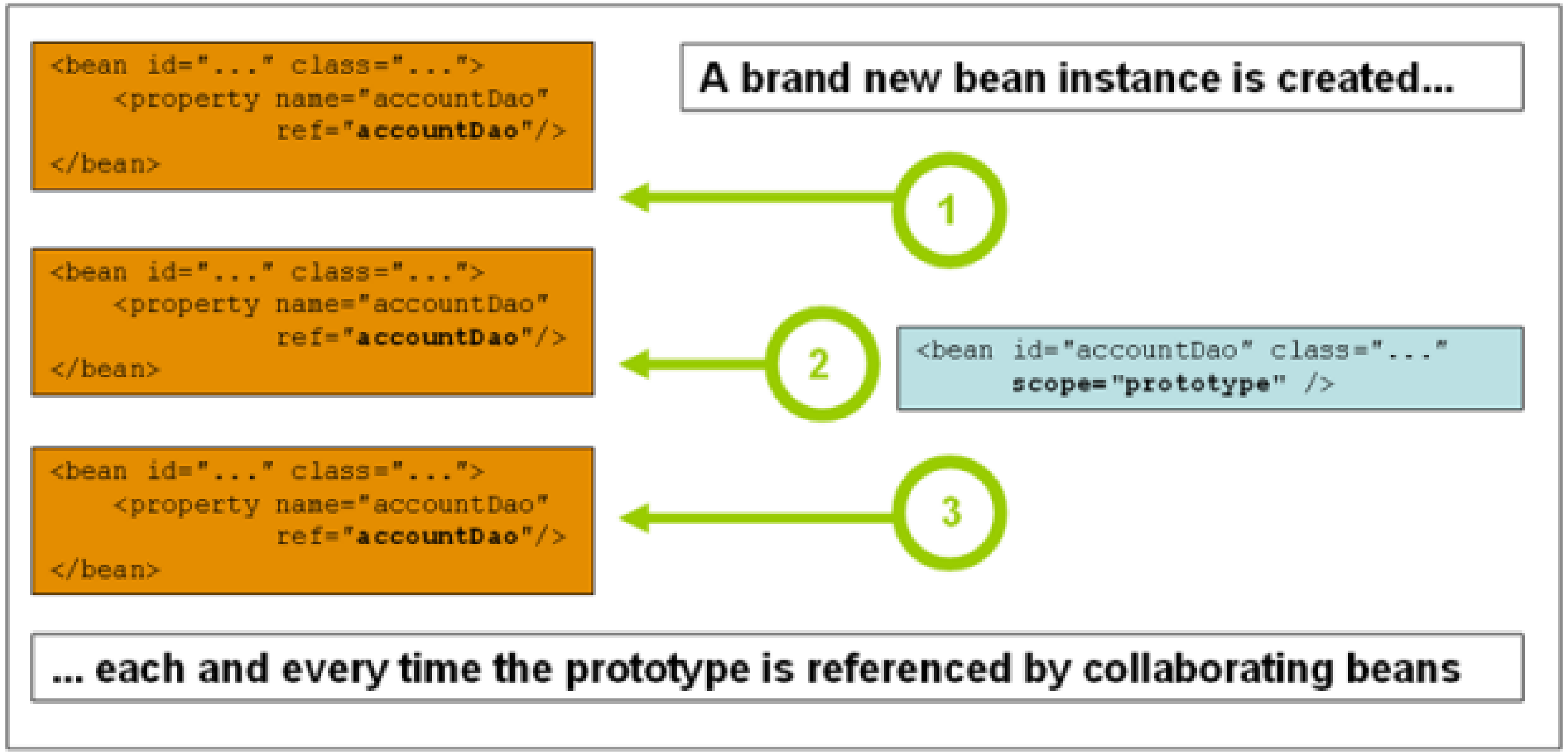
```
<bean id="..." class="...">
    <property name="accountDao"
              ref="accountDao"/>
</bean>
```

```
<bean id="accountDao" class="..." />
```

... and this same shared instance is injected into each collaborating object

scope property

- Scope “prototype”



```
<bean id="..." class="...">
    <property name="accountDao"
              ref="accountDao"/>
</bean>
```

```
<bean id="..." class="...">
    <property name="accountDao"
              ref="accountDao"/>
</bean>
```

```
<bean id="..." class="...">
    <property name="accountDao"
              ref="accountDao"/>
</bean>
```

```
<bean id="accountDao" class="..."
      scope="prototype" />
```

A brand new bean instance is created...
1
2
3
... each and every time the prototype is referenced by collaborating beans

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:

```java
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();
    }
}
```

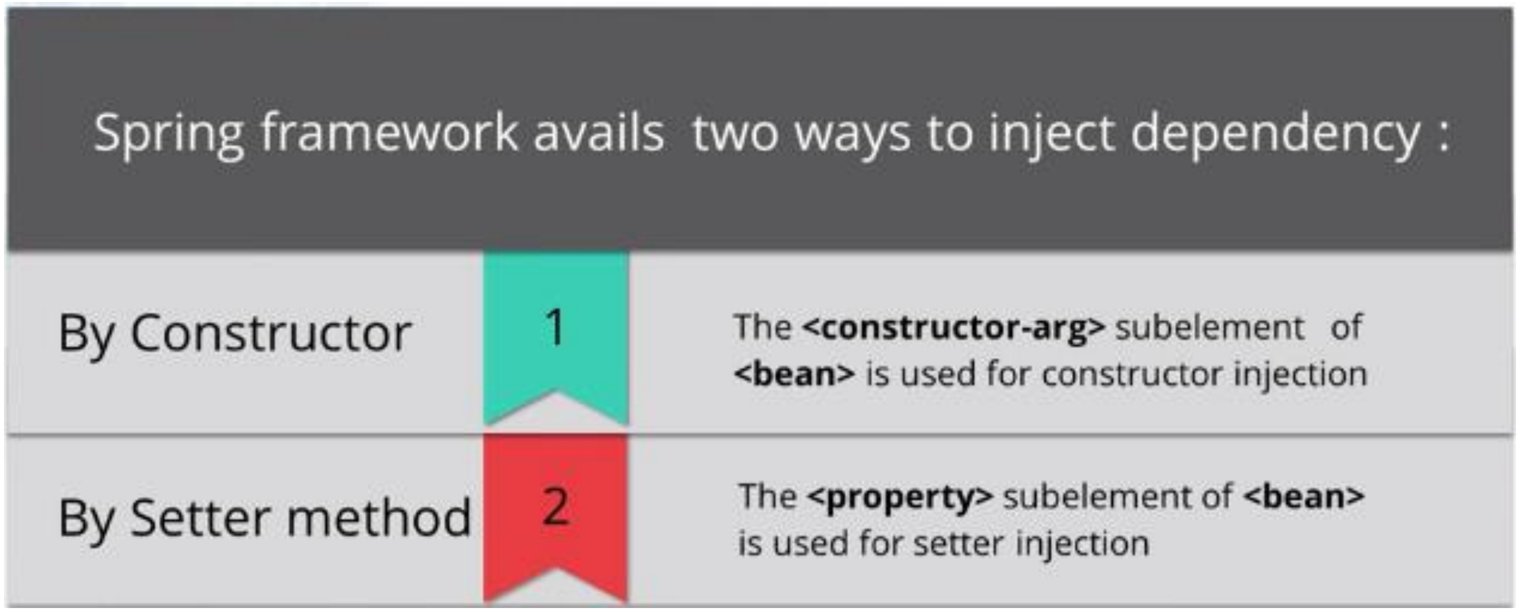
Spring DI

- 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

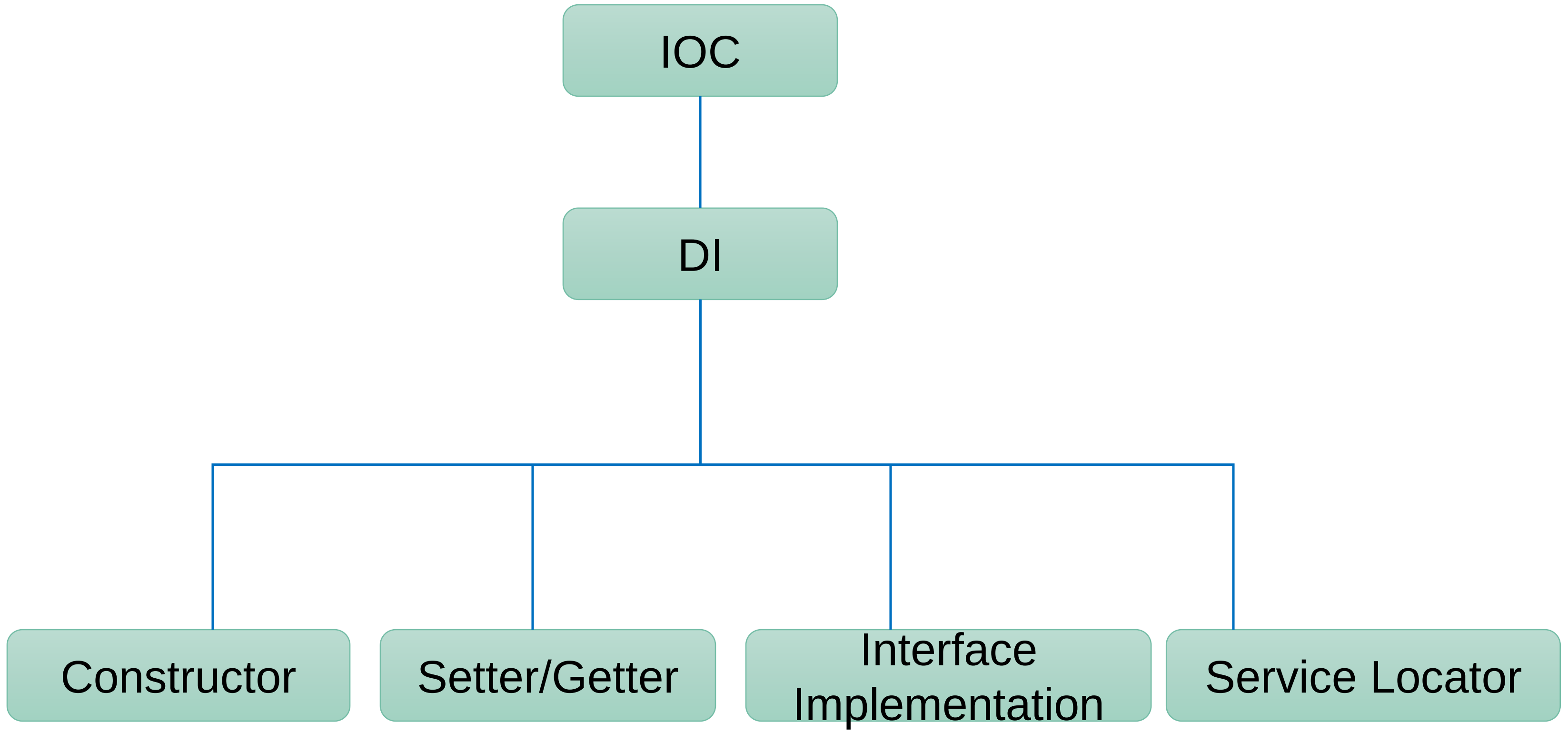
- **By Setter**

```
<?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

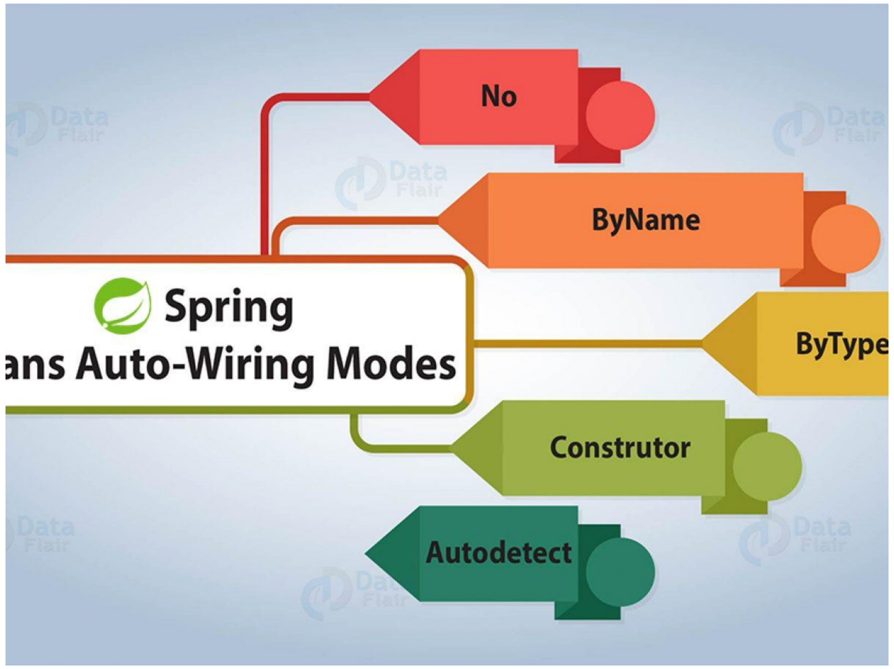


Section 5

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 default. The default mode is no.



Example classes

```
public class Department {
    private String deptName;

    public String getDeptName() {
        return deptName;
    }

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

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

    // getter, setter and constructor methods

    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**.

```xml
..
<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;
   }
}
```

Summary

- Overview of the Spring Framework
- Spring IoC
- Spring Bean
- Dependency Injection
- Autowiring in Spring
- Questions and Answers

THANK YOU!