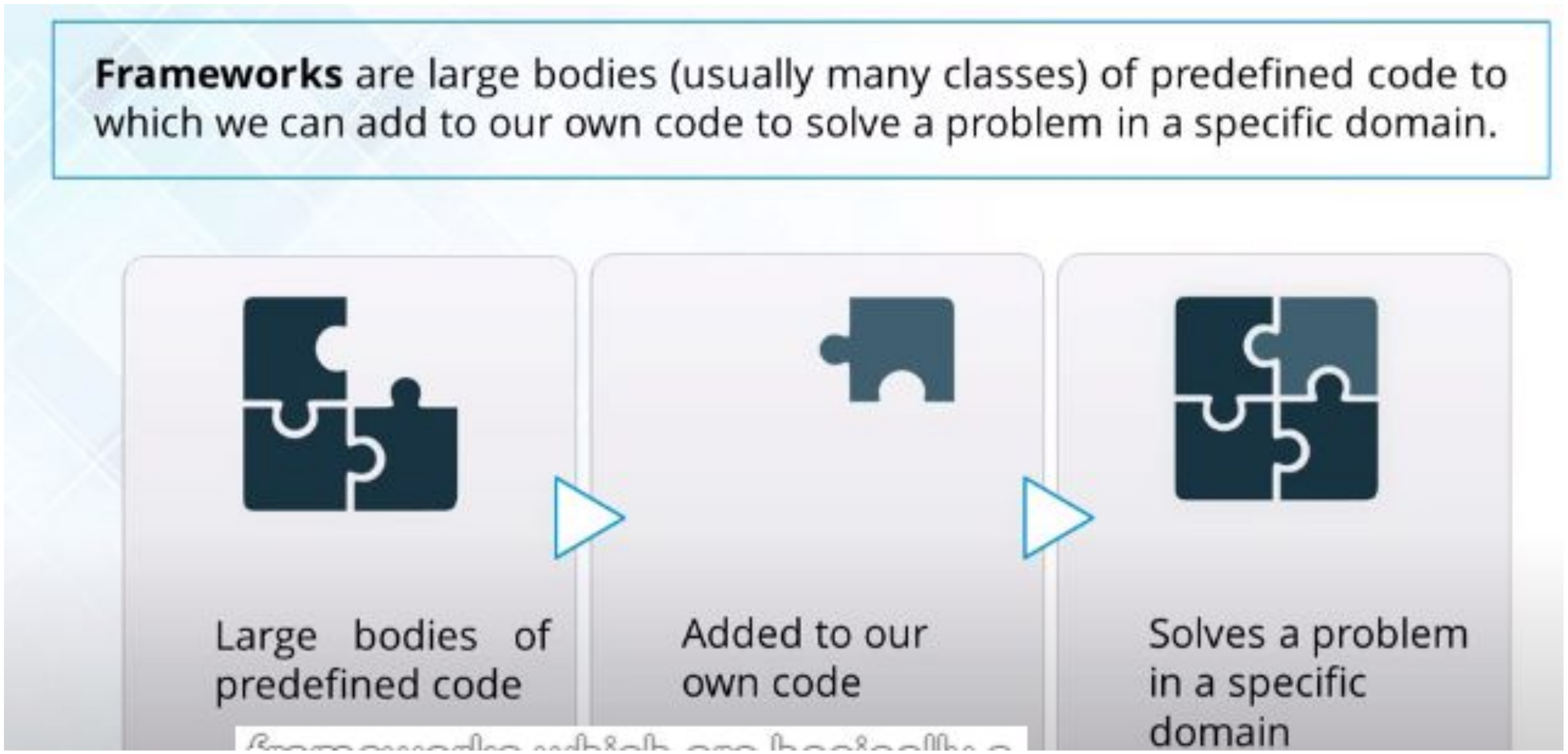
Spring Frameworks

What is Java Framework?


Frameworks are large bodies (usually many classes) of predefined code to which we can add to our own code to solve a problem in a specific domain.

Large bodies of predefined code

Added to our own code

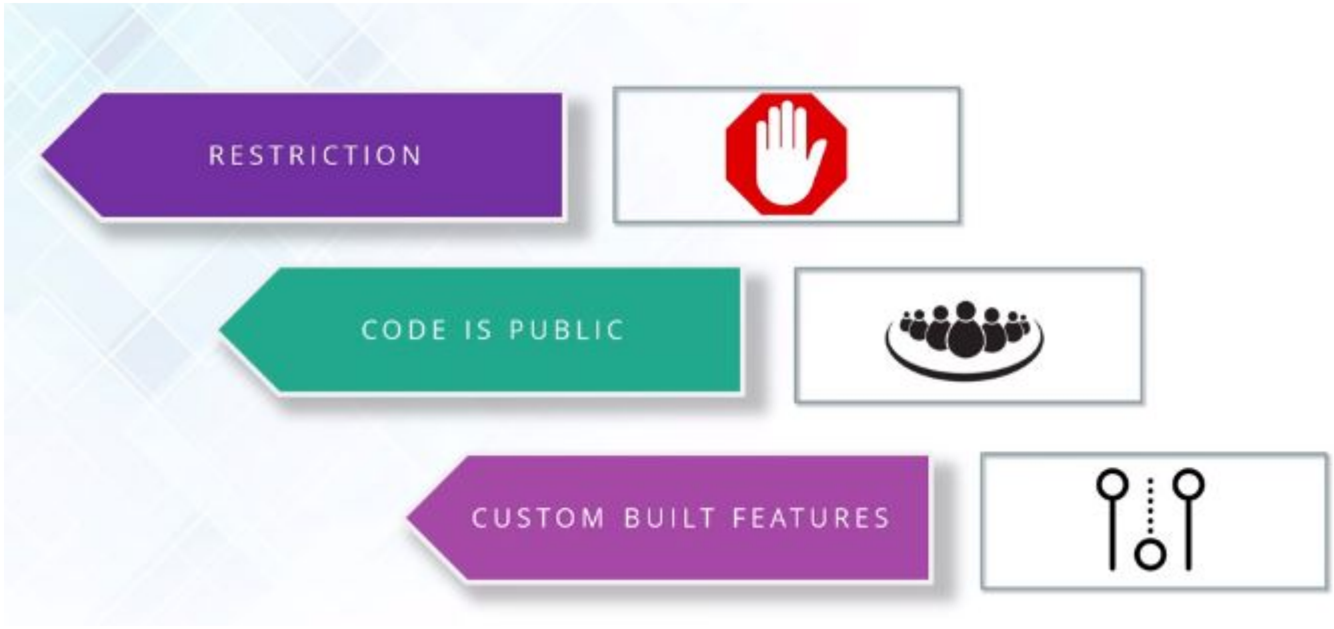
Solves a problem in a specific domain

Advantages of Java Framework

EFFICIENCY
SECURITY
EXPENSE
SUPPORT

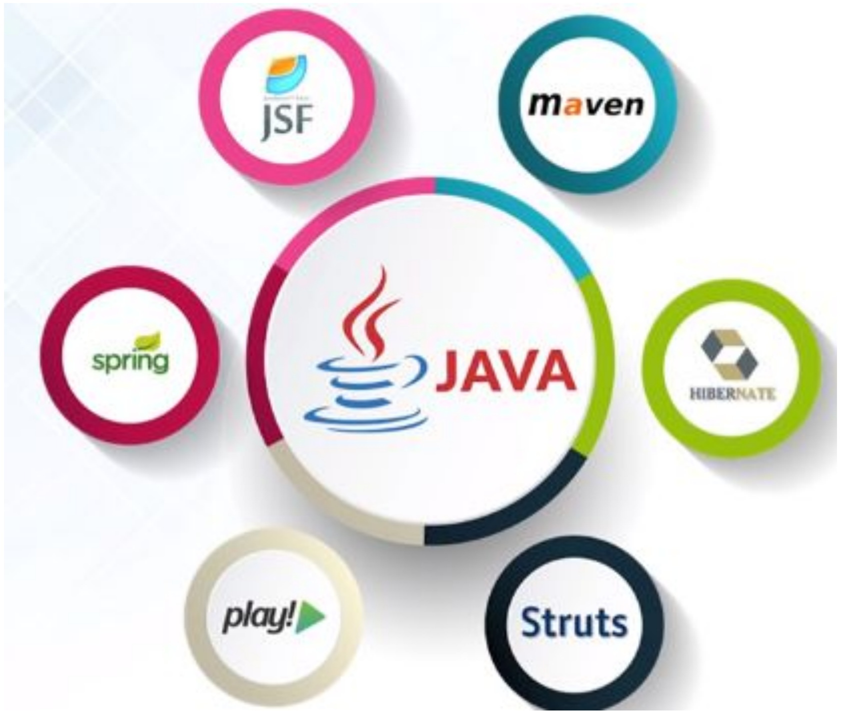
- All above features are available when we use the API

Disadvantages



- Need the use the API the way it is, Cannot modify the API
- Code is public: Same code is exposed to the public
- Internals of the API is not exposed to the developer (specially web service)

Java Framework



Spring

Team...We got one request from the client to develop an application but the challenge is that we have got very little time frame to complete this Project
Harry

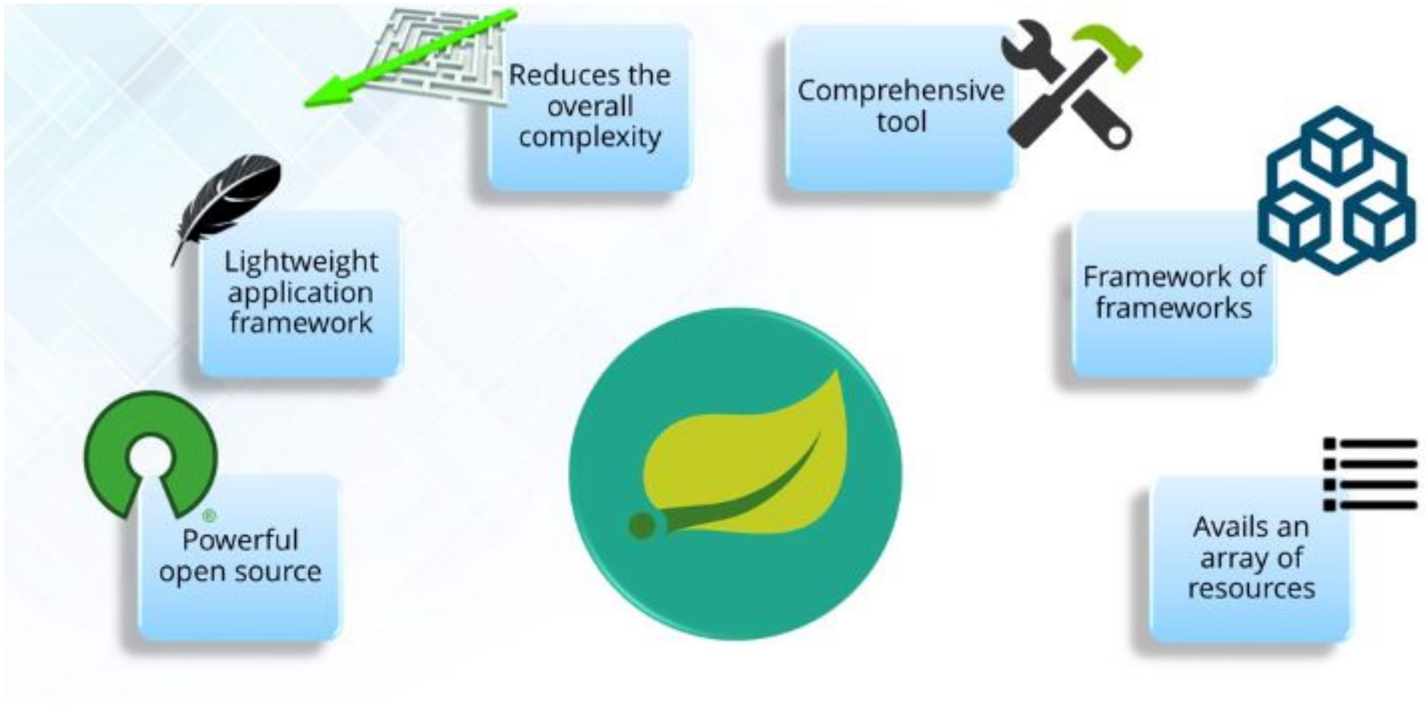
- Need Something Fast, flexible and efficient

Why Spring Framework?

- Spring Framework makes the easy development of J2EE Applications

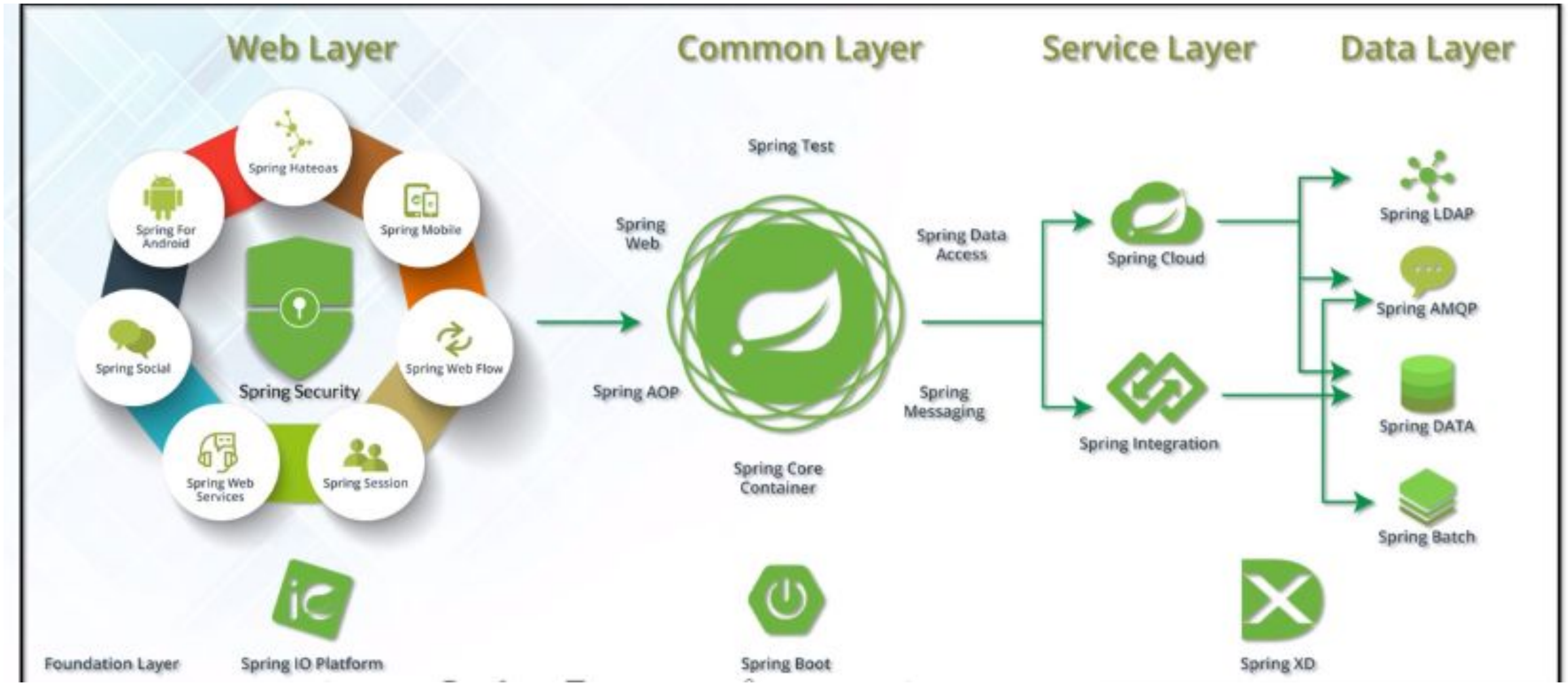
- It provides more flexibility as Spring MVC is based on interfaces, So someone can make required changes in the implementation without effecting the client code.
- Spring framework is based on interfaces so we can write our own code
- Reduce the development time as APIs will be used to develop the application

Introduction to Spring Framework


Reduces the overall complexity
Comprehensive tool
Lightweight application framework
Framework of frameworks
Powerful open source
Avails an array of resources

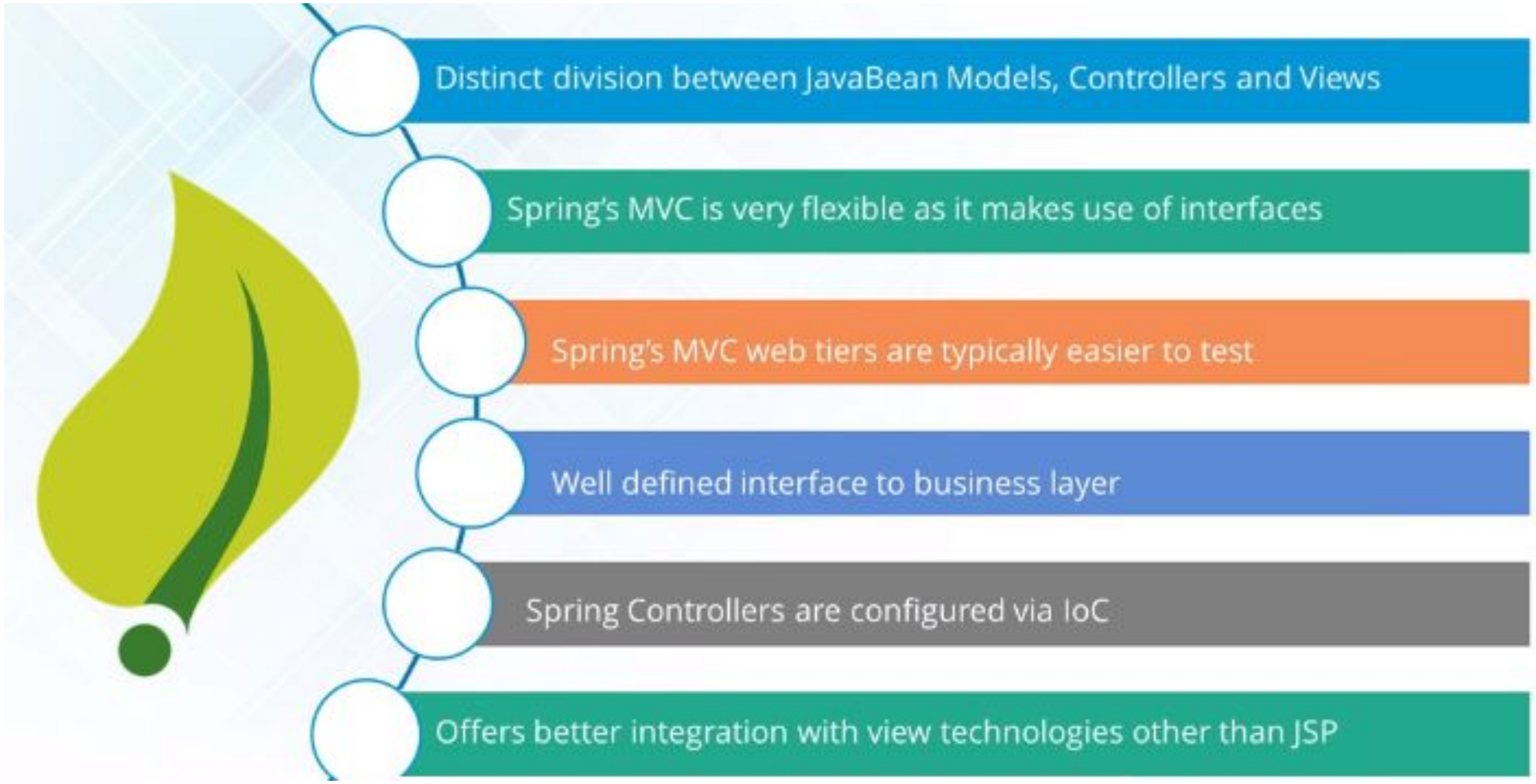
Introduction to Spring Framework

- Spring Framework ecosystem

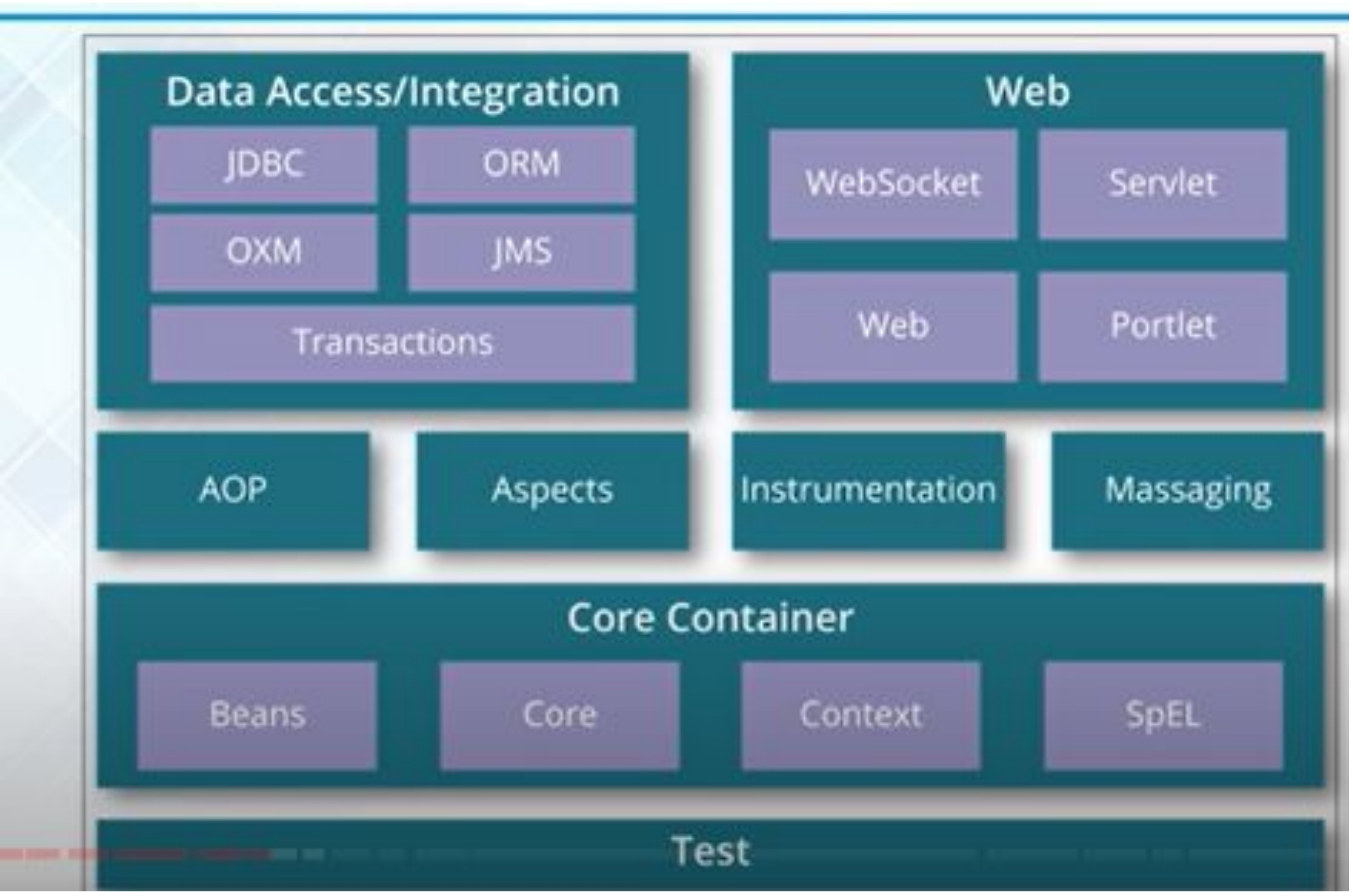


Introduction to Spring Framework

- Why Spring?


Distinct division between JavaBean Models, Controllers and Views
Spring's MVC is very flexible as it makes use of interfaces
Spring's MVC web tiers are typically easier to test
Well defined interface to business layer
Spring Controllers are configured via IoC
Offers better integration with view technologies other than JSP

Spring Architecture

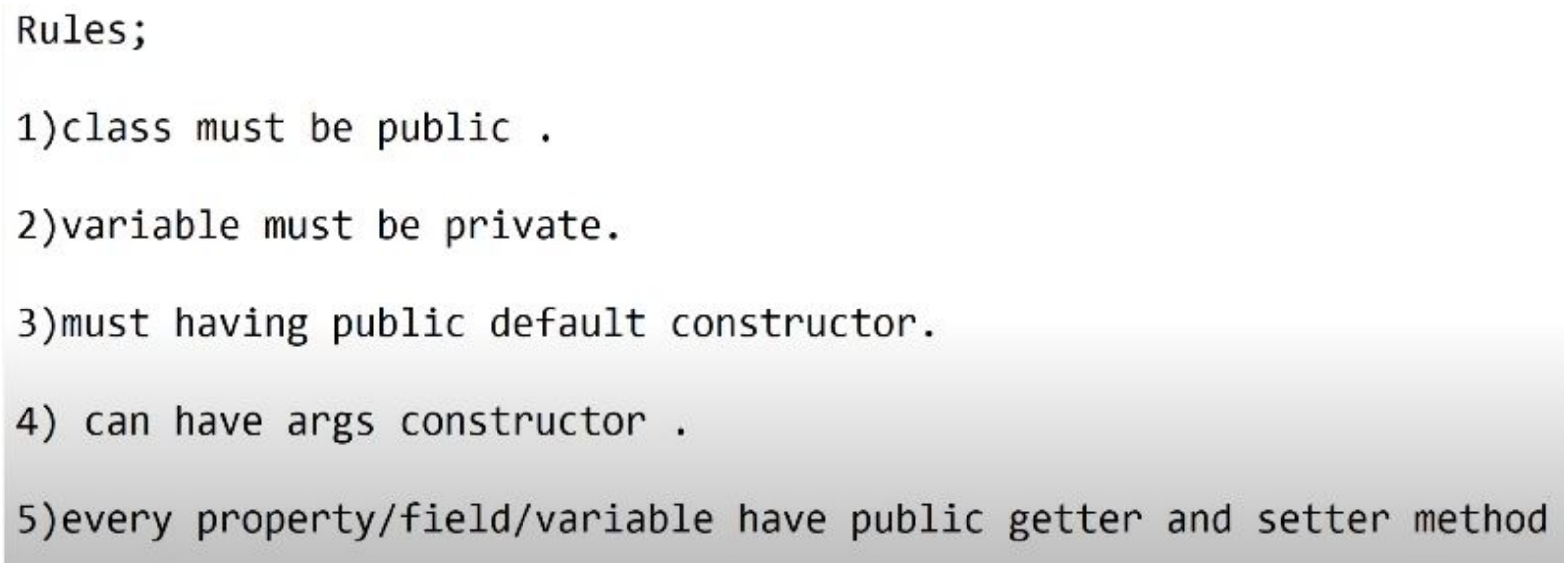


POJO Programming Model

- POJO: Plain Old Java Objects
  - It is a Java object that doesn't extend or implement some specialized classes and interfaces respectively.
  - all normal Java objects are POJO
  - In POJO model, it is recommended that, interfaces should be explicitly implemented whenever you want to pick and choose the methods of the interface

POJO Programming Model

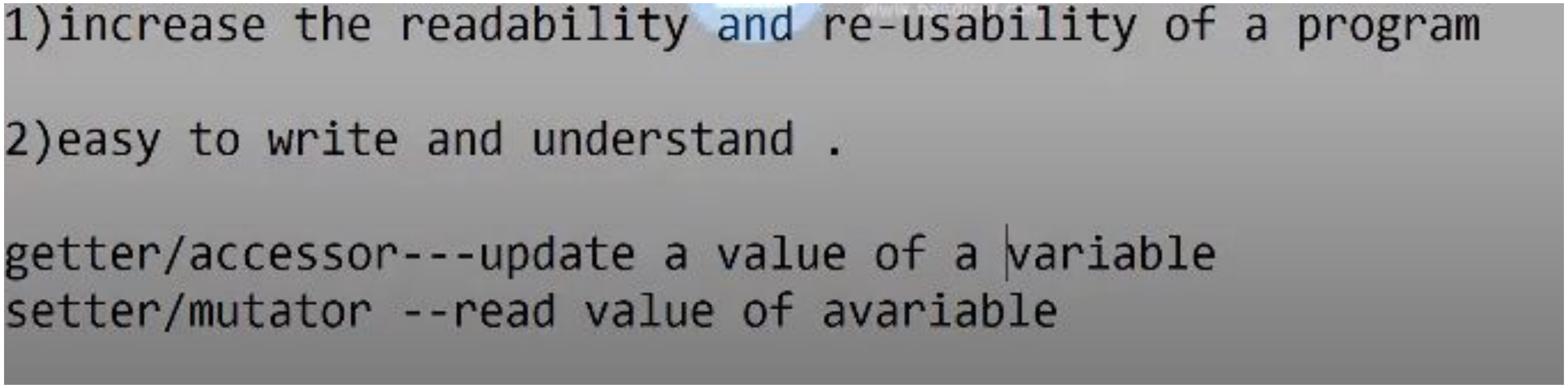
- POJO Classes Rules



```
Rules;

1)class must be public .

2)variable must be private.

3)must having public default constructor.

4) can have args constructor .

5)every property/field/variable have public getter and setter method
```

POJO Programming Model

- Advantages of POJO Classes



```
1)increase the readability and re-usability of a program

2)easy to write and understand .

getter/accessor---update a value of a variable
setter/mutator --read value of avariable
```

POJO Programming Model

- Example

```
public class Employee
{
String name;
public String id;
private double salary;

public Employee()
{}

public Employee(String name, String id,
double salary)
{
this.name = name;
this.id = id;
this.salary = salary;
}
Public void setName(String name)
{ this.name=name;
```

```
public String getName()
{
return name;
}
public String getId()
{
return id;
}
public Double getSalary()
{
return salary;
}
}
```

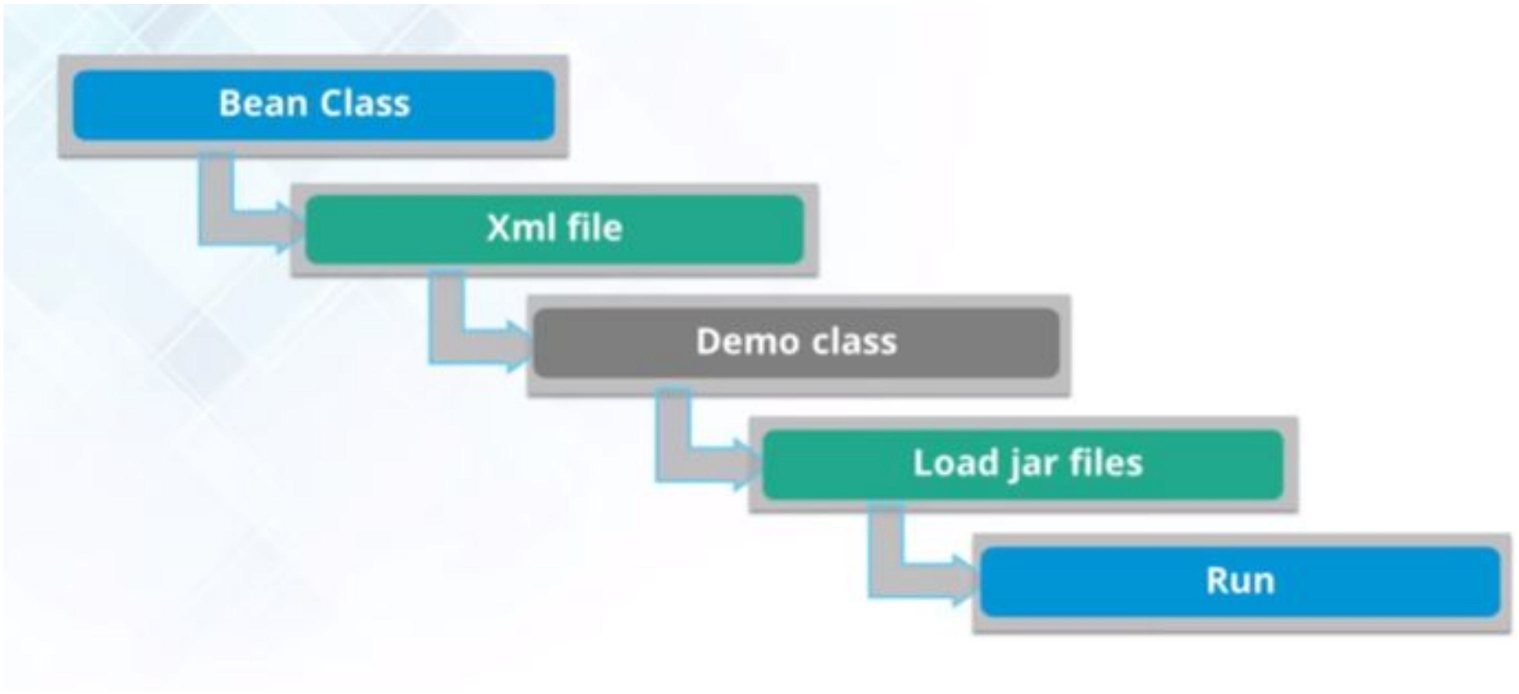
POJO Programming Model

- Example

```
public class Myprogram
{
Employee s1= new Employee(“Ram”, 1, 28);
 System.out.println(“Name:”+s1.getName());
System.out.println(“Name:”+s1.getId());
System.out.println(“Name:”+s1.getSalary());
Employees2= new Employee();
s2.setName(“Geeta”);
System.out.println(“Name:”+s2.getName());

}
```

Introduction to Spring Framework

- Five steps Spring Coding



Lightweight Containers

- Spring IOC container
- How Class is constructed?
  - Attribute
  - methods
- Object Construction
  - Classname refname=new Classname()
  - Refname.arrtibute1= value
  - Refname.arrtibute1= value
  - ………….
- Spring
  - Core : IOC: Inversion of Control
  - No need to create objects
  - Objects shall be configured in an XML file by the developer
  - Spring container module: responsible to construct java objects by parsing XML file

Lightweight Containers

- Spring IOC container
- Benefit
- The XML file is not part of source code so it can grow and shrink
- Values can be manipulated accrodingly

Spring IOC container

- Example

```
public class Employee
{
String name;
int id;
double salary;

public Employee()
{
}

public Employee(String name, String id,
double salary)
{
this.name = name;
this.id = id;
this.salary = salary;
}

Public void setName(String name)
{ this.name=name;}

public String getName()
{return name;}

Public void setId(int  id)
{ this.id=id;}

public String getId()
{return id;}

Public void setSalary(int  salary)
{ this.salary=salary;}

public Double getSalary()
{return salary;}

}
```

Spring IOC container

- Example

```
public class Myprogram
{
public static void main(String[] args) {
//Object Construction by Developer
Employee e1= new Employee();
e1.setName(“Geeta”);
e1.setId(21);
e1.setSalary(21000.20);
 System.out.println(“EmpDetails:”+e1);

//Spring Way| IOC enable Spring Core
Framework
// Link to XML File XML\Employee.xml
```

Spring IOC container

- Example

```
// An API named Resource from Spring Framework
Resource resource= new ClassPathResource(“employee.xml”);
BeanFactory factory=new XmlBeanFactory(resource);
 //BeanFactory is Spring Core Container that shall parse the XML and
construct the objects
Employee e2=(Employee) factory.getBean(“emp1”);
Employee e3=factory.getBean(“emp2”.Employee.class);

//Objects are constructed by spring core container and we are using the
reference to the object
System.out.println(“EmpDetails:”+e2);
}
```

Spring IOC container

ApplicationContext API is implementation of Bean Package

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

public class Myprogram
 {
   public static void main(String[] args) {
      ApplicationContext context = new
ClassPathXmlApplicationContext(“Employee.xml");
      Employee e1 = (Employee) context.getBean(“emp1");
        }
}
```

ApplicationContext creates object even if it is not requested. However, BeanFactory creates object only if it is requested.

Spring IOC container

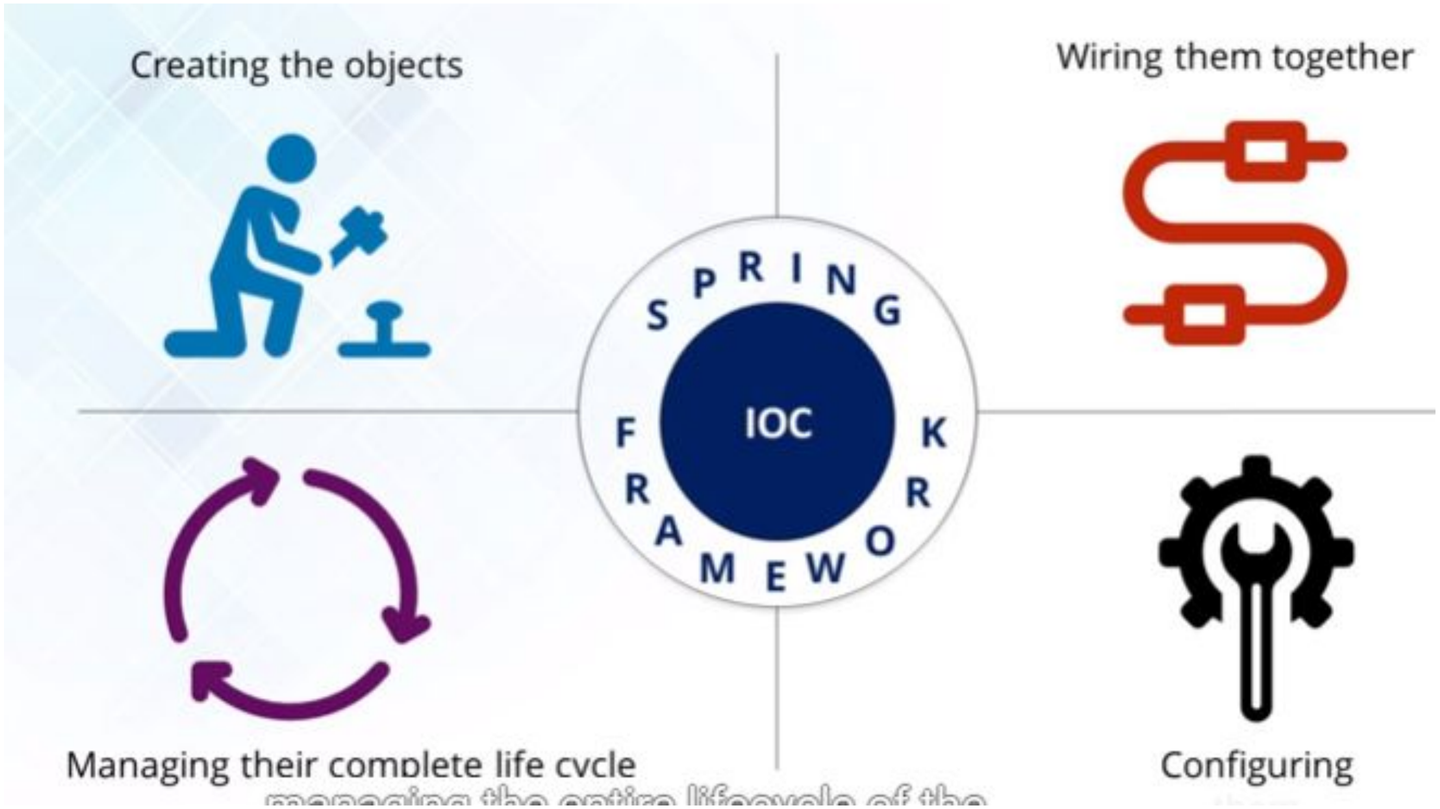
- Spring Inversion Control
  - Objects not created by developer in code
  - Object is Constructed by Spring core container
  - Two API’s
    - BeanFactory
    - ApplicationContext
    - ApplicationContext is built on top of Bean package
    - Bean factory constructs the object when its requested by calling getBean Method
    - ApplicationContext constructs the objects even though if its is not requested.

Spring IOC container

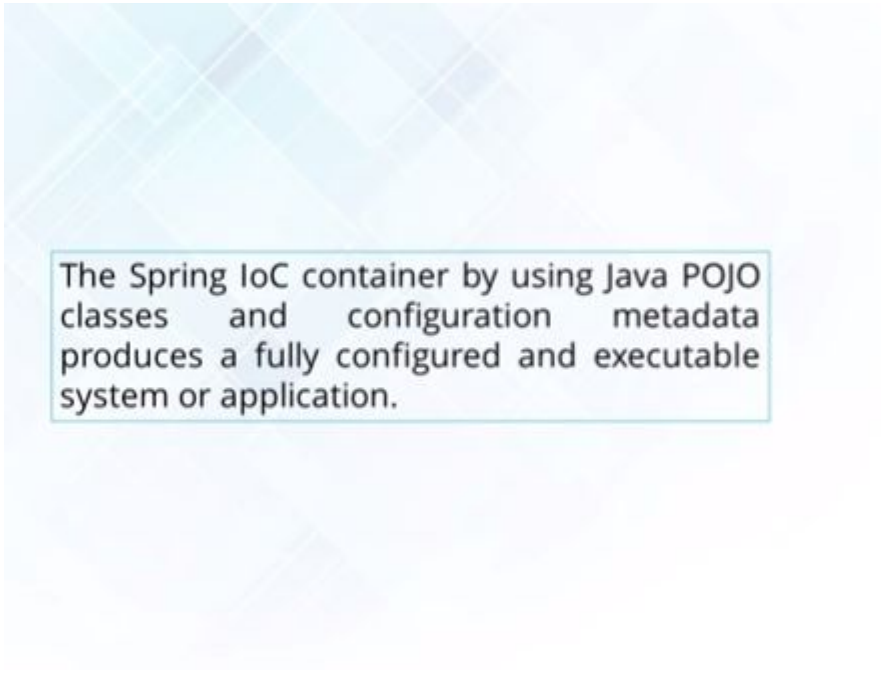
```
public class Employee
{
String name;
int id;
double salary;

public Employee()
{
System.out.println(" Objects Constructed….." );
}

public Employee(String name, String id,
double salary)
{
this.name = name;
this.id = id;
this.salary = salary;
}
```

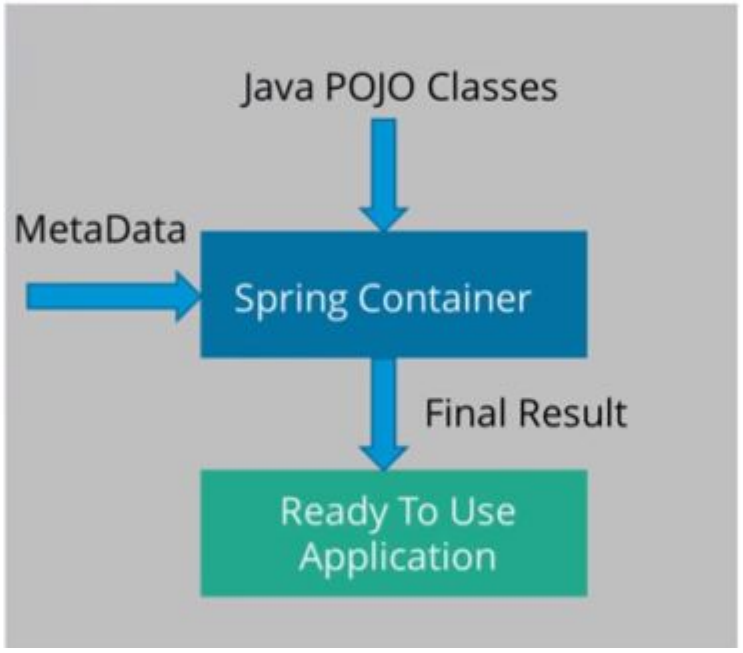
```
Public void setName(String name)
{ this.name=name;}

public String getName()
{return name;}

Public void setId(int  id)
{ this.id=id;}

public String getId()
{return id;}

Public void setSalary(int  salary)
{ this.salary=salary;}

public Double getSalary()
{return salary;}

}
```

Spring IOC container



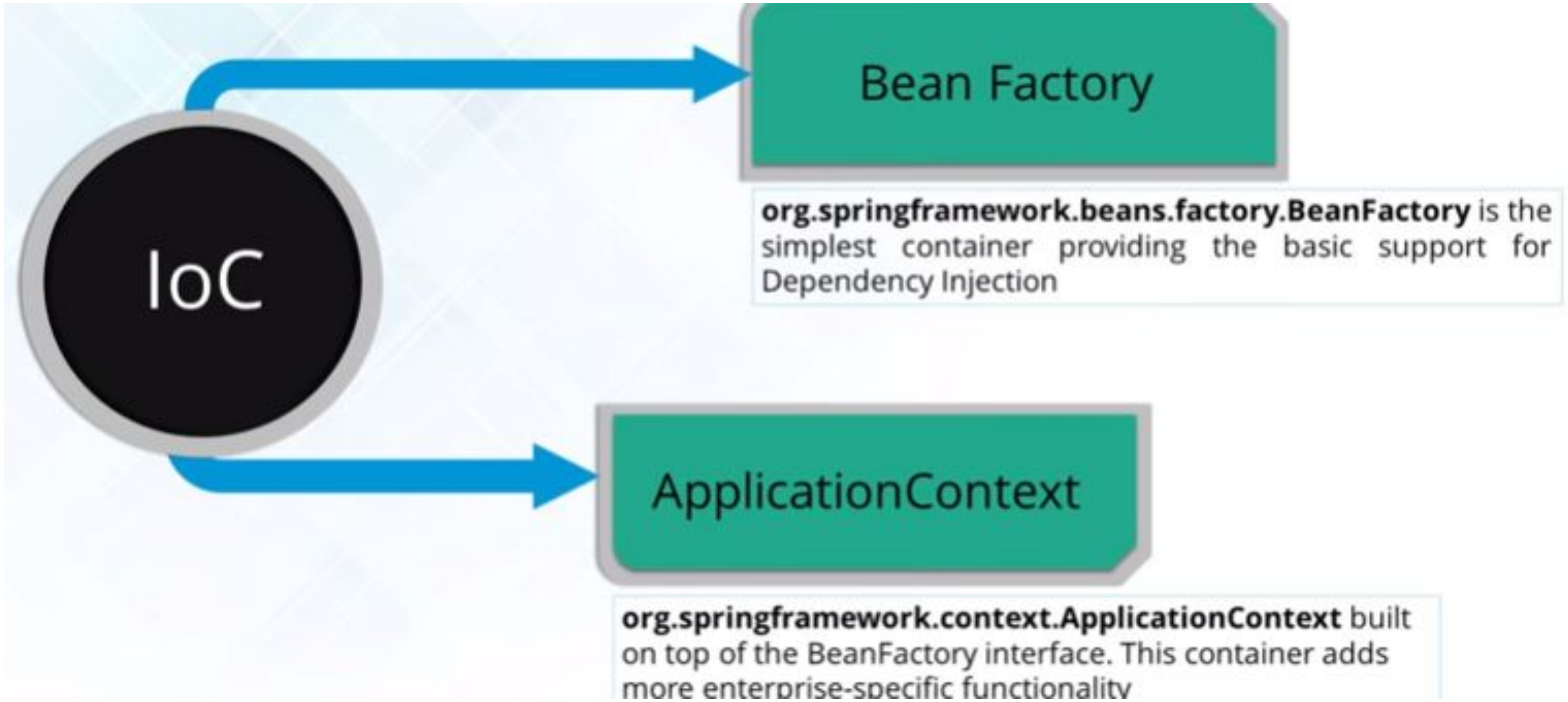
Spring IOC container features


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



- Metadata: XML e.g employee.xml
- POJO: Employee Class

Lightweight Containers

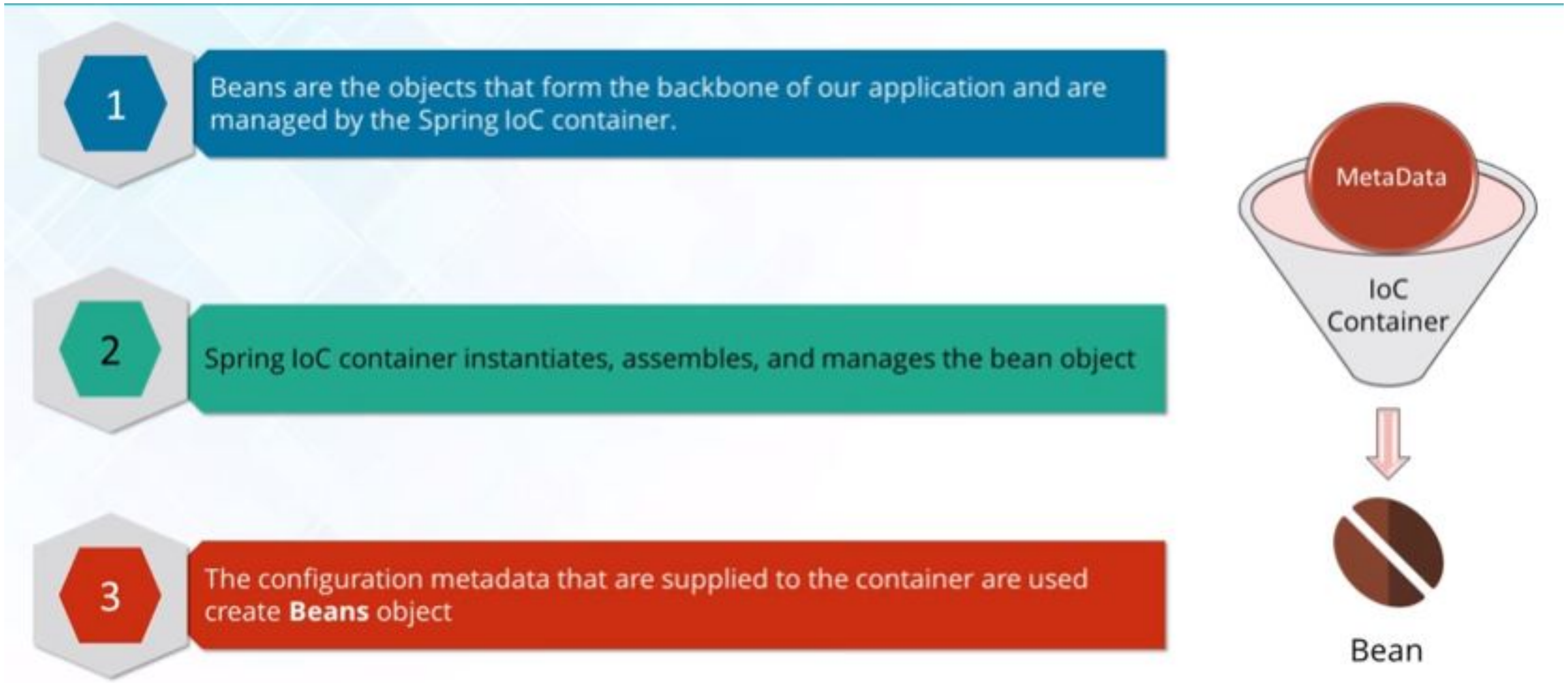


org.springframework.beans.factory.BeanFactory is the simplest container providing the basic support for Dependency Injection

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

Bean

- Bean Object

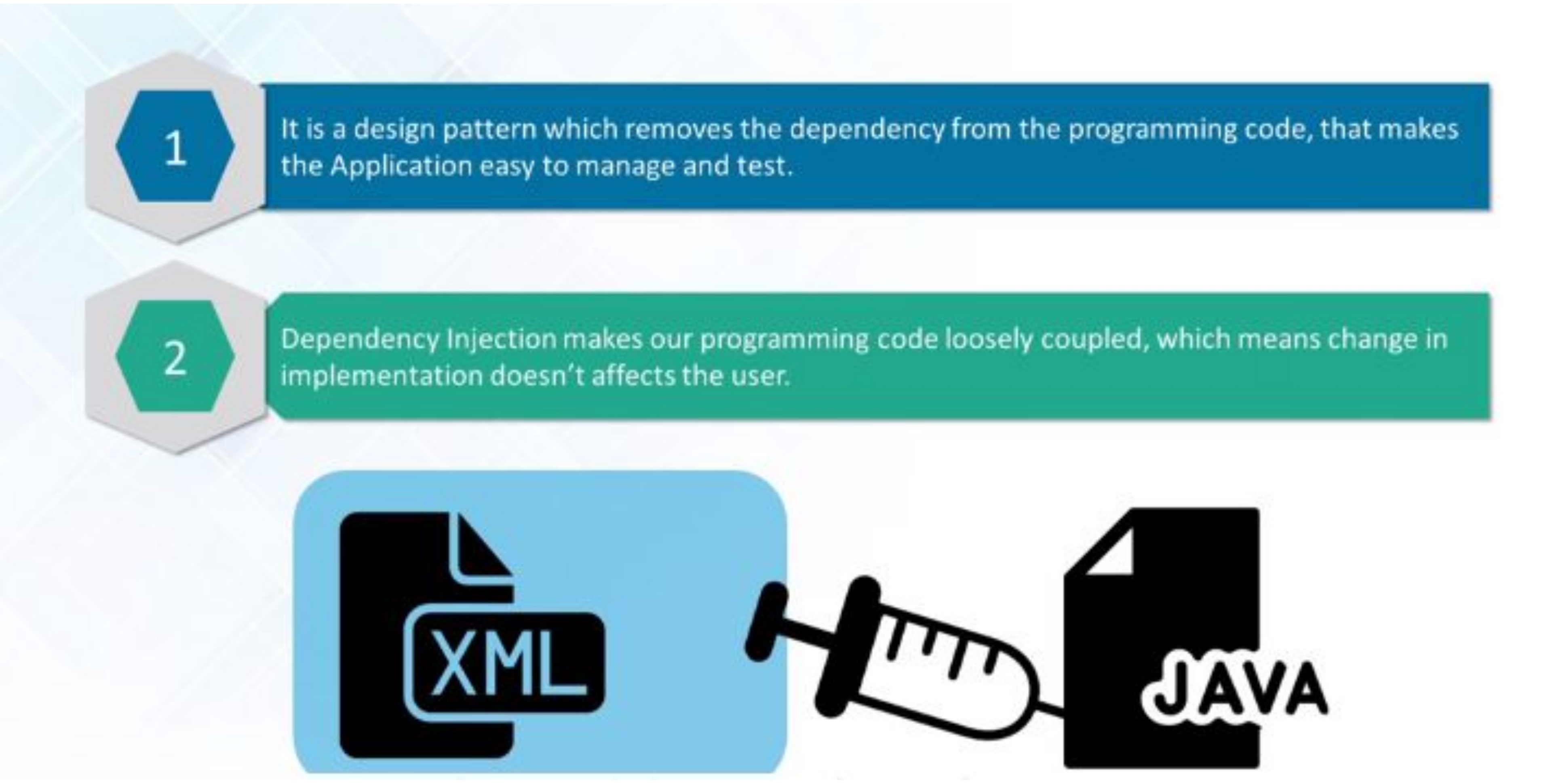


1 Beans are the objects that form the backbone of our application and are managed by the Spring IoC container.

2 Spring IoC container instantiates, assembles, and manages the bean object

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

Dependency Injection with Spring



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

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

Dependency Injection with Spring

- Constructor Injection
  - Constructor-based DI is accomplished by the container invoking a constructor with a number of arguments, each representing a dependency.

```
public class Employee
{
String name;
int id;
Address addrs
}
public Address
{
int aptno;
String city;
int pin;
}
```

Dependency Injection with Spring

- How we will create employee object?

```
Employee e=new Employee()
Constructor of employee
Employee()
{ name=‘rahul”;
Id=20;
address=new Address();
}
```

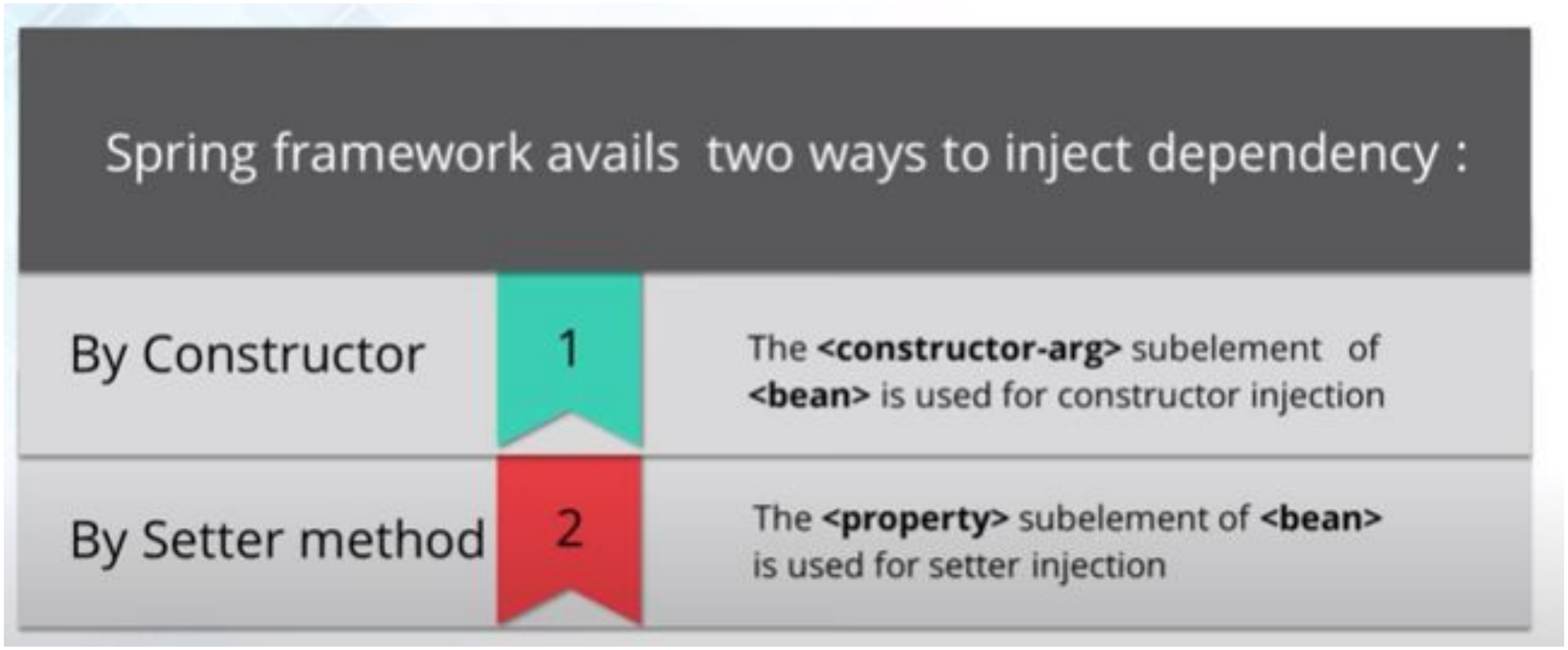
- High Dependency : any change in Address class needs to be reflected in Employee class as well
- Leads to high coupling
- If Employee class uses Address object without mentioning Address class in it | we can overcome all such problems

Dependency Injection Framework

Dependency Injection with Spring

- Never create object inside another class using new operator
- DI Create all objects with dependencies in the application and just use it
- Reduce Dependency
- Achieve loose coupling
- Less code
- More readable code
- More maintainable code

Dependency Injection with Spring



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

Constructor Injection

```
Employee( Address address){
addrs=address;
}
Address a=new Address()
Employee e =new Employee(a);
```

Reduce Dependency

Achieve loose coupling

Constructor Injection

```
Public class Address{
String city;
String state;
int pin;
Public Address(){
}
Public Address(String city, String state, int pin){
This.city=city;
This.state=tate;
This.pin=pin;
}
// getter and setter methods
}
```

```
Public class employee{
Int id;
String name;
Address address;
Public Employees(){
}
// Constructor Injection
Public Employee(Address
address){
This.address=address;
}
// getter and setter methods
}
```

Constructor Injection

```
<?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 = “add1" class = "Vesit.Address">
    <property name = “Citye" value = “Mumbaii"/>
    <property name = “state" value = “Maharashtra"/>
    <property name = “pin" value = “615702/>
  </bean>
<bean id = "emp1" class = "Vesit.Employee">
    <property name = "name" value = "Rishi"/>
    <property name = "id" value = "12"/>
    <constructor-arg ref="add1"/>
  </bean>
</beans>
```

In Employee class constructor injection will take place

Constructor Injection

```
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationConte
xt;

public class Myprogram
 {
   public static void main(String[] args) {
      ApplicationContext context = new
ClassPathXmlApplicationContext(“Employee.xml");
      Employee e1 = (Employee) context.getBean(“emp1");
         }
}
```

Setter Injection

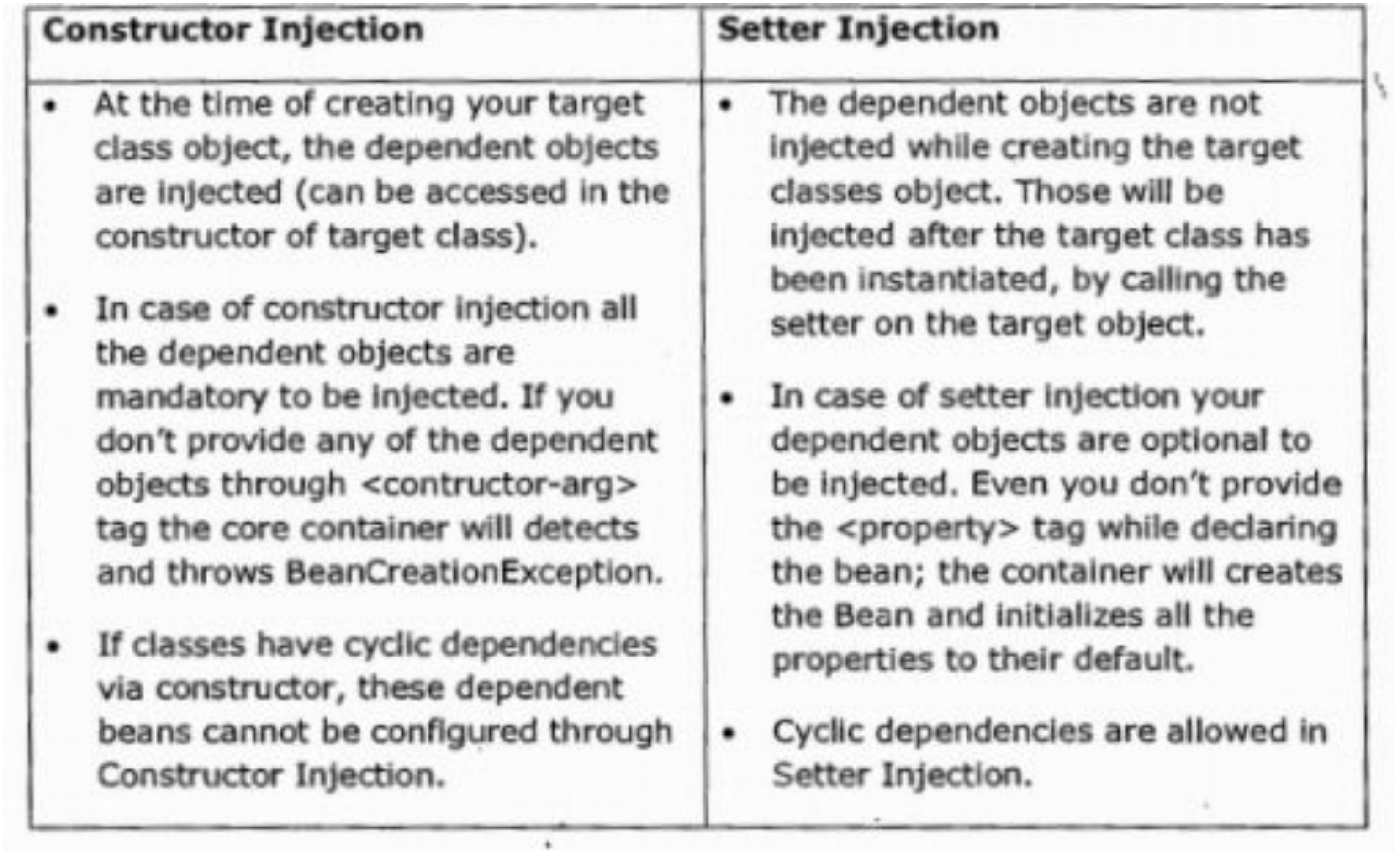
```
Public class Address{
String city;
String state;
int pin;
Public Address(){
}
Public Address(String city, String state, int pin){
This.city=city;
This.state=tate;
This.pin=pin;
}
// getter and setter methods
}
```

```
Public class employee{
Int id;
String name;
Address address;
Public Employees(){
}
// Constructor Injection
Public Employee(Address address){
This.address=address;
}
// getter and setter methods
Public Address getAddress(){
return address;
}
Public void setAddress( Address address){
This.address=address;
}
}
```

Setter Injection

```
<?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 = “add1" class = "Vesit.Address">
    <property name = “Citye" value = “Mumbaii"/>
    <property name = “state" value = “Maharashtra"/>
    <property name = “pin" value = “615702/>
  </bean>
<bean id = "emp1" class = "Vesit.Employee">
    <property name = "name" value = "Rishi"/>
    <property name = "id" value = "12"/>
    <property name = “address"  ref= “add1"/>
  </bean>
</beans>
```

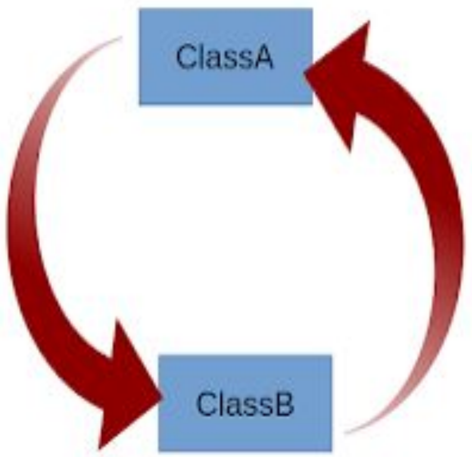
Constructor vs Setter Injection



Constructor Injection	Setter Injection
• At the time of creating your target class object, the dependent objects are injected (can be accessed in the constructor of target class). • In case of constructor injection all the dependent objects are mandatory to be injected. If you don't provide any of the dependent objects through <contructor-arg> tag the core container will detects and throws BeanCreationException. • If classes have cyclic dependencies via constructor, these dependent beans cannot be configured through Constructor Injection.	• The dependent objects are not injected while creating the target classes object. Those will be injected after the target class has been instantiated, by calling the setter on the target object. • In case of setter injection your dependent objects are optional to be injected. Even you don't provide the <property> tag while declaring the bean; the container will creates the Bean and initializes all the properties to their default. • Cyclic dependencies are allowed in Setter Injection.

Circular Dependency

- if you use predominantly constructor injection, it is possible to create an unresolvable circular dependency scenario.
- For example: Class A requires an instance of class B through constructor injection, and class B requires an instance of class A through constructor injection.
- If you configure beans for classes A and B to be injected into each other, the Spring IoC container detects this circular reference at runtime, and throws a BeanCurrentlyInCreationException.



Circular dependency

Overriding Bean

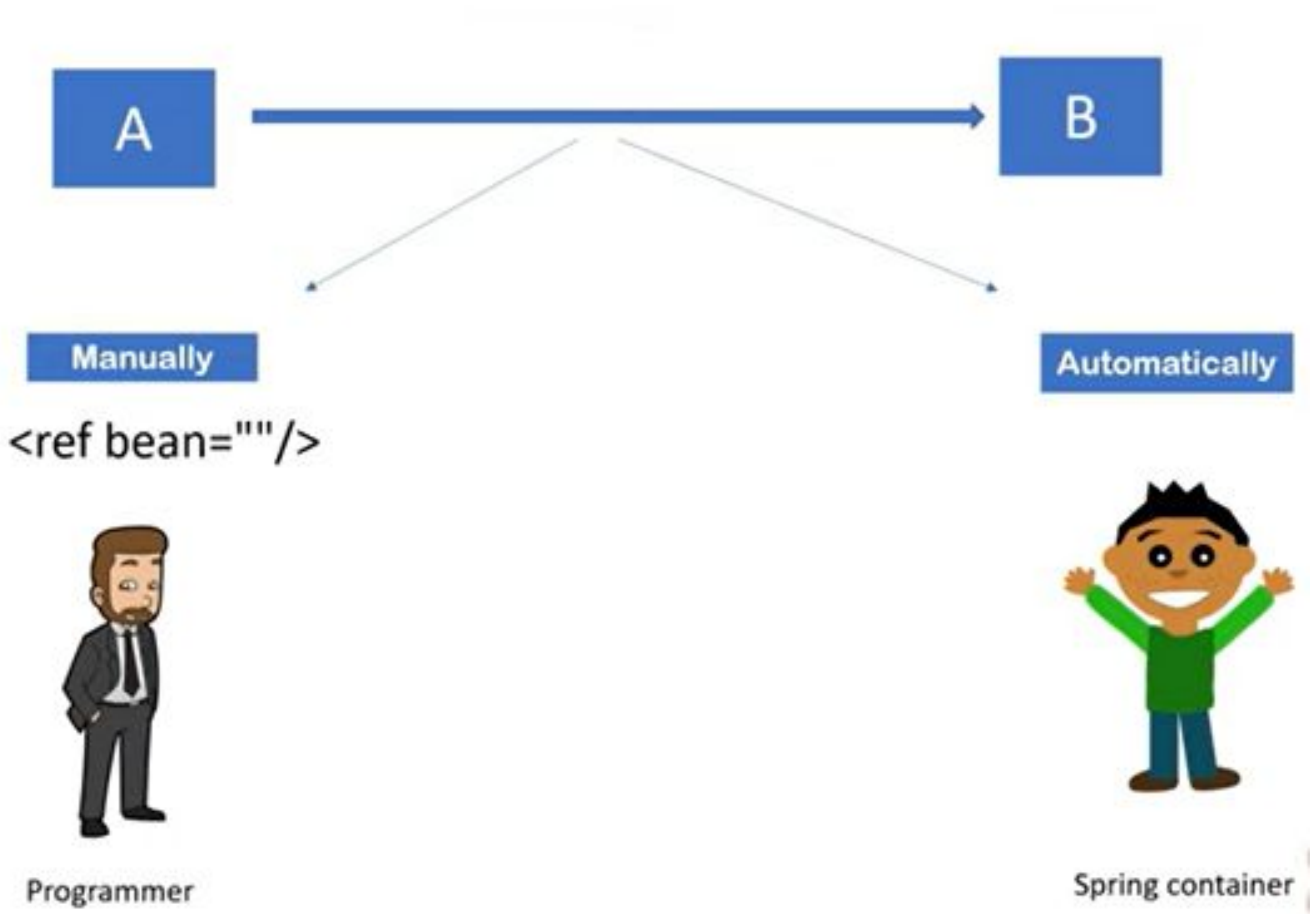
Auto Wiring Bean Looksup

Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection.

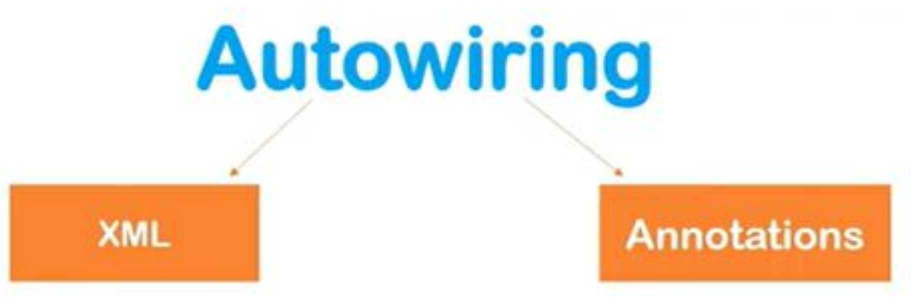
Autowiring can't be used to inject primitive and string values. It works with reference only.

- Advantage
  - It requires the **less code** because we don't need to write the code to inject the dependency explicitly.
- Disadvantage
  - No control of programmer.
  - It can't be used for primitive and string values.

Auto Wiring

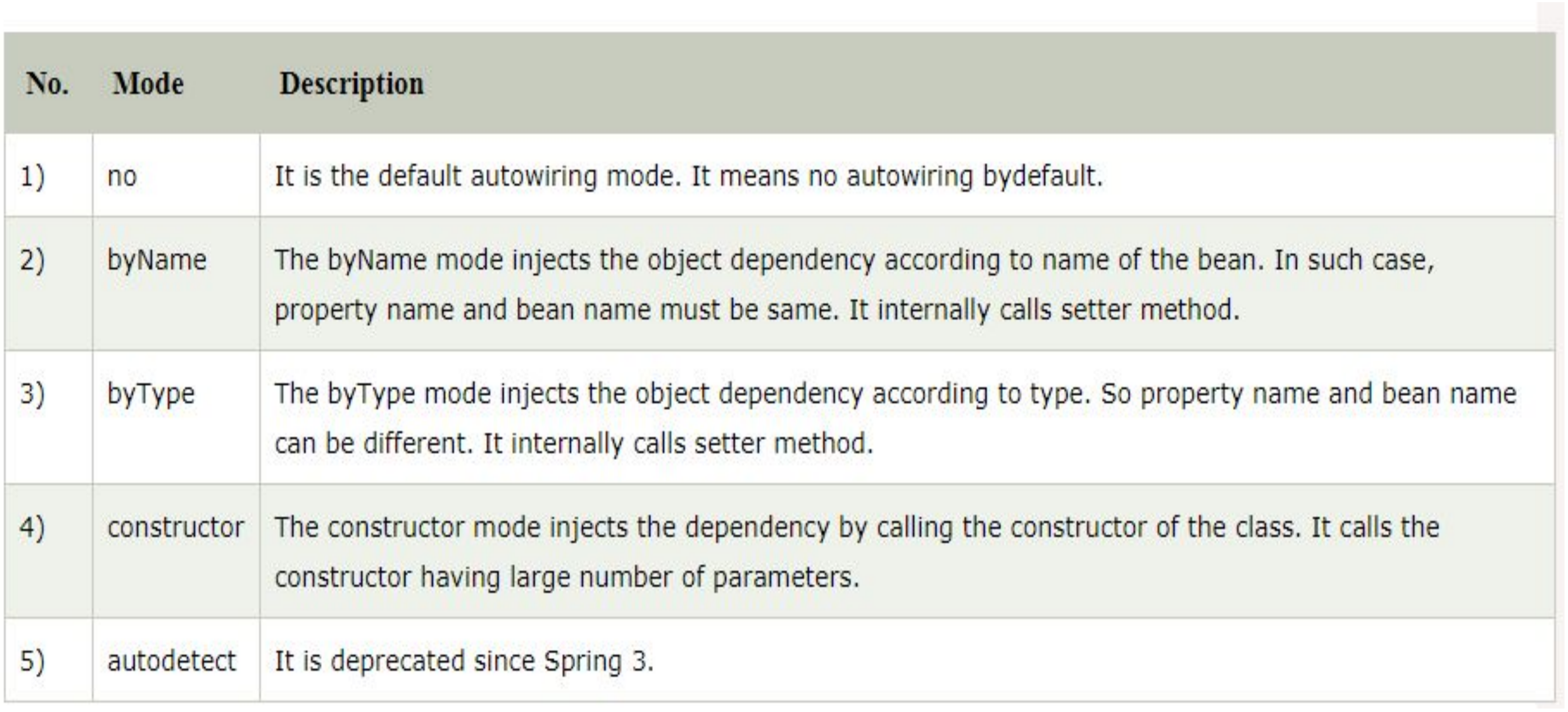


Auto Wiring



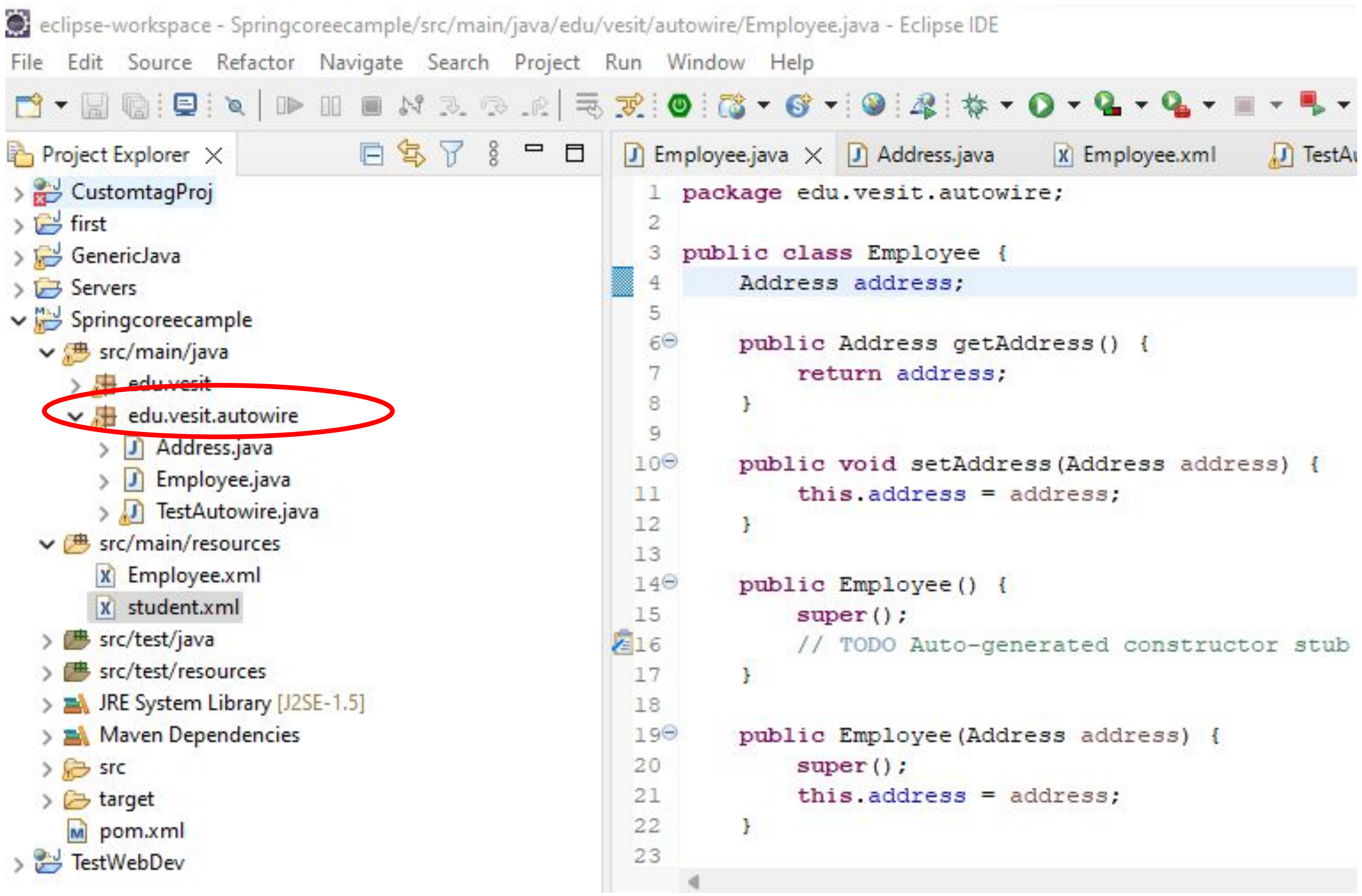
@Autowired

Autowiring Modes XML



No.	Mode	Description
1)	no	It is the default autowiring mode. It means no autowiring bydefault.
2)	byName	The byName mode injects the object dependency according to name of the bean. In such case, property name and bean name must be same. It internally calls setter method.
3)	byType	The byType mode injects the object dependency according to type. So property name and bean name can be different. It internally calls setter method.
4)	constructor	The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters.
5)	autodetect	It is deprecated since Spring 3.

Autowiring Modes XML (Example)



```java
package edu.vesit.autowire;

public class Employee {
    Address address;

    public Address getAddress() {
        return address;
    }

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

    public Employee() {
        super();
        // TODO Auto-generated constructor stub
    }

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

```

Autowiring Modes XML (Example)

```
package edu.vesit.autowire;

public class Employee {
Address address;

public Address getAddress() {
return address;
}

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

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

@Override
public String toString() {
return "Employee [address=" + address + "]";
}
}
```

Autowiring Modes XML (Example)

```java
package edu.vesit.autowire;

public class Address {

private String street;
private String city;
public String getStreet() {
return street;
}
public void setStreet(String street) {
this.street = street;
}
public String getCity() {
return city;
}
public void setCity(String city) {
this.city = city;
}
@Override
public String toString() {
return "Address [street=" + street + ", city=" + city + "]";
}
}
```

Autowiring Modes XML (Example)

```
<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 class="edu.vesit.autowire.Address" name="address">
<property name="street" value="Bell Pepper"></property>
<property name="city" value="Thane"></property>
</bean>

<bean class="edu.vesit.autowire.Employee" name="emp1" autowire="byName">
```

Autowiring Modes XML (Example)

```
package edu.vesit.autowire;

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

public class TestAutowire {

public static void main(String[] args) {
ApplicationContext context = new
ClassPathXmlApplicationContext("employee.xml");
Employee emp=context.getBean("emp1", Employee.class);
System.out.println(emp);
}

}
```

Autowiring Modes XML (Example)

```
Employee [address=Address [street=Bell Pepper, city=Thane]]
```

Autowiring Modes XML (Example)

```
<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 class="edu.vesit.autowire.Address" name="address1">
<property name="street" value="Bell Pepper"></property>
<property name="city" value="Thane"></property>
</bean>

<bean class="edu.vesit.autowire.Employee" name="emp1" autowire="byName">
```

OUTPUT

```
Employee [address=null]
```

Autowiring Modes XML (Example)

```
<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 class="edu.vesit.autowire.Address" name="address1">
<property name="street" value="Bell Pepper"></property>
<property name="city" value="Thane"></property>
</bean>
<bean class="edu.vesit.autowire.Employee" name="emp1" autowire="byType">
</bean>
</beans>
```

OUTPUT

```
Employee [address=Address [street=Bell Pepper, city=Thane]]
```

Autowiring Modes XML (Example)

```
<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 class="edu.vesit.autowire.Address" name="address1">
<property name="street" value="Bell Pepper"></property>
<property name="city" value="Thane"></property>
</bean>

<bean class="edu.vesit.autowire.Address" name="address2">
<property name="street" value="Golf court"></property>
<property name="city" value="Mumbai"></property>
</bean>
<bean class="edu.vesit.autowire.Employee" name="emp1" autowire="byType">
</bean>
</beans>
```

OUTPUT

```
Error creating bean with name 'emp1‘
No qualifying bean of type 'edu.vesit.autowire.Address' available: expected single matching bean but found
2: address1,address2
```

Autowiring Modes XML (Example)

```
Call byName and byType use setter injection
```

Autowiring Modes XML (Example)

```
<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 class="edu.vesit.autowire.Address" name="address">
<property name="street" value="Bell Pepper"></property>
<property name="city" value="Thane"></property>
</bean>

<bean class="edu.vesit.autowire.Address" name="address2">
<property name="street" value="Golf court"></property>
<property name="city" value="Mumbai"></property>
</bean>
<bean class="edu.vesit.autowire.Employee" name="emp1" autowire="constructor">
</bean>
</beans>
```

Constructor injection will be called

Autowiring Modes @Autowired (Example)

```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-3.1.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.1.xsd">

<context:annotation-config />


<!--  <bean class="edu.vesit.autowire.annotation.Address" name="address1">
<property name="street" value="Bell Pepper"></property>
<property name="city" value="Thane"></property>
</bean>
-->

<bean class="edu.vesit.autowire.annotation.Address" name="address">
<property name="street" value="Golf Street"></property>
<property name="city" value="Mumbai"></property>
</bean>
<bean class="edu.vesit.autowire.annotation.Employee" name="emp2">
</bean>
</beans>
```

Autowiring Modes @Autowired (Example)

```
package edu.vesit.autowire.annotation;
import org.springframework.beans.factory.annotation.Autowired;

public class Employee {
@Autowired //by property
Address address;

public Address getAddress() {
return address;
}
public void setAddress(Address address) {
this.address = address;
}
public Employee() {
super();
// TODO Auto-generated constructor stub
}
public Employee(Address address) {
super();
System.out.println("HI I am in Constructor");
this.address = address;
}
@Override
public String toString() {
return "Employee [address=" + address + "]";
}
}
```

Example

https://docs.google.com/document/d/1M1IyOHORnP_NBChwstRFdhG8zY0J7NcbfRYdukWumoE/edit

@Autowired Annotation

`@Autowired` annotation to auto wire bean on the setter method, constructor or a field. Moreover, it can autowire property in a particular bean. We must first enable the annotation using below configuration in configuration file.If you are using Java based configuration, you can enable annotation-driven injection by using below spring configuration:

```
 @Configuration

 @ComponentScan("guru.springframework.autowiringdemo")

 public class AppConfig {}
```

As an alternative, we can use below XML based configuration in Spring:

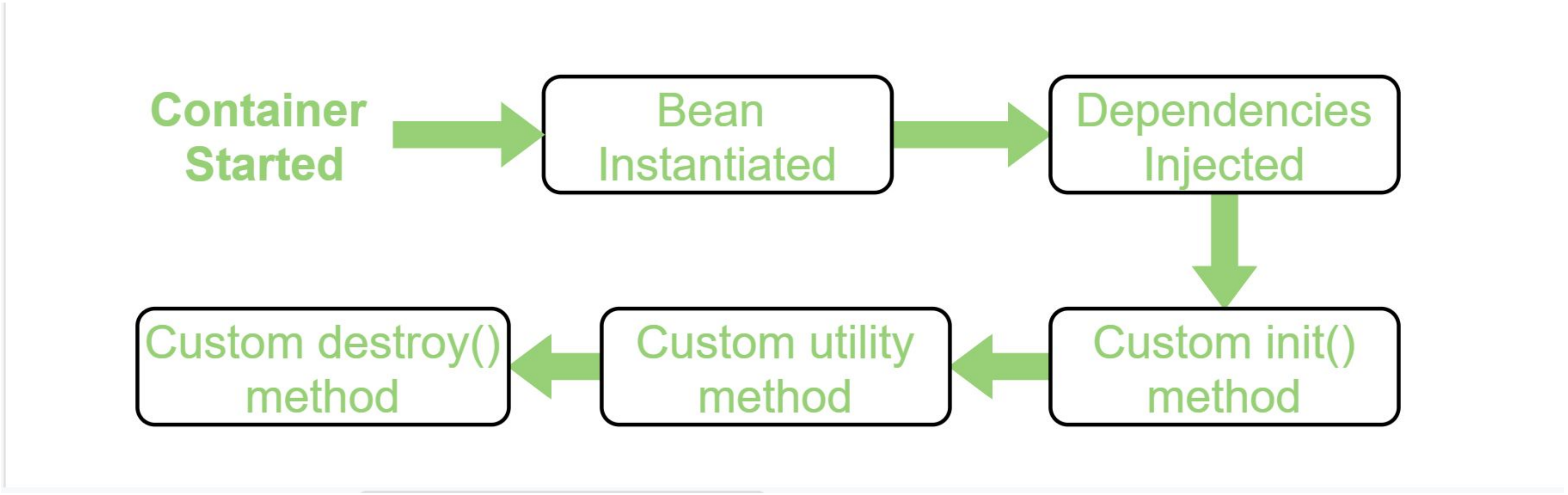
```
<context:annotation-config />
```

Spring Manage Beans

- Lifecycle of Object
  - when & how it is born
  - how it behaves throughout its life
  - when & how it dies
- bean life cycle
  - when & how the bean is instantiated
  - action it performs until it lives
  - when & how it is destroyed

Spring Manage Beans

Bean life cycle is managed by the spring container



Spring Manage Beans

- Spring provides three ways to implement the life cycle of a bean
  - **By XML**
  - **Spring Interface**
  - **Annotation**

Links

- https://www.youtube.com/watch?v=rMLP-NEPgnM
- https://www.thejavaprogrammer.com/spring-hello-world-example/
- https://www.youtube.com/watch?v=TtikLIZstkE