

Spring-Core

What is Spring Framework :

1. Spring Framework was developed by Rod Johnson in 2003.
2. Spring framework makes the deployment of javaEE applications easy.
3. Spring framework is a dependency injection framework

Enterprise Edition:

1. Enterprise means it's designed for large organizations or businesses (like banks, hospitals, or online stores) that need complex, secure, and scalable applications.
2. Edition means it's a specific version of Java that focuses on enterprise-level tasks, unlike the simpler Java SE (Standard Edition), which is used for smaller applications.
3. So, Java EE is a special version of Java created to building big, complex applications to make businesses easier and more organized!

Dependency Injection

1. Dependency injection is an design pattern
2. For eg. when we create the object of one class in another class the application becomes tightly coupled
3. To avoid that spring uses IOC container to inject object and it's dependencies into the class during runtime

IOC(Inversion of Control) Container

1. IOC container is a pre-defined program which is responsible for
 - a. Creation of object
 - b. Holding the object into the memory
 - c. And injecting the object in another object as required
2. So the complete life cycle of an object from the creation to the destruction is maintained by IOC container
3. IOC container can inject
 - a. Primitive DataTypes
 - i. byte,short,int,long,float,double,char,boolean
 - b. Non primitive/ Collection DataTypes
 - i. String,List,Set,Map
 - c. Reference type
 - i. Other class Object
4. Dependency injection can be done by two ways
 - a. Setter/Property based injection

- i. Use setter methods to create an object
- b. constructor based injection
 - i. Uses constructor to create an object

Application Context

1. We use application context to fetch or get the object from the IOC container
2. ApplicationContext is an interface which extends BeanFactory
3. Since ApplicationContext is an interface we can not create Object of it but we can create the Object of Its Subclasses
4. Some commonly use subclasses of ApplicationContext are
 - a. ClassPathXmlApplicationContext
 - i. loads Spring Beans from XML configuration files located in the classpath.
 - b. FileSystemXmlApplicationContext
 - i. loads Spring Beans from XML configuration files located in the file system.
 - c. AnnotationConfigApplicationContext\
 - i. loads Spring Beans from Java-based configuration classes annotated with @Configuration.
 - d. WebApplicationContext:
 - i. This is an interface that represents the application context for web applications.
 - e. GenericApplicationContext:
 - i. This is a generic implementation of the ApplicationContext interface which can be customized as needed.

Configuration.xml

1. To inject an object we need to provide some information of beans and it's dependencies to the IOC container
2. We provide that Information using Configuration.xml(the name can be anything)
3. Configuration.xml has <beans> tag, inside that tag we can configure as many beans as we want using <bean> tag
4. We define some xmlns (XML Namespace) and xsi(XML Schema Instance) in <beans> tag
5. Bean is nothing but java POJO class with getters and setters

Config.xml <beans> tag's name space configuration.

```
<?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"
  xmlns:p="http://www.springframework.org/schema/p"
  xmlns:c="http://www.springframework.org/schema/c"
  xmlns:util="http://www.springframework.org/schema/util"
  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
    http://www.springframework.org/schema/util
    http://www.springframework.org/schema/util/spring-util.xsd">
```

Spring Modules

1. **Spring Core Container**
 - a. Provides the fundamental functionality of the Spring framework, including IoC (Inversion of Control) and DI (Dependency Injection). This includes modules like `spring-core`, `spring-beans`, `spring-context`, and `spring-expression`.
2. **Spring AOP (Aspect-Oriented Programming)**
 - a. Supports aspect-oriented programming by allowing the separation of cross-cutting concerns from the business logic. This includes modules like `spring-aop` and `spring-aspects`.
3. **Spring JDBC (Java Database Connectivity)**
 - a. Simplifies database access by providing JDBC abstractions and utilities. This includes modules like `spring-jdbc` and `spring-tx`.
4. **Spring ORM (Object-Relational Mapping)**
 - a. Provides integration layers for popular ORM frameworks like Hibernate, JPA, and JDO. This includes modules like `spring-orm`.
5. **Spring OXM (Object-XML Mapping)**
 - a. Provides support for XML mapping, allowing conversion between Java objects and XML representations. This includes modules like `spring-oxm` and supports technologies such as JAXB, Castor, XMLBeans, and XStream.
6. **Spring JMS (Java Message Service)**
 - a. Offers support for messaging within Spring applications, allowing for the integration of JMS providers. This includes modules like `spring-jms`.
7. **Spring WebSocket**
 - a. Offers support for WebSocket-based communication in Spring applications. This includes modules like `spring-websocket`.
8. **Spring Web Servlet**
 - a. Provides foundational support for building web applications using the servlet API. This includes modules like `spring-web`.
9. **Spring Security**
 - a. Provides comprehensive security services for Java EE-based enterprise software applications. This includes modules like `spring-security-core`, `spring-security-config`, `spring-security-web`, etc.
10. **Spring Data**
 - a. Simplifies data access by providing a consistent approach to data access, regardless of the data store being used. This includes modules like `spring-data-jpa`, `spring-data-mongodb`, `spring-data-redis`, etc.
11. **Spring Batch:**
 - a. Supports batch processing and the development of robust batch applications.

12. Spring Test:

- a. Provides support for testing Spring components with JUnit or TestNG.

13. Spring Boot:

- a. Simplifies the process of building production-ready applications by providing auto-configuration, embedded servers, and other opinionated features.

14. Spring Cloud:

- a. Provides tools and libraries for building cloud-native applications.

Setter/Properties bases injection

It is a dependency injection method in which the Spring container injects dependencies into a bean using **setter methods** after creating the bean instance.

1] Primitive/Non-primitive DataType Injection

1] Using Value Tag

```
<bean class="" name="">  
    <property name="">  
        <value> some value </value>  
    </property>  
</bean>
```

2] Using value attribute

```
<bean class="" name="" >  
    <property name="" value="" />  
</bean>
```

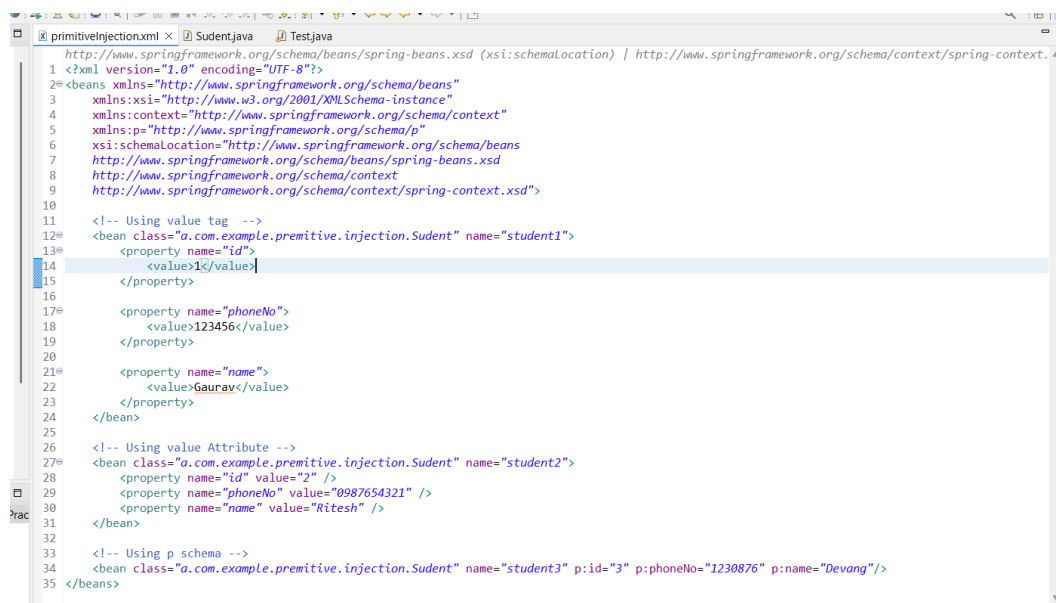
3] Using p schema

```
<bean class="" name="" p:propertyName="someValue" />
```

```

1 package a.com.example.primitive.injection;
2
3 public class Sudent {
4
5     private int id;
6     private long phoneNo;
7     private String name;
8
9
10    public Sudent() {
11        super();
12        // TODO Auto-generated constructor stub
13    }
14
15    public Sudent(int id, long phoneNo, String name) {
16        super();
17        this.id = id;
18        this.phoneNo = phoneNo;
19        this.name = name;
20    }
21
22    public int getId() {
23        return id;
24    }
25
26    public void setId(int id) {
27        System.out.println("StudentId from setter");
28        this.id = id;
29    }
30
31    public long getPhoneNo() {
32        return phoneNo;
33    }
34
35 }

```



```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xmlns:context="http://www.springframework.org/schema/context"
5       xmlns:p="http://www.springframework.org/schema/p"
6       xsi:schemaLocation="http://www.springframework.org/schema/beans
7                           http://www.springframework.org/schema/beans/spring-beans.xsd
8                           http://www.springframework.org/schema/context
9                           http://www.springframework.org/schema/context/spring-context.xsd">
10
11    <!-- Using value tag -->
12    <bean class="a.com.example.primitive.injection.Sudent" name="student1">
13        <property name="id">
14            <value>1</value>
15        </property>
16
17        <property name="phoneNo">
18            <value>123456</value>
19        </property>
20
21        <property name="name">
22            <value>Gaurav</value>
23        </property>
24    </bean>
25
26    <!-- Using value Attribute -->
27    <bean class="a.com.example.primitive.injection.Sudent" name="student2">
28        <property name="id" value="2" />
29        <property name="phoneNo" value="0987654321" />
30        <property name="name" value="Ritesh" />
31    </bean>
32
33    <!-- Using p schema -->
34    <bean class="a.com.example.primitive.injection.Sudent" name="student3" p:id="3" p:phoneNo="1230876" p:name="Devang"/>
35 </beans>

```

```

1 package a.com.example.primitive.injection;
2
3 import org.springframework.context.ApplicationContext;
4
5 public class Test {
6
7     public static void main(String[] args) {
8
9         ApplicationContext context = new ClassPathXmlApplicationContext("com/example/primitive/injection/primitiveInjection.xml");
10         Sudent student1 = (Sudent) context.getBean("student1");
11         Sudent student2 = (Sudent) context.getBean("student2");
12         Sudent student3 = (Sudent) context.getBean("student3");
13         System.out.println(student1);
14         System.out.println(student2);
15         System.out.println(student3);
16     }
17 }

```

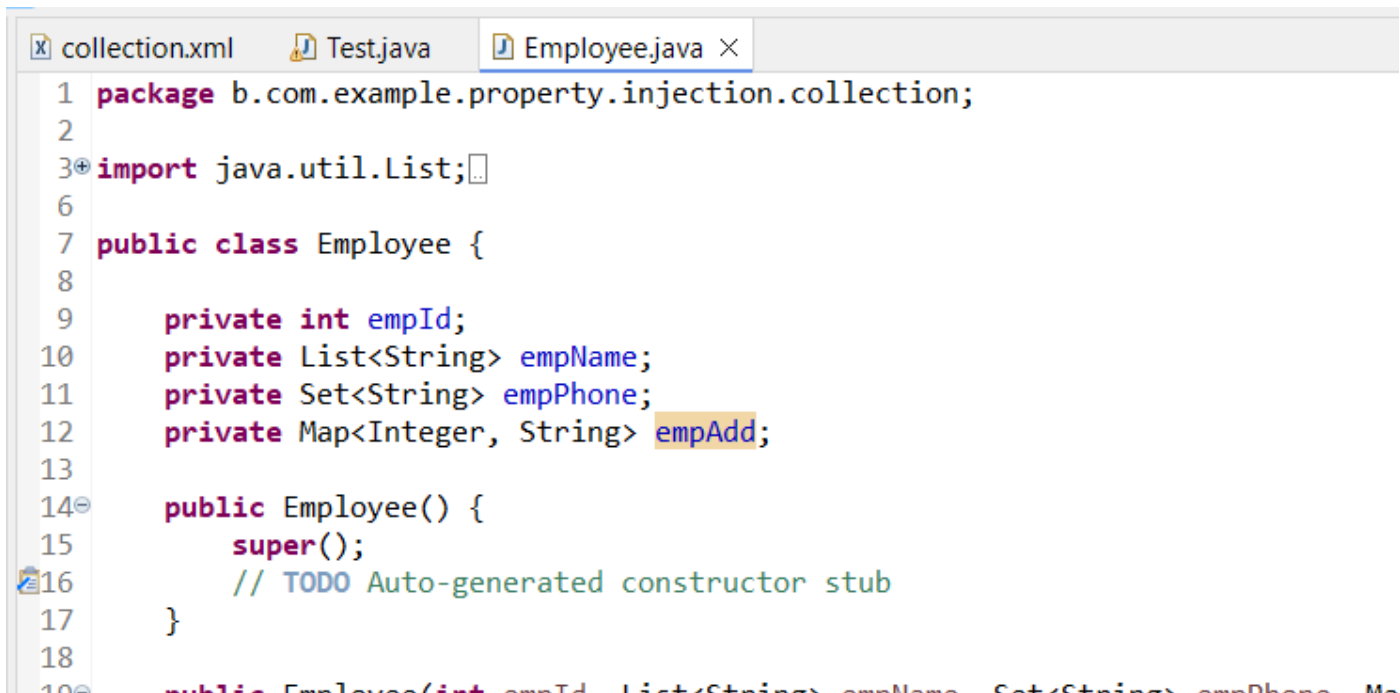
2] Collection type Injection

1] to inject List, Set or Map

```
<bean class="" name="">
  <property name="">
    <list>
      <value> value1</value>
      <value> value2</value>
      <value> value3</value>
    </list>
  </property>

  <property name="">
    <set>
      <value> value1</value>
      <value> value2</value>
      <value> value3</value>
    </set>
  </property>

  <property name="">
    <map>
      <entry key="" value="">
      <entry key="" value="">
      <entry key="" value="">
    </map>
  </property>
</bean>
```



```
collection.xml Test.java Employee.java x
1 package b.com.example.property.injection.collection;
2
3 import java.util.List;
4
5
6
7 public class Employee {
8
9     private int empId;
10    private List<String> empName;
11    private Set<String> empPhone;
12    private Map<Integer, String> empAdd;
13
14    public Employee() {
15        super();
16        // TODO Auto-generated constructor stub
17    }
18
19    public Employee(int empId, List<String> empName, Set<String> empPhone, Map<Integer, String> empAdd) {
20        this.empId = empId;
21        this.empName = empName;
22        this.empPhone = empPhone;
23        this.empAdd = empAdd;
24    }
25
26    public int getEmpId() {
27        return empId;
28    }
29
30    public void setEmpId(int empId) {
31        this.empId = empId;
32    }
33
34    public List<String> getEmpName() {
35        return empName;
36    }
37
38    public void setEmpName(List<String> empName) {
39        this.empName = empName;
40    }
41
42    public Set<String> getEmpPhone() {
43        return empPhone;
44    }
45
46    public void setEmpPhone(Set<String> empPhone) {
47        this.empPhone = empPhone;
48    }
49
50    public Map<Integer, String> getEmpAdd() {
51        return empAdd;
52    }
53
54    public void setEmpAdd(Map<Integer, String> empAdd) {
55        this.empAdd = empAdd;
56    }
57
58 }
```

<https://www.springframework.org/docs/core/context/spring-context.xml>

```
<bean class="b.com.example.property.injection.collection.Employee" name="emp">
  <property name="empId" value="1" />

  <property name="empName">
    <!-- <value>Dipak</value> if list have only one value -->
    <list>
      <value>Gaurav</value>
      <value>Rutik</value>
      <value>Devang</value>
      <null/>
    </list>

    <!-- To create an empty list -->
    <!-- <list></list> -->
  </property>

  <property name="empPhone">
    <set>
      <value>1234</value>
      <value>5678</value>
      <value>0987</value>
    </set>
  </property>

  <property name="empAdd">
    <map>
      <entry key="1" value="Kayan" />
      <entry key="2" value="Dombivali" />
      <entry key="3" value="Thane" />
    </map>
  </property>
```

```
5
6 public class Test {
7
8   public static void main(String[] args) {
9
10     ApplicationContext context = new ClassPathXmlApplicationContext("b/com/example/property/injection/collection/collection.xml");
11     Employee emp = (Employee)context.getBean("emp");
12     System.out.println(emp);
13   }
14 }
15
16 }
```

3] Reference type Injection

To Inject Object reference of one class Into Another class

<bean class="" name="injectionBean" /> (bean which will be injected in other beans)

Using ref tag

```
<property name="" >
    <ref bean="injectionBean" />
</property>
```

Using ref attribute

```
<property name="" ref="injectionBean" />
```

Using p schema

```
<bean class="" name="" p:referencePropertyName-ref="injectionBean"/>
```

```
<!-- Using ref tag -->
<bean class="com.example.property.injection.reference.B" name="obOfB" p:varFrom="Variable From B"/>
<!-- Using ref tag -->
<bean class="com.example.property.injection.reference.A" name="obOfA0">
    <property name="varFrom" value="Var From obOfA0" />
    <property name="objectOfB" >
        <ref bean="obOfB" />
    </property>
</bean>

<!-- Using ref attribute -->
<bean class="com.example.property.injection.reference.A" name="obOfA">
    <property name="varFrom" value="Var From A"/>
    <property name="objectOfB" ref="obOfB" />
</bean>

<!-- Using P schema -->
<bean class="com.example.property.injection.reference.A" name="obOfA2"
    p:varFrom="Var From A2"
    p:objectOfB-ref="obOfB"/>
```


Constructor Based Injection

Injecting Object of bean Using Constructors

1] Primitive Injection

```
2
3 public class Student {
4     private int a;
5     private int b;
6
7     public Student(int a, int b) {
8         super();
9         System.out.println("Assignin vaues to int a int b");
10        this.a = a;
11        this.b = b;
12    }
13
14    @Override
15    public String toString() {
16        return "Student [a=" + a + ", b=" + b + "]";
17    }
18
19 }
```

```
<bean class="com.example.cunstructor.primitive.injection.Student" name="student1">
  <!-- Using Value Tag -->
  <constructor-arg>
    <value>10</value>
  </constructor-arg>

  <constructor-arg>
    <value>20</value>
  </constructor-arg>
</bean>

<!-- Using value attribute -->
<bean class="com.example.cunstructor.primitive.injection.Student" name="student2">
  <constructor-arg value="30" />
  <constructor-arg value="40" />
</bean>

<!-- Using c schema -->
<bean class="com.example.cunstructor.primitive.injection.Student" name="student3"
  c:a="60" c:b="70"/>
```

```
5 public class Test {
6
7     public static void main(String[] args) {
8
9         ApplicationContext context = new ClassPathXmlApplicationContext("d.com/example/cunstructor/primitive/injection/cpi.xml");
10
11         Student student1 = (Student) context.getBean("student1");
12         System.out.println(student1);
13
14         System.out.println("-----");
15         Student student2 = (Student) context.getBean("student2");
16         System.out.println(student2);
17
18         System.out.println("-----");
19         Student student3 = (Student) context.getBean("student3");
20         System.out.println(student3);
21     }
22 }
23
24 }
```

2] Collection type Injection

```
6
7 public class Emp {
8
9     private List<String> emoName;
10    private List<String> empBiwiKaName;
11    private Set<Long> empPhone;
12    private Map<Integer, String> empInfo;
13
14    public Emp(List<String> emoName, Set<Long> empPhone, Map<Integer, String> empInfo) {
15        super();
16        this.emoName = emoName;
17        this.empPhone = empPhone;
18        this.empInfo = empInfo;
19    }
20
```

```
<bean class="com.example.cunstructor.collection.injection.Emp" name="emp1">
    <constructor-arg>
        <list>
            <value>Gaurav</value>
            <value>Diksha</value>
            <null/>
        </list>
    </constructor-arg>
    <constructor-arg>
        <set>
            <value>12345</value>
            <value>67890</value>
        </set>
    </constructor-arg>
    <constructor-arg>
        <map>
            <entry key="1" value="Gaurav" />
            <entry key="2" value="Diksha" />
        </map>
    </constructor-arg>
</bean>
```

```
5
6 public class Test {
7
8     public static void main(String[] args) {
9         // TODO Auto-generated method stub
10
11         ApplicationContext context = new ClassPathXmlApplicationContext("e/com/example/cunstructor/collection/injection/cci.xml");
12         Emp emp1 = (Emp) context.getBean("emp1");
13
14         System.err.println(emp1);
15
16     }
17 }
18
```

3] Reference type

```
5 public class A {
6
7     private String varFrom;
8     private B obOfB;
9     private List<B> ListOfB;
10
11
12     public A(List<B> listOfB) {
13         super();
14         ListOfB = listOfB;
15     }
16
17     public A(String varFrom, B obOfB) {
18         super();
19         this.varFrom = varFrom;
20         this.obOfB = obOfB;
21     }
22 }
```

```
3 public class B {
4
5     private String varFrom;
6
7     public B(String varFrom) {
8         super();
9         this.varFrom = varFrom;
10     }
11 }
```

```
<!-- Using ref Tags -->
<bean class="com.example.cunstructor.ref.injection.B" name="obOfB1" c:varFrom="Varoable from obOfB1" />
<bean class="com.example.cunstructor.ref.injection.B" name="obOfB2" c:varFrom="Varoable from obOfB2" />
<bean class="com.example.cunstructor.ref.injection.B" name="obOfB3" c:varFrom="Varoable from obOfB3" />

<!-- Using ref Tags -->
<bean class="com.example.cunstructor.ref.injection.A" name="obOfA1">
    <constructor-arg value="Variable from obOfA" />
    <constructor-arg>
        <ref bean="obOfB1"/>
    </constructor-arg>
</bean>

<!-- Using ref attribute -->
<bean class="com.example.cunstructor.ref.injection.A" name="obOfA2">
    <constructor-arg value="Variable from obOfA2" />
    <constructor-arg ref="obOfB2" />
</bean>

<!-- Using c schema -->
<bean class="com.example.cunstructor.ref.injection.A" name="obOfA3"
    c:varFrom="Variable From obOfA3"
    c:obOfB-ref="obOfB3" />

<!-- ref collection -->
<bean class="com.example.cunstructor.ref.injection.A" name="obOfA4">
    <constructor-arg>
        <list>
            <ref bean="obOfB1"/>
            <ref bean="obOfB2"/>
            <ref bean="obOfB3"/>
        </list>
    </constructor-arg>
</bean>
```

```

6 public class Test {
7
8     public static void main(String[] args) {
9         // TODO Auto-generated method stub
10
11         ApplicationContext context = new ClassPathXmlApplicationContext("f/com/example/cunstructor/ref/injection/ref.xml");
12         A ob0fA1 = (A) context.getBean("ob0fA1");
13         A ob0fA2 = (A) context.getBean("ob0fA2");
14         A ob0fA3 = (A) context.getBean("ob0fA3");
15         A ob0fA4 = (A) context.getBean("ob0fA4");
16
17         System.out.println(ob0fA1);
18         System.out.println(ob0fA2);
19         System.out.println(ob0fA3);
20         System.err.println("-----");
21         System.out.println(ob0fA4.getList0fB());
22

```

Spring Ambiguity Problem

In constructor based injection if the class is having multiple constructors with same length of parameter but there datatype is different then First preference will be always given to Constructor which is taking String as an parameter and if Constructor with String parameter does not exist in that case IOC container will Look for the constructor from top to bottom order and will call first Constructor based on number of argument passed in config.xml.

If the ambiguity cannot be resolved, the container throws an exception.

`org.springframework.beans.factory.UnsatisfiedDependencyException`

To resolve this we use type attribute where we specify type of an argument we are passing so that IOC container will know which container to be called exactly.

```

2
3 public class Student {
4
5     private int a;
6     private int b;
7     private String cunstructorCalled;
8
9     public Student(int a, int b) {
10         super();
11         this.a = a;
12         this.b = b;
13         this.cunstructorCalled = "int int";
14     }
15
16     public Student(String a, String b) {
17         super();
18         this.a = Integer.parseInt(a);
19         this.b = Integer.parseInt(b);
20         this.cunstructorCalled = "String String";
21     }
22

```

```

<!-- Ambiguity Issue -->
<!-- Will consider values as String -->
<bean class="com.example.cunstructor.injection.ambiguity.Student" name="student1" >
    <constructor-arg value="10" />
    <constructor-arg value="20" />
</bean>

<!-- Solution -->
<bean class="com.example.cunstructor.injection.ambiguity.Student" name="student2" >
    <constructor-arg value="10" type="int" />
    <constructor-arg value="20" type="int" />
</bean>

```

```

3
6 public class Test {
7
8     public static void main(String[] args) {
9
10         ApplicationContext context = new ClassPathXmlApplicationContext("g/com/example/cunstructor/injection/ambiguity/ambiguity.xml");
11         Student student1 = (Student) context.getBean("student1");
12         System.out.println(student1);
13
14         Student student2 = (Student) context.getBean("student2");
15         System.out.println(student2);
16
17     }
18 }
19
20

```

Spring Bean Lifecycle

In Spring Framework, a bean follows a well-defined lifecycle that includes instantiation, dependency injection, initialization, usage, destruction, and garbage collection. The Spring Container manages these phases automatically.

Phases of Spring Bean Lifecycle

1. Instantiation (Object Creation)

- The Spring Container creates an instance of the bean using:
 - Constructor-based instantiation (default constructor or parameterized constructor).
 - Factory methods (e.g., `@Bean` annotated method in Java-based configuration).

2. Dependency Injection (Property Population)

- Once the bean is created, Spring inject dependencies into it.
- This can be done using:
 - Field injection (`@Autowired`)
 - Setter injection
 - Constructor injection

3. Bean Post-Processing (Before Initialization) - Global Level

- Before the initialization starts, Spring provides a way to modify the bean globally using the BeanPostProcessor interface.
- It calls the postProcessBeforeInitialization() method.
- These modifications are applied to all beans globally and are used for modifying or validating beans before their initialization phase.

4. Initialization Phase

- In this phase, once the properties are set, we can perform custom initialization logic
- The following methods are used:
 - **@PostConstruct** (Annotation-based approach)
 - **init-method="init"** (XML-based approach)
 - **afterPropertiesSet()** (via **InitializingBean** interface)

5. Bean Post-Processing (After Initialization)

- After initialization, BeanPostProcessor calls postProcessAfterInitialization(), allowing further modifications before the bean is ready to use.
- This is a global step and applies to all beans.

6. Bean is Ready to Use

- Now, the bean is **fully initialized** and can be used within the application.

7. Destruction Phase (Cleanup Before Removal)

- Before the bean is removed from the container, Spring **performs cleanup operations**.
- The following methods are used:
 - **@PreDestroy** (Annotation-based approach)
 - **destroy-method="cleanup"** (XML-based approach)
 - **destroy()** (via **DisposableBean** interface)

8. Garbage Collection (Non-Spring Phase)

- After destruction, the bean becomes eligible for garbage collection, and the JVM removes it when necessary.

Ways to call init() and destroy() method using three ways

- Xml
- Spring interface
- Annotation

1] Using xml

Define the init and destroy method in bean

Note : the name of the both function can be anything

```
public void init() {  
    System.out.println("Inside init() method");  
}  
  
public void destroy() {  
    System.out.println("Inside destroy() method");  
}
```

Pass the name of that method in bean tag's init-method and destroy-method attribute

```
<bean class="com.example.spring.lifecycle.LifeCycleBean" name="bean1" init-method="init" destroy-method="destroy">  
    <property name="var" value="some value" />  
</bean>
```

To call the destroy method we have to call registerShutdownHook() method of AbstractApplicationContext interface.

```
public static void main(String[] args) {  
    AbstractApplicationContext applicationContext = new ClassPathXmlApplicationContext("com/example/spring/lifecycle/be  
    LifeCycleBean bean = (LifeCycleBean) applicationContext.getBean("bean1");  
    System.out.println(bean);  
  
    // To call destroy method using registerShutdownHook() which is present in AbstractApplicationContext interface.  
    applicationContext.registerShutdownHook();  
}
```

registerShutdownHook() is a method in the Spring Framework that registers a shutdown hook with the JVM. This ensures that when the application is stopped, the Spring container automatically calls the **destroy()** methods of beans, allowing proper resource cleanup.

2] Using Interface

We can call init() and destroy() by implementing two interface InitializingBean, DisposableBean and by overriding their respective methods afterPropertiesSet() and destroy()
In this case also we can call registerShutdownHook() for destroy()

```
public class Student implements InitializingBean, DisposableBean {

    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("Study Done, Ready for exam");
    }

    @Override
    public void destroy() throws Exception {
        System.out.println("Leaving School");
    }

}
```

```
<bean class="com.example.spring.lifecycle.using.interfac.Student" name="bean1">
    <property name="studyStatus" value="Giving Exam" />
</bean>

</beans>
```

```
public class Test {
    public static void main(String[] args) {
        AbstractApplicationContext context = new ClassPathXmlApplicationContext("com/example/spring/lifecycle/using/interfac/blcui.xml");
        Student std1 = (Student) context.getBean("bean1");
        System.out.println(std1);

        context.registerShutdownHook();
    }
}
```

3] Using Annotations

We can call init and destroy using using annotation as well

- We can use **@PostConstruct** to call init()
- And **@PreDestroy** for destroy()

Note :

1. The **@PostConstruct** and **@PreDestroy** are the part of javaEE and since in the newer version of java javaEE is deprecated we can not use those annotations directly we have to add some dependency in pom.xml
2. To enable all the annotation we have to add **<context:annotation-config />** tag in xml

```
<!-- To enable all the Annotations -->
<context:annotation-config />

<bean class="com.example.spring.lifecycle.using.annotation.Bean" name="bean1">
    <property name="beanStatus" value="using bean" />
</bean>

</beans>
```



```

    @PostConstruct
    public void start() {
        System.out.println("After Intiaization, init called");
    }

    @PreDestroy
    public void end() {
        System.out.println("Before distruction, destroy called");
    }
}

```

```

public class Test {

    public static void main(String[] args) {

        AbstractApplicationContext context = new ClassPathXmlApplicationContext("j:/com/example/spring/lifecycle/using/annotation/blcua.xml");
        Bean bean = (Bean) context.getBean("bean1");
        System.out.println(bean);
        context.registerShutdownHook();
    }

}

```

Spring Bean ID, Name, and Alias

In Spring, beans are identified using **id**, **name**, and **alias**. These attributes help in referencing beans within the Spring container.

1. **id** Attribute

- The **id** attribute is a **unique identifier** for a bean.
- It **must be unique** within the Spring configuration file.
- It **cannot contain special characters or spaces** (except underscores).
- **Example:**
`<bean id="myBean" class="com.example.MyClass" />`

2. **name** Attribute

- The **name** attribute is an **alternative to id** and can hold multiple names separated by commas, semicolons, or spaces.
- If the **id** is not specified, the first value in **name** is treated as the bean's identifier.
- Unlike **id**, **name allows special characters**.
- **Example:**
`<bean name="bean1, myBean" class="com.example.MyClass" />`

3. **alias** Element

- Spring provides the **<alias>** tag to create **additional names (aliases) for a bean**.

- This helps in using different names for the same bean in different contexts.

Example:

```
<bean id="mainBean" class="com.example.MyClass" />
```

- `<alias name="mainBean" alias="alternativeBean" />`

4. Important Points

- A bean can have multiple names using `name` or `<alias>`.
- `id` is always unique, but `name` allows multiple references.
- Spring treats `name` as `id` if `id` is not provided.

Autowiring in Spring

What is Autowiring?

Autowiring is a feature in the Spring Framework that automatically injects dependencies into a bean without needing explicit configurations. Instead of manually defining dependencies, Spring finds and injects the required objects automatically.

Advantages of Autowiring

- **Saves time** – No need to manually define dependencies.
- **Less code** – Reduces the number of `<bean>` definitions.
- **Automatic dependency management** – Spring injects dependencies based on name or type.

Disadvantages of Autowiring

- **Less control** – Spring makes the decisions, not the developer.
- **Cannot inject primitive and String values.**
- **Ambiguity** – If multiple beans of the same type exist, Spring may throw an error.

Types of Autowiring

Spring provides two ways to perform autowiring:

1. **XML-Based Autowiring**
2. **Annotation-Based Autowiring**

1 XML-Based Autowiring

Spring provides different modes of autowiring when using XML configuration:

1.1 No Autowiring (**no**)

- If we don't specify autowiring in the `<bean>` tag, it defaults to **no**.
- Dependencies must be set manually using `<property>` or `<constructor-arg>`.

1.2 Autowiring by Name (**byName**)

- Uses `autowire="byName"` in the `<bean>` tag.

- Spring looks for a bean whose **ID matches the variable name** or reference object in the class.
- Uses **setter methods** for injection.

Example:

```
<bean id="employee" class="com.example.Employee" autowire="byName" />
<bean id="address" class="com.example.Address" />
```

If the `Employee` class has a variable `address`, Spring will inject the `Address` bean automatically.

1.3 Autowiring by Type (`byType`)

- Uses `autowire="byType"` in the `<bean>` tag.
- Spring looks for a bean of the **same type** as the reference variable in the class.
- Uses **setter methods** for injection.
- If **only one bean** of the required type exists, it is injected.
- If **multiple beans** of the same type exist **without a unique qualifier**, Spring throws a `NoUniqueBeanDefinitionException`.

Example:

```
<bean id="employee" class="com.example.Employee" autowire="byType" />
<bean id="address" class="com.example.Address" />
```

If the `Employee` class has a variable of type `Address`, Spring will inject the `Address` bean automatically.

1.4 Autowiring by Constructor (`constructor`)

- Uses `autowire="constructor"` in the `<bean>` tag.
- Spring resolves constructor-based dependencies **by type, not by name**.
- If **multiple beans** of the same type exist **without a unique qualifier**, Spring throws a `NoUniqueBeanDefinitionException`.
- Works best with `@Primary` or `@Qualifier` when multiple beans exist.

Example:

```
<bean id="employee" class="com.example.Employee" autowire="constructor" />
<bean id="address" class="com.example.Address" />
```

If `Employee` has a constructor that takes an `Address`, Spring will inject it automatically.

2 Annotation-Based Autowiring

To inject dependencies automatically in Spring, we use the `@Autowired` annotation. It tells Spring to **automatically find and inject** a matching bean into the dependent class.

Ways to Use @Autowired

We can use @Autowired in three ways:

1. On Fields
2. On Setter Methods
3. On Constructors

How @Autowired Works (Step-by-Step)

1. Spring first looks for a bean of the required type (class type).
2. If only one bean of that type exists, it is injected.
3. If multiple beans exist:
 - Spring checks if any bean name matches the variable name (Fallback Mechanism).
 - If a bean with the same name as the reference variable exists, it is injected.
 - If no bean name matches and multiple beans of the same type exist, Spring throws a `NoUniqueBeanDefinitionException`.
4. To resolve conflicts, we use @Qualifier to specify the exact bean we want to inject.

```
<bean class="l.com.example.spring.autowiring.annotation.Address" name="address1">
  <property name="add" value="Kalyan" />
</bean>

<bean class="l.com.example.spring.autowiring.annotation.Address" name="address2">
  <property name="add" value="Dombivali" />
</bean>

<bean class="l.com.example.spring.autowiring.annotation.TestBean" name="address">
  <property name="a" value="10" />
</bean>

<bean class="l.com.example.spring.autowiring.annotation.Emp" name="emp1" />
</beans>
```

1] Annotation on field

```
public class Emp {

    @Autowired
    @Qualifier("address1")
    private Address address;

    public Emp() {
        super();
    }
}
```

2] Annotation on setter()

```

@Autowired
@Qualifier("address2")
public void setAddress(Address address) {
    System.out.println("Setting Value using setter");
    this.address = address;
}

```

3] Annotation on constructor

```

@Autowired
public Emp(@Qualifier("address2") Address address) {
    super();
    this.address = address;
    System.out.println("Settting value Using Constructoe");
}

```

Spring Standalone Collection

1. Normally, when we define a **collection** inside a **<bean>**, it is limited to that bean's scope only.
2. However, using Spring's **util** schema, we can **define collections** (List, Set, Map, Properties) **separately**.
3. These **standalone collections** can then be **referenced in multiple beans**, promoting **reuse and cleaner XML configuration**.

```

import java.util.Set;

public class StandAlonCollection {

    private List<String> friends;
    private Set<Long> phoneNo;
    private Map<String, Integer> age;
    private Properties props;

    public List<String> getFriends() {
        return friends;
    }

    public void setFriends(List<String> friends) {
        this.friends = friends;
    }
}

```

```

http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util.xsd">

<util:list list-class="java.util.LinkedList" id="friendList">
  <value>Diksha</value>
  <value>Rutik</value>
  <value>Atish</value>
</util:list>

<util:set set-class="java.util.HashSet" id="phone">
  <value>1234</value>
  <value>5678</value>
  <value>9101</value>
</util:set>

<util:map map-class="java.util.TreeMap" id="ageOfFriends" >
  <entry key="Diksha" value="27" />
  <entry key="Rutik" value="22" />
  <entry key="Atish" value="23" />
</util:map>

<util:properties id="mysql">
  <prop key="driver">com.mysql.jdbc.Driver</prop>
  <prop key="user">root</prop>
  <prop key="pass">root</prop>
  <prop key="url">jdbc:mysql://localhost:3306</prop>
</util:properties>

<bean class="com.example.spring.stanfalone.collection.StandAlonCollection" name="bean1">
  <property name="friends" ref="friendList" />
  <property name="phoneNo" ref="phone" />
  <property name="age" ref="ageOfFriends" />
  <property name="props" ref="mysql" />
</bean>

```

Stereotype Annotations

Spring stereotype annotations are special annotations used to mark classes as Spring-managed components so that Spring can automatically detect and register them in the Spring container (IOC container).

Explanation:

In Spring, we don't always need to define beans manually in a configuration file (XML or Java-based). Instead, we can use **stereotype annotations** to tell Spring which classes should be **automatically managed** as beans.

These annotations are:

1. **@Component** – Generic annotation for any Spring-managed component.
2. **@Service** – Specifically used for **service layer** classes.
3. **@Repository** – Used for **DAO (Data Access Object) classes** that interact with the database.
4. **@Controller** – Used for **Spring MVC controllers** to handle web requests.

Note :

The `<context:component-scan>` tag is used in Spring XML configuration to enable component scanning. This allows Spring to automatically detect and register beans annotated with stereotype annotations (`@Component`, `@Service`, `@Repository`, `@Controller`) inside the specified package and its subpackages.

```
http://www.springframework.org/schema/util/spring-util.xsd">  
  
<context:component-scan base-package="n.com.example.stereotype.anno" ></context:component-scan>  
  
<util:list list-class="java.util.ArrayList" id="studentAdd">
```

Retrieving Beans from IoC Container (Spring Stereotype Annotations)

1. When we annotate a class with a stereotype annotation (`@Component`, `@Service`, `@Repository`, `@Controller`), Spring automatically registers it as a bean in the IoC container.
2. By default, Spring uses the class name as the bean name but follows camelCase convention:
 - a. The first letter of the class name is converted to lowercase.
 - b. Example:
 - i. `Student` → "student"
 - ii. `StudentInfo` → "studentInfo"

```
5  
6 @Component  
7 public class Student {  
8  
9     private int studentId;  
10    private String studentName;  
11  
12    public int getStudentId() {  
13        return studentId;  
14    }  
15 }
```

```
5 public class StereotypeAnno {  
6  
7     public static void main(String[] args) {  
8         ApplicationContext context = new ClassPathXmlApplicationContext("n/com/example/stereotype/anno.xml");  
9         Student student = context.getBean("student", Student.class);  
10        System.out.println(student);  
11    }  
12 }  
13  
14 }
```

3. Instead of using the default naming convention, we can **explicitly specify a bean name** inside the `@Component` annotation.

```
@Component("studentBean")  
public class Student {  
  
    private int studentId;  
}
```

```
public static void main(String[] args) {  
    ApplicationContext context = new ClassPathXmlApplicationContext("n/com/example/stereotype/anno.xml");  
    Student student = context.getBean("studentBean", Student.class);  
    System.out.println(student);  
}
```


Setting Values in Stereotype Annotations Using @Value

In Spring, we can use the `@Value` annotation to inject values into properties of a class that is annotated with a stereotype annotation (`@Component`, `@Service`, `@Repository`, `@Controller`).

```
5
6 @Component("studentBean")
7 public class Student {
8
9     @Value("1")
10    private int studentId;
11    @Value("Gaurav")
12    private String studentName;
13}
```

Setting Collection in Stereotype Annotations

We can inject collections into a bean that is annotated with a stereotype annotation (`@Component`, `@Service`, `@Repository`, `@Controller`) using the following methods:

1. Standalone collection defined in XML (`<util:list>`)
2. Spring Expression Language (SpEL) in `@Value` annotation

This allows Spring to fetch and inject collections from the IoC container into the bean automatically.

```
<context:component-scan base-package="n.com.example.stereotype">
<util:list list-class="java.util.ArrayList" id="studentAdd">
  <value>Mumbai</value>
  <value>Delhi</value>
  <value>Pune</value>
</util:list>
</beans>
```

```
@Value("Gaurav")
private String studentName;
@Value("#{studentAdd}")
private List<String> studentAddress;

public int getStudentId() {
    return studentId;
}
```

Spring Bean Scope

In Spring, we can define the scope of a bean to control how Spring manages its instances in the IoC (Inversion of Control) container. There are five types of bean scopes:

1. Singleton (Default)
2. Prototype
3. Request (*for web applications*)
4. Session (*for web applications*)
5. GlobalSession (*for web applications*)

Singleton Scope (Default)

In singleton scope, Spring creates only one instance of the bean, and the same instance is returned every time it is requested from the IoC container.

Key Points:

- This is the default scope in Spring.
- The same bean instance is shared across the application.
- Saves memory as only one object is created.
- Suitable for stateless beans (e.g., Service or DAO classes).

Example :

```
4 import org.springframework.stereotype.Component;
5
6 @Component("student")
7 public class Student {
8
9     @Value("10")
10    private int id;
11    @Value("Gaurav")
12    private String name;
13
14    public int getId() {
```

- Since the same object is returned, the hashCode of all references will be the same.
- Modifying the object will reflect the changes in all references because they all point to the same instance in the IoC container.

```
5
6 public class BeanScope {
7
8     public static void main(String[] args) {
9         ApplicationContext context = new ClassPathXmlApplicationContext("o/com/example/bean/scope/config.xml");
10
11         Student student1 = context.getBean("student", Student.class);
12         System.out.println(student1);
13         System.out.println(student1.hashCode());
14         student1.setName("Devang");
15
16         Student student2 = context.getBean("student", Student.class);
17         System.out.println(student2);
18         System.out.println(student2.hashCode());
19
20         Student student3 = context.getBean("student", Student.class);
21         System.out.println(student3);
22         System.out.println(student3.hashCode());
23     }
24 }
25
```

Spring IDE Console Output:

```
<terminated> BeanScope [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (Mar 16, 2025, 1:53:10 PM - 1:53:11 PM) [pid: 21312]
Student [id=10, name=Gaurav]
2074820378
Student [id=10, name=Devang]
2074820378
Student [id=10, name=Devang]
2074820378
```

Prototype Scope (Default)

- In prototype scope, the IoC container creates a new instance of the bean every time it is requested.
- Unlike singleton scope, where the same object is returned each time, prototype scope ensures that each request gets a fresh instance.
- This scope is useful when we need stateful beans, where each instance maintains its own separate state.

There are **two ways** to define the scope of a Spring bean:

1. Using `@Scope` Annotation (Java-based Configuration)
2. Using `scope` Attribute in XML Configuration

1. Using `@Scope` Annotation (Java-based Configuration)

We can use the `@Scope` annotation along with `@Component`, `@Service`, `@Repository`, or `@Bean` to specify the bean's scope.

```
7 @Component("student")
8 @Scope("prototype")
9 public class Student {
10
11     @Value("10")
12     private int id;
13     @Value("Gaurav")
14
```

```
6 public class BeanScope {
7
8     public static void main(String[] args) {
9         ApplicationContext context = new ClassPathXmlApplicationContext("o/com/example/bean/scope/config.xml");
10
11         Student student1 = context.getBean("student", Student.class);
12         System.out.println(student1);
13         System.out.println(student1.hashCode());
14         student1.setName("Devang");
15
16         Student student2 = context.getBean("student", Student.class);
17         System.out.println(student2);
18         System.out.println(student2.hashCode());
19
20         Student student3 = context.getBean("student", Student.class);
21         System.out.println(student3);
22         System.out.println(student3.hashCode());
23     }
24 }
25
```

Problems Servers Terminal Data Source Explorer Properties Console × Git Staging

<terminated> BeanScope [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (Mar 16, 2025, 2:59:17 PM – 2:59:20 PM) [pid: 24376]

Student [id=10, name=Gaurav]
70323523
Student [id=10, name=Gaurav]
1205445235
Student [id=10, name=Gaurav]
454104863

- Each call to `context.getBean("student", Student.class)` **creates a new object**.
- The **hashcodes are different** for each object, proving that new instances are created.
- Any **modification to one object does not affect other objects**, since they are **completely separate instances**.

Using **scope** Attribute in XML Configuration

- We can define the bean's scope in the `beans.xml` file using the `scope` attribute.

Example:

```
<bean id="student" class="com.example.Student" scope="prototype" />
```

Now, `Student` bean will be created every time it is requested.

Spring Expression Language (SpEL)

Spring Expression Language (SpEL) is a powerful expression language that allows us to dynamically evaluate expressions in Spring applications. It is mainly used to inject values into beans at runtime.

Types of Values We Can Inject Using Spring Expression Language (SpEL)

- Literals (String, Integer, Double, Boolean, Character)
- Mathematical & Logical Expressions (+, -, *, /, >, <, &&, ||)
- Ternary Operator (Conditional expressions using ? :)
- Bean Properties (Injecting values from another bean)
- Method Invocation (Calling instance/static methods)
- Collections (Lists, Sets, Maps from XML or beans)
- Class References (`T(ClassName).method()`)
- System Properties & Environment Variables (`systemProperties['propertyName']`)
- Regular Expressions (`matches()` method)

Syntax :

```
@Value("#{expression}")  
private DataType variableName;
```

Example :

Literals :

```
7 public class TestBean {  
8  
9     // Literals (String, Integer, Double, Boolean, Character)  
10    @Value("#{10}")  
11    private int a;  
12  
13    @Value("#{ 'Test' }")  
14    private String b;  
15  
16    @Value("#{1.1}")  
17    private double c;  
18  
19    @Value("#{true}")  
20    private boolean d;  
21 }
```

Mathematical & Logical Expressions :

```
// Mathematical & Logical Expressions
@Value("#{1 + 1}")
private int e;

@Value("#{2 * 2}")
private int f;

@Value("#{true && false}")
private boolean g;

@Value("#{true || false}")
private boolean h;
```

Ternary Expression

```
33 private boolean h;
34
35 // Ternary Expression
36 @Value("#{10 > 20? 1 : 100}")
37 private int j;
38
39 public int getA() {
```

Result

```
6
7 public class Spel {
8     public static void main(String[] args) {
9
10         ApplicationContext context = new ClassPathXmlApplicationContext("p/com/example/spel/config.xml");
11         TestBean bean = context.getBean("testBean", TestBean.class);
12         System.out.println(bean);
13     }
14 }
15
```

Problems Servers Terminal Data Source Explorer Properties Console × Git Staging

<terminated> Spel [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (Mar 16, 2025, 5:48:30 PM – 5:48:31 PM) [pid: 24812]

TestBean [a=10, b=Test, c=1.1, d=true, e=2, f=4, g=false, h=true, j=100]

```

2
3 public class GeneralUtil {
4
5     public static final String STR_CONSTANT = "This is static variable";
6
7     private String msg;
8
9     public static int addition(int a, int b) {
10         return a+b;
11     }
12
13     public GeneralUtil() {
14         super();
15         // TODO Auto-generated constructor stub
16     }
17
18     public GeneralUtil(String msg) {
19         super();
20         this.msg = msg;
21     }
22
23     public String getMsg() {
24         return msg;
25     }
26

```

Static Variables in SpEL

syntax

```

@Value("#{ T(class-name).variabel-name }")
private DataType variableName;

```

Note : In Spring Expression Language (SpEL), **T** stands for **"Type Reference"** and is used to access static fields and methods of a class.

```

// Static variables
@Value("#{T(java.lang.Math).PI}")
private double sqrRoot;

@Value("#{T(p.com.example.spel.GeneralUtil).STR_CONSTANT}")
private String strVar;

public int getA() {
    return a;
}

```

Static Methods in SpEL

syntax

```
@Value("#{ T(class-name).method-name(param) }")  
private DataType variableName;
```

```
45  
46 // Static Methods  
47 @Value("#{ T(java.lang.Math).sqrt(144) }")  
48 private double sqrRoot;  
49  
50 @Value("#{ T(p.com.example.spel.GeneralUtil).addition(2,5) }")  
51 private int sumOfInt;  
52
```

Object Instantiation using SpEL

syntax

```
@Value("#{ new class-name(param) }")  
private DataType variableName;
```

```
// Object  
@Value("#{ new String('This is gaurav') }")  
private String myName;  
  
@Value("#{ new p.com.example.spel.GeneralUtil('This value is set by SpEL') }")  
private GeneralUtil util;
```

Result

```
7 public class Spel {  
8     public static void main(String[] args) {  
9  
10        ApplicationContext context = new ClassPathXmlApplicationContext("p/com/example/spel/config.xml");  
11        TestBean bean = context.getBean("testBean", TestBean.class);  
12        System.out.println(bean);  
13        System.out.println(bean.getUtil().getMsg());  
14    }  
15 }
```

Problems Servers Terminal Data Source Explorer Properties Console X Git Staging

<terminated> Spel [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (Mar 16, 2025, 6:26:20 PM – 6:26:21 PM) [pid: 18064]

TestBean [pi=3.141592653589793, strVar=This is static variable, sqrRoot=12.0, sumOfInt=7, myName=This is gaurav, util=p.com.example.spel.GeneralUtil@7d9f158f]

This value is set by SpEL

Java Based Configuration

In Java-based configuration, we completely remove the XML configuration from our Spring application. Instead of defining beans in an XML file, we use Java classes and annotations to configure our Spring container.

Key Annotations:

@Configuration

- Marks the class as a configuration class, meaning it will be used to define Spring beans.
- This replaces the `configuration.xml` file.

@ComponentScan

- Specifies the package where Spring should scan for components, services, and repositories annotated with `@Component`, `@Service`, `@Repository`, and `@Controller`.
- This replaces the `<context:component-scan>` tag in XML.

```
5
6 @Component("student")
7 public class Student {
8
9     @Value("Gaurav")
10    private String studentName;
11
```

```
5 import org.springframework.context.annotation.ComponentScan;
6
7 @Configuration
8 @ComponentScan(basePackages = "q.com.example.java.base.config")
9 public class JavaConfig {
10 }
11
```

```
5
6 public class TestMain {
7
8     public static void main(String[] args) {
9         ApplicationContext context = new AnnotationConfigApplicationContext(JavaConfig.class);
10         Student student = context.getBean("student", Student.class);
11         System.out.println(student);
12     }
13 }
14
```

Important Note:

Since we are now using annotation-based configuration, we need to use the `AnnotationConfigApplicationContext` class to initialize the Spring container instead of `ClassPathXmlApplicationContext`.

♦ Key Difference:

- In XML-based configuration, we pass the XML file path to `ClassPathXmlApplicationContext`.
- In Java-based configuration, we pass the `ClassName.class` of our configuration class to `AnnotationConfigApplicationContext`.

Method 2: Using `@Bean` Instead of `@Component`

Instead of annotating our beans with `@Component` and using `@ComponentScan` to let Spring automatically scan the package, we can manually define beans using the `@Bean` annotation.

How Does It Work?

- We annotate a method with `@Bean` inside a `@Configuration` class.
- This method returns an instance of the bean.
- We can retrieve the bean using the method name or specify custom names inside `@Bean()`.

Why Use `@Bean`?

- More control over bean creation.
- Allows us to define multiple names for a bean.
- Useful when we **don't** want to modify the original class with `@Component`.

Example

```
4
5 public class Student {
6
7     @Value("Gaurav")
8     private String name;
9
10    public String getName() {
11        return name;
12    }
13 }
```

```

5
6 @Configuration
7 public class JavaConfig {
8
9     @Bean
10    public Student getStudent() {
11        Student student = new Student();
12        return student;
13    }
14 }
15

```

```

public static void main(String[] args) {

    ApplicationContext context = new AnnotationConfigApplicationContext(
        Student bean = context.getBean("getStudent", Student.class);
        System.out.println(bean);
    }
}

```

Or

```

5
6 @Configuration
7 public class JavaConfig {
8
9     @Bean(name = {"student", "firstStudent"})
10    public Student getStudent() {
11        Student student = new Student();
12        return student;
13    }
14 }
15

```

```

12    Student bean = context.getBean("student", Student.class);
13    System.out.println(bean);
14
15    Student bean2 = context.getBean("firstStudent", Student.class);
16    System.out.println(bean2);
17 }
18
19 }

```

Problems Servers Terminal Data Source Explorer Properties Console Git Sta

<terminated> TestMain (1) [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (Mar 16, 2025

Student [name=Gaurav]

Student [name=Gaurav]

Injecting Object

```
2
3 public class Samosa {
4
5     public void displayPrice() {
6         System.out.println("My price is 10 ruppes");
7     }
8 }
9
```

```
4 import org.springframework.context.annotation.Configuration;
5
6 @Configuration
7 public class JavaConfig {
8
9     @Bean(name = {"student", "firstStudent"})
10    public Student getStudent() {
11        Student student = new Student(getSamosa());
12        return student;
13    }
14
15    @Bean
16    public Samosa getSamosa() {
17        Samosa s = new Samosa();
18        return s;
19    }
20 }
21
```

```
6 public class TestMain {
7
8     public static void main(String[] args) {
9
10        ApplicationContext context = new AnnotationConfigApplicationContext(JavaConfig.class);
11
12        Student bean = context.getBean("student", Student.class);
13        System.out.println(bean);
14        bean.getSamosa().displayPrice();
15
16    }
17
18 }
19
```