Constructor Injection

Here IOC container uses parameterized constructor to create target class object.

In this process it assigns/injectes dependant object to the Target class Object.

syntax:

<bean id ='' class=''>

<constructor-arg name='' ref=''/>

</bean>

if we place<constructor-arg> tag for 'n' time under bean tag, then IOC container uses

"n-param" constructor to create the Spring bean class Object.

 <ref> is used in Spring when you want to **inject another bean (i.e., an object)** into a property of your class.  
👉 It links to another bean defined in the config file using its id.

 <value> is used when you want to **inject a simple value** like a number, text, or date into a property.  
👉 It just puts in a plain value like "Pavan" or 100

Eg: SpringCoreDependencyInjectionWithXMLConstructor

**⚙️ Sequential Execution Steps by IOC Container:**

1. **Start ApplicationContext**:
   * ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
   * Container loads beans.xml into memory as **in-memory DOM structure**.
2. **Bean Request Initiated**:
   * context.getBean("wmg", WishMessageGenerator.class);
   * Container checks internal cache for bean id "wmg".
3. **"wmg" Not in Cache → Parse XML**:
   * Finds <bean id="wmg" class="WishMessageGenerator">
   * Sees <constructor-arg name="time" ref="currentTime"/> → needs dependency injection.
4. **Resolve Dependency "currentTime"**:
   * Checks internal cache → **not found**.
   * Looks in XML → finds:

<bean id="currentTime" class="java.time.LocalTime" factory-method="now"/>

1. **Factory Method Invocation**:
   * Container loads class java.time.LocalTime.
   * Invokes static method now():

Method m = LocalTime.class.getMethod("now");

LocalTime time = (LocalTime) m.invoke(null);

* + ✅ **Creates LocalTime object**
  + Stores it in cache as "currentTime".

1. **Create Target Bean "wmg"**:
   * Loads class WishMessageGenerator.
   * Finds constructor: WishMessageGenerator(LocalTime time)
   * Instantiates:

WishMessageGenerator wmg = new WishMessageGenerator(time);

* + ✅ **Creates WishMessageGenerator object using constructor injection**
  + Stores wmg in internal cache.

1. **Return to Caller**:
   * Returns **fully constructed** WishMessageGenerator object to the caller.

**✅ Objects Created:**

* ✔️ LocalTime (via factory method now())
* ✔️ WishMessageGenerator (via constructor injection)

**✅ IOC Cache:**

currentTime → LocalTime

wmg → WishMessageGenerator

Most **predefined classes like java.util.Date or java.util.Calendar** do **not have setters** like setYear(), setMonth() in modern versions, and many of them are either deprecated or not Spring-friendly for direct property injection.

To simulate this properly in Spring, you must **wrap** the predefined object in a custom class.

Eg: DependantPredefinedClassPrimitiveInjection

**1️. Application Starts**

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

WishMessageGenerator wmg = context.getBean("wmg", WishMessageGenerator.class);

**2️. You Request Bean wmg**

Spring sees:

<bean id="wmg" class="WishMessageGenerator">

<constructor-arg name="calendarHolder" ref="calendarHolder"/>

</bean>

➡️ This means WishMessageGenerator needs a dependency: calendarHolder

So before creating wmg, Spring must create the calendarHolder bean first.

**3️. Spring Looks for calendarHolder**

It finds:

<bean id="calendarHolder" class="CalendarHolder" init-method="init">

<property name="year" value="1999"/>

<property name="month" value="6"/>

<property name="day" value="20"/>

</bean>

**4️. Spring Creates the Dependent Object calendarHolder Using Setter Injection**

* Loads class:  
  Class.forName("CalendarHolder");
* Creates object using zero-arg constructor:

CalendarHolder ch = new CalendarHolder();

* Injects values using setters:

ch.setYear(1999);

ch.setMonth(6);

ch.setDay(20);

* Calls the init() method:

ch.init(); // if defined, e.g., to calculate or validate date

✅ Now the calendarHolder bean is ready and stored in Spring's internal cache.

**5️. Spring Creates the Target Object wmg Using Constructor Injection**

* Now Spring goes back to wmg
* Calls the constructor of WishMessageGenerator like:

WishMessageGenerator wmg = new WishMessageGenerator(calendarHolder);

✅ Dependency is injected at object creation time.

**6️. Final Bean Cache:**

| **Bean ID** | **Object** |
| --- | --- |
| calendarHolder | CalendarHolder instance |
| wmg | WishMessageGenerator instance with calendarHolder injected |

**7️. Now Your Code Can Use the Bean**

wmg.showWishMessage();

// internally uses calendarHolder's values like year, month, day

Eg: SetterAndConstrutorInjectionForBean

### ****Step-by-Step Internal Flow:****

### 1️. ApplicationContext is created:

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

### 2️. You request:

Person p1 = context.getBean("person1", Person.class);

### 3️. Spring reads this bean:

<bean id="person1" class="Person">

<property name="pname" value="Pavan"/>

...

</bean>

### 4️. Spring creates the target object using ****zero-arg constructor****:

Person person1 = new Person(); // zero-arg constructor

### 5️. Spring injects values using setter methods:

person1.setPname("Pavan");

person1.setPid(9);

person1.setPaddress("Vijayawada");

person1.setIncome(200000.00f);

### 6️. Final Object Created:

Person{pname="Pavan", pid=9, paddress="Vijayawada", income=200000.00f}

✅ **Setter Injection** completed — values set after object is created.

## ✅ Explanation for person2 (Constructor Injection)

### ****Step-by-Step Internal Flow:****

### 1️. You request:

Person p2 = context.getBean("person2", Person.class);

### 2️. Spring reads:

<bean id="person2" class="Person">

<constructor-arg name="pname" value="Devadatta"/>

...

</bean>

### 3️. Spring prepares all constructor arguments:

String pname = "Devadatta";

int pid = 6;

String paddress = "Vijayawada";

float income = 200000.00f;

### 4️.Spring creates object using parameterized constructor:

Person person2 = new Person("Devadatta", 6, "Vijayawada", 200000.00f);

### 5️. Final Object Created:

Person{pname="Devadatta", pid=6, paddress="Vijayawada", income=200000.00f}

✅ **Constructor Injection** completed — values injected at creation time.

Usage of index/type/name attribute in the <constructor-arg>

===================================================

If there are multiple params in constructor and if they have same datatype, then to resolve the

parameter binding from the container we need to go either index/name(recommended)

Eg: IndexTypeNameAttributeInConstructor

**Circular Dependency Injection?**

It means **two or more classes depend on each other**.

For example:

java

CopyEdit

class A {

private B b;

public void setB(B b) { this.b = b; }

}

class B {

private A a;

public void setA(A a) { this.a = a; }

}

Here, A needs B, and B needs A – this is a **circular dependency**.

**🚫 Constructor Injection ❌ (Not Supported)**

If both classes use **constructor injection**, Spring **cannot resolve** them because:

* Spring tries to create object A → needs B → B needs A again → Infinite loop → ❌

java

CopyEdit

class A {

B b;

A(B b) { this.b = b; }

}

class B {

A a;

B(A a) { this.a = a; }

}

Spring throws an error here.

**✅ Setter Injection ✔ (Supported)**

In **setter injection**, Spring creates empty objects first, then injects dependencies using setters.

* Spring creates A and B first
* Then sets A.setB(B) and B.setA(A) safely

This avoids the loop.

**💡 Summary**

| **Injection Type** | **Supports Circular Dependency?** |
| --- | --- |
| Constructor | ❌ No |
| Setter | ✅ Yes |

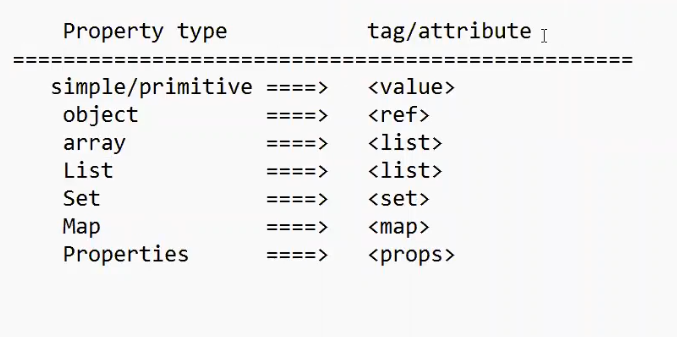
🔒 **Best Practice:** Avoid circular dependencies

Difference between constructor and setter injection

| **Feature** | **Constructor Injection** | **Setter Injection** |
| --- | --- | --- |
| **Injection Method** | Through constructor | Through setter methods (setX()) |
| **Syntax (XML)** | <constructor-arg .../> | <property name='' value=''/> or <property name='' ref=''/> |
| **Object Creation** | * When you use **constructor injection**, Spring creates the **dependent objects first**, and **injects them into the target class constructor** while creating the target object. * So, the **target object and its dependencies are created at the same time**, in one step. | "In setter injection, Spring first creates all dependent beans, then creates the target object using a zero-argument constructor, and finally calls setter methods to inject those dependencies." |
| **When Injection Happens** | At object creation | After object creation |
| **Speed** | Faster – all values set at once | Slightly slower – properties set after object is created |
| **Circular Dependency** | ❌ Not supported | ✅ Supported |
| **Optional Properties** | ❌ Not ideal – all values must be passed | ✅ Suitable – can set selected properties |
| **Immutability** | ✅ Promotes immutability | ❌ Not suitable for immutability |
| **Best Used For** | Mandatory dependencies | Flexible/optional dependencies or cyclic dependencies |

Collection Injection

=> It is all about injecting values to array, collection type bean properties through Dependancy injection.



**✅ Meaning of package bean;**

* **bean** usually means this package contains **JavaBeans** — simple POJOs (Plain Old Java Objects) that follow a specific convention:
  + Have private fields.
  + Provide public getter and setter methods.
  + Have a no-argument constructor.
  + Are serializable (optional, depending on use case).

**🧠 Why name it bean?**

* In Spring or other frameworks, we often create **model or data-holder classes** (like MarksInfo, Student, etc.).
* These classes are often treated as **Beans** managed by Spring.
* So, developers place them in a package called bean to logically organize code.

**📁 Example folder structure:**

src/

└── bean/

└── MarksInfo.java ← A JavaBean class for storing marks

Note:

In Spring XML configuration, factory-method="parse" is used

to tell the Spring container how to create a bean instance using a static

factory method instead of a constructor.

| **Property** | **factory-method="now"** | **factory-method="parse"** |
| --- | --- | --- |
| Purpose | To get the **current date/time** when the bean is created. | To **create a specific date/time** by parsing a string. |
| Method Type | Static method: now() | Static method: parse(String text) |
| Argument | ❌ No arguments needed | ✅ Requires a string as argument |
| Example Value | — | "2025-05-23" |

Collection Injection:

Eg: CollectionInjectionArray

Eg: CollectionInjectionList

Eg: CollectionInjectionSet

Eg:CollectionInjectionMap

Eg: CollectionInjectionProperties

NullInjection

===========

In constructor injection, all params must participate in injection process otherwise it would result in "Exception".

If constructor param type is object/reference type and we are not ready with value then we can go for null injection.

This is very handy(useful) when we are working with predeifned classes as a spring bean, that is a spring bean will have limited no of Overloaded constructors and no setter injection support is available.

Syntax:

| **Style** | **Example** |
| --- | --- |
| By index | <constructor-arg index="2"><null/></constructor-arg> |
| By name | <constructor-arg name="dob"><null/></constructor-arg> |
| By order | <constructor-arg><null/></constructor-arg> |

Eg: BeanInheritance

Eg: BeanInheritanceConstructor

Eg: BeanId