**✅ Autowiring in Spring (a.k.a Dependency Injection)**

**➤ Definition:**

Assigning the dependent class object to the target class object is called **Dependency Injection** or **Autowiring**.

**🧩 Types of Autowiring**

**1. Explicit Autowiring / Manual Injection**

* You manually specify dependencies using XML configuration.

**a. Using Setter Injection:**

<property name="courier" ref="blueDart"/>

**b. Using Constructor Injection:**

<constructor-arg name="courier" ref="blueDart"/>

**2. Autowiring / Auto-Injection**

* Spring will automatically detect and wire beans.

<bean id="flipkart" class="in.ineuron.bean.Flipkart" autowire="byName"/>

**Autowire Modes:**

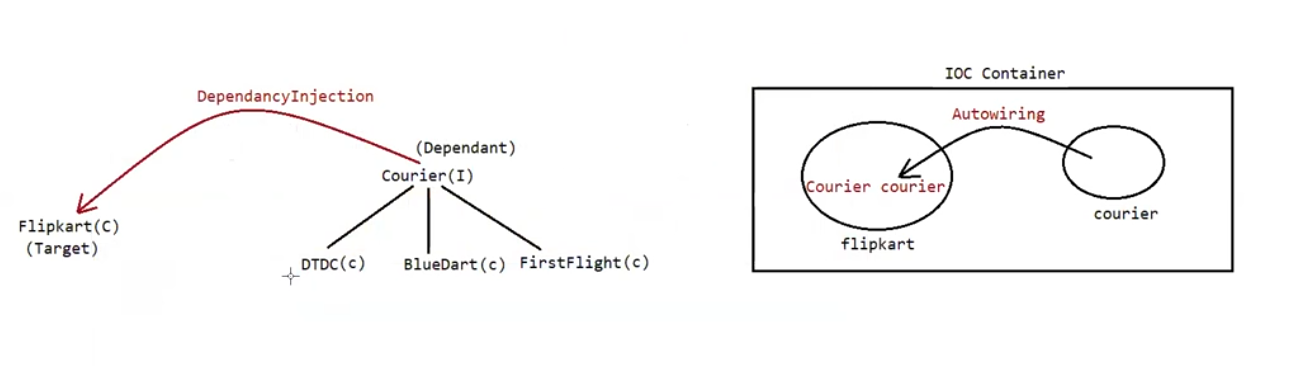
* **byName**: Matches bean id with property name.
* **byType**: Matches bean class type with property type.
* **constructor**: Matches constructor arguments by type or name.

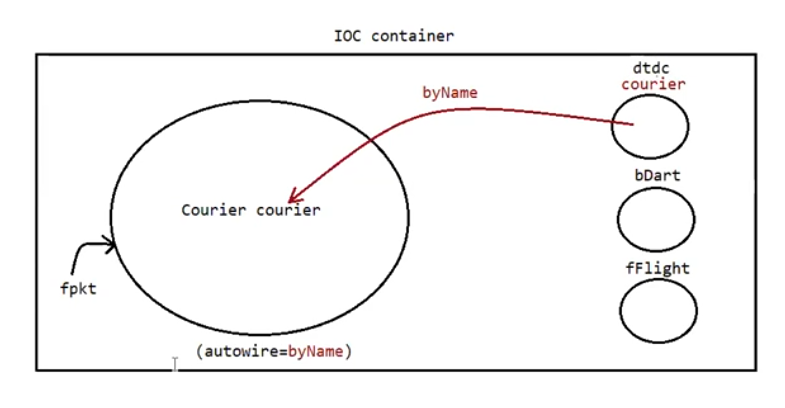
**⚠️ Limitations of Autowiring**

1. Works **only on object/reference-type** properties.
2. May cause **ambiguity problems** if multiple beans of same type exist.
3. **Reduces readability** of Spring configuration (applicationContext.xml).

Note : lazy-init="true": Bean is not instantiated unless explicitly requested.

Eg: AutoWireByName





**🔧 autowire="byName" in Spring**

**✅ How It Works:**

* **Injection Type:** Setter Injection
* **Mechanism:** The Spring container matches the **property name** in the target class with a **bean ID** in the configuration.
* If they match, the dependency is automatically injected via the setter method.

**🧩 Example:**

**Java Class**

private Courier courier; // property name = "courier"

**Spring XML**

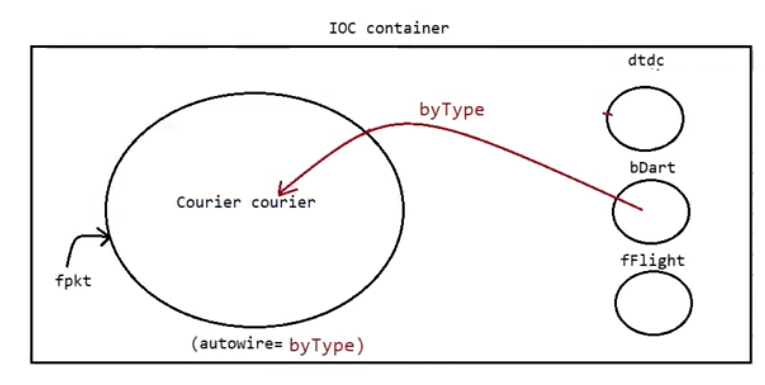
<bean id="courier" class="in.ineuron.bean.BlueDart"/>

<bean id="fpkt" class="in.ineuron.bean.Flipkart" autowire="byName"/>

➡ Since the property name is courier and a bean with id "courier" exists, Spring injects it automatically using the setter method setCourier().

**🚫 No Ambiguity Problem**

* Bean IDs in the Spring IoC container are **unique**, so there's no chance of multiple beans conflicting when using byName.



Eg: AutowireByType

**🔧 autowire="byType" in Spring**

**✅ How It Works:**

* **Injection Type:** Setter Injection
* **Mechanism:** Spring searches for a **bean with a matching type** (not name).
* It injects the bean based on the **class type of the property** in the target class.

**🧩 Example:**

**Java Property**

private Courier courier; // Type: Courier

**Spring XML**

<bean id="blueDart" class="in.ineuron.bean.BlueDart"/>

<bean id="fpkt" class="in.ineuron.bean.Flipkart" autowire="byType"/>

➡ Spring finds a bean of type Courier and injects it using setCourier().

**⚠️ Ambiguity Issue**

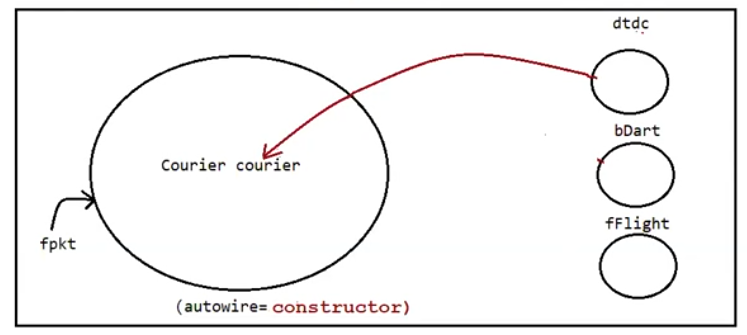
* If **multiple beans of the same type** exist (e.g., BlueDart, DTDC, etc., all implement Courier), Spring gets confused.

**✅ Solution:**

* Use primary="true" to **mark one bean as the default**:

<bean id="blueDart" class="in.ineuron.bean.BlueDart" primary="true"/>

Eg: AutowireByConstructor



**🔧 autowire="constructor" in Spring**

**✅ How It Works:**

* **Injection Type:** Constructor Injection (using a parameterized constructor).
* Spring injects the dependency via the class constructor

**"The constructor parameter's type should match a bean's class/type for autowiring to succeed."**

**🔍 Explanation:**

* Spring uses the **type of the constructor parameter** to resolve which bean to inject.
* **It does NOT use the constructor parameter name** to match the bean ID in XML-based configuration (unless you're using advanced setups like -parameters compiler flag + annotations).

**✅ Summary:**

| **Match Criteria** | **Used by Spring autowire="constructor"?** |
| --- | --- |
| Constructor **type** | ✅ Yes |
| Constructor **name** | ❌ No (unless explicitly configured) |
| Bean **type** | ✅ Yes |
| Bean **id** | ❌ Not directly (used in manual injection, not autowiring) |

**🧩 Example:**

public class Flipkart {

private Courier courier;

public Flipkart(Courier courier) {

this.courier = courier;

}

}

**Spring XML**

<bean id="courier" class="in.ineuron.bean.BlueDart"/>

<bean id="fpkt" class="in.ineuron.bean.Flipkart" autowire="constructor"/>

➡ Spring finds the matching bean (courier) and injects it through the constructor.

**🛡️ No Ambiguity Problem (Usually)**

* Since **constructor parameter names are unique**, ambiguity is less common.
* But if multiple beans of the same type exist (e.g., multiple Courier beans), ambiguity may still occur.

**✅ Note:**

If the **constructor param name matches a bean ID** and one of those beans is marked with primary="true", that bean will be injected **with highest priority**.

<bean id="courier" class="in.orcas.bean.DTDC" primary="true"/>

### ✅ Purpose of autowire-candidate="false"

You're using this attribute to **exclude specific beans (BlueDart, FirstFlight) from autowiring**, resolving ambiguity when using:

<bean id='fpkt' class='in.ineuron.bean.Flipkart' autowire="byType">

<bean id='bDart' class='in.ineuron.bean.BlueDart' autowire-candidate="false"/>

<bean id='dtdc' class='in.ineuron.bean.DTDC' primary="true"/>

<bean id='courier' class='in.ineuron.bean.FirstFlight' autowire-candidate='false'/>

**🎯 Target Bean (Flipkart):**

<bean id='fpkt' class='in.ineuron.bean.Flipkart' autowire="byType">

<property name='regNo' value='12345'/>

</bean>

✅ Since Flipkart expects a dependency of type Courier, and only dtdc is a candidate, Spring injects dtdc.

**🧠 Final Notes:**

* Use autowire-candidate="false" for beans **you don’t want Spring to inject automatically**.
* Ensure only one candidate remains for autowire="byType" or use primary="true" for preference among multiple.

What is the difference b/w them?

autowire = no => Disables the autowiring, programmer should explicity perform Autowiring.

autowire-candidate =false => it makes the spring bean not to participate in Autowiring.

**autowire-candidate="false"?**

This tells Spring:

"Do **not consider this bean** when performing **automatic dependency injection** like autowire="byType" or @Autowired."

But it **does NOT block manual injection**.

**🔍 Let's Understand the XML Example Step-by-Step:**

<bean id='bDart' class='in.ineuron.bean.BlueDart' autowire-candidate='false'/>

* ✅ This bean exists in the Spring container.
* ❌ But it **won’t be picked** when Spring tries to inject something automatically (e.g., using @Autowired, autowire="byType").

<bean id='fpkt' class='in.ineuron.bean.Flipkart' autowire='no'>

<constructor-arg ref='bDart'/>

</bean>

* autowire='no' disables automatic injection for Flipkart.
* Instead, you're manually saying:  
  **"Inject the bDart bean into the Flipkart constructor."**

**✅ So, what is happening?**

Even though bDart has autowire-candidate="false", Spring **doesn’t stop you from manually injecting it** using <constructor-arg ref="bDart"/>.

This is called **explicit dependency injection**, and it always works.

Eg: SingletonScope

**1. Singleton Scope (default):**

Spring creates **only one instance per container** for a bean defined with scope="singleton", and reuses it every time getBean() is called.

**🧾 applicationContext.xml:**

<bean id="wmg" class="in.ineuron.bean.WishMessageGenerator" scope="singleton">

<property name="date" ref="dt"/>

</bean>

This means:

* Spring will create **one instance** of WishMessageGenerator.
* The date dependency is injected via setter.

**🧪 Java Code:**

WishMessageGenerator generator1 = factory.getBean("wmg", WishMessageGenerator.class);

WishMessageGenerator generator2 = factory.getBean("wmg", WishMessageGenerator.class);

System.out.println("Generator1 class object reference :: " + generator1.hashCode());

System.out.println("Generator2 class object reference :: " + generator2.hashCode());

**✅ Output (Example):**

Generator1 class object reference :: 12345678

Generator2 class object reference :: 12345678

✅ Both generator1 and generator2 point to the **same object** → because of **singleton** scope.

**✅ Summary:**

* **Singleton scope** ensures a **single shared object**.
* Efficient for **stateless beans** like services or DAOs.
* Use hashCode() to verify if two references point to the same instance.

Eg: PrototypeScope

**🔄 Prototype Scope:**

**🔹 Key Points:**

* Spring **creates a new object** every time getBean() is called.
* Spring **does not store** prototype beans in its internal cache.

**📁 applicationContext.xml:**

<bean id="wmg" class="in.ineuron.bean.WishMessageGenerator" scope="prototype">

<property name="date" ref="dt"/>

</bean>

This tells Spring:

* Always return a **new instance** of WishMessageGenerator whenever it's requested.

**🧪 ClientApp.java:**

WishMessageGenerator generator1 = factory.getBean("wmg", WishMessageGenerator.class);

WishMessageGenerator generator2 = factory.getBean("wmg", WishMessageGenerator.class);

System.out.println("Generator1 class object reference :: " + generator1.hashCode());

System.out.println("Generator2 class object reference :: " + generator2.hashCode());

**✅ Expected Output:**

Generator1 class object reference :: 12345678

Generator2 class object reference :: 87654321

Each getBean() call returns a **different object** → because of **prototype** scope.

### ****ApplicationContext Container****

* **1.** It is an extension of **BeanFactory**.
* **2.** Implementation classes of ApplicationContext interface:
  + **a.** FileSystemXmlApplicationContext – for **standalone** applications; loads XML config from any filesystem path.
  + **b.** ClassPathXmlApplicationContext – for **standalone** applications; loads XML config from the classpath.
  + **c.** XmlWebApplicationContext – for **Spring MVC** web apps using XML configuration.
  + **d.** AnnotationConfigApplicationContext – for **standalone** apps using **Java-based** configuration (@Configuration).
  + **e.** AnnotationConfigWebApplicationContext – for **Spring MVC** web apps using **Java-based** configuration.

**🔹 Pre-instantiation of Singleton Beans**

* When using ApplicationContext in Spring, all beans defined with **singleton scope** are created **at startup**, not when they are first requested.
* This is **different** from BeanFactory, which instantiates beans **lazily** (i.e., when getBean() is called).

**🔹 Relevance in Spring MVC**

* The DispatcherServlet, which acts as the **Controller** in the MVC pattern, is configured as a **singleton**.
* It is also set with **load-on-startup** in web.xml or Spring Boot's auto-configuration so that it initializes **early** in the app lifecycle.

**🔄 Flow Illustrated:**

UI (JSP)

|

v

Controller (DispatcherServlet) <-- load-on-startup, singleton

|

v

Service Layer <-- singleton

|

v

DAO Layer <-- singleton

|

v

Database

**💡 Summary:**

* All layers (Controller, Service, DAO) are **singleton beans**, and they are instantiated at the time of container startup because of ApplicationContext.
* This ensures readiness and reduces latency during request handling.
* Pre-instantiation is particularly helpful for wiring dependencies and detecting configuration issues early.

<bean id='bDart' class='in.ineuron.bean.BlueDart' lazy-init="true"/>

🔍 Meaning:

lazy-init="true" tells the Spring container not to instantiate the bean when the ApplicationContext is initialized.

Instead, the bean will be created only when it is first requested (i.e., on-demand).

### Support of I18N (Internationalization)

Making our application work for all different locales is called **I18N**.

* **Locale** = Language + Country
* **Examples**:
  + en-US (English - United States)
  + en-BR → (Incorrect: should be pt-BR for Portuguese - Brazil)
  + hi-IN (Hindi - India)
  + fr-FR (French - France)
  + de-DE (German - Germany)

Eg: Internationalization