**Spring - IoC Containers**

* The Spring container is at the core of the Spring Framework. The container will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction.
* The Spring container uses DI to manage the components that make up an application. These objects are called Spring Beans.
* The container gets its instructions on what objects to instantiate, configure, and assemble by reading the configuration metadata provided.
* The configuration metadata can be represented either by XML, Java annotations, or Java code.
* The following diagram represents a high-level view of how Spring works.
* The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.

**Types:**

Spring provides the following two distinct types of containers.

[**Spring BeanFactory Container**](https://www.tutorialspoint.com/spring/spring_beanfactory_container.htm)

[**Spring ApplicationContext Container**](https://www.tutorialspoint.com/spring/spring_applicationcontext_container.htm)

**Differences :**

|  |  |  |
| --- | --- | --- |
|  | **BeanFactory** | **ApplicationContext** |
|  | This is the simplest container providing the basic support for DI | This container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners. |
|  | This is defined by the *org.springframework.beans.factory.BeanFactory*  interface.*.* | This container is defined by the *org.springframework.context.ApplicationContext* interface. |
|  |  |  |

**Note :** The *ApplicationContext* container includes all functionality of the *BeanFactory*container, so it is generally recommended over *BeanFactory*. BeanFactory can still be used for lightweight applications like mobile devices or applet-based applications where data volume and speed is significant.

**Using BeanFactory**

The XmlBeanFactory is the implementation class for the BeanFactory interface. To use the BeanFactory, we need to create the instance of XmlBeanFactory class as given below:

Resource resource=new ClassPathResource("applicationContext.xml");

BeanFactory factory=new XmlBeanFactory(resource);

The constructor of XmlBeanFactory class receives the Resource object so we need to pass the resource object to create the object of BeanFactory.

**Using ApplicationContext**

The ClassPathXmlApplicationContext class is the implementation class of ApplicationContext interface. We need to instantiate the ClassPathXmlApplicationContext class to use the ApplicationContext as given below:

ApplicationContext context =

    new ClassPathXmlApplicationContext("applicationContext.xml");

The constructor of ClassPathXmlApplicationContext class receives string, so we can pass the name of the xml file to create the instance of ApplicationContext.

**Dependency Injection in Spring**

The Dependency Injection is a design pattern that removes the dependency of the programs. In such case we provide the information from the external source such as XML file. It makes our code loosely coupled and easier for testing. In such case we write the code as:

class Employee{

Address address;

Employee(Address address){

this.address=address;

}

public void setAddress(Address address){

this.address=address;

}

}

In such case, instance of Address class is provided by external souce such as XML file either by constructor or setter method.

### Two ways to perform Dependency Injection in Spring framework

Spring framework provides two ways to inject dependency

* By Constructor
* By Setter method

**Difference between constructor and setter injection**

There are many key differences between constructor injection and setter injection.

1. **Partial dependency**: can be injected using setter injection but it is not possible by constructor. Suppose there are 3 properties in a class, having 3 arg constructor and setters methods. In such case, if you want to pass information for only one property, it is possible by setter method only.
2. **Overriding**: Setter injection overrides the constructor injection. If we use both constructor and setter injection, IOC container will use the setter injection.
3. **Changes**: We can easily change the value by setter injection. It doesn't create a new bean instance always like constructor. So setter injection is flexible than constructor injection.

# Autowiring in Spring

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 of Autowiring

It requires the **less code** because we don't need to write the code to inject the dependency explicitly.

# Disadvantage of Autowiring

No control of programmer.

## **Autowiring Modes**

There are many autowiring modes:

|  |  |  |
| --- | --- | --- |
| **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. |

## **Example of Autowiring**

Let's see the simple code to use autowiring in spring. You need to use autowire attribute of bean element to apply the autowire modes.

1. <bean id="a" **class**="org.sssit.A" autowire="byName"></bean>

Let's see the full example of autowiring in spring. To create this example, we have created 4 files.

1. **B.java**
2. **A.java**
3. **applicationContext.xml**
4. **Test.java**

**B.java**

This class contains a constructor and method only.

1. **package** org.sssit;
2. **public** **class** B {
3. B(){System.out.println("b is created");}
4. **void** print(){System.out.println("hello b");}
5. }

**A.java**

This class contains reference of B class and constructor and method.

1. **package** org.sssit;
2. **public** **class** A {
3. B b;
4. A(){System.out.println("a is created");}
5. **public** B getB() {
6. **return** b;
7. }
8. **public** **void** setB(B b) {
9. **this**.b = b;
10. }
11. **void** print(){System.out.println("hello a");}
12. **void** display(){
13. print();
14. b.print();
15. }
16. }

**applicationContext.xml**

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
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-3.0.xsd">
9. <bean id="b" **class**="org.sssit.B"></bean>
10. <bean id="a" **class**="org.sssit.A" autowire="byName"></bean>
12. </beans>

**Test.java**

This class gets the bean from the applicationContext.xml file and calls the display method.

1. **package** org.sssit;
2. **import** org.springframework.context.ApplicationContext;
3. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
4. **public** **class** Test {
5. **public** **static** **void** main(String[] args) {
6. ApplicationContext context=**new** ClassPathXmlApplicationContext("applicationContext.xml");
7. A a=context.getBean("a",A.**class**);
8. a.display();
9. }
10. }

Output:

*b is created*

*a is created*

*hello a*

*hello b*