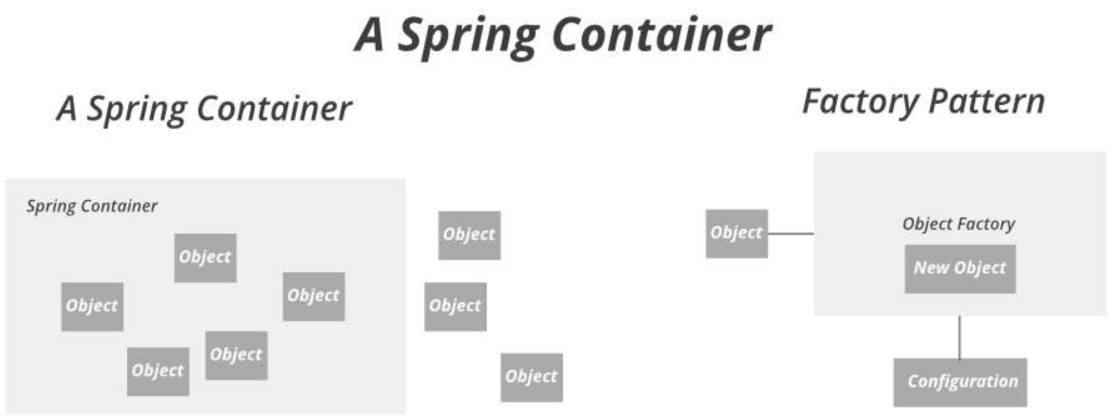
Spring - BeanFactory

Last Updated : 23 Jul, 2025

The first and foremost thing when we talk about Spring is dependency injection which is possible because Spring is a container and behaves as a factory of **Beans**. Just like the **BeanFactory** interface is the simplest container providing an advanced configuration mechanism to instantiate, configure, and manage the life cycle of beans.

Beans are Java objects that are configured at run-time by Spring IoC Container. BeanFactory represents a basic **IoC container** which is a parent interface of **ApplicationContext. BeanFactory** uses Beans and their dependencies metadata to create and configure them at run-time. BeanFactory loads the bean definitions and dependency amongst the beans based on a configuration file (XML) or the beans can be directly returned when required using Java Configuration. There are other types of configuration files like LDAP, RDMS, properties files, etc. BeanFactory does not support Annotation-based configuration whereas ApplicationContext does.



Let us first go through some of the methods of Bean Factory before landing up on implementation which are shown below in tabular format:

Method	Description
containsBean(String name)	Does this bean factory contain a bean definition or externally registered singleton instance with the given name?
getAliases(String name)	Return the aliases for the given bean name, if any.
getBean(Class<T> requiredType)	Return the bean instance that uniquely matches the given object type, if any.
getBean(Class<T> requiredType, Object... args)	Return an instance, which may be shared or independent, of the specified bean.
getBean(String name)	Return an instance, which may be shared or independent, of the specified bean.
getBean(String name, Class<T> requiredType)	Return an instance, which may be shared or independent, of the specified bean.
getBean(String name, Object... args)	Return an instance, which may be shared or independent, of the specified bean.
getBeanProvider(Class<T> requiredType)	Return a provider for the specified bean, allowing for lazy on-demand

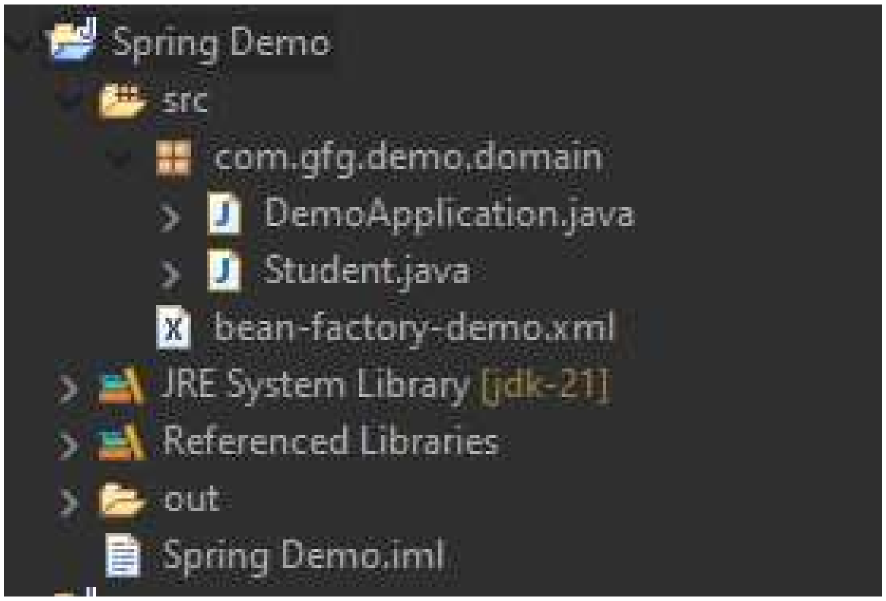
Method	Description
	retrieval of instances, including availability and uniqueness options.
getBeanProvider(ResolvableType requiredType)	Return a provider for the specified bean, allowing for lazy on-demand retrieval of instances, including availability and uniqueness options.
getType(String name)	Determine the type of the bean with the given name.
getType(String name, boolean allowFactoryBeanInit)	Determine the type of the bean with the given name.
isPrototype(String name)	Is this bean a prototype? That is, will getBean(java.lang.String) always return independent instances?
isSingleton(String name)	Is this bean a shared singleton? That is, will getBean(java.lang.String) always return the same instance?
isTypeMatch(String name, Class<?> typeToMatch)	Check whether the bean with the given name matches the specified type.
isTypeMatch(String name, ResolvableType typeToMatch)	Check whether the bean with the given name matches the specified type.

Procedure:

- First, create a Spring project using start.spring.io.
- Create a POJO class.
- Configure the Student bean in the **bean-factory-demo.xml** file.
- Then write it to application class.

Project Structure:

After creating all packages and classes, the project structure will look like below:



```
Spring Demo
  src
    com.gfg.demo.domain
      DemoApplication.java
      Student.java
    bean-factory-demo.xml
  JRE System Library [jdk-21]
  Referenced Libraries
  out
  Spring Demo.iml
```

Step-by-Step Implementation to Configure Bean Factory in Spring

Step 1: Create a Student POJO class.

Now we will define bean inside the Student class file.

Student.java:

```java
// Java Program where we are
// creating a POJO class

// POJO class
public class Student {

  // Member variables
  private String name;
  private String age;

  // Constructor 1
  public Student() {
  }

  // Constructor 2
  public Student(String name, String age) {
    this.name = name;
    this.age = age;
  }

  // Method inside POJO class
  @Override
  public String toString() {

    // Print student class attributes
    return "Student{" + "name='" + name + '\'' + ", age='" + age + '\'' +
'}';
  }
}
```

Step 2: Configure the Student bean in the ***bean-factory-demo.xml*** file.

XML Bean Configuration:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans/"
        xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans/
        https://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="student" class="com.gfg.demo.domain.Student">
        <constructor-arg name="name" value="Tina"/>
        <constructor-arg name="age" value="21"/>
    </bean>
</beans>
```

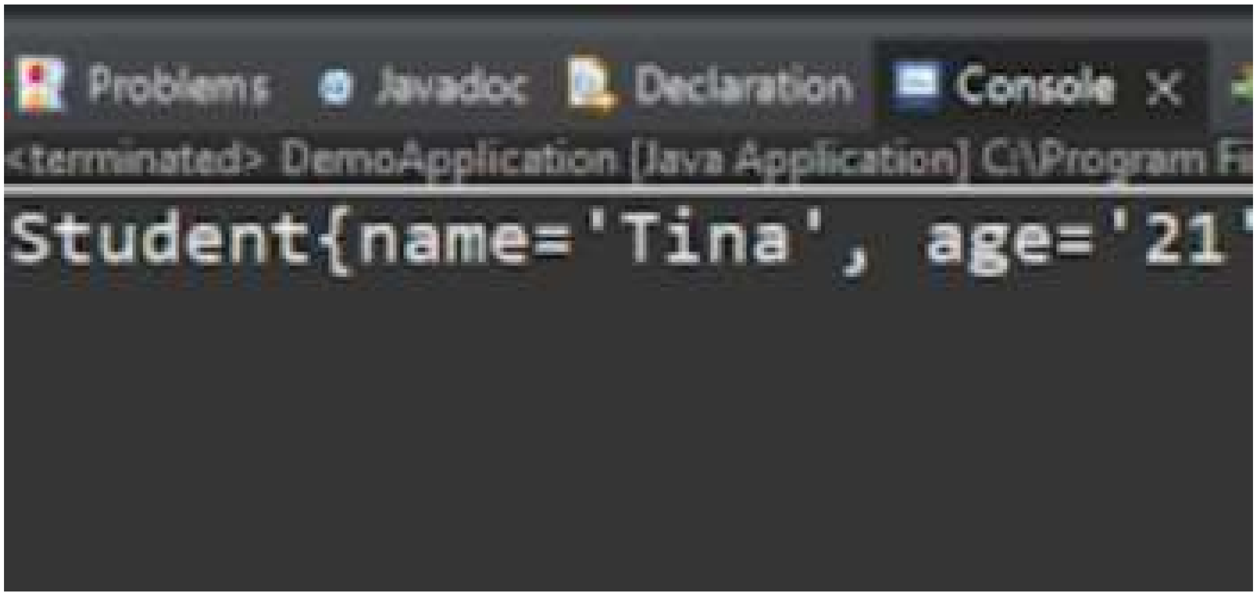
Step 3: Now let's write the main class file.

```java
@SpringBootApplication
// Main class
public class DemoApplication
{
// Main driver method
  public static void main(String[] args)
  {
    // Creating object in a spring container (Beans)
    BeanFactory factory = new ClassPathXmlApplicationContext("bean-factory-
demo.xml");
    Student student = (Student) factory.getBean("student");

    System.out.println(student);
    }
}
```

Output:

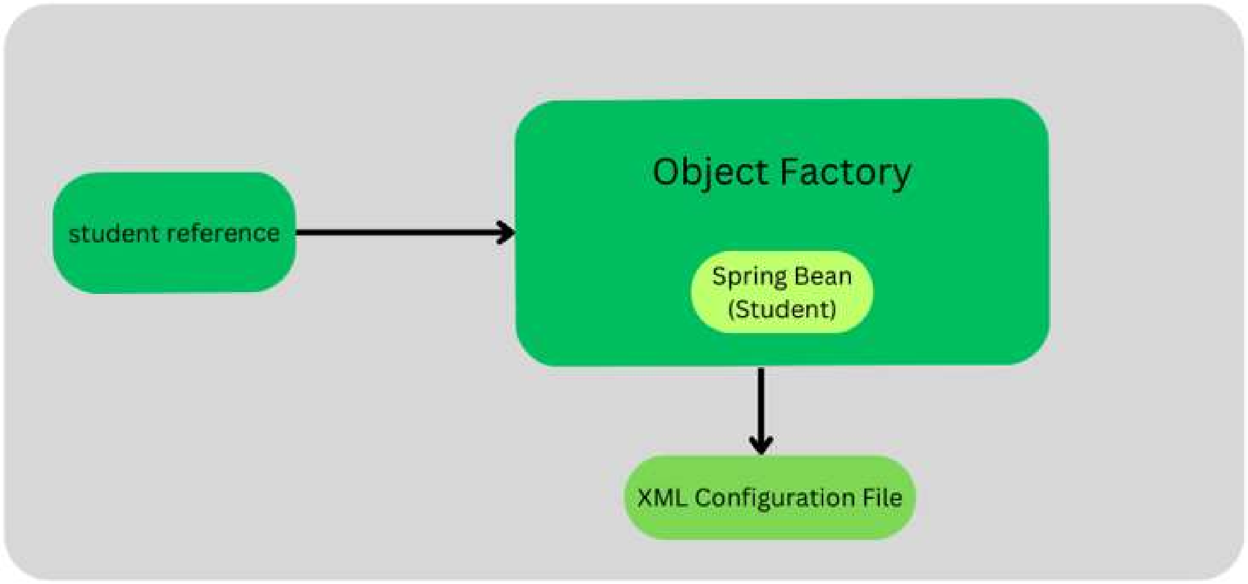
```
Student{name='Tina', age='21'}
```



```
<terminated> DemoApplication [Java Application] C:\Program Fi
Student{name='Tina', age='21'
```

> ***Note: XmlBeanFactory class** is deprecated.*

Let's understand the above code with visuals:



The program flow is something like this:

- First of all, the Bean factory reads the XML configuration file and as per the specifications defined in it, it creates the bean of the student POJO.

- Then the student reference asks for the student object from the object factory.
- Then finally, the spring object factory hands over the spring bean (student) to its reference. Here, note that the bean returned by the object factory is of "**Object**" type, so we have to typecast it into our desired bean.